
Introduction to Analysis and Modeling in Biology with Python

Simen Tennøe^{1,2}

Andreas V. Solbrå^{1,3}

Milad H. Mobarhan¹

Svenn-Arne Dragly^{1,3}

Lex Nederbragt^{4,5}

¹Centre for Integrative Neuroplasticity (CINPLA), Department of Biosciences,
University of Oslo

²Department of Informatics, University of Oslo

³Department of Physics, University of Oslo

⁴Centre for Ecological and Evolutionary Synthesis, Department of Biosciences,
University of Oslo

⁵Centre for Bioinformatics, Department of Informatics, University of Oslo

Aug 10, 2022

Contents

1 Using Python as a Scientific Calculator	8
1.1 Getting started with Python	10
1.1.1 Starting Jupyter Notebook	11
1.1.2 Creating your first notebook	11
1.2 Writing code	13
1.2.1 Writing your first Python code	13
1.2.2 Variables: Storing data	14
1.2.3 Calculations with numbers	16
1.2.4 Converting between strings and numbers	20
1.2.5 How functions are used	21
1.2.6 Order of operations	22
1.2.7 Commenting the code	23
1.2.8 Debugging the code	24
1.2.9 Using packages to get pre-made functions	24
1.3 Computer science glossary	26
1.4 Summary	27
2 Protein Secretion Analysis	29
2.1 Insulin and blood glucose levels	30
2.2 Measuring insulin production	31
2.3 Plotting insulin production	31
2.3.1 Lists: Storing many data elements in one variable	32
2.3.2 Plots: Plotting data over time	36
2.3.3 Using list slicing to examine the linear growth phase	38
2.3.4 Plotting the breakfast response	40
2.4 Exploring the effects of obesity on insulin secretion	43
2.4.1 Multiple lines in the same plot	43
2.4.2 Different line styles	46
2.4.3 Saving the plot to file	49
2.4.4 Plotting insulin secretion rates for non-obese and obese subjects	49
2.5 Modeling the linear growth of insulin production	50
2.5.1 Revisiting the effect of obesity on the secretion rate of insulin	53
2.5.2 Linear functions for insulin secretion in non-obese and obese subjects	54
2.6 Simulating insulin secretion using our model	54
2.6.1 How soon after a meal does insulin production start?	56

2.7	Visualizing 24 hours of insulin secretion	58
2.7.1	Plotting two series in one plot	60
2.8	Summary	62
2.8.1	Lists	62
2.8.2	Plotting	62
2.8.3	Reading from file	63
3	Modeling unlimited bacterial population growth	64
3.1	How bacteria multiply: binary fission	66
3.2	Analyzing <i>E. coli</i> population growth	67
3.2.1	Visualizing the phases of bacterial population growth	67
3.2.2	Plotting with a logarithmic scale	69
3.2.3	Examining the different growth phases	72
3.3	Examining the growth rate	76
3.3.1	Exponential growth with logarithmic scale	76
3.3.2	Calculating growth rate and doubling time from data	78
3.4	Exponential growth: A simple model for bacterial population growth	79
3.5	While loops: Repeating a piece of code many times	85
3.5.1	Understanding the while loop	86
3.5.2	Common errors when writing while loops	88
3.5.3	Using while loops to loop over elements in a list	89
3.6	Implementing exponential growth	90
3.7	Adding time to our model	93
3.8	Limitations of the exponential growth model	98
3.9	Difference equations: A mathematical look at what we have done	101
3.9.1	A brief overview of difference equations	101
3.10	Summary	102
3.10.1	While-loops	102
3.10.2	Mathematical symbols and their corresponding variable names	103
3.10.3	Exponential growth	103
3.10.4	Chapter glossary	104
4	Models for limited bacterial population growth	105
4.1	Logistic growth: A more detailed model	106
4.1.1	Requirements of our new model	106
4.1.2	The logistic growth equation	109
4.1.3	A comparison of logistic and exponential growth	113
4.2	Modeling the death phase	117
4.2.1	Simple model for the death phase	117
4.2.2	Detailed model with long-term stationary phase	122
4.3	Modeling the lag phase	126
4.4	Modeling all phases of the bacterial population growth	127
4.5	Summary	130
4.5.1	Mathematical symbols and their corresponding variable names	130
4.5.2	Lag phase	131
4.5.3	Logistic growth	131
4.5.4	Simple death phase	132

4.5.5	Advanced death phase	132
4.5.6	Complete model of the bacterial population growth	133
5	Modeling plant population growth	135
5.1	One-year model for plant population growth	136
5.1.1	Implementing the first year	138
5.1.2	Implementing the following years	141
5.1.3	Expressing the model as a difference equation	143
5.1.4	Examining low survival rate	144
5.1.5	Examining different survival and germination rates	146
5.1.6	Estimating model parameters	146
5.2	Variable germination rate	148
5.3	Two-year model for plant population growth	152
5.3.1	Adding two-year-old seeds to the model	152
5.3.2	Variable germination in the two-year model	158
5.3.3	Subplots: Showing two plots in the same figure	159
5.3.4	Expressing the two-year model as a difference equation	161
5.4	Summary	162
5.4.1	Mathematical symbols and their corresponding variable names	163
5.4.2	One-year model of annual plant population growth	163
5.4.3	Two-year model of annual plant population growth	164
5.4.4	Range	165
5.4.5	Subplots	165
6	Modeling inheritance	167
6.1	Mendelian genetics	168
6.1.1	Mendel’s model of traits and genes	171
6.1.2	Using a Punnett square to determine the offspring of two parents	171
6.1.3	Implementing a Punnett square	173
6.2	Functions: Reusing code	178
6.2.1	Creating functions	179
6.2.2	Improving code using functions	180
6.2.3	Creating a Punnett square function	182
6.2.4	Looking at several traits simultaneously	183
6.3	Implementing Mendel’s model	187
6.3.1	Random pollination	187
6.3.2	If tests: Taking different branches in code	188
6.3.3	Comparison operators	189
6.3.4	If-else-tests: Taking an alternative branch	191
6.3.5	A complete virtual experiment	192
6.3.6	What do our results mean for Mendel’s model?	194
6.3.7	A note on randomness in computers	195
6.4	More on functions	196
6.4.1	Default function values	196
6.4.2	Global and local variables	197
6.4.3	Common errors in functions	198
6.4.4	Docstrings	199

6.5	Summary	199
6.5.1	Biology glossary	200
6.5.2	Mendel's model	200
6.5.3	Functions	200
6.5.4	If-tests	202
6.5.5	Random choice in Python	203
7	DNA sequence analysis	204
7.1	The structure of DNA	206
7.2	Counting nucleotides	208
7.2.1	A simple program for counting nucleotides	209
7.2.2	Using a <code>for</code> loop to count nucleotides	210
7.2.3	Shorthands for common variable operations	213
7.2.4	If-elif-else-tests to count all four nucleotides	214
7.2.5	Dictionaries: Storing nucleotide counts	216
7.3	Calculating GC-content	220
7.4	Transcription of DNA to mRNA	222
7.5	Translating a mRNA strand to a protein	224
7.5.1	Converting codons to amino acids	224
7.5.2	Using the range function in for loops	226
7.5.3	Reading frames	228
7.5.4	Translation from start to stop	231
7.5.5	Translation without a stop codon	235
7.5.6	Logical operators for combining boolean expressions	235
7.5.7	A complete translation function	236
7.6	More on dictionaries	238
7.6.1	Different types of keys in dictionaries	238
7.6.2	Removing a key-value pair from a dictionary	238
7.6.3	Assigning the same dictionary to two different variables	239
7.6.4	Check if a key is in a dictionary	239
7.6.5	Accessing a key that does not exist	240
7.7	Summary	240
7.7.1	Chapter glossary	240
7.7.2	Strings	241
7.7.3	For loops	242
7.7.4	Nested loops	242
7.7.5	<code>range</code>	243
7.7.6	Dictionary	244
7.7.7	Short-hand syntax for common operations	244
7.7.8	If-elif-else tests	244
7.7.9	Logical operators for combining boolean expressions	245
8	Mutations and DNA	246
8.1	Point mutations	248
8.1.1	Importing functions from files	248
8.1.2	Insertions and deletions	249
8.1.3	Substitutions	250

8.2	Searching for the sickle cell mutation	251
8.2.1	Sickle-cell and normal hemoglobin	252
8.2.2	Reading FASTA files with the open-function	253
8.2.3	Comparing two mRNA strands	256
8.3	Implementing restriction fragment analysis	258
8.3.1	Cloning	258
8.3.2	Restriction cutting	258
8.3.3	Gel electrophoresis	259
8.3.4	Implementing restriction cutting of linear DNA	260
8.3.5	Finding DNA fragments	264
8.3.6	Restriction fragment analysis of normal and sickle-cell hemoglobin	266
8.3.7	Finding the cut locations on a circular DNA	268
8.3.8	Implementing restriction cutting of circular DNA	274
8.3.9	Redoing the restriction fragment analysis of normal and sickle-cell hemoglobin	275
8.4	Writing to files	277
8.5	Summary	278
8.5.1	Point mutations	278
8.5.2	Restriction fragment analysis	279
8.5.3	Importing functions from a file	279
8.5.4	Reading files	280
8.5.5	Writing to files	280
8.5.6	Traversing two or more lists simultaneously with zip	280
9	Modeling epidemics	282
9.1	The SIR model of infectious disease	283
9.1.1	Deriving the SIR model equations	285
9.2	Arrays: Like lists with support for mathematics	287
9.2.1	Similarities and differences between lists and arrays	288
9.2.2	Mathematical operations on arrays	290
9.2.3	Creating arrays	292
9.3	Implementing the SIR model	294
9.3.1	Calculating the death tolls from the SIR model	296
9.4	Disease on a grid: a spatial SIR model	298
9.4.1	Working on a spatial grid	298
9.4.2	Rules of the spatial SIR model	300
9.4.3	Using random numbers to represent probabilities	301
9.4.4	Tuples: unchangeable lists	302
9.4.5	Finding the neighbors	303
9.4.6	Special behavior on the boundary	304
9.4.7	A complete spatial SIR model	304
9.5	Exploring the spatial SIR model	308
9.6	The effect of vaccinations	312
9.7	Pandemics: Disease on a world map	314
9.7.1	A naive vaccination strategy	316
9.7.2	A limited number of vaccines	317
9.7.3	Random patient zero	319
9.7.4	Globalization	322

9.8 Summary	325
9.8.1 Arrays	325
9.8.2 linspace	326
9.8.3 Tuples	326
9.8.4 SIR model	327
9.8.5 Spatial SIR model	327
Bibliography	328
A Additional information	329
A.1 Packages	329
A.2 Terminal basics	331
A.3 Lists with mixed contents	332
Index	333

Chapter 1

Using Python as a Scientific Calculator

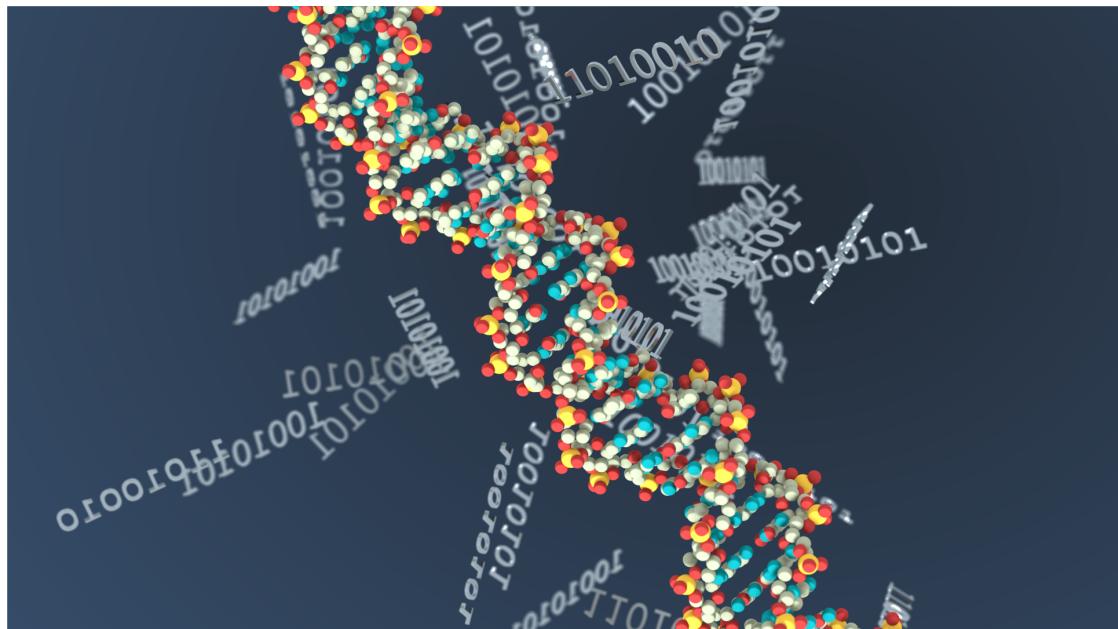


Figure 1.1: Computers store instructions to run programs in machine code, which are sequences of zeros and ones. Cells store instructions to create proteins in DNA, which are sequences of amino acids.

Our world is filled with great diversity of life. Everything from the smallest single-celled organisms to the largest animals make use of materials and energy to reproduce, and features are inherited from parents to children. What is truly amazing is how all living things encode their hereditary information in their DNA. Cells are small chemical factories which produce proteins

used for movement, sensing, signaling, structure, chemical reactions, and much more. The recipe for producing these proteins is stored in the cells' DNA.

The “building blocks” of DNA are molecules called adenine (A), thymine (T), cytocine (C), and guanine (G). Together, they make up the DNA’s four-letter alphabet - A, T, C, and G - and when put together, they define the information stored in the DNA. This is very similar to how we use our own alphabet to combine letters into words and sentences that contain information. Think of it as a cooking recipe: where humans read the letters in a cooking recipe to get instructions on how to make food, cells read their DNA to get instructions on how to make different proteins. Computers work in a similar way, but in their case, the instructions are stored as a long sequence of zeros and ones. The zeros and ones are called *machine code* and are read by a CPU (central processing unit), which is responsible for carrying out the instructions in the code.

When working with computers, you will often find that different programming languages and operating systems do not work together. If a program is made to only work on a Windows operating system, it will not work on a macOS system. We have not been able to make all computers understand the same machine code. One could imagine that nature, through 3.6 billion years of evolution, would suffer from the same issue and come up with multiple ways of encoding hereditary information, but all known species share the same DNA “language”.

While computers use zeros and ones to read their instructions, we do thankfully not have to think about machine code when we are working with computers. This is also true for programming. When you program, you will not be typing long sequences of zeros and ones to instruct the computer to do something. Rather, you will type something much more similar to our everyday language, like this:

```
while the_bacteria_are_alive():
    measure_the_number_of_bacteria()
```

Try to read the above out loud while ignoring the underscores, colon, and parentheses. It should read something like ”while the bacteria are alive, measure the number of bacteria”. This is a small piece of *Python code*. It could for instance be used to control a machine in a laboratory that automatically measures the number of bacteria in a petri dish as long as the bacteria are alive.

Python is a programming language. It is named after the comedy series *Monty Python’s Flying Circus*, but the creators of the language chose a pair of python snakes as their logo, as seen in Figure 1.2. Python is one of many programming languages you can use to write computer programs. The words we use in Python are mostly the same as in English language. However, the syntax is very different. In the English language, the syntax defines the structure of the sentences. In Python, it defines what the program does and in what order. When we *run*, or *execute*, Python code, the computer turns your code into machine code and follows the instructions you have given it. You will also learn that Python requires the user to be much more precise when giving the instructions. When you send an e-mail to a friend, you can get away with a few spelling mistakes. Python takes every instruction exactly as it is, and will not understand what you mean if anything is spelled the wrong way.

In the rest of this chapter, you will learn the basics of writing your very first program and how to run it. This program will be used to print a few messages on the screen. Before we can do so, we will show you where to download and how to install Python.



Figure 1.2: The Python logo.



Learning outcomes

After working with this chapter you know:

- how to use the computer as a scientific calculator, and
- how to implement simple models for heart rate and yearly temperature fluctuations.

The programming concepts we introduce in this chapter are:

- writing your own program,
- using variables,
- using functions, and
- importing packages.

1.1 Getting started with Python

We recommend using a program called Jupyter Notebook when working with the Python code in this book. There are also other ways to work with Python, so you are free to use other tools if you prefer. However, we assume that you are using Jupyter Notebook throughout this book. This book is in fact available as a Jupyter Notebook, so you may be looking at one already. If that is the case, you may skip the parts below on how to start Jupyter Notebook and jump straight to creating your first notebook in Section 1.1.2.

To install Python on your computer, we recommend that you download Anaconda which contains not only Python, but a large number of packages that are useful for scientific programming. The latest version of Anaconda is found at [anaconda.org](https://www.anaconda.org). Go to this webpage, find the download link and follow the instructions to install Anaconda.

1.1.1 Starting Jupyter Notebook

How to start the Jupyter Notebook depends on the operating system you are running.

- In Ubuntu, start Jupyter Notebook by opening the Dash menu, search for “Terminal”, open it and type `jupyter notebook`.
- In macOS, open the Terminal app. This is found under Applications > Utilites > Terminal. Once the terminal is open, type `jupyter notebook`.
- In Windows, start the Jupyter Notebook by opening the start menu and search for “Jupyter Notebook”. Open the application.

In all the above cases, a web browser should appear with Jupyter Notebook open, which should look something like in Figure 1.3.

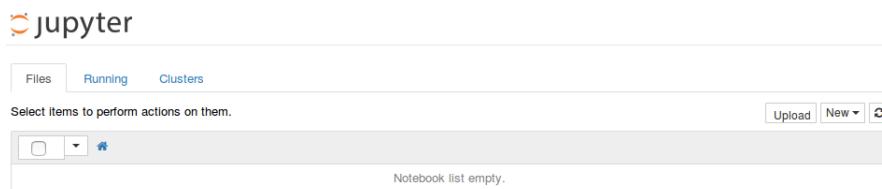


Figure 1.3: Jupyter Notebook once started.

1.1.2 Creating your first notebook

From the notebook front page, you can browse your file system and create new files. Click the “New” button and select “Python 3” from the menu as in Figure 1.4.

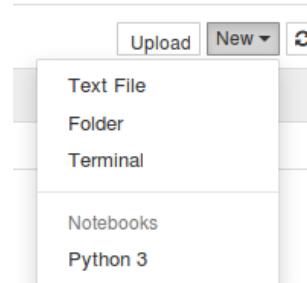


Figure 1.4: Click “New” and “Python 3” to create a new notebook.

This creates a new notebook in the folder you currently are in. The notebook will open in a new tab in your browser, and it should look similar to Figure 1.5.

The new notebook has one *cell*. Note that this has nothing to do with cells in biology, but is simply the name of the rectangle, much like cells in spreadsheets like Excel. The different uses

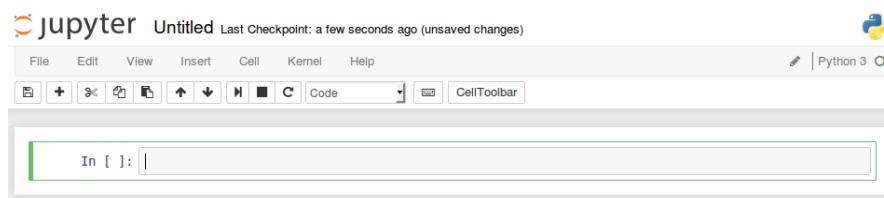


Figure 1.5: Jupyter Notebook showing an empty notebook with one cell.

of the word 'cell' is an example of the many terms used differently in programming and biology.

We will write our Python code inside these cells. You can try it out right now by writing the following text in the cell:

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

```
Hello, world!
```

This line of code will print the text "Hello, world!" to the screen. Run the code in the cell by pressing **Shift+Return**. That is, click on the cell with the mouse, then hold down the Shift button while pressing the Return button. (The Return button is sometimes called the Enter button.) The result of running our code appears below the cell, as in Figure 1.6.

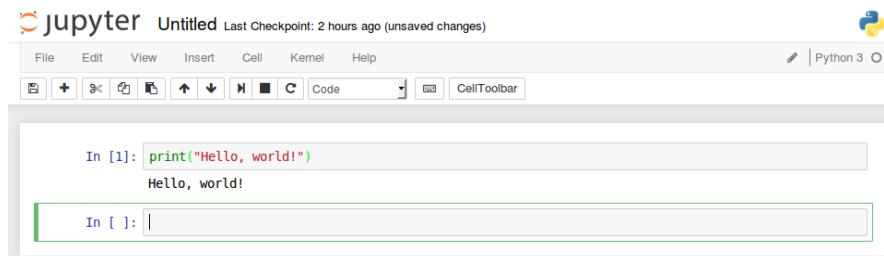


Figure 1.6: Type the code in the cell and hit **Shift+Return** on your keyboard. This will run the code, which in this case prints the text "Hello, world!" below the cell.

If you are working in a new notebook, in addition to running the existing cell, a new, empty cell will be created. This is in contrast to when you are working with this book chapter in notebook form, in which case the next cell will be selected after you have executed the cell with **Shift+Return**.

We can continue editing the first cell, or write some new code in the second cell. To run the code in any selected cell, press **Shift+Return** on your keyboard.

Save the changes to your notebook by clicking **File > Save and Checkpoint** or by hitting **Ctrl+S** on your keyboard in Linux or Windows or **Cmd+S** on macOS.

Reading Python code in this book. Throughout this book, we will show code in Jupyter Notebook cells with the result from running the code immediately below it:

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

```
| Hello, world!
```

In Jupyter notebooks, any code that is on the last (or only) line in a cell will also be executed and its resulting output (if any) printed, but printing in this way will sometimes give a result that appears different.

```
"Hello, world!"
```

```
| 'Hello, world!'
```

In this case, the output now has a ' (single quotation mark) on each side. The difference is technical, but for all our purposes only aesthetic.

1.2 Writing code

Computer code is instructions that the computer performs in order. The computer goes line by line and does exactly what you have told it to do. You can tell it to perform a vast range of instructions, such as showing a line of text or analyze the entire human genome. The computer does not interpret or understand the meaning of your commands. It just does whatever you tell it to do, even when it makes no sense. Think of it as a young child that has not yet been taught how to make a meal, but can help you out by performing simple tasks. You can not simply tell the child to cook a lasagna, but you can give instructions to put pasta and sauce in layers, and some cheese on the top.

The process of writing code consists of four steps:

1. Determine which instructions you need to give the computer.
2. Write down the instructions as code on the computer.
3. Run the code and check the validity of the results. This step is very important. Mistakes are often made and not always caught by the computer. If you told the computer to calculate something using the wrong numbers, it will gladly do so and give you the wrong result.
4. Find errors in the code and correct them. This process is called *debugging* and is often the most time consuming part of writing code.

The step-by-step procedure of programming is unfamiliar for most at first, and requires much training to master. To practice the skill of programming, you need to write large amounts of code and debug it until it works.

1.2.1 Writing your first Python code

In the previous section you wrote and ran your very first Python code:

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

```
| Hello, world!
```

This code consists of two parts: The *function* `print()` and the *string* "Hello, world!". The `print()` function takes everything you put inside its parentheses and prints it to your screen. In this case, we put the string of text "Hello, world!" inside the parentheses. Functions perform specific actions and will be explained in more detail later.

"Hello, world!" is a string. Strings in Python are enclosed with a " (double quotation mark) on each side, or a ' (single quotation mark) on each side. Both types are used, and you are free to choose, but make sure to never use one sign to start the string and another to end it, as Python will not understand this input. In this book we will always use ". Everything inside the quotation marks is the string itself, so the marks are not considered part of the string.

1.2.2 Variables: Storing data

Programming can be far more powerful and efficient than your typical calculator. One thing that makes it so powerful is the ability to store intermediate results in what is called a *variable*. So far we have just printed output below the cell. No line of code was dependent on the previous lines. By storing the text in a variable, we keep it around and print it at a later time. The following code stores the text in a variable named `greeting`:

```
greeting = "Hello, world!"
```

Variables in Python are *assigned* a value by using a single = (equal sign). Left of the equal sign is the name of the variable, which you choose (in this case, the variable name is `greeting`). Right of the equal sign is the information you want to store in a variable, which in this case is the text Hello, world!.

The process of associating a variable with a value is called *assignment*.

Note how the string has quotation marks around it, while the variable name is written without quotation marks. This is what makes a variable different from a string. Variables can be referenced later to use their content. A string is on the other hand just text and must be stored in a variable or used immediately.

Variable names can consist of letters (uppercase or lowercase), numbers, and the _ (underscore) symbol. They cannot start with a number. Note that you cannot use a space in a variable name, if you want a variable name that consists of two or more words, such as "number of bacteria", then the most common way to do this in Python is to name the variable `number_of_bacteria`. Python distinguishes uppercase and lowercase letters, so `Animal` and `animal` are two different variables.

We now print the text by *passing* the name of the variable, `greeting`, to the `print()` function, which means to insert the `greeting` inside the parentheses after `print()`:

```
greeting = "Hello, world!"
```

```
print(greeting)
```

```
Hello, world!
```

Quotation marks. The result of the above code looks identical to the output of our first code. Note that the name of the variable `greeting` is passed to the `print()` function without quotation marks around it. If we used quotation marks around `greeting` we get:

```
greeting = "Hello, world!"  
print("greeting")
```

```
greeting
```

The difference is that the quotation marks tells the computer that it should treat "`greeting`" as a string and print the string to the screen. Without the quotation marks, we tell the computer to look up the variable named `greeting` and print the *contents* of the variable.

Variables can be used to store the result of anything that is to the right of the equal sign, and give it a name. In the above case, the operation was simply that we stated the string, but the result of calculations such as adding two numbers can also be stored in a variable. We will return to this in Section 1.2.3.

Naming variables. In our example, we named the variable `greeting`, which is an arbitrary name. We could have named it `test`, `my_string`, `something_cool`, or anything we felt like. Using clear and informative variable names is useful when the amount of code becomes large, such as when you are writing entire programs. Choosing good variable names makes it much easier to understand what the code does. You should therefore always try to use descriptive variable names. In the code above, `greeting` tells us the variable contains a greeting. This would not have been as clear if we for instance used the variable name `g`. Sometimes an exception is made when we are implementing mathematical models. As an example, you may have heard of Einsteins equation relating the mass of an object to its energy,

$$E = mc^2, \quad (1.1)$$

where E is the energy, m is the mass and c is the speed of light. If we wanted to make a program which implements this equation, it would be acceptable to call the relevant variables `E`, `m` and `c`, rather than `energy`, `mass` and `speed_of_light`. The reason is that we would want the code to look like the mathematics, as this makes it easier to implement everything correctly.

Reserved names. There exists several reserved keywords in Python. These cannot be used as variables because they already have a different meaning:

```
and      del      from     not      while  
as       elif     global    or       with  
assert   else     if        pass    yield  
break   except   import   print  
class   exec    in       raise
```

```
continue  finally  is      return
def       for      lambda  try
```

We are going to use several of these in later chapters. In the meantime you should just make note of the reserved keywords and avoid using them as variable names.

Replacing the value of an existing variable. If we assign multiple values successively to the same variable name, the last value will be kept, and any previous values are forgotten. Take the code below as an example. We store the string `Hello, world!` to the variable `greeting` first, but then replace it with the string `Goodbye..` When we pass the `greeting` variable to the `print()` function, only the final string is printed on the screen:

```
greeting = "Hello, world!"
greeting = "Goodbye."
print(greeting)
```

| Goodbye.

However, if we write the following, the output is `Hello, world!:`

```
greeting = "Hello, world!"
print(greeting)
greeting = "Goodbye."
```

| Hello, world!

This happens because the code is executed line by line, and the last value of the variable is all that matters when we call the `print()` function.



Notice

Although lines in a cell are executed in order, cells in the notebook can be executed out-of-order and this can have unintended consequences.

1.2.3 Calculations with numbers

So far we have only used strings in our code. The real strength of scientific programming comes from working with numbers.

```
my_number = 3
```

Here, we do not use quotation marks, because quotation marks would specify that the variable is a text string. Storing the string `"3"` and the number `3` are two completely different things. With numbers, we can perform calculation, such as addition, which is done with `+` (addition sign).

```
print(1 + 2)
```

| 3

Similarly, we can do subtraction using `-` (minus sign),

```
print(4 - 3)
```

| 1

multiplication using `*` (star sign),

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

| 4

and division using `/` (slash sign):

```
print(10/2)
```

| 5.0

We calculate powers of numbers, a^b , by writing `a**b`. In Python, 3^2 thus becomes:

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

| 9

The result of a calculation can be stored to a variable:

```
my_number = 1 + 2
print(my_number)
```

| 3

It is important to note that Python first processes whatever is to the *right* of the equal sign and then assigns the result to the variable on the *left*. In this case, 1 is added to 2 first, which gives 3 as a result. Once calculated, the result is assigned to the variable `my_number`.

We can also do several operations on the same line. This snippet calculates $20 - 3 \times 4$ and stores the result, which is the integer 8, to the variable `a`:

```
a = 20 - 3*4
print(a)
```

| 8

In addition, calculations can be performed using existing variables, which is very useful for larger calculations.

```
a = 1  
b = 3  
c = b*b - a  
print(c)
```

8

This code stores the result of $b*b - a$, which, with the current variables, is the calculation $3*3 - 1$ and stores the result 8 in the variable c . Then we print c and see the result is 8.

You might have noticed that in the example of division, the printed result was 5.0, rather than 5, which would have looked more like the other examples. Numbers in Python come in different forms. The most common ones are *floating point* and *integer* numbers. Integer numbers (or *integers* or *ints*) are "whole" numbers, such as 1, 24, 967 and -511. Floating point numbers (or *floats*) are decimal point numbers, such as 1.5, 24.91, 967.2 and -511.412. Since the result of adding, subtracting or multiplying two integers is always another integer, these operations will return an int when they receive integers as input. For division, there is no such guarantee, as the result might be a decimal number. Because of this, a division always returns a float. If we want Python to use one specific type of number, we can convert them using the functions `int()` and `float()`. `print()` shows these differently, so a simple example that shows this difference is:

```
a = int(1)  
b = float(1)  
print(a)  
print(b)
```

1
1.0

The following is an example where we use Python as a scientific calculator.

Example 1.2.1. *Calculating your target heart rate.*

Measuring your heart rate while exercising gives you useful information about the intensity of your exercise. The rule of thumb for calculating your maximum heart rate, HR_{max} , is that your maximum heart rate is about 220 beats per minute, minus your age:

$$HR_{max} = 220 - \text{age}. \quad (1.2)$$

We calculate this in Python by

1. defining a variable `age` with the value of your age (we assume that you are 21 years old),
2. defining a new variable `maximum_heart_rate` and set it to $220 - \text{age}$, and
3. print the maximum heart rate.

The code then becomes:

```
age = 21
maximum_heart_rate = 220 - age
print(maximum_heart_rate)
```

199

This tells us that if you are 21 years old, your maximum heart rate is about 199 beats per minute. The target heart rate, HR_{target} , for a specific exercise intensity is found by multiplying the maximum heart rate with the desired intensity:

$$HR_{target} = HR_{max} \times \text{intensity} = (220 - \text{age}) \times \text{intensity}. \quad (1.3)$$

The intensity is a number between 0 and 1, where 0 means 0%, 0.5 means 50% and 1 means 100%, etc. Let us calculate your target heart rate at 70% exercise intensity:

```
age = 21
intensity = 0.7
maximum_heart_rate = 220 - age
target_heart_rate = maximum_heart_rate * intensity
print(target_heart_rate)
```

139.29999999999998

So if you are 21 years old and want to train at 70% intensity, your target heart rate should be about 139.3 beats per minute.

Notice how we are reusing the variable `maximum_heart_rate` to calculate `target_heart_rate`. This is an example of how variables is used for storing information that we need later on.

A surprising result is that the result was not printed as 139.3, but rather as 139.29999999999998. This is because of the way numbers are represented in a computer. In the decimal system, some fractions can be represented with a finite number of decimals, like $1/2 = 0.5$, or $7/10 = 0.7$, while others cannot, such as $1/3 = 0.33333333\dots$, where the 3s repeat forever. The computer uses a binary number system, where even fewer fractions are represented exactly. When we write `intensity = 0.7`, the computer attempts to approximate 0.7, but it cannot represent it with perfect precision. These small errors are called *round-off errors*, and can be important, but they will not cause problems in the examples found in this book.

Python has a function for rounding off numbers. This function is called `round()` and rounds a number to a certain decimal point. The `round()` function takes in two numbers, one to be rounded, and one that specifies the number of decimal places to include. For example, to round `target_heart_rate` in the example above to one digit after the decimal point, we use:

```
target_heart_rate_rounded = round(target_heart_rate, 1)
print(target_heart_rate_rounded)
```

139.3

As you see from the output, `target_heart_rate` is rounded to one digit after the decimal point.

1.2.4 Converting between strings and numbers

If a string is formatted exactly like a number, which means that it only consists of digits and possibly a decimal point, we can also convert the string to a number by writing `float(my_string)` or `int(my_string)`, where `my_string` is the string.

```
my_string = "1"
my_number = int(my_string)
print(type(my_string))
print(type(my_number))
```

```
<class 'str'>
<class 'int'>
```

We will deliberately ignore the `class` part of the output. We see that `my_number` is a variable of type integer, (`int`) and that the `my_string` variable has the content of the `my_number` variable converted to type string (`str`).

We can always convert a number to a string by writing `str(my_number)`, where `my_number` is the number.

```
my_number = 1
my_string = str(my_number)
print(my_number)
print(my_string)
```

```
1
1
```

Addition can be performed on strings as well, but the result is very different from addition of numbers, as the following example shows:

```
my_number = 1 + 2
my_string = "1" + "2"
print(my_number)
print(my_string)
```

```
3
12
```

The first line is the addition of one plus two, while the second line is a concatenation (joining) of the two strings "1" and "2".

We can also do multiplication of string and an integer. In the same way that $3a = a + a + a$ in mathematics, a multiplication of an int and a string adds the string to itself 3 times:

```
3*"Hello"
```

```
'HelloHelloHello'
```

1.2.5 How functions are used

Above, we converted numbers to strings and strings to numbers. This is a case of using functions. Let us take some time to understand how functions are used, before we move on.

A function has a name, which comes in front of the parenthesis. If you see `float(a)` in your code, `float` is the name of the function. Inside the parentheses are *arguments* that the function receives. In this case, there is one argument, and it holds the value of `a`. The number of arguments a function expects depends on how the function is defined, and can be zero, one, or multiple. In a Jupyter Notebook you can get information about any function by writing the name of the function followed by a question mark:

```
float?
```

```
Init signature: float(self, /, *args, **kwargs)
Docstring:
float(x) -> floating point number

Convert a string or number to a floating point number, if possible.
Type:           type
```

```
Init signature: float(self, /, *args, **kwargs)
Docstring:
float(x) -> floating point number

Convert a string or number to a floating point number, if possible.
Type:           type
```

Note that in the Jupyter Notebook, this text will appear in a window at the bottom. You close that window using the small 'x' at the top right.

This helptext is a so-called *docstring*, which holds information about the function. Ignore the first line about "Init signature" for now. The rest of the output tells us that the functions name is `float`, it takes an argument, `x`, and returns a floating point number. Then it explains that the function is used to "Convert a string or number to a floating point number, if possible". Note that this docstring refers to the argument as `x`, but it is not necessary to name the argument `x` to use the function. The argument can be any variable with any name you like:

```
my_number = "3.4"
print(float(my_number))
```

```
3.4
```

or even just a value:

```
print(float("2.3"))
```

```
2.3
```

When a line of code with a function is run, the function performs an action that has been defined before. You can create functions yourself, which we return to in Chapter 6, or you can use functions defined by others. A function may perform an action and give you something back, which is to say that the function *returns* a value. We can store whatever is returned to a variable:

```
a = float("2.3")
print(a)
```

| 2.3

As you might have noticed, our first encounter with functions was `print()`. The `print()` function does not return a result, but it performs an action, which is to print all arguments it receives to the screen.

1.2.6 Order of operations

When performing multiple operations in a single line of code, it is important to know the order in which the operations are performed. These rules are familiar to many rules from mathematics, but we repeat them here:

1. First, we calculate everything inside parentheses,
2. then we do all function calls,
3. then we do all powers,
4. then we do all multiplications and divisions, and
5. finally, we do all additions and subtractions.

This means that in the below code, the division is evaluated first, then the addition:

```
a = 1 + 2/2
print(a)
```

| 2.0

In the following code, the parenthesis is evaluated first, and because the parenthesis contains the addition, the addition is evaluated before the division:

```
b = (1 + 2)/2
print(b)
```

| 1.5

As seen in the above cases, `a` will be 2.0 and `b` will be 1.5.

1.2.7 Commenting the code

We can add comments to our code to make it easier to understand. In Python, everything that follows a hashtag symbol is a comment:

```
# this is a comment
```

The content of a comment is ignored by Python when running the code, which means that you can write normal text, rather than something the computer should understand. Note that the space after the hashtag symbol is optional, but recommended for readability. Here is an example of the previous code with comments added:

```
# the parenthesis is important here
# this adds one plus two, then divides the result by two
b = (1 + 2)/2
# print the result to the screen
print(b)
```

| 1.5

This code with comments, and the previous code without, both do the same thing, but the comments can be helpful to understand the code better. Good comments increase the readability of code and is very helpful when trying to understand what the code does. Short code, like above, does not really need comments, but comments are very useful in larger programs. A comment should ideally not repeat the code with words, but instead contain information that is not obvious from the code. As such the comment `# print the result to the screen` is not necessary. It requires practice to write good, informative comments and it is something you should get into the habit of always doing. Let us show you an example on how to use comments well.

Example 1.2.2. *A Fahrenheit to Celsius converter.* Here we make a simple code that converts temperatures in degrees Fahrenheit to degrees Celsius. If F is the temperature in Fahrenheit, and C is the temperature in Celsius, this conversion is made by the formula,

$$C = (F - 32)/1.8 . \quad (1.4)$$

We use this formula to convert a temperature from Fahrenheit to Celsius, and then print out the result with some additional text.

```
F = 10          # degrees Fahrenheit
C = (F - 32)/1.8 # degrees Celsius

# Convert F and C to strings and concatenate with additional text
output_string = str(F) + " degrees Fahrenheit is equal to " + str(C) + " degrees Celsius."
print(output_string)
```

| 10 degrees Fahrenheit is equal to -12.22222222222221 degrees Celsius.

We can now change the variable F depending on which temperature we want to convert from. The comments explain and summarize what the code does. Note how the first two comments are added at the end of the line instead of on their own line. Both ways are possible in Python.

Those two first comments tell you what F and C are and which units they have. The last comment summarizes what the piece of code below it does.

1.2.8 Debugging the code

What happens if we add a number to a string? This happens if we in the above example forgot to use `str()` around F and C.

```
F = 10          # degrees Fahrenheit
C = (F - 32)/1.8  # degrees Celsius

# Convert F and C to strings and join together with additional text
output_string = F + " degrees Fahrenheit is equal to " + C + " degrees Celsius."
print(output_string)
```

```
-----
TypeError                                 Traceback (most recent call last)
Input In [38], in <cell line: 5>()
      2 C = (F - 32)/1.8  # degrees Celsius
      3 # Convert F and C to strings and join together with additional text
----> 5 output_string = F + " degrees Fahrenheit is equal to " + C + " degrees Celsius."
      6 print(output_string)

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

The error message tells us that Python does not know how to add (use the operand `+`) an `int` and a `str`. Python does not know how to add an integer (a number) with a string (a piece of text). We also see the line where the problem is encountered, namely line 5, which has an arrow pointing it out.

Python will produce error messages whenever your code has statements which do not make sense to the computer. An experienced programmer will look at this message and quickly realize the mistake, but most error messages look strange the first few times you encounter them. You should always look for the last error message, the file name (none is shown in this case) and the line number. It is completely normal that sorting out bugs in your code, commonly called debugging, takes just as much of your time, if not more, than writing the code.

1.2.9 Using packages to get pre-made functions

We have so far used some simple mathematical operators, that always are available in Python. Now we want to use some more advanced mathematical functions. These are not part of 'standard' Python. To use these, we need to import them from a *package*. A package is Python code written by someone else (usually a group of programmers) that can be imported into Python so you get access to it. Other words used for packages are *modules* or *libraries*. We will be using several packages throughout this book, and many more are available from the internet. A package that contains many functions we need for scientific calculations is named `pylab`, and we use functions from this package throughout the book. Pylab provides functions for calculating square roots or logarithms that can be included using the `import` command. Typing

```
from pylab import sqrt
```

gives you access to the `sqrt()` function from the `pylab` package. Typing `sqrt(x)` calculates the square root of `x`, which is the equivalent of \sqrt{x} in mathematical notation.

After importing the `sqrt()` function, we may use it to calculate the square root of a number, and print the result to the screen:

```
from pylab import sqrt
print(sqrt(4))
```

| 2.0

As seen above, the result is 2.0, which is the square root of 4.

Some of the most common mathematical functions in the `pylab` package are listed in following table.

Function name	description
<code>sin(x)</code>	the sine of x , $\sin(x)$
<code>cos(x)</code>	the cosine of x , $\cos(x)$
<code>tan(x)</code>	the tangent of x , $\tan(x)$
<code>factorial(x)</code>	the factorial of x , $x!$
<code>exp(x)</code>	e raised to the power of x , e^x
<code>sqrt(x)</code>	the square root of x , \sqrt{x}
<code>log(x)</code>	the natural (base e) logarithm of x , $\ln(x)$
<code>log10(x)</code>	the base 10 logarithm of x , $\log_{10}(x)$
<code>pi</code>	π to numerical precision, 3.14159...
<code>e</code>	e to numerical precision, 2.71828...

If we want access to everything in the `pylab` packaged we use `*`, instead of using the name of the function we want:

```
from pylab import *
print(pi)
```

| 3.141592653589793

In this book, we use this method of importing everything as our preferred method because this simplifies the introduction to Python in general. However there are some disadvantages to this, which we explain in Appendix A.1.

We round off this section with an example of using trigonometric functions to estimate the average temperature of an area.

Example 1.2.3. *Using a simple temperature model.* The average temperature, measured in degrees Celsius, during the course of a year in Oslo, the capital of Norway, is modeled fairly well by the expression

$$T(x) = 6 + 10 \sin\left(\frac{2\pi}{365}x - 1.9\right), \quad (1.5)$$

where T is the average temperature, and x is time, measured in days, starting at December 31st (such that $x = 1$ is January 1st and so on). Let us make a small program that will allow us to check the average temperature at the 100th day of the year (April 10th).

```
from pylab import *
x = 100 # day of the year
T = 6 + 10*sin(2*pi*x/365 - 1.9) # average temperature (in Celsius)
print("Average temperature on day", x, ":", T, "degrees Celsius")
```

```
Average temperature on day 100 : 4.223682866976491 degrees Celsius
```

We can easily modify the program to find the average temperature for any other day of the year, but we leave this as an exercise to the reader.

Note that we use comma in the `print()` function to separate the content which we want printed to the screen. This is different from what we have done previously, where floats and integers were converted to string objects using `str()`. Here, instead of converting the variables `x` and `T` to strings, we just use comma to separate them.

1.3 Computer science glossary

There are several important words that programmers use when they talk about programming. We have listed a few of them here so you can look them up if you encounter words you do not recognize:

- **Code or source code:** are instructions given to the computer in the form of text.
- **Scripts:** Pieces of code, typically contained in a file or notebook, that can be interpreted by the computer.
- **Programs:** Computer code that is ready to run on its own. Python and Jupyter Notebook are programs that run the scripts you write in this book. Excel and Word are other examples of programs.
- **App or Application:** Another word for program.
- **Cell:** Part of a Jupyter Notebook. Cells can be runnable, similar to a script, and the output is shown below the cell.
- **Running or executing:** Running or executing a program or script is to make the computer perform the instructions in the program or script.
- **Algorithm:** The recipe the computer follows to solve a specific problem. Much in the same way as you follow instructions for doing a biology laboratory exercise.
- **Implementation:** The process of writing and testing code.
- **Verification or testing:** Done to make sure that a program works as intended.

- **Errors or bugs:** Errors or bugs occur if the code does not work as intended. Debugging is seen by many as the most difficult and time-consuming task of programming.
- **Input or input data:** The information given to the program.
- **Output:** The result from a program. It can either be a few lines of text written to the screen or several gigabytes of data written to a file.
- **Packages, modules or libraries:** These contain premade code that others have written. It can be used for plotting, moving files, creating graphics and much more.

1.4 Summary

This chapter had two main parts. The first was about how to install and setup all necessary software to be able to work through this book. In the second part, we learned the very basics of programming and how to use Python as a scientific calculator.

Variables. Data is stored in variables. This is called assignment. Variables make it possible for us to give the data a name, which can be used to access the data later in our program. Good variable names make the code easier to read.

Types of variables. There are several types of variables in Python. Each type has special properties. We have been working on the following types:

- *Floats.* Floats are rational numbers, meaning numbers with decimal points, such as 3.2 and 1.5.
- *Integers.* Integers are the natural numbers, often called whole numbers, such as 1, 2 and 9.
- *Strings.* Strings are lines of text. They must be enclosed by quotes, as in "The text goes here".

Functions. A function is a piece of previously defined code, either by yourself or others that performs specific tasks. A function can take zero, one or multiple arguments as input (inside the parenthesis) and may return a value. When in a Jupyter Notebook you can get information about any function by writing the name of the function followed by a question mark, ?. The functions introduced in this chapter are listed below:

Syntax	Description
<code>print()</code>	Prints the arguments to screen
<code>int()</code>	Converts the argument to integer
<code>float()</code>	Converts the argument to float
<code>round()</code>	Rounds a number to a certain decimal point

Order of operations. When performing multiple operations in a single line of code the order in which the operations are performed is the following:

1. First, we calculate everything inside parentheses,
2. then we do all function calls,
3. then we do all powers,
4. then we do all multiplications and divisions,
5. finally, we do all additions and subtractions.

Comments. The main point of comments is to make the program easier to read. They begin with a hashtag and everything after this character is ignored when the program is run.

Packages. A package is a collection of functions for performing specific tasks that we can use. A package that contains many functions we need for scientific calculations, is named pylab. This package is used throughout the book. There are multiple ways to import packages in Python. In this book, we use

```
from pylab import *
```

as our preferred method because this simplifies the introduction to Python in general. You can also use the name of the specific function you want instead of *:

```
from pylab import sqrt
```

Chapter 2

Protein Secretion Analysis

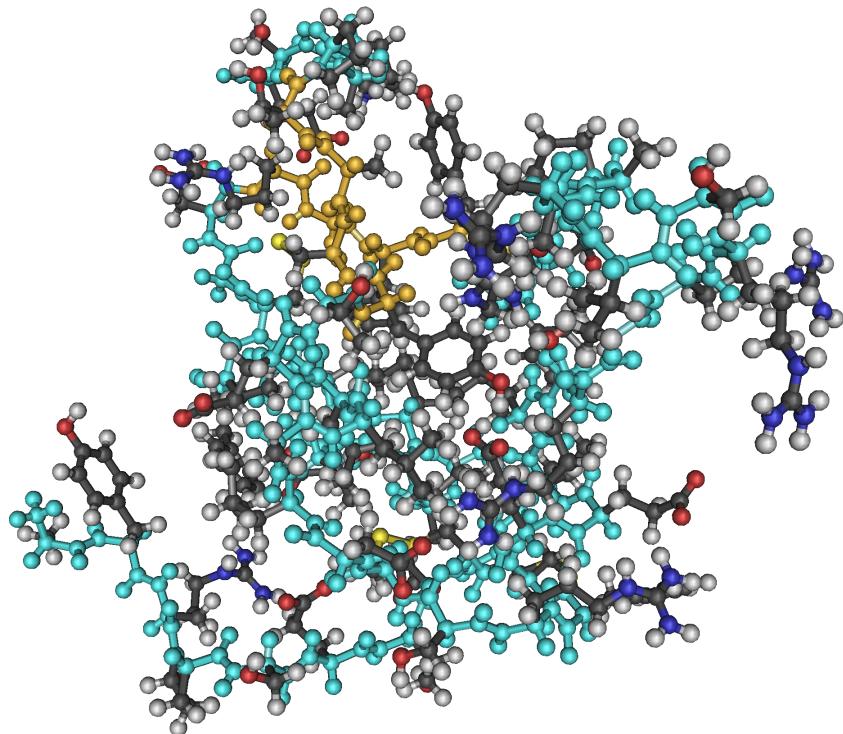


Figure 2.1: 3D atomic rendering of Insulin-Like Growth Factor. [6]

In this chapter, we will be looking at linear growth of a biological process: insulin production by the pancreas. We will compare the amount of insulin in the blood between non-obese and obese (overweight) people. We are going to use Python to examine experimental measurements of insulin in the blood. The size of datasets in biology are ever increasing and have become so large it is easy to get lost in them. If we tried to go through the data manually, we would

be unable to draw conclusions from it, because there is too much information for a human to process. When we create a Python program to perform an analysis, there is also very little effort required to redo the analysis later, or perform the same analysis on another dataset.



Learning outcomes

After working with this chapter you will know:

- the role of insulin in regulating blood sugar levels,
- how insulin is produced,
- the effect of eating a meal on insulin levels, and
- the difference between insulin levels and production between non-obese and obese people.

The programming concepts we introduce in this chapter are:

- lists,
- visualizing data with plots,
- saving plots, and
- reading data from file.

2.1 Insulin and blood glucose levels

Insulin is a hormone which plays a key role in the regulation of blood glucose levels. Insulin helps control blood glucose levels by signaling to the liver cells, muscle cells, and fat cells to take up glucose from the blood. Insulin therefore helps cells to take in glucose to be stored or used for energy.

When there is sufficient energy available for cellular processes, insulin signals the liver to take up glucose and store it as glycogen.

Diabetes is a chronic condition associated with abnormally high and low levels of sugar (glucose) in the blood. The absence or insufficient production of insulin, or an inability of the body to properly use insulin are the causes for diabetes. Diabetes has become a global health problem, affecting >170 million individuals worldwide.

There are a number of different types of diabetes. The most common ones are called type 1 and type 2. People with type 1 diabetes cannot make insulin. Therefore, these people will need insulin injections to allow their body to process glucose and avoid complications from high blood sugar (hyperglycemia). People with type 2 diabetes do not respond well or are resistant to insulin.

They will need to adjust their diet, and some may need insulin shots to help them better control blood glucose levels.

In addition to its role in controlling blood sugar levels, insulin is also involved in the storage of fat. Insulin's function in fat storage has been an area of much interest. In particular the association between *obesity* (overweight) and insulin resistance is an area of enormous public health impact, with hundreds of scientific publications in the last years focused on the possible mechanisms that underlie this association. It has been suggested that obesity can lead to increased insulin levels in the blood, subsequently leading to insulin resistance and type 2 diabetes.

2.2 Measuring insulin production

We want to model the release of insulin from the pancreas in response to rising blood glucose concentration as a result of (for example) eating food. We will use data from experiments on humans. Insulin secretion was evaluated by measuring insulin concentration in the blood in 14 non-obese and 15 obese subjects during a 24 hour period. Here we will focus on the early 2 hours of the day.

The data will represent the mean of for each group (non-obese and obese). The insulin measurements were done one hour before and after breakfast, with samples drawn every 15 minutes.

We will first look at the results for the non-obese group only. These measurements are shown in the table below:

Time, minutes	Insulin ($\mu\text{U}/\text{ml}$)
0	7.1
15	7.2
30	7.1
45	10.0
60	26.7
75	43.3
90	56.4
105	56.8
120	49.7

Insulin concentration is reported as microunits per milliliter ($\mu\text{U}/\text{ml}$). A 'unit' here is the standard 'enzyme unit', a measure related to catalytic activity.

2.3 Plotting insulin production

The dataset from this experiment tells us how the insulin concentration in the subject's blood changes during a normal morning. We can see that insulin level are constant for a while before the subjects have breakfast and insulin levels rise. But drawing conclusion by just looking at the numbers is not easy. If the table had tens of thousands of data points, the task of making sense of the data manually would be nearly impossible! We humans are visual creatures and

get a much better understanding of data when we visualize it. Plotting the data is therefore always a good first step when we examine data for the first time. It will guide further analysis by indicating possible interesting features that we would like to examine.

Before we can visualize our data using Python, we have to enter the data into our program. We use a feature of Python, and many other programming languages, called *lists* for storing our data.

2.3.1 Lists: Storing many data elements in one variable

Up until now, all variables we have used contained a single number or string. Lists are one way of grouping a set of data into one variable. A list is a collection of different objects, such as floats, integers, and/or strings. A list is enclosed by a [on the left side, and a] (square bracket signs) on the right side, and each element in the list is separated by a comma.

We store our time data in one list that we name `t` (for time):

```
t = [0, 15, 30, 45, 60, 75, 90, 105, 120]
print(t)
```

```
[0, 15, 30, 45, 60, 75, 90, 105, 120]
```

And we do the same with the insulin concentration data in a list called `N` (for non-obese):

```
N = [7.1, 7.2, 7.1, 10, 26.7, 43.3, 56.4, 56.8, 49.7]
print(N)
```

```
[7.1, 7.2, 7.1, 10, 26.7, 43.3, 56.4, 56.8, 49.7]
```

Lists can also contain elements of different types at the same time, such as integers, floats, and strings:

```
my_list = [31, 4.2, 9, "hi!", 7]
print(my_list)
```

```
[31, 4.2, 9, 'hi!', 7]
```

Lists are a good way to store large amounts of data. Other options exist, but lists are the easiest to use.

Retrieving list elements. Every element in a list is associated with an index, which represents the position of that element in the list. The first element has index 0, the second element index 1, the third index 2, and so on, as seen in Figure 2.2. It is important to note that we start counting from 0 instead of 1. This is called *zero-based indexing*, and is illustrated in Figure 2.2.

Zero-based indexing is typical for many programming languages, but there are some languages, such as *Matlab* and *R*, which use *one-based indexing*, where the first element has index 1. The

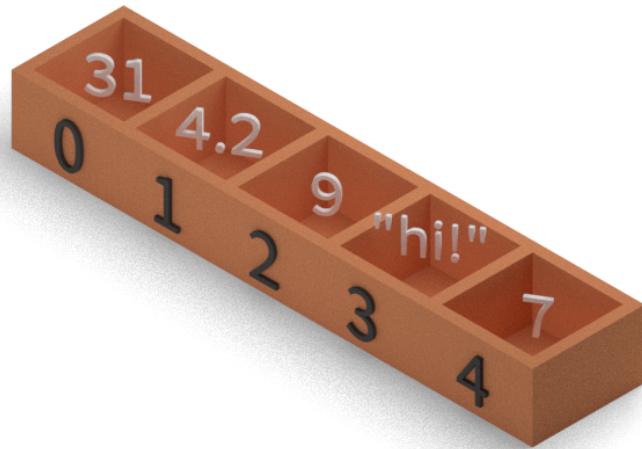


Figure 2.2: You can think of lists as a series of indexed (or numbered) boxes, where the first box has index 0. Each box can contain one element of different types.

reason for using zero-based indexing are both historical and technical, but for us, the most important thing is to accept the convention, and remember it when we write our own code.

Looking at the list containing the insulin secretion data `N`, we have 9 elements, and 9 corresponding indices, starting from 0 and ending at 8. The comment shows the index of each element in our `N` list.

```
N = [7.1, 7.2, 7.1, 10, 26.7, 43.3, 56.4, 56.8, 49.7]
# [ 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 ]      Indices
```

To access a certain element in our list `N`, we write square brackets after the list name, and put the index in between them. This means that to access the second element we write `N[1]`:

```
print(N[1])
```

7.2

Let us also print out the first element, index 0, and sixth element, index 5:

```
print(N[0])
print(N[5])
```

```
7.1  
43.3
```

What happens if we try to use an index that is not in our list, for example, index 10?

```
print(N[10])
```

```
-----  
IndexError                                                 Traceback (most recent call last)  
Input In [53], in <cell line: 1>()  
      1 print(N[10])  
  
IndexError: list index out of range
```

This results in an `IndexError`, with the message that the index we used is out of range.

Retrieve elements with negative indices. We can also use negative indices to access elements in a list. In this case the indices count from the end of the list instead of the beginning, so that `N[-1]` is the last element in the list, `N[-2]` is the second to last element and so on:

```
N = [7.1, 7.2, 7.1, 10, 26.7, 43.3, 56.4, 56.8, 49.7]  
#   [-9, -8 , -7 , -6,  -5 , -4 , -3 , -2 , -1 ]       Indices  
  
print(N[-1])  
print(N[-2])
```

```
49.7  
56.8
```

Make a copy of a list. We can assign two variables to the same list as follows:

```
N = [7.1, 7.2, 7.1, 10, 26.7, 43.3, 56.4, 56.8, 49.7]  
N_copy = N
```

Let us see what happens when we change an element in the copy:

```
N_copy[0] = 6.5  
print(N)  
print(N_copy)
```

```
[6.5, 7.2, 7.1, 10, 26.7, 43.3, 56.4, 56.8, 49.7]  
[6.5, 7.2, 7.1, 10, 26.7, 43.3, 56.4, 56.8, 49.7]
```

You might have expected that changing `N_copy` did not change `N`, but it does! The same goes if we only change `N`:

```
N = [7.1, 7.2, 7.1, 10, 26.7, 43.3, 56.4, 56.8, 49.7]  
N_copy = N  
N[0] = 6.5
```

```
print(N)
print(N_copy)
```

```
[6.5, 7.2, 7.1, 10, 26.7, 43.3, 56.4, 56.8, 49.7]
[6.5, 7.2, 7.1, 10, 26.7, 43.3, 56.4, 56.8, 49.7]
```

This behavior can be very useful, but it can also be the source of your most frustrating debugging session, should you ever forget it.



Notice

When you assign a new variable name to a string or a number, the two variables are independent of each other. This is not the case for lists! Both variables refer to the same list.

If you do want to make a new list, which has all the same elements as the original list, but is independent from the original, you use the following syntax:

```
N = [7.1, 7.2, 7.1, 10, 26.7, 43.3, 56.4, 56.8, 49.7]
N_copy = N.copy()    # copy the content in N
N_copy[0] = 6.5
print(N)
print(N_copy)
```

```
[7.1, 7.2, 7.1, 10, 26.7, 43.3, 56.4, 56.8, 49.7]
[6.5, 7.2, 7.1, 10, 26.7, 43.3, 56.4, 56.8, 49.7]
```

As the example shows, `N` is unaffected in this case. The syntax `N.copy()` is an example of the *dot notation*, where we call a function (technically called a *method*) that cannot be used on its own, but only works with a Python element (technically called an *object*), such as lists and strings. In this case the Python element is a list, and the function we call is `.copy()`.

Finally, note that if we use `=` to override the copy with something new, the original list is not affected:

```
N = [7.1, 7.2, 7.1, 10, 26.7, 43.3, 56.4, 56.8, 49.7]
N_copy = N.copy()
N_copy = "Hello, world!"
print(N)
print(N_copy)
```

```
[7.1, 7.2, 7.1, 10, 26.7, 43.3, 56.4, 56.8, 49.7]
Hello, world!
```

Likewise, setting a new value to `N` with `=` would not change `N_copy`.

Counting list elements. When working with lists, we often want to know the number of elements in the list without counting them manually. We do this by using the function `len()`, as shown below:

```
N = [7.1, 7.2, 7.1, 10, 26.7, 43.3, 56.4, 56.8, 49.7]
# [ 0 , 1 , 2 , 3, 4 , 5 , 6 , 7 , 8 ]      Indices
length_of_list = len(N)
print(length_of_list)
```

| 9

2.3.2 Plots: Plotting data over time

Storing the data in a list is a good start, but to really get a feel for what the data contains, we need more. Visualizing data is a very important tool in science. Here, we show how to use Python to make drawings, or *plots*, of data. The package we use for plotting is `matplotlib`, which is included in the `pylab` package. We import `pylab` as follows:

```
from pylab import *
```

We now have access to everything in the Matplotlib package and can plot the insulin secretion measurement `N`, against time, `t`. For this we use the command `plot(t, N)`. This command plots time, `t`, on the *x*-axis against the insulin concentration, `N`, on the *y*-axis, and we get a curve that shows how the secretion measurements change with time.

We always need call the function `show()` to make sure the plot is shown:

```
from pylab import *
plot(t, N)
show()
```

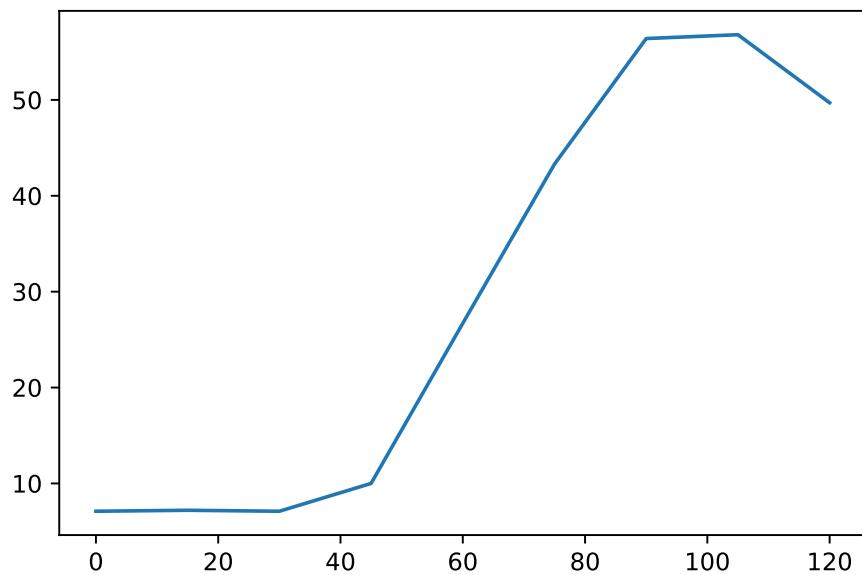


Figure 2.3: Our first plot for insulin secretion.

Even though we have plotted our data, we are not finished. The rule of thumb is that we should be able to understand the plot without any additional information. Thus, to make plots easier to understand they should have

- a title,
- labels on the x - and y -axes, and
- if the axis has labels, units added in parenthesis to the labels

All of the above is easily added after the `plot()` command, but before `show()`. The `show()` function must always be the last thing we call when we make changes to a plot, otherwise we do not see the changes.

```
from pylab import *
plot(t, N)
title("Insulin secretion over time")
xlabel("Time, t (minutes)")
ylabel("Insulin concentration (microU/ml)")
show()
```

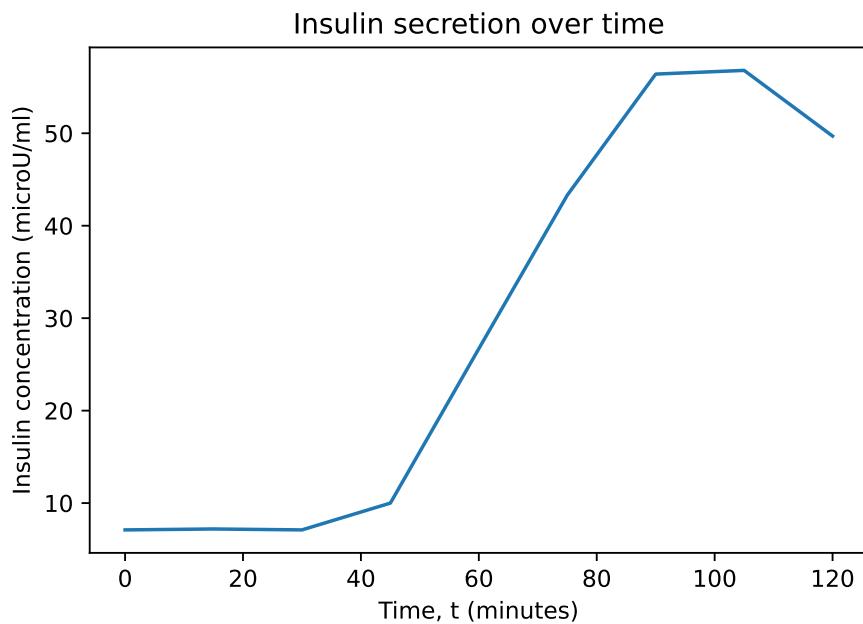


Figure 2.4: A plot for insulin secretion with axis labels and title.

With the axis labels in place, this new figure is easier to understand than Figure 2.3. From this plot we see that insulin production is constant for the first 30 minutes and that it takes about 45 minutes between eating breakfast and the first increase in insulin production. After about 90 minutes the secretion level plateaus, meaning it no longer increases.

2.3.3 Using list slicing to examine the linear growth phase

Before we discuss the part of Figure 2.4 showing the increase in insulin production, we wish to extract elements of our lists that correspond to this phase only and plot them separately. Python has a nice method for extracting parts of a list that helps us. Such parts of a list are called *sublists* or *slices*.

Sublists. This operation is done by using the colon symbol when indexing a list. $N[i:j]$ is the sublist that starts with index i and continues up to, but not including j , as illustrated in Figure 2.5. The comments show you the indices as well as the indices associated with the sublist.

```
my_list = [31,    4.2,     9, "hi!",      7]
#       [ 0,      1,      2,      3,      4]      Indices
#           [ 1,      2,      3]                  [1:4]
print(my_list[1:4])
```

```
[4.2, 9, 'hi!']
```

$N[i:]$ is the sublist that starts with index i and continues to the end of N :

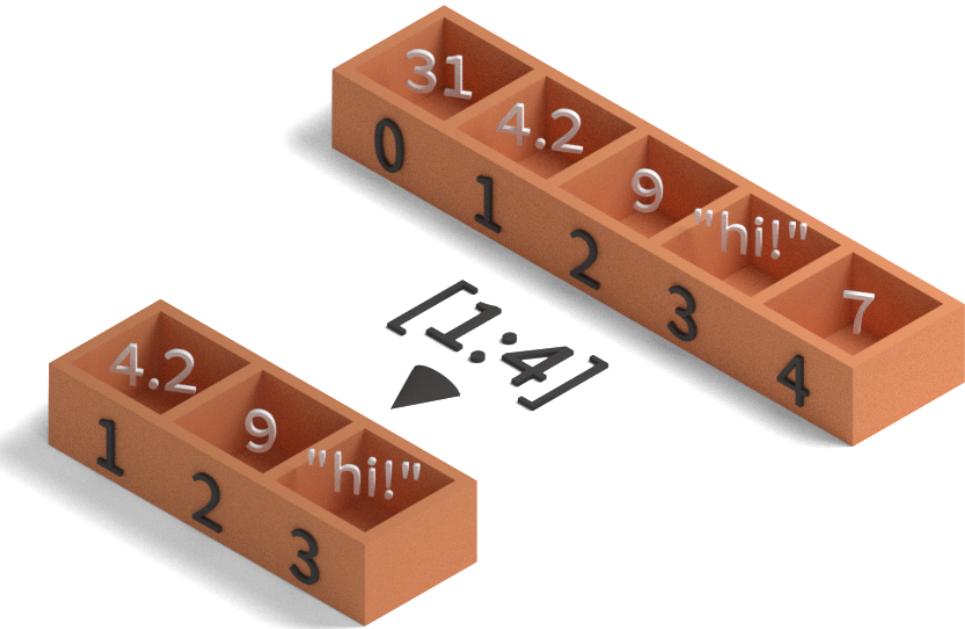


Figure 2.5: Lists can be sliced into smaller sublists. This produces a new list with copies of the elements included in the slice.

```
N = [7.1, 7.2, 7.1, 10, 26.7, 43.3, 56.4, 56.8, 49.7]
# [ 0 , 1 , 2 , 3, 4 , 5 , 6 , 7 , 8 ]      Indices
# [ 3, 4 , 5 , 6 , 7 , 8 ]                  [3:]

print(N[3:])
```

```
[10, 26.7, 43.3, 56.4, 56.8, 49.7]
```

Similarly, $N[:j]$ is the sublist that starts with index 0 and goes up to, but not including j :

```
N = [7.1, 7.2, 7.1, 10, 26.7, 43.3, 56.4, 56.8, 49.7]
# [ 0 , 1 , 2 , 3, 4 , 5 , 6 , 7 , 8 ]      Indices
# [ 0 , 1 , 2 , 3, 4 ]                      [:5]

print(N[:5])
```

```
[7.1, 7.2, 7.1, 10, 26.7]
```

We can also use negative indices when creating sublists. $N[1:-1]$ is a sublist containing all elements except the first and the last

```
N = [7.1, 7.2, 7.1, 10, 26.7, 43.3, 56.4, 56.8, 49.7]
# [ 0 , 1 , 2 , 3, 4 , 5 , 6 , 7 , 8 ]      Indices
```

```
# [ -9, -8 , -7 , -6, -5 , -4 , -3 , -2 , -1 ]      Neg. ind.
#      [ 1,   2,   3,   4,   5,   6,   7 ]           [1:-1]

print(N[1:-1])
```

[7.2, 7.1, 10, 26.7, 43.3, 56.4, 56.8]

Slicing a list returns a copy. An important point to note is that a sublist is always a copy of the original list, if we modify the sublist we do not modify the original list and vice versa:

```
N = [7.1, 7.2, 7.1, 10, 26.7, 43.3, 56.4, 56.8, 49.7]
# [ 0 , 1 , 2 , 3, 4 , 5 , 6 , 7 , 8 ]      Indices
#      [ 1,   2,   3]           [1:4]

N2 = N[1:4]
print(N2)
```

[7.2, 7.1, 10]

```
N2[0] = 6.5

print(N2)
print(N)
```

[6.5, 7.1, 10]
[7.1, 7.2, 7.1, 10, 26.7, 43.3, 56.4, 56.8, 49.7]

As you see from the output `N` is not affected by the change in `N2`. We can slice the entire list by writing `N[:]`. This gives the same result as `N.copy()`:

```
N = [7.1, 7.2, 7.1, 10, 26.7, 43.3, 56.4, 56.8, 49.7]
# [ 0 , 1 , 2 , 3, 4 , 5 , 6 , 7 , 8 ]      Indices
N2 = N[:]
N2[0] = 6.5

print(N2)
print(N)
```

[6.5, 7.2, 7.1, 10, 26.7, 43.3, 56.4, 56.8, 49.7]
[7.1, 7.2, 7.1, 10, 26.7, 43.3, 56.4, 56.8, 49.7]

2.3.4 Plotting the breakfast response

We now have the tools that we need to continue plotting the insulin secretion. We are interested in plotting only the part showing an *increase* in secretion, which happens during the time interval from 45 to 90 minutes, and corresponds with the list elements from index 3 until (but not including) index 7. The first three elements of our measurements are those before breakfast. We thus need to create a slice of our list from index 3 until index 7: `t[3:7]` and `N[3:7]`.

To only visualize the part of our data showing an increase in production, we replace the plot statement from last section by:

```
plot(t[3:7], N[3:7])
title("Insulin secretion over time, linear phase")
xlabel("Time, t (minutes)")
ylabel("Insulin concentration (microU/ml)")
show()
```

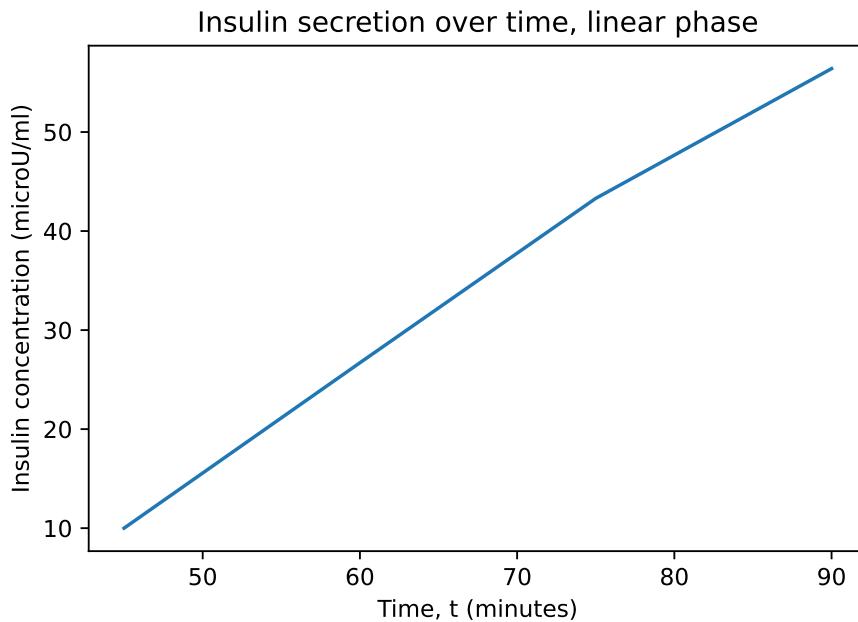


Figure 2.6: A plot for the linear part of the insulin secretion.

Both the time list, `t`, and the secretion list, `N`, need to be sliced, otherwise the number of elements in these two lists does not match and we get an error and will not be able to plot the data. As an alternative, we could also create new sliced versions of the time and secretion lists, and plot them instead.

```
t = [0, 15, 30, 45, 60, 75, 90, 105, 120]
# [0, 1, 2, 3, 4, 5, 6, 7, 8] Indices
# [3, 4, 5, 6] [3:7]

t_linear = t[3:7]

N = [7.1, 7.2, 7.1, 10, 26.7, 43.3, 56.4, 56.8, 49.7]
# [0, 1, 2, 3, 4, 5, 6, 7, 8] Indices
# [3, 4, 5, 6] [3:7]

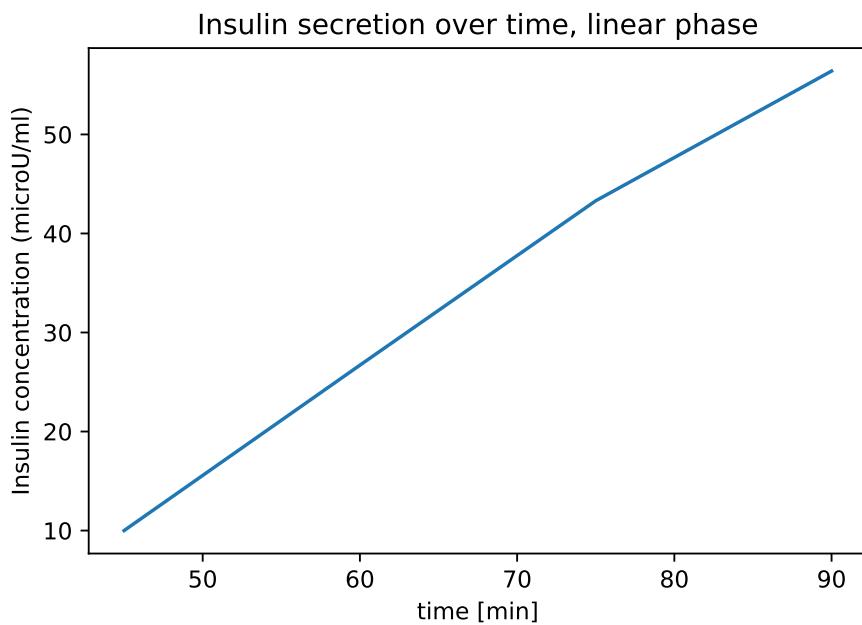
N_linear = N[3:7]

print(t_linear)
print(N_linear)
```

```
[45, 60, 75, 90]  
[10, 26.7, 43.3, 56.4]
```

We now can plot these as follows:

```
plot(t_linear, N_linear)  
title("Insulin secretion over time, linear phase")  
xlabel("time [min]")  
ylabel("Insulin concentration (microU/ml)")  
show()
```



Looking at the plot in figure 2.6, let us try to determine how much the secretion increases between 45 and 60 minutes and again from 60 to 75 minutes and from 75 to 90 minutes. We see that these increases are more or less the same: about $15\mu\text{U}/\text{ml}$ no matter which two timepoints we choose. We can now estimate the *rate of increase* in insulin production: Insulin production increases about $15\mu\text{U}/\text{ml}$ in 15 minutes, in other words, the *secretion rate* is about $1\mu\text{U}/\text{ml}$ per minute.

This type of increase, where the number of individuals increases with a fixed rate over time, is called *linear growth* (or linear increase). This property can be deduced from the plot. You will often encounter this type of linear response and being able to recognize that it is a linear relationship gives you the information that is required to model such phenomena. If the secretion rate is not limited in any way, secretion will continue forever at a constant rate.

2.4 Exploring the effects of obesity on insulin secretion

In the last section we saw that insulin secretion increased after breakfast. Now we want to have a look at the data from both non-obese and obese subjects and compare the data. Here, we focus on the secretion rate and compare it between the non-obese and obese subjects. We want to see if obese subjects exhibit a different secretion rate of insulin after breakfast when compared to non-obese subjects.

To explore the effects of obesity on insulin secretion rate we have to use both data series. We want to compare the collected data from both obese and non-obese subjects. The insulin secretion measurement data of non-obese and obese subjects is shown in the table below.

Time, minutes	Insulin non-obese ($\mu\text{U}/\text{ml}$)	Insulin obese ($\mu\text{U}/\text{ml}$)
0	7.1	18.4
15	7.2	19.8
30	7.1	18.4
45	10.0	20.3
60	26.7	63.3
75	43.3	97.4
90	56.4	120.2
105	56.8	118.7
120	49.7	121.0

We could compare the two tables directly, but seeing trends in the data will become much easier when we plot the two datasets in the same plot.

2.4.1 Multiple lines in the same plot

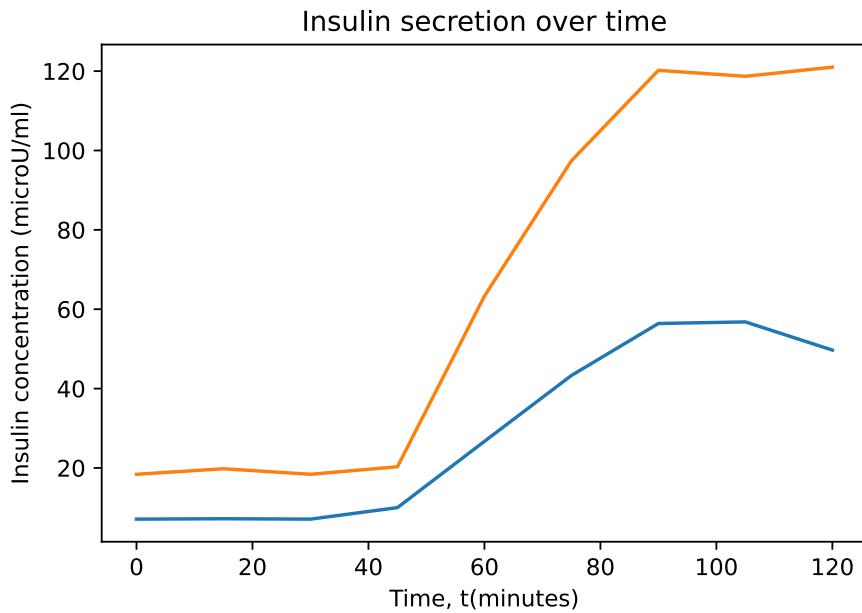
Adding a second curve to our plot is done by storing the second dataset in a new variable `O`, (for Obese):

```
0 = [18.4, 19.8, 18.4, 20.3, 63.3, 97.4, 120.2, 118.7, 121.0]
```

We then plot the second dataset after the first plot command with `plot(t, N)`.

```
from pylab import *
plot(t, N)
plot(t, O)
title("Insulin secretion over time")
xlabel("Time, t(minutes)")
ylabel("Insulin concentration (microU/ml)")

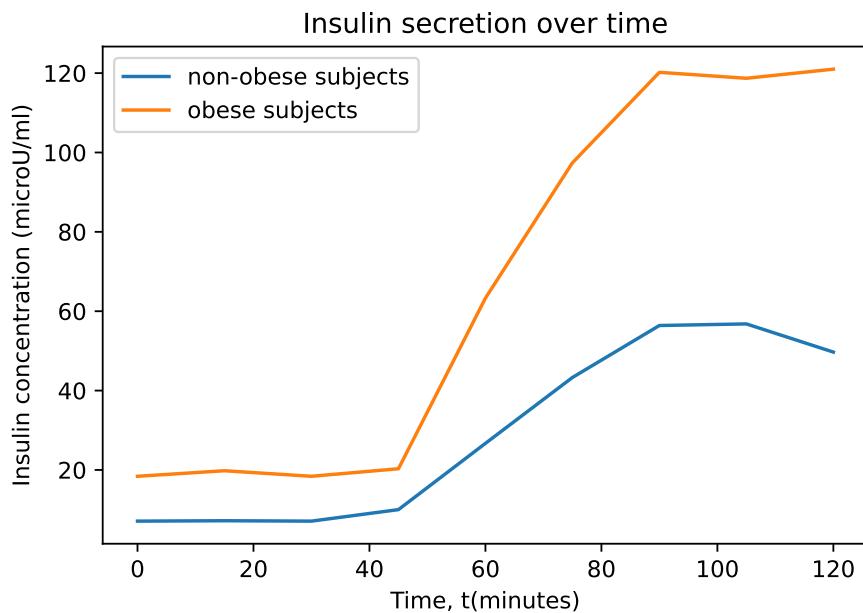
show()
```



We see that we automatically get a different color for the second line. Whenever we have more than one line in the plot, we should add a legend to it. This lets us put a label on each line, so we can tell which is which. A label is added to the plot argument, as follows:

```
plot(t, N, label = "non-obese subjects")
plot(t, O, label = "obese subjects")

title("Insulin secretion over time")
xlabel("Time, t(minutes)")
ylabel("Insulin concentration (microU/ml)")
legend()
show()
```



After we have plotted the curve we then call the `legend()` function for the legend to show. The full program for plotting both datasets (showing all datapoints) with labels in the same plot is then:

```
from pylab import *
t = [0, 15, 30, 45, 60, 75, 90, 105, 120]
N = [7.1, 7.2, 7.1, 10, 26.7, 43.3, 56.4, 56.8, 49.7]
O = [18.4, 19.8, 18.4, 20.3, 63.3, 97.4, 120.2, 118.7, 121.0]

plot(t, N, label = "non-obese subjects")
plot(t, O, label = "obese subjects")
title("Insulin secretion over time")
xlabel("Time, t(minutes)")
ylabel("Insulin concentration (microU/ml)")
legend()
show()
```

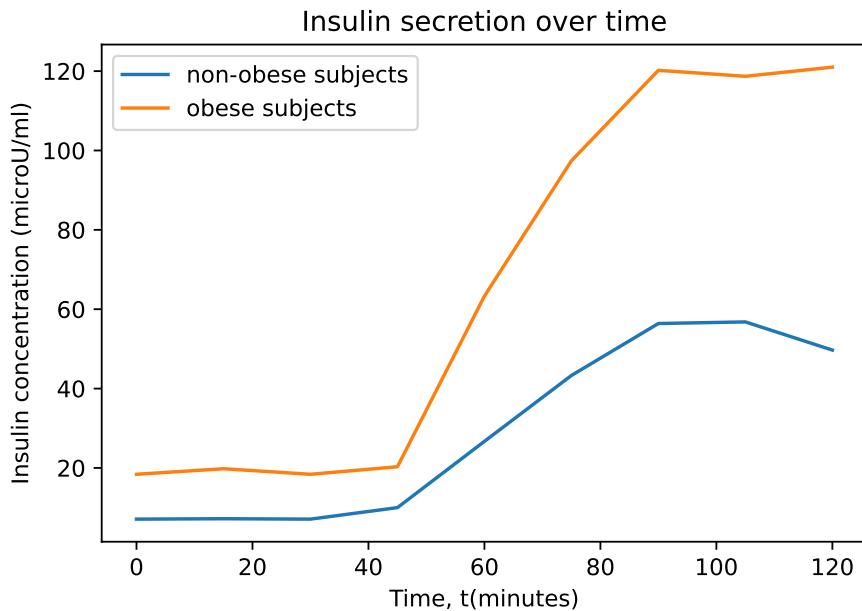


Figure 2.7: A plot comparing insulin secretion for non-obese and obese subjects.

2.4.2 Different line styles

As we see from our last plots, the individual curves get distinct line colors by default. The default settings of matplotlib are chosen to work well even for people with color blindness and when printing the plots in grey scale. We therefore recommend using these, but it is possible to change them. You can add custom colors by adding a color character to the line style in the plot command. For instance, if you want the first line to be green, we add "g-" to the plot command for the first curve: `plot(t, N, "g-", label = "non-obese subjects")`.

If we want the second curve to consist of blue circles, we add "bo" to the second plot command: `plot(t, O, "bo", label = "obese subjects")`.

The style "g-" represents a green (g) line (-), while "bo" is a blue (b) circle (o). The line style specifications are added after the x and y arguments of the plotting command and before any label argument.

We now plot the two in the same figure together with labels and a legend:

```

plot(t, N, "g-", label = "non-obese subjects")
plot(t, O, "bo", label = "obese subjects")

title("Insulin secretion over time")
xlabel("Time, t [minutes]")
ylabel("Insulin secretion, C [microU/ml]")
legend()
show()

```

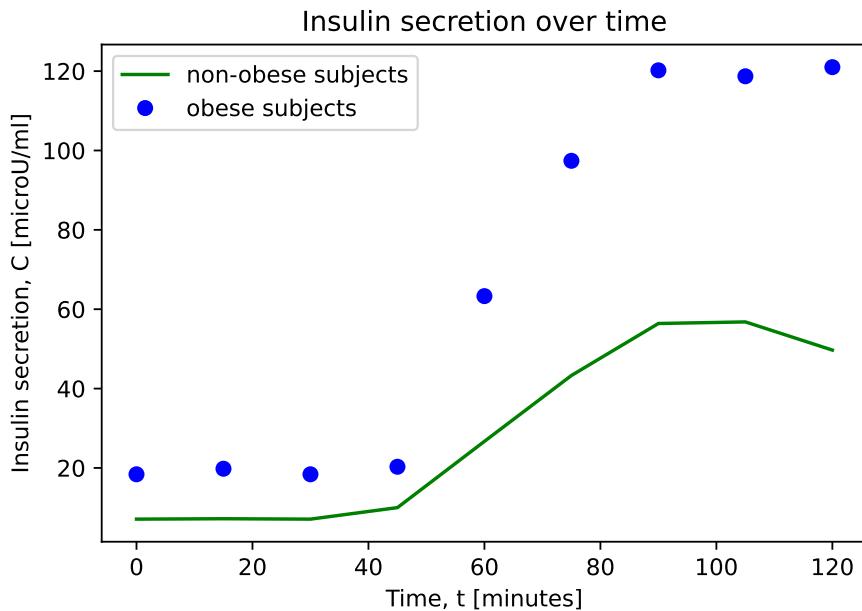


Figure 2.8: A plot comparing insulin secretion for non-obese and obese subjects demonstrating distinct line styles.

There are several different formatting options available in matplotlib. Some available colors include:

color	argument	color	argument
blue:	"b"	green:	"g"
red:	"r"	cyan:	"c"
magenta:	"m"	yellow:	"y"
black:	"k"	white:	"w"

And some available line styles are:

line style	argument	line style	argument
solid:	"--"	dashed:	"---"
dash dot:	"-.."	dotted:	

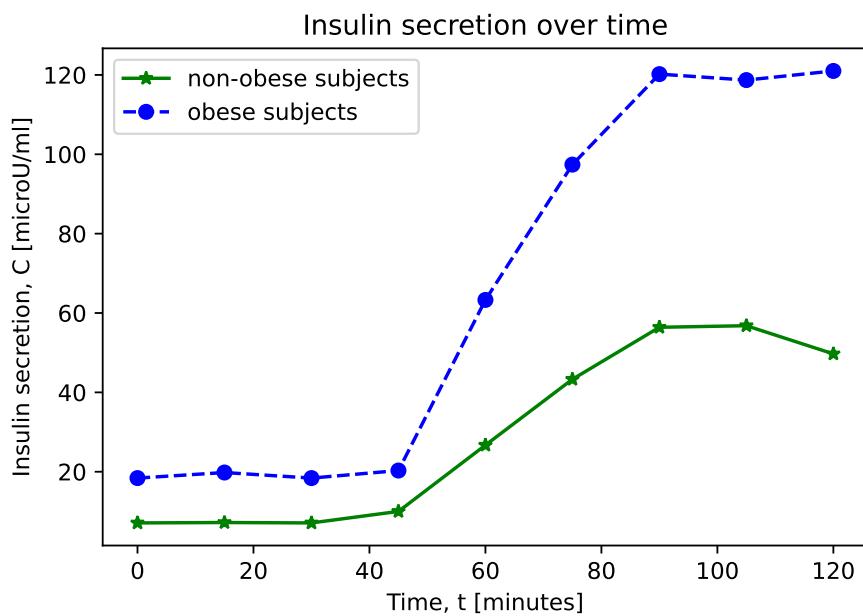
There are also several options for choosing markers for each point. This can be used in addition to having a line style, for example `b--o` are a blue (`b`) dashed line (`--`) with circles (`o`) at each point.

marker	argument	marker	argument
point:	". "	pixel:	", "
circle:	"o"	square:	"s"
triangle up:	"^"	triangle down:	"v"
triangle left:	triangle right:		
octagon:	"8"	pentagon:	"p"
hexagon:	"h"	star:	"*"
plus:	"+"	x:	"x"
diamond:	"D"		

A reasonable combination would be different line styles for both datasets, each with different colors for the datapoints:

```
plot(t, N, "g-*", label = "non-obese subjects")
plot(t, O, "b--o", label = "obese subjects")

title("Insulin secretion over time")
xLabel("Time, t [minutes]")
yLabel("Insulin secretion, C [microU/ml]")
legend()
show()
```



Explicit commands for plotting styles

Using "g-*" is a shorthand for the more explicit `color = "green"`, `linestyle = "solid"`, `marker = "*"`. Similarly, "b--o" is a shorthand for `color = "blue"`, `linestyle = "dashed"`, `marker = "o"`.

The longer versions of the plotting commands would thus be:

```
plot(t, N, color = "green", linestyle = "solid", marker = "*", label = "non-obese subjects")
plot(t, O, color = "blue", linestyle = "dashed", marker = "o", label = "obese subjects")
```

Note that `linestyle` often is abbreviated to `ls`. For some plotting commands, you can not use these abbreviations so it is useful to know the longer versions.

2.4.3 Saving the plot to file

To include the plots you have created in documents we need a way to save them to file. This is done by the `savefig("filename.extension")` command. The figure file will be saved in the directory where your program is located. The extension after the filename determines the format of the saved plot. A few of the most used are:

- Portable Network Graphics (PNG): `.png`
- Encapsulated PostScript (EPS): `.eps`
- Portable Document Format (PDF): `.pdf`
- Scalable Vector Graphics (SVG): `.svg`

PNG files are recommended for sharing images online, because they are well supported by all web browsers. EPS, PDF, and SVG are recommended for figures used in print, such as in reports or on a large poster, because they use vector graphics, which makes possible to scale the figure to enormous sizes without losing quality. The following command saves the previous plot to a PNG file named `insulin_secretion.png`:

```
savefig("insulin_secretion.png")
```

This command must be placed *before* the `show()` command.

2.4.4 Plotting insulin secretion rates for non-obese and obese subjects

The final program for plotting the two datasets, using the options we have discussed above is:

```
from pylab import *

t = [0, 15, 30, 45, 60, 75, 90, 105, 120]
N = [7.1, 7.2, 7.1, 10, 26.7, 43.3, 56.4, 56.8, 49.7]
O = [18.4, 19.8, 18.4, 20.3, 63.3, 97.4, 120.2, 118.7, 121.0]

plot(t, N, "g-", label = "non-obese subjects")
plot(t, O, "b-", label = "obese subjects")

title("Insulin secretion over time")
xlabel("Time, t [minutes]")
ylabel("Insulin secretion, C [microU/ml]")
legend()
savefig("secretion_rate.png")
show()
```

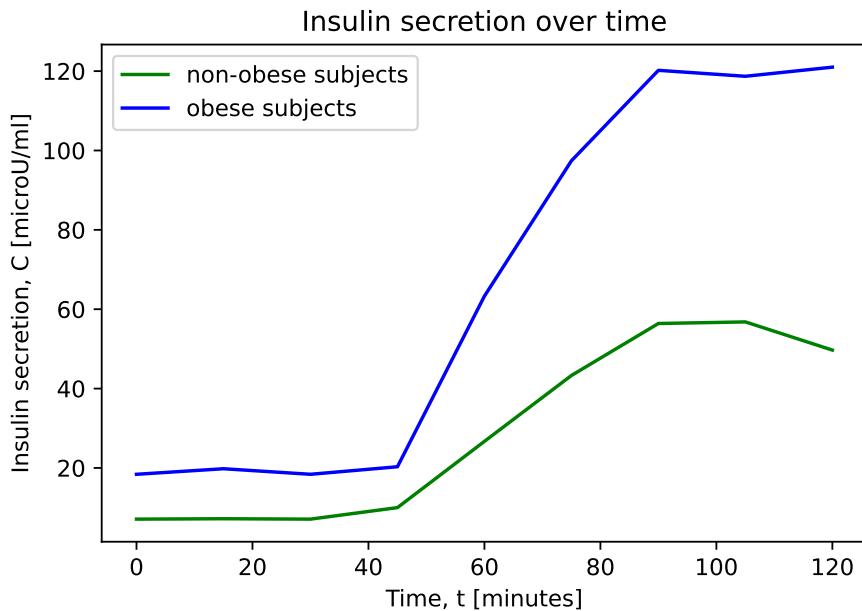


Figure 2.9: A plot comparing insulin secretion for non-obese and obese subjects, utilizing distinct line styles.

The two datasets are clearly different, illustrating the effect of obesity on insulin secretion. First, there is a higher pre-meal insulin concentration in obese subjects. Second, the increase in insulin concentration after breakfast appears steeper. The measure of how fast insulin is produced is called secretion rate. We will define this fully later, but for now, we just say that secretion rate is higher in obese subjects.

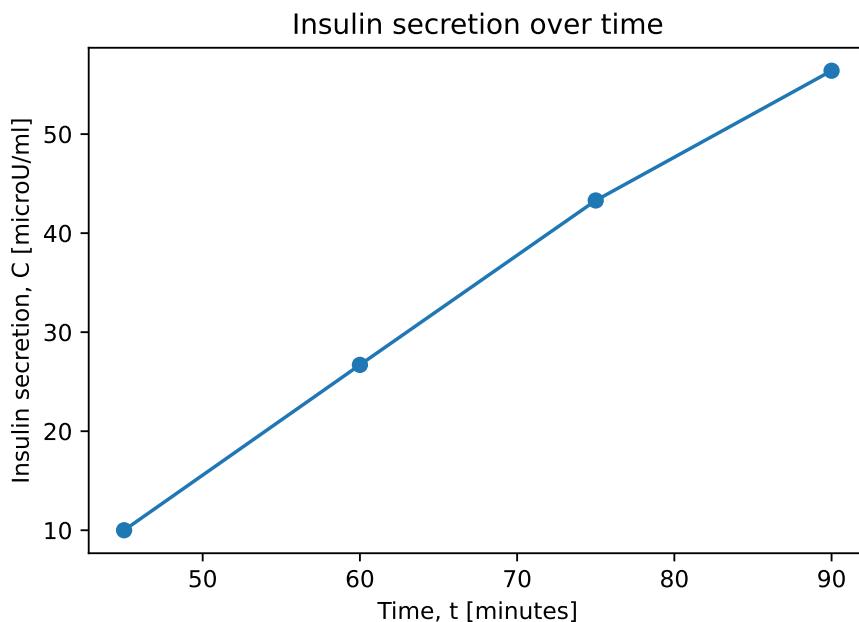
We have now identified insulin secretion as a response to breakfast as a linear process. As a next step, we want to quantify the secretion rate, and use this to make a linear growth *model* for insulin secretion.

2.5 Modeling the linear growth of insulin production

We observed in our plots that following ingestion of the breakfast, insulin concentration increased linearly until it plateaus. We repeat here the relevant plot for the non-obese subjects. The plot was based on showing only the data from the linear increase phase by slicing the data from the non-obese subjects:

```
from pylab import *
t = [0, 15, 30, 45, 60, 75, 90, 105, 120]
N = [7.1, 7.2, 7.1, 10, 26.7, 43.3, 56.4, 56.8, 49.7]
plot(t[3:7], N[3:7], "-o")
xlabel("Time, t [minutes]")
ylabel("Insulin secretion, C [microU/ml]")
```

```
title("Insulin secretion over time")
show()
```



This plot shows an almost a straight line. We use this together with the mathematical definition of a linear function to find the secretion rate, which is a measure for how fast the insulin is secreted. We should first make it exactly clear what we mean by a straight line.



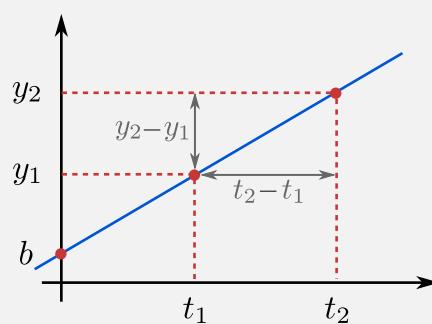
Linear functions

A straight line, or *linear function*, is written mathematically as:

$$y = at + b, \quad (2.1)$$

where a is the slope (steepness) of the line and b is where the line intercepts the y -axis. The slope of a linear function is found by taking two points on the graph and divide the change in y by change in t . This is:

$$a = \frac{\text{change in } y}{\text{change in } t}. \quad (2.2)$$



The above figure shows a linear function with the slope given as

$$a = \frac{y_2 - y_1}{t_2 - t_1}. \quad (2.3)$$

Note that it does not matter which two points we choose for calculating the slope. A linear function has the same slope everywhere.

We use the mathematical definition of a straight line to describe the line in our insulin secretion plot as:

$$N = at + b, \quad (2.4)$$

i.e. the concentration (Units) of insulin, is given as a linear function of time t . All we have done here is to set $y = N$. We are first interested in the slope of this line, a , which is in our case is called the *secretion rate*. A steeper slope means that the secretion is happening at a faster rate, which means it takes a shorter time to release the same amount of insulin.

Let us use now derive the slope of the line. There are many methods that can be used to find this slope. Here we show a simple method, by choosing the first and last element of the part of the graph showing linear growth. These correspond, as we have seen, to the fourth and seventh element in N .

```
t = [0, 15, 30, 45, 60, 75, 90, 105, 120]
N = [7.1, 7.2, 7.1, 10, 26.7, 43.3, 56.4, 56.8, 49.7]

N_1 = N[3]
N_2 = N[6]
t_1 = t[3]
t_2 = t[6]
```

Using the formula for deriving the slope, we can obtain the secretion rate by dividing the difference in insulin secretion by the time difference:

```
a = (N_2 - N_1)/(t_2 - t_1)
print("Secretion rate:", a, "microU/ml per minute")
```

```
| Secretion rate: 1.031111111111111 microU/ml per minute)
```

As you see from the output the secretion rate is approximately $1.03\mu\text{U}/\text{ml}$ per minute, which is very close to the secretion rate of $1\mu\text{U}/\text{ml}$ per minute that we estimated from looking at the figure. The estimate above is obtained by a very simple method. Better (and more sophisticated) methods exists, but they are outside the scope of this book.

We have now calculated the secretion rates, a , in our function. Now we need the intercept b , the starting amount of insulin. If we use a value for N from and the corresponding value for t (from the linear growth phase of our dataset), we can calculate b , and thus arrive at the intercept b mathematically. We can use either the first or the last point of the linear growth phase. Let use the first datapoint, $10\mu\text{U}/\text{ml}$ at 45 minutes, this gives us $N = 10$, $a = 1.03$ and $t = 45$. These values we can use in

$$N = at + b \quad (2.5)$$

Inserting the numbers we have chosen gives us:

$$\begin{aligned} 10 &= 1.03 \times 45 + b \\ b &= 10 - 1.03 \times 45 \\ b &= -36.35. \end{aligned}$$

So the linear equation becomes:

$$N = 1.03t - 36.35 \quad (2.6)$$

2.5.1 Revisiting the effect of obesity on the secretion rate of insulin

We will now have a look at insulin secretion rates in obese subjects and compare it to the rates in non-obese subjects, using both data series (lists). As we have already done most of these things, we just need to combine what we have learned so far.

We use the same data as above, and perform the same calculations for the data of the obese subjects.

```
t = [0, 15, 30, 45, 60, 75, 90, 105, 120]
N = [7.1, 7.2, 7.1, 10, 26.7, 43.3, 56.4, 56.8, 49.7]

N_1 = N[3]
N_2 = N[6]
t_1 = t[3]
t_2 = t[6]

a_N = (N_2 - N_1)/(t_2 - t_1)
print("Secretion rate of non-obese subjects:", a_N, "U/ml per minute")

O = [18.4, 19.8, 18.4, 20.3, 63.3, 97.4, 120.2, 118.7, 121.0]

O_1 = O[3]
O_2 = O[6]

a_O = (O_2 - O_1)/(t_2 - t_1)
```

```
print("Secretion rate of obese subjects:", a_0, "U/ml per minute)")
```

```
Secretion rate of non-obese subjects: 1.03111111111111 U/ml per minute)
Secretion rate of obese subjects: 2.22 U/ml per minute)
```

The secretion rate seems to be doubled in obese subjects when compared to non-obese subjects. Let us check this:

```
d = a_0/a_N
print("Ratio between secretion rate observed for obese and non-obese subjects:", d)
```

```
Ratio between secretion rate observed for obese and non-obese subjects: 2.1530172413793105
```

We can also round this number to make it more readable:

```
rounded_d = round(d, 2)
print("Ratio between secretion rate observed for obese and non-obese subjects:", rounded_d)
```

```
Ratio between secretion rate observed for obese and non-obese subjects: 2.15
```

We see that the rate of the insulin secretion after breakfast in obese subjects is a little more than double that of the non-obese subjects.

2.5.2 Linear functions for insulin secretion in non-obese and obese subjects

We previously discussed the mathematical definition of the concentration of insulin secretion over time (for the non-obese subjects) with this linear function:

$$N = at + b, \quad (2.7)$$

Similarly, for insulin secretion in the obese dataset, we derive (check this for yourself):

$$O = 2.22 \times t - 79.6 \quad (2.8)$$

Note that the intercept and slope both are about two times higher for the obese dataset.

It becomes clear from the various analyses that we performed that obese subjects have much higher secretion rates of insulin as well as reach much higher maximum levels of insulin concentrations in their body.

2.6 Simulating insulin secretion using our model

We now have a mathematical model for the linear growth of insulin secretion after a meal. An important next step is to check how good our model compares to the real data. For this, we need

to *implement* the model in Python, simulate insulin secretion for the same time interval as we used to derive the model, and compare the results with the measured insulin secretion.

In this chapter, we will make a very simple implementation. We can do this, because linear growth is represented by a linear function, which is visualised as a straight line. In order to draw a straight line, all we need is two datapoints.

To simulate the insulin secretion using our model, we therefore use two timepoints, calculate the secretion as predicted by the model at these two timepoints, and draw a straight line through the two predicted points. For now, let us use the interval from 0 to 90 minutes, so $t = 0$ and $t = 90$. We use two new lists, `t_sim` and `N_sim` (“sim” stands for “simulated”). The first list then becomes:

```
t_sim = [0,90]
```

To calculate the value of `N_sim` corresponding to `t_sim = 0` we need to look at the intercept b , see the mathematical definition of a linear function as described in Section 2.5. Earlier, we determined this intercept to be -36.35 . We could also calculate this value by entering 0 for t in our equation (2.6) for linear growth $N = 1.03t - 36.35$. This then also leads to $N = -36.35$. We can now set up our list `N_sim`:

```
N_sim = []
N_sim.append(-36.35)
```

Note that initiating a list with one element also could be done in one step:

```
N_sim = [-36.35]
```

Next, we need to calculate the value of `N_sim` corresponding to `t_sim = 90` minutes.

For this, we can again use our linear growth equation (2.6), by entering $t = 90$ minutes:

```
# N = 1.03t -36.35 for t = 90 minutes
print(1.03 * 90 - 36.65)
```

```
56.05000000000004
```

We add the result of this calculation to the list `N_sim`

```
N_sim.append(1.03 * 90 - 36.65)
print(N_sim)
```

```
[-36.35, 56.05000000000004]
```

Now our lists have the datapoints we need, and we can plot them together with the measured data. We use a dotted line for the simulated data:

```
plot(t, N, label = "Non-obese subjects")
plot(t_sim, N_sim, ":" , label = "Linear equation non-obese dataset")
xlabel("Time, t [minutes]")
ylabel("Insulin secretion, C [microU/ml]")
```

```
title("Insulin secretion over time")
legend()
show()
```

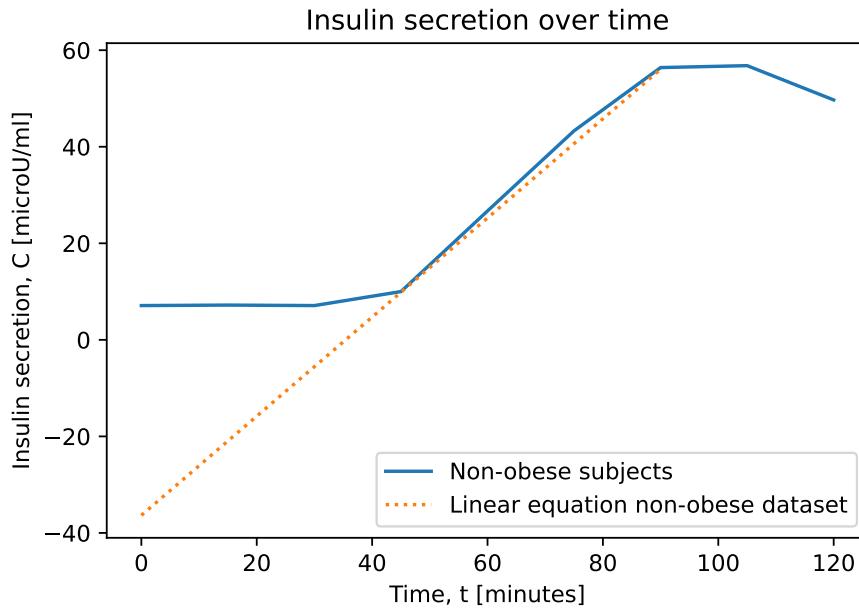


Figure 2.10: A plot comparing measured insulin secretion with the linear model.

We see from the figure that the measured insulin secretion overlaps with the simulated one, at least for the part of the graph where there is an linear increase. However, for a better match, we should let the simulation start at the start of the increase of insulin secretion. But when is the time point where secretion starts to increase? It seems to be somewhere between the measurements at 30 and 45 minutes. But can we be more precise? Could we use our mathematical model to help us answer this question?

2.6.1 How soon after a meal does insulin production start?

Using our simulation of insulin secretion, we now want to ask the following question: How soon after breakfast does the insulin secretion start to increase?

To answer this, we need one more thing, and that is the pre-meal level of insulin. Looking at the data and the graph, using $7.1\mu\text{U}/\text{ml}$ for this would be an acceptable choice. Let us add this pre-meal level to our plot with another dotted line. Since this value is constant, we can use the command `axhline` to plot a horizontal line. To make it appear at a value of 7.1 as a red-colored dotted line, and add a label for it to the legend, we use `axhline(7, ls = ":" , color = "red", label = "Pre-meal secretion")`. Note that with `axhline`, we have to use `ls` and `color` explicitly, we cannot use the shorthand notation we use with the `plot()` function (as explained in Section 2.4.2).

```
plot(t, N, label = "Non-obese subjects")
```

```

plot(t_sim, N_sim, ":" , label = "Linear equation non-obese dataset")
axhline(7, ls = ":" , color = "red", label = "Pre-meal secretion")
xlabel("Time, t [minutes]")
ylabel("Insulin secretion, C [microU/ml]")
title("Insulin secretion over time")
legend()
show()

```

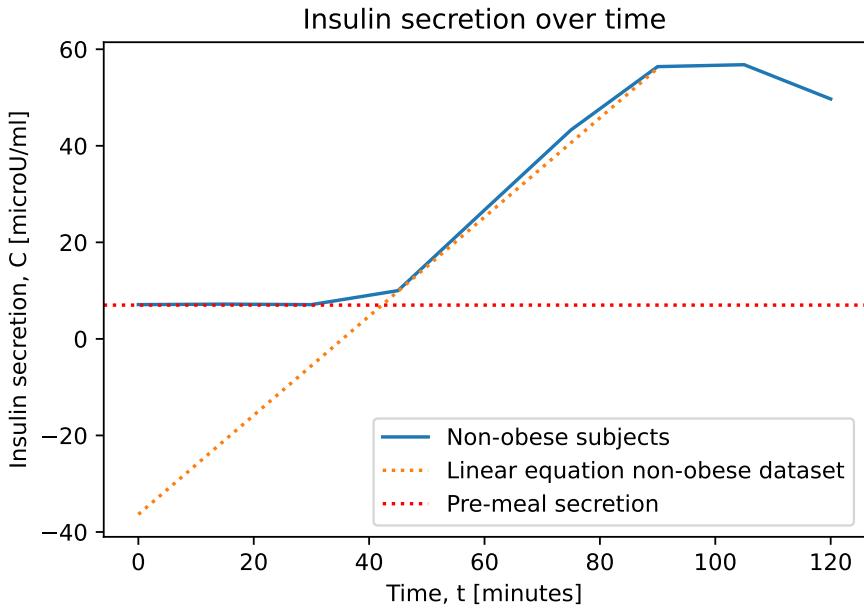


Figure 2.11: A plot comparing measured insulin secretion with the linear model and the pre-meal secretion.

How can we now determine the onset of the increase in insulin secretion following a meal? Looking at the figure, the insulin secretion starts to go up between timepoints 30 and 45 minutes. But we can also use our linear growth model to answer this question. We then need to rephrase the question to “at which timepoint starts the insulin secretion to be above the pre-meal secretion levels?” This is the same as asking “from which timepoint does our model exceed $7.1\mu\text{U}/\text{ml}$?” In other words, we need find the point our dotted lines in the plot cross, or solve $N = 1.03t - 36.35$ for $N = 7.1$.

Doing this gives us

$$7.1 = 1.03t - 36.35 \quad (2.9)$$

$$7.1 + 36.35 = 1.03t \quad (2.10)$$

$$t = \frac{43.45}{1.03} \quad (2.11)$$

$$t = 42.2 \text{ minutes} \quad (2.12)$$

We see that, using our model of insulin secretion, and some mathematics, we can answer a biological question rather precisely.

2.7 Visualizing 24 hours of insulin secretion

The profiles of insulin secretory rates we saw in Figure 2.9 revealed that insulin secretion rates were elevated in the obese group both under basal fasting conditions (morning before breakfast) and in response to the breakfast. But how do insulin secretion rates vary over a 24 hour period, and are the responses that we observed during breakfast similar for lunch and dinner?

So far, we have been using lists to plot our measurements. However, especially for larger datasets, we would like to avoid writing those lists manually and instead loading data from files into our programs. Loading this data from file also saves us time, because we no longer have to type in all the data by hand. Further, it enables us to analyze data from multiple experiments by loading multiple files and plotting them side by side.

We begin with a file containing the insulin secretion data we have used for the non-obese subjects. A format often used for files of this type is the `.csv` (Comma Separated Values) format. Microsoft Excel, LibreOffice Calc and other spreadsheet programs have an option to write data to csv files. The `.csv` file for our experiment looks like this:

```
minutes,non-obese
0,7.1
15,7.2
30,7.1
45,10
60,26.7
75,43.3
90,56.4
105,56.8
120,49.7
```

The file contains column headers as first line: `minutes` and `non-obese`, separated by a comma. Each next line contains two numbers, also separated by a comma, where the first number is the timepoint, and the second is the insulin measurement.

We can read this file with Pandas. Pandas is a Python package for scientific analysis that, among other things, contains the function `pandas.read_csv()`, which is used to read such `.csv` files:

```
import pandas
data = pandas.read_csv("insulin_prod_non_obese.csv")
print(data)
```

```
      minutes  non-obese
0          0       7.1
1         15       7.2
2         30       7.1
3         45      10.0
4         60      26.7
```

```
5      75      43.3
6      90      56.4
7     105      56.8
8     120      49.7
```

This loads the contents of the .csv file into a variable called `data` and prints those contents to the screen.

We can access the individual columns as `data["minutes"]` and `data["non-obese"]`, i.e. using the column headers (names) inside quotation marks inside square brackets. This is syntax particular to the pandas package, but also similar to working with another type of variable we will encounter later. To access the data from both columns similarly to the above code snippets, we convert them to lists using the `list()` function:

```
t = list(data["minutes"])
N = list(data["non-obese"])
print(t)
print(N)
```

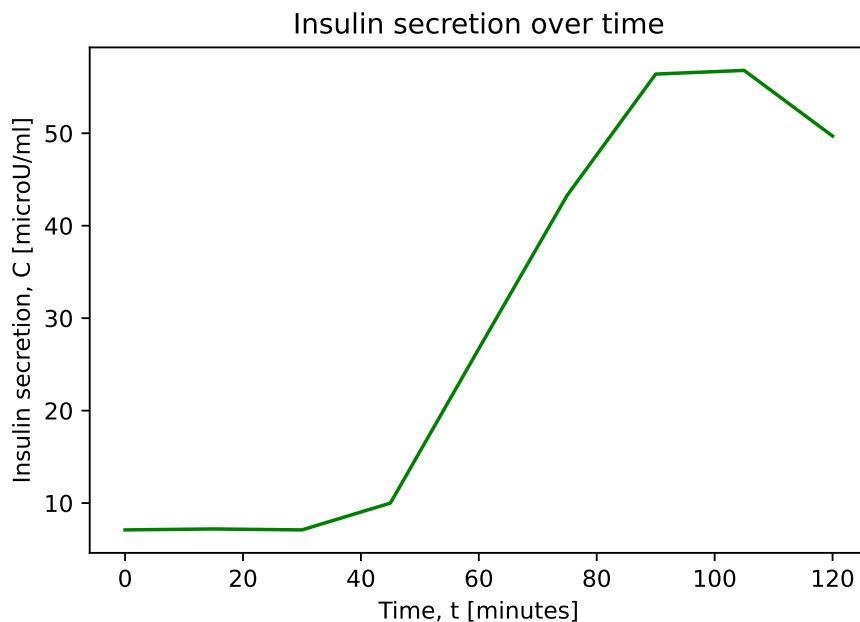
```
[0, 15, 30, 45, 60, 75, 90, 105, 120]
[7.1, 7.2, 7.1, 10.0, 26.7, 43.3, 56.4, 56.8, 49.7]
```

Putting this together with the `plot()` function, we are able to plot the data in the file:

```
from pylab import *
import pandas

data = pandas.read_csv("insulin_prod_non_obese.csv")
t = list(data["minutes"])
N = list(data["non-obese"])

plot(t, N, "g-")
xlabel("Time, t [minutes]")
ylabel("Insulin secretion, C [microU/ml]")
title("Insulin secretion over time")
show()
```



This plot is identical to our previous plot, Figure 2.4. If there had been any differences between the plots we would know there was bug in our code since the data and plotting code used are equal.

2.7.1 Plotting two series in one plot

We are now ready to plot a large data set containing 24 hours of insulin secretion in both non-obese and obese subjects. We use the same code but now read from a data file with measurements from a longer time period.

After loading the data into a data frame let's have a look at the data first. As it is a large data file we only look at the top part of the data frame by using index [:10]:

```
import pandas

data = pandas.read_csv("insulin_prod_all.csv")
print(data[:10])
```

	minutes	non-obese	obese
0	0	7.1	18.4
1	15	7.2	19.8
2	30	7.1	18.4
3	45	10.0	20.3
4	60	26.7	63.3
5	75	43.3	97.4
6	90	56.4	120.2
7	105	56.8	118.7
8	120	49.7	121.0
9	135	42.0	98.7

Now let us plot the two separate time series. For this we need to make three lists, one for the time points, and the two for subject groups.

```
from pylab import *
t_all = list(data["minutes"])
N_all = list(data["non-obese"])
O_all = list(data["obese"])

plot(t_all, N_all, "g-", label = "non-obese subjects")
plot(t_all, O_all, "b-", label = "obese subjects")
xlabel("Time, t [minutes]")
ylabel("Insulin secretion, C [microU/ml]")
title("Measured insulin secretion over 24 hours")
legend()
show()
```

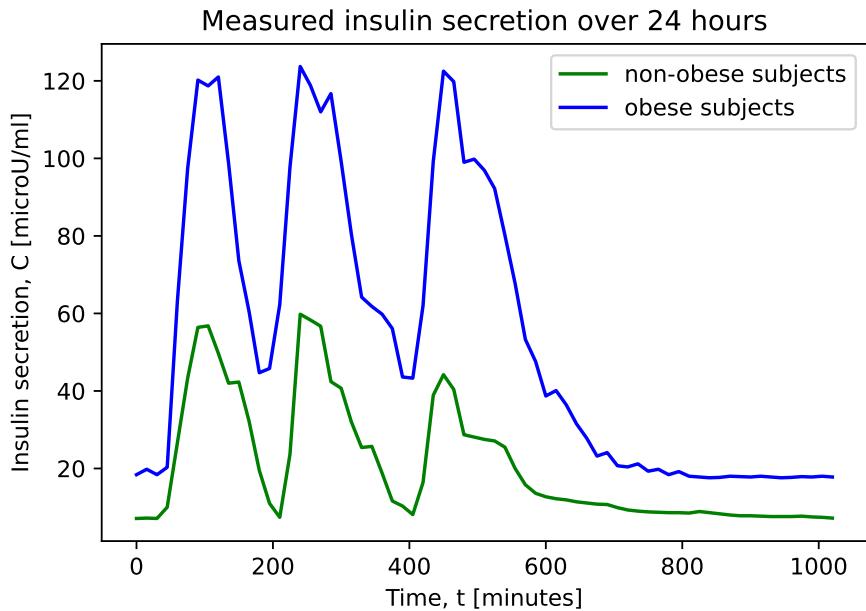


Figure 2.12: Insulin secretion measured for 24 ours in non-obese and obese subjects.

There is a lot of information that can be extracted from this plot. For example, the insulin secretion in response to the meals is very obvious. Furthermore, insulin secretion did not fall as rapidly as it did after breakfast than it did after dinner and lunch in both subject groups.

The two datasets are clearly different, illustrating the effect of obesity on insulin secretion in comparison to insulin secretion in non-obese subjects. Basal insulin secretion (when fasting) is much lower in non-obese subjects than obese subjects. Also, in obese subjects insulin secretion did not drop to basal as was the case in non-obese subjects. Furthermore, the difference in insulin secretion rate observed after breakfast was much lower during dinner and lunch, as insulin secretion levels were already elevated between meals.

2.8 Summary

In this chapter we have analyzed linear growth of insulin secretion. The important computer science concepts you learned in this chapter are lists, including sublists/slices, plotting, and reading from a file. You have learned to store data in Python, either storing the data in lists or reading from a file, as well as plotting the data. You have also learned to visualize the insulin secretion data, as well as modeling the linear growth rate mathematically, and simulate linear insulin secretion using an implementation in Python.

2.8.1 Lists

A list is used to collect a number of objects in an ordered sequence. A list element can be any Python object, including numbers, strings, and other lists, for instance. An example of a list is `[1, 4.4, "insulin"]`. Note that only a subset of these are explained in the present chapter.

Syntax	Description	Result
<code>L = []</code>	Initialize an empty list	[]
<code>L = [1, 4.4, "insulin"]</code>	Initialize a list	[1, 4.4, "insulin"]
<code>len(L)</code>	number of elements in list L	3
<code>L[1]</code>	Index a list, get element 1	4.4
<code>L[-1]</code>	Get last element in a list	"insulin"
<code>L[1:3]</code>	Slice: copy data to sublist	[4.4, "insulin"]
<code>L.index(4.4)</code>	Find index of first occurrence of 4.4	1
<code>L.append(2)</code>	Add 2 to the end of L	[1, 4.4, "insulin", 2]
<code>L.insert(1, "a")</code>	Insert "a" before index 1	[1, 'a', 4.4, 'insulin', 2]
<code>del L[1]</code>	Delete an element (index 2)	[1, 4.4, 'insulin', 2]
<code>L.remove(4.4)</code>	Remove first element with value 4.4	[1, 'insulin', 2]
<code>L + [1, 3]</code>	Merge two lists	[1, 'insulin', 2, 1, 3]
<code>L.count("insulin")</code>	Count occurrences of "insulin"	1
<code>L.copy()</code>	Copy the list	[1, 'insulin', 2, 1, 3]

The following operations are only valid if we have a list with only floats/ints or strings, and they are shown on the list `L = [4, 2, 10]`:

Syntax	Description	Result
<code>min(L)</code>	The smallest element in L	2
<code>max(L)</code>	The largest element in L	10
<code>sum(L)</code>	Add all elements in L	16
<code>sorted(L)</code>	Return sorted version of list L	[2, 4, 10]

2.8.2 Plotting

You have learned the following plot commands:

Syntax	Description
<code>plot(t, N, "g-", label = "non-obese subjects")</code>	Plot t and N as a green line "g-" with a label
<code>plot(t, O, "bo", label = "obese subjects")</code>	Plot t2 and O as yellow circles "yo" with a label
<code>xlabel("Time, t [minutes]")</code>	Label for x-axis
<code>ylabel("Insulin secretion, C [microU/ml]")</code>	Label for y-axis
<code>title("Insulin secretion over time")</code>	Title of figure
<code>legend()</code>	Show the legend in the plot
<code>savefig("name_of_plot.png")</code>	Save the plot as name_of_plot.png
<code>show()</code>	Show the plot

2.8.3 Reading from file

You can read in data from a file with `pandas.read_csv()`.

```
import pandas
data = pandas.read_csv("insulin_prod_non_obese.csv")
print(data)
```

	minutes	non-obese
0	0	7.1
1	15	7.2
2	30	7.1
3	45	10.0
4	60	26.7
5	75	43.3
6	90	56.4
7	105	56.8
8	120	49.7

We access the individual columns as `data["minutes"]` and `data["non-obese"]`. You convert the columns to lists with the `list()` function:

```
t = list(data["minutes"])
N = list(data["non-obese"])
print(t)
print(N)
```

[0, 15, 30, 45, 60, 75, 90, 105, 120]
[7.1, 7.2, 7.1, 10.0, 26.7, 43.3, 56.4, 56.8, 49.7]

For those interested, the data used in this chapter is derived from this scientific publication: <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC329589/>, specifically from figure 1.

Chapter 3

Modeling unlimited bacterial population growth

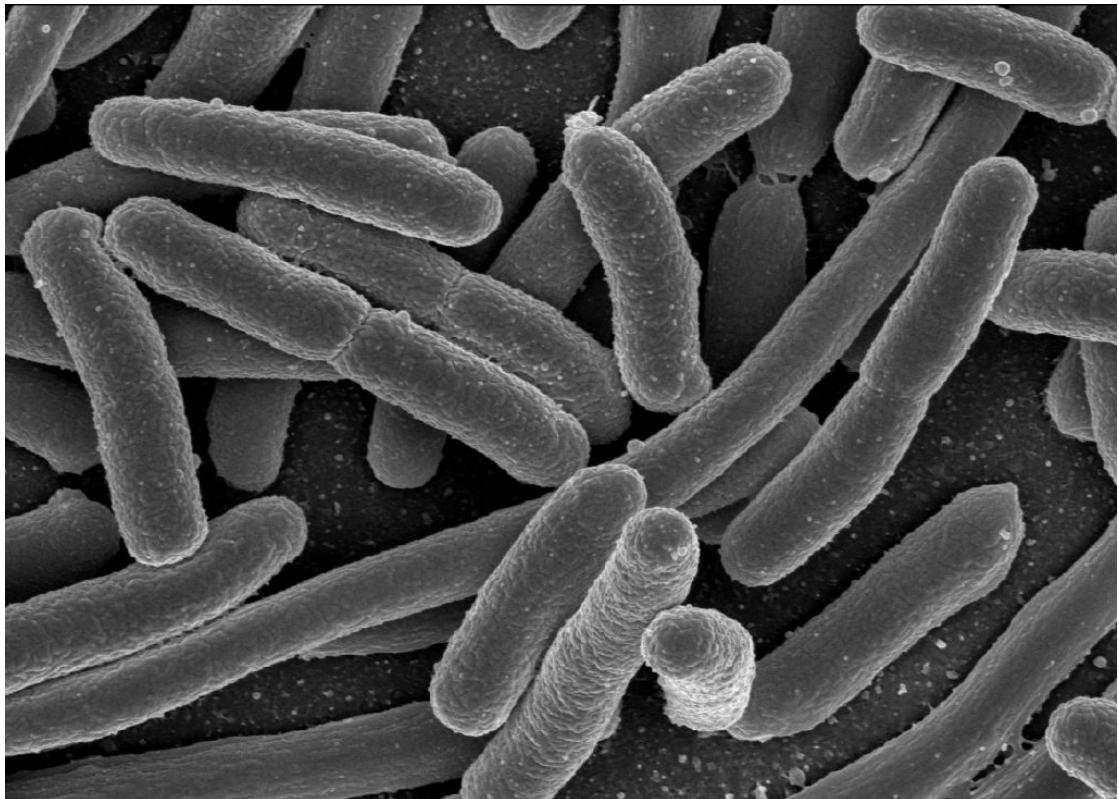


Figure 3.1: Scanning electron micrograph of *E. coli* bacteria [4]. In this chapter we simulate the growth of a population of *E. coli* bacteria in a virtual experiment on a computer.

Among single-celled organisms, the *Escherichia coli* (*E. coli*) bacterium is a popular *model organism*. A model organism is a non-human organism that is extensively studied to understand particular biological phenomena. Discoveries in *E. coli* provide insight into the workings of other organisms.

There are many reasons why *E. coli* is used as a model organism. Two of these are that, firstly, the genome of *E. coli* is quite simple, and secondly, *E. coli* is easy to work with in the laboratory. It grows fast and is relatively safe to study. The time it takes for one generation to reproduce and thus create a new generation is about 20 minutes, so a large number of generations can be studied in a short time span. A genetic change in one bacteria can thus quickly become an entire population that can be studied further. Most of our present concepts of molecular biology, including the basic mechanisms of genetics such as DNA replication and protein synthesis are derived from studies of *E. coli*. In addition, many important techniques in molecular biology were initially developed using *E. coli*, for example *molecular cloning*. Molecular cloning is a technique used when one wants to copy a piece of DNA many times, it is possible to do this by introducing the DNA into a bacteria and then let the bacteria multiply many times. We can then extract the DNA from the bacteria.

In this chapter we use *E. coli* as a model organism for studying population growth. Bacterial population growth is a type of *population dynamics*, which is the study of *population size* (number of individuals) over time.

Bacterial population growth can be studied both by analyzing data from real experiments or by creating virtual experiments. We start this chapter with a short overview on how bacteria reproduce. We will then determine the growth characteristics of an *E. coli* population from real growth data. We then *simulate* the bacterial population growth on the computer by imitating what we know about the real experiment, and compare the results from the simulation with the real growth data.



Learning outcomes

After working with this chapter you know:

- how bacteria reproduce and what binary fission is,
- the different phases in the bacterial population growth curve,
- the meaning of generation time, doubling time, and growth rate,
- how to create and implement an exponential growth model,
- how to compare the model results to experimental data, and
- the limitations of the exponential growth model.

The programming concept we introduce in this chapter is `while` loops.

3.1 How bacteria multiply: binary fission

Before we start analyzing the bacterial population growth we go briefly through how bacteria reproduce. Bacteria are single-celled organisms that are only a few micrometers in length. They belong to a group of organisms called *prokaryotic* organisms. The genetic material of a cell is stored in the DNA, and the DNA is stored in structures called *chromosomes*. The cells of prokaryotic organisms have no inner membrane around the nucleus, and the chromosomes float freely around the cell. In comparison, *eukaryotic* organisms are organisms whose cells have a membrane surrounding the cell nucleus, as well as other membrane-bound parts (organelles). For eukaryotic organisms, the chromosomes are stored in the cell nucleus. Plants and animals are examples of eukaryotic organisms.

In bacterial populations the individual cells usually increase in size until they divide into two new cells. This process is called *binary fission* which is a form of *asexual reproduction* where two daughter cells arise from a single parent cell. The daughter cells are thus identical to the original cell, as long as no mutations (random changes in the DNA sequence) occur. Due to binary fission a population of bacteria potentially doubles in size every generation.

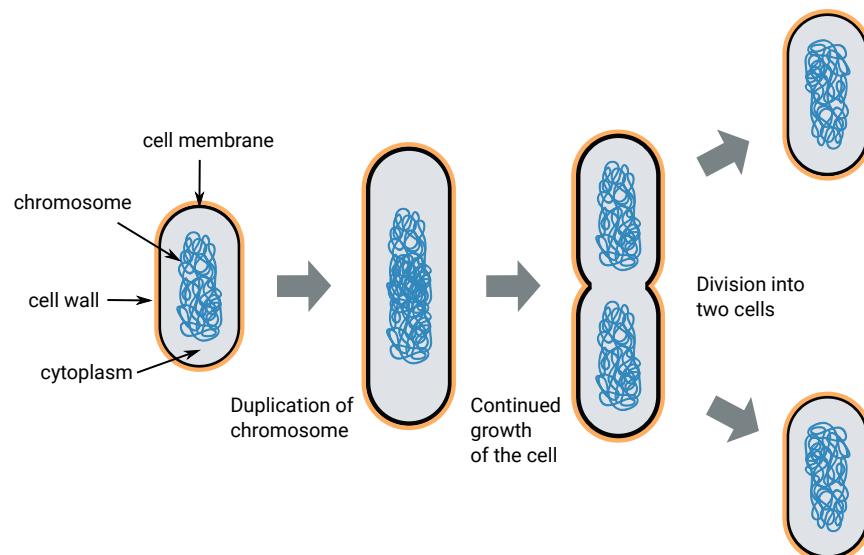


Figure 3.2: Binary fission in a bacterium. A bacterial cell is shown with structures indicated. The cytoplasm is the material inside the cell, excluding the chromosome, which contains the DNA. The cell membrane surrounds the cytoplasm and allows substances to pass through, while the cell wall is an outer protective layer. Binary fission is the process where this single cell duplicates its chromosomes, grows to about double its original size, and divides itself into two cells.

When bacteria reproduce they grow to approximately twice their original length by enlarging the cell membrane, cell wall, and overall volume. As the bacterium increases in size, the chromosome is duplicated and the copies are pulled to separate ends of the bacterium, Figure 3.2. These ends are eventually separated by the growth of a new cell wall that divides the parent cell into two daughter cells. The (average) time between these divisions of cells is called *generation time*.

For organisms that can reproduce more than once (for example, us humans), the definition of generation time is somewhat more involved and is outside the scope of this chapter. The generation time is highly variable and depends on a number of factors, including physical and nutritional factors. In the case of the bacterium *E. coli*, the generation time can be as short as 20 minutes under ideal conditions.

3.2 Analyzing *E. coli* population growth

We want to study the population growth of *E. coli* in the laboratory. This is done by growing *E. coli* in a closed environment, for example on a solid support, or in a flask, with appropriate reagents supporting growth, called *growth medium*. The bacteria have a fixed amount of this growth medium, which for our experiment is a liquid that supports the growth of microorganisms. No fresh medium is provided during the experiment, so the nutrient concentrations decline with time while the concentrations of waste products increase. We measure the number of bacteria every 30 minutes.

In most studies, the number of *E. coli* bacteria is usually quantified by techniques such as *optical density*, where the amount of light absorbed by the bacteria culture is related to the population size. However, we will only consider the number of individuals to avoid unnecessary complications.

3.2.1 Visualizing the phases of bacterial population growth

The experimental data for the growth of our *E. coli* population is available in a file called `ecoli_all.csv`. As we have seen in Section 2.7 we can load this data from a file using the `pandas` package.

```
import pandas

data = pandas.read_csv("ecoli_all.csv")

print(data)
```

	t	E
0	0	10010
1	30	9951
2	60	10042
3	90	25587
4	120	76327
..
59	1770	371768
60	1800	365374
61	1830	353612
62	1860	360434
63	1890	356572

[64 rows x 2 columns]

There are two columns in the file labelled `t` for timepoints, and `E` for measured number of *E. coli*. The final timepoint is 1890 minutes, which means that we have almost 32 hours of growth data. As we did before, we access the individual columns as `data["t"]` and `data["E"]`, and to access the data from both columns we convert them again to lists using the `list()` function. Finally, we plot the data.

```

1 import pandas
2 from pylab import *
3
4 data = pandas.read_csv("ecoli_all.csv")
5 t = list(data["t"])
6 E = list(data["E"])
7
8 plot(t, E, "o-")
9 xlabel("Time, t (minutes)")
10 ylabel("Population size, E")
11 title("Measured bacterial population growth")
12 show()

```

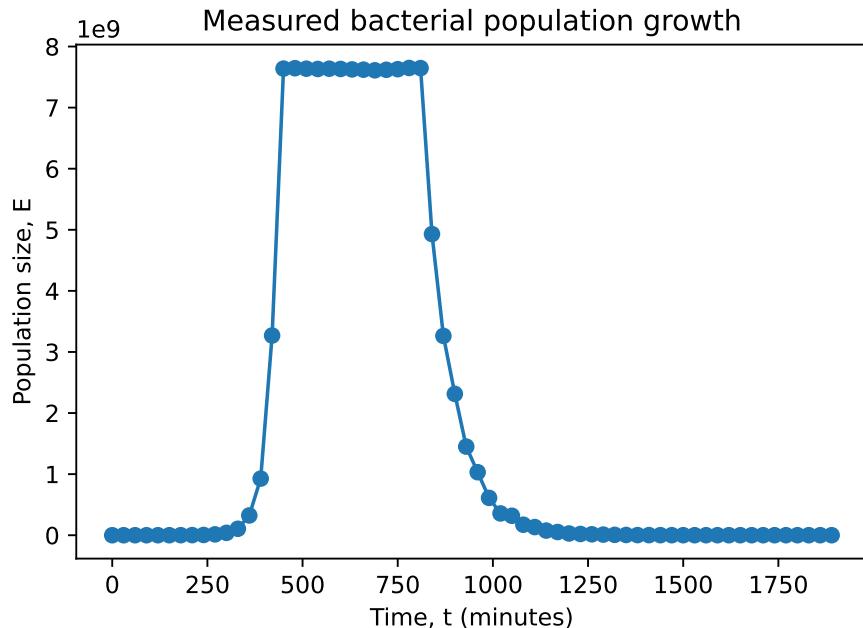


Figure 3.3: Bacterial population growth, all phases.

Note that the values on *y*-axis may be given in *scientific notation*. This means all numbers on *y*-axis are written in the form

$$m \times 10^9, \quad (3.1)$$

where *m* is the value on the *y*-axis. In Python 10^9 is written as `1e9`.

We can see that the population growth goes through distinct phases. First, there seems to be little growth, then the population grows very fast after which it reaches a plateau, and after

some time without growth it declines again. Before we discuss these phases in more detail, we will improve our plot so we can obtain more information from it.

The problem with the current plot is that because when the population reaches its maximum size, those numbers are very big (into the billions), which makes it difficult to see patterns for the datapoints with the smaller population sizes. In the next section, we introduce a method to make the details at all scales equally visible.

3.2.2 Plotting with a logarithmic scale

Our bacteria undergo binary fission, which means one bacterium become 2, 2 bacteria become 4, and so on:

$$1, \quad 2, \quad 4, \quad 8, \dots \quad (3.2)$$

This process is illustrated in Figure 3.4.

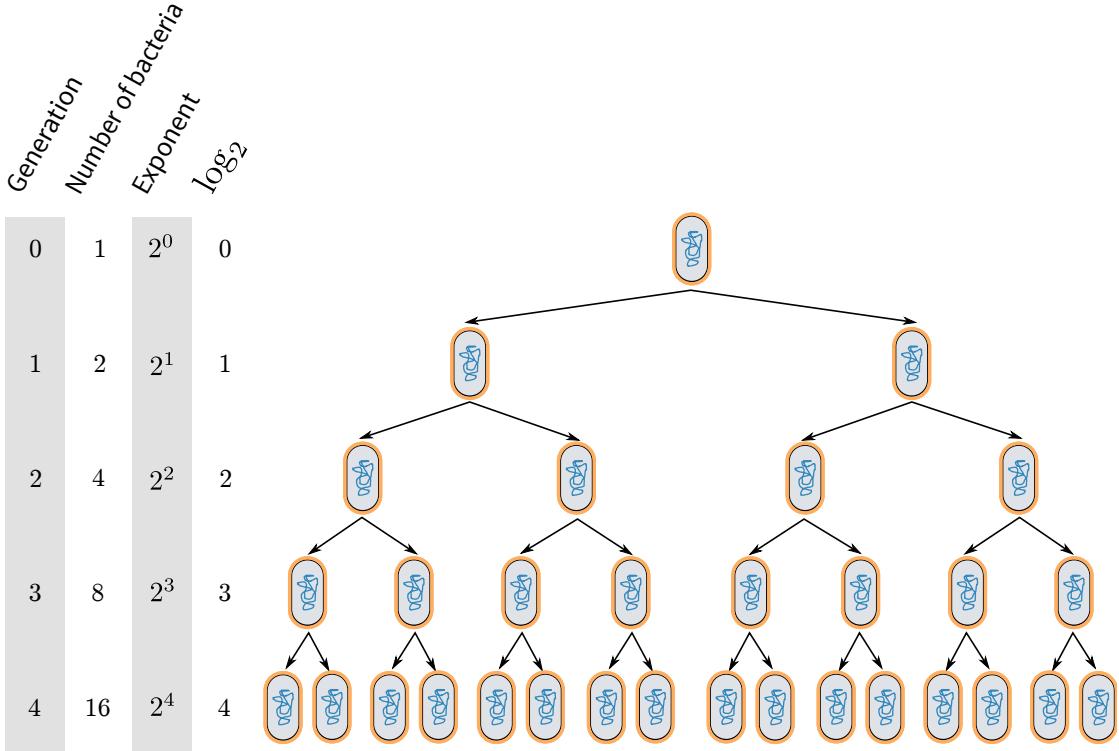


Figure 3.4: Binary fission leads to an exponential relation between the generations.

When we, in the exponential growth phase, go from one generation to the next, the number of bacteria doubles. We can thus write the number of bacteria as an exponent of two,

$$2^0 = 1, \quad 2^1 = 2, \quad 2^2 = 4, \quad 2^3 = 8, \dots \quad (3.3)$$

Now we will look at *logarithms*. The base 2 logarithm of a number is defined by the equation

$$\log_2(2^n) = n \quad (3.4)$$

This might look a bit cryptic. To find the base 2 logarithm of a number, you must ask yourself one question. “What power must I raise 2 to, in order for the result to be my number?” What is the base 2 logarithm of 8? Well, to get 8, I must raise 2 to the power 3, because $2^3 = 8$. So the base 2 logarithm of 8 is 3.

If we take \log_2 of the numbers of bacteria after each doubling, we get:

$$\log_2(2^0) = 0, \quad \log_2(2^1) = 1, \quad \log_2(2^2) = 2, \quad \log_2(2^3) = 3, \dots \quad (3.5)$$

When the number of bacteria doubles, the logarithm of the number of bacteria only increases by one, as you see above. We can use this effect to reduce the difference between large and small numbers in our plot. Instead of plotting the actual number of bacteria we plot the logarithm of the number of bacteria. Such a scale is called a *logarithmic scale* and is common to use in cases where the numbers being plotted span a large range. We do not need to use logarithms of base 2, we can use any base we would like, such as base 10, and still get this effect. In the rest of this chapter we are going to use \log_{10} . Doing this we plot the *logarithm* of the number of bacteria over time, instead of the number of bacteria over time.

The resulting graph is a *semilogarithmic* graph since only one of the axes has logarithmic scaling (the y -axis). A comparison of linear and logarithmic scaling is shown in Figure 3.6. Notice how the starting point and endpoint are the same on both lines, but the numbers in between are differently spaced. Small numbers take up much more space in the logarithmic scale, compared to a regular scale, making it possible to show differences in both large and small numbers at the same time.

To plot our data with logarithmic scale, we add the following command after the `plot()` command:

```
yscale("log")
```

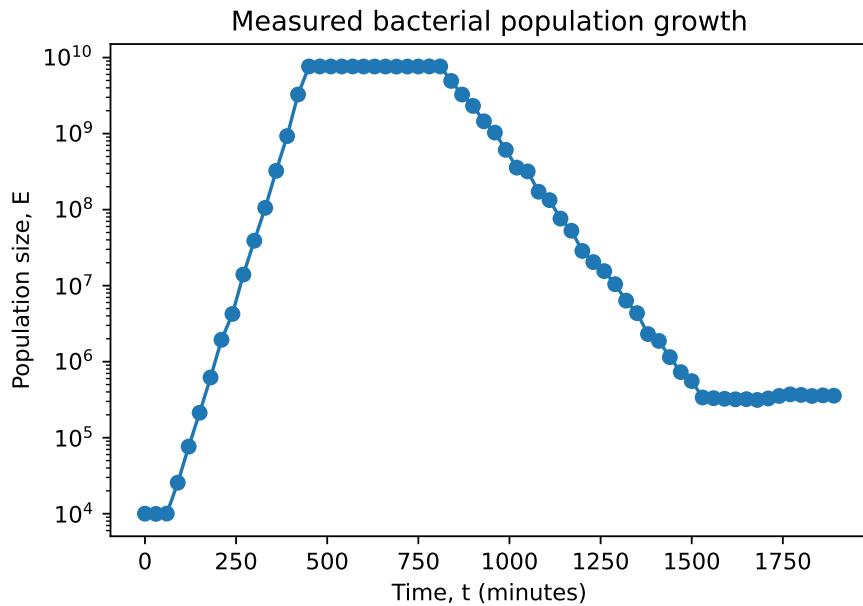
This line makes the scaling of the y -axis logarithmic (with respect to base 10).

Let us use this to create a new plot:

```

1 import pandas
2 from pylab import *
3
4 data = pandas.read_csv("ecoli_all.csv")
5 t = list(data["t"])
6 E = list(data["E"])
7
8 plot(t, E, "o-")
9 yscale("log") # New
10 xlabel("Time, t (minutes)")
11 ylabel("Population size, E")
12 title("Measured bacterial population growth")
13 show()

```

Figure 3.5: Bacterial population growth, all phases with logarithmic y -axis.

You can identify the plot as a semilogarithmic plot by looking at the *minor tick marks*, the little lines to the left, of the y axis. In a regularly scaled plot, these tick marks would be regularly spaced, with the same distance between them. In a (semi)logarithmic plot, the *minor tick marks* have different distances, with the largest distance between the smallest numbers, while the *major tick marks* are evenly spaced.

Note that we could also take the logarithm of all the values in E and make a plot of those values, the resulting graph will have the exact same shape as the Figure 3.5. The only change is that the values on the y -axis will be in range 4-10 instead of $10^4 - 10^{10}$.

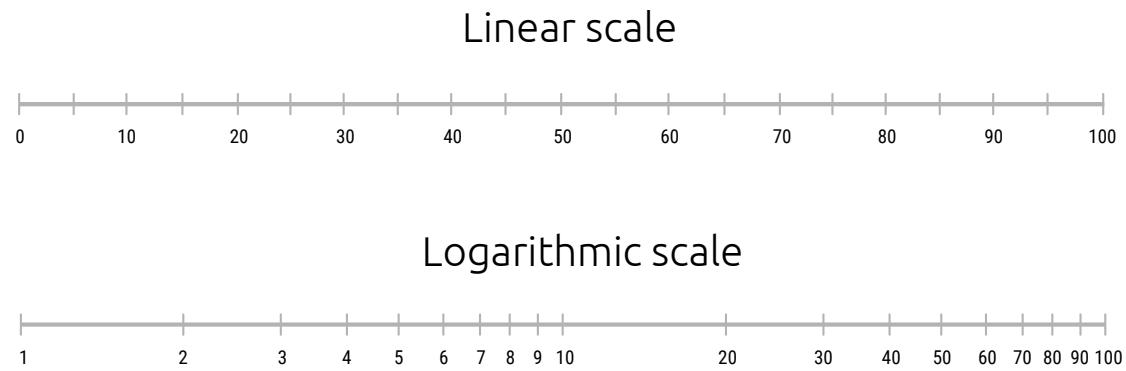


Figure 3.6: Comparison of linear and logarithmic scale.

3.2.3 Examining the different growth phases

As noted earlier, the bacterial population growth is going through different distinct phases. We have marked all the phases of the population growth in Figure 3.7. We will now describe each of them.

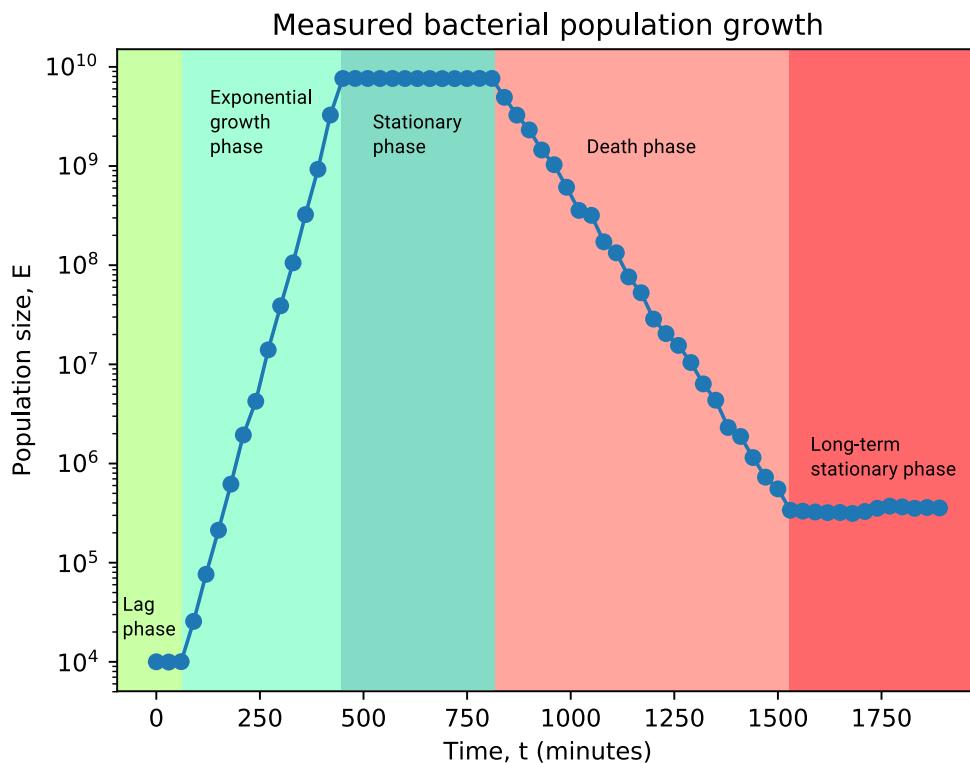


Figure 3.7: All phases of bacterial population growth illustrated on a logarithmic plot of the experimental growth data.

The lag phase. Focusing on the earliest timepoints, and comparing Figure 3.3 with the semilogarithmic plot in Figure 3.7, we see that although the plot with normal axes seems to show no growth for at least the first 250 minutes, the semilogarithmic plot indicates that growth actually starts after the third timepoint, which is 60 minutes. This indicates that the bacterial population is more or less fixed at the beginning, before it starts to grow more and more rapidly. This delay is due to the *lag phase*.

When microorganisms are introduced into a fresh medium, population growth usually begins only after a period of time has passed. In our case, this period lasts for about one hour before the bacteria start to divide. During this period there are some fluctuations in the population number, a few bacteria die and a few bacteria are born, but generally the bacteria do not reproduce. The duration of this phase depends on the history of the bacterial culture and the conditions for

growth. During the lag phase the cells synthesize components, such as enzymes needed to use different nutrients in the new environment. The microorganisms may also have been injured when introduced to the new medium, and use the lag phase to recover.

The exponential growth phase. Once the bacteria start growing, they do so very rapidly. Let us have a look at the first timepoints of this phase by generating a plot showing only the time interval from 60 to 240 minutes:

```
plot(t[2:9], E[2:9], "o-")
xlabel("Time, t (minutes)")
ylabel("Population size, E")
title("Measured bacterial population growth")
show()
```

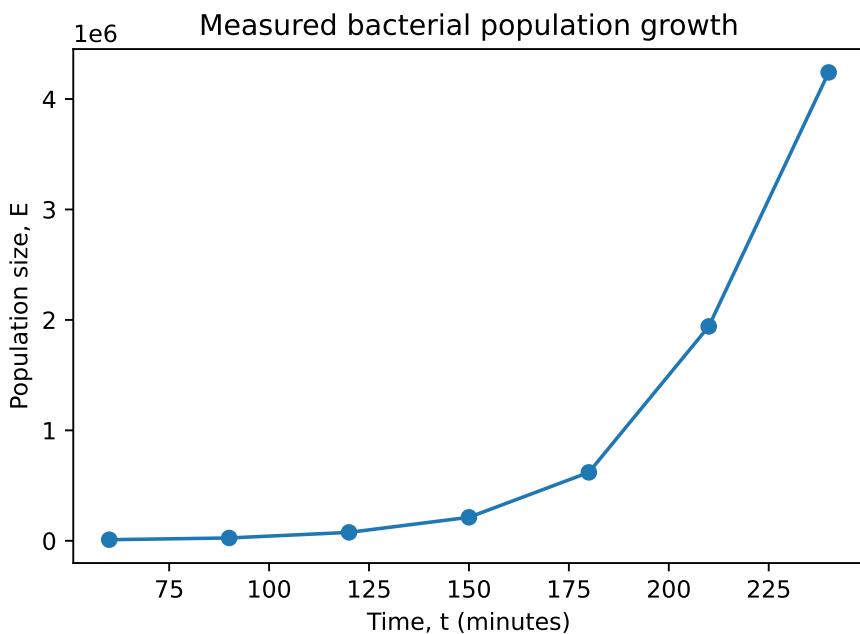


Figure 3.8: A plot of the exponential growth part of bacterial population growth data.

Look at the plot in Figure 3.8, and try to determine the time it takes before the population is doubled from 1 million to 2 million. Now try again from 2 million to 3 million. We see that these times are more or less the same (around 20 minutes) no matter which two points we choose. This type of population growth, where the number of individuals doubles after a fixed time, is called *exponential growth*. The time it takes for the population to double in size is called the *doubling time*. This property can be estimated from the plot and we will do that later in this chapter.



Difference between doubling time and generation time

Doubling time is the time it takes before the population is doubled. Generation time is the (average) time between the divisions of cells. In the case of binary fission, where there is no mortality among the bacteria, the doubling time and the generation time are equal, but in most cases the doubling time and generation time are different.

You are often going to encounter this type of curve and being able to recognize that it is exponential growth gives you much information about the phenomena. Importantly, when plotting exponential growth on a semilogarithmic scale, it will show as a straight line:

```
plot(t[2:9], E[2:9], "o-")
yscale("log")
xlabel("Time, t (minutes)")
ylabel("Population size, E")
title("Measured bacterial population growth")
show()
```

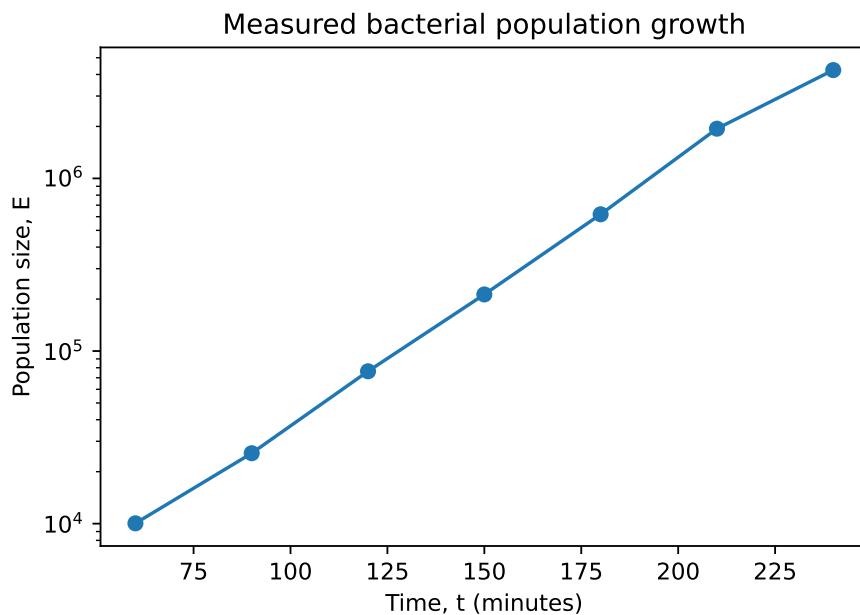


Figure 3.9: A semilogarithmic plot of the exponential growth part of bacterial population growth data, showing the growth as a straight line.

If the population growth is not limited in any way, the bacterial population continues to grow forever with an ever increasing rate.

The stationary phase. As expected, the exponential growth cannot be unlimited. From Figure 3.7 we see that the population growth flattens after some time. The population growth is limited by:

1. the fact that essential nutrients have been exhausted,
2. accumulated waste products which inhibit further growth.

At this point, the population reaches the *stationary phase*, where there is no net increase or decrease in the number of bacteria. Although there is no population growth, binary fission still occurs, but it is balanced by the number of bacteria that die.

The amount of nutrition available in the environment decreases and the amount of waste products increases with time. Due to this, after a while the number of bacteria that die becomes greater than the number of bacteria added through binary fission. When this happens, the population enters the next phase.

The death phase. This phase is called the *death phase* and in it, the number of bacteria begins to decrease. From Figure 3.3 we see that the shape of the curve in the death phase is similar to the shape of the curve in the exponential growth phase. In the death phase, the population is exponentially *decreasing*.

The long-term stationary phase. In the long term, some of the bacteria may survive and keep the population size more or less fixed. This is called the *long-term stationary phase*.

Comparing the plot with normal axes, Figure 3.3, with the semilogarithmic plot, Figure 3.5, shows some interesting differences. In Figure 3.3 we see that the numbers on the y -axis span a large range. Our starting population is $10\ 010 = 1.001 \times 10^4$, while the maximum number of bacteria is more than $10^9 = 1\ 000\ 000\ 000$. The smaller values are almost indistinguishable in the bottom of the graph. It is impossible to see the starting and final population size from the graph without zooming in, and we cannot see how long the lag phase lasts. Neither can we tell if we have more or fewer bacteria in the long term stationary phase compared to the lag phase. Figure 3.5 has some important advantages. It is much easier to tell when the population goes through the different phases in the growth cycle. Furthermore, we see that the number of bacteria in the long term stationary phase is higher than the starting population. This was not possible to observe by looking at Figure 3.3. Another very interesting observation is the linearity in the semilogarithmic plot during the exponential growth phase and death phase. The graph in these phases is a straight line which increases and decreases, respectively. This is in contrast to the plot with normal axes, where the population size increased more and more rapidly. This observation is a very important property of exponentially increasing (or decreasing) functions: an exponential function is a straight line in a semilogarithmic graph.

Comparing the slope (steepness) for the exponential growth phase with the slope for the death phase, we see that the latter has a smaller slope (in absolute values). In other words, the line in the growth phase rises faster than the line in the death phase declines. This means the growth in the exponential growth phase happens faster than the decline in the death phase. In the next section, we will learn how we can quantify this growth.

3.3 Examining the growth rate

In the previous section, we saw that plotting the bacterial growth data with logarithmic scaling immediately gave a more informative plot. We now use this representation of the data to extract even more information about the population growth. Earlier in this chapter, we used a graph (Figure 3.8) to estimate the doubling time. We did that by reading the time it took to double the number of cells from the plot. However, if we wish to have a more rigid estimate of the doubling time, reading values from an exponentially growing plot is not the best method. It can be done much more precisely using the semilogarithmic plot.

3.3.1 Exponential growth with logarithmic scale

In this section we find the relation between the so-called *growth rate* and the doubling time for a population.

We have seen earlier that if the bacterial population grows exponentially, and this growth is visualised using a semilogarithmic plot, it will show as a straight line, see Figure 3.5. We can use this fact together with the mathematical definition of a line that we saw in Section 2.5 to find the growth rate, which is a measure for how fast the bacterial population grows, as well as the doubling time.

Remember that we defined a straight line, or linear function, mathematically as

$$y = at + b, \quad (3.6)$$

where a is the slope (steepness) of the line and b is where the curve intercepts the y -axis. We determine a and b by taking two points on the graph and divide the change in y by change in t :

$$a = \frac{y_2 - y_1}{t_2 - t_1}. \quad (3.7)$$

Our bacterial population growth also shows a straight line, however, only after when plotted as a semilogarithmic plot, or after taking the logarithm of all the numbers of bacteria and plotting those.

We thus have to take this into account when we use the linear function for the growth of our bacterial population. We now rewrite the mathematical definition of a straight line to describe the line in our plot of exponential growth to:

$$\log_{10}(E) = at + b, \quad (3.8)$$

i.e. the logarithm of the number of bacteria, $\log_{10}(E)$, is given as a linear function of time t . All we have done here is to set $y = \log_{10}(E)$. We are mainly interested in the slope of this line, a , which is related to the *growth rate*. As we are taking the logarithm of the number of bacteria, a in our formula is the *logarithm* of the growth rate. A steeper slope means that the population is doubling at a faster rate, which means a shorter doubling time. It is therefore reasonable to

assume that the (logarithm of the) growth rate a is closely related to the doubling time. Let us use mathematics to derive the details of the relation between doubling time and growth rate. In order to do so, we must remind ourselves of some of the rules of logarithms.



Logarithm rules

For any two positive numbers x and y , we have the following relations:

$$\log(xy) = \log(x) + \log(y) \quad (3.9)$$

$$\log(x/y) = \log(x) - \log(y) \quad (3.10)$$

Now, notice that if we have the number of bacteria, E_1 and E_2 , at two different times, t_1 and t_2 . We then have the following relations:

$$\log_{10}(E_1) = at_1 + b, \quad (3.11)$$

$$\log_{10}(E_2) = at_2 + b. \quad (3.12)$$

We subtract the first equation from the second to get

$$\begin{aligned} \log_{10}(E_2) - \log_{10}(E_1) &= at_2 + b - (at_1 + b) \\ &= at_2 - at_1 \\ &= a(t_2 - t_1). \end{aligned} \quad (3.13)$$

This removes b from our equations. If we rearrange the terms we get

$$\begin{aligned} a(t_2 - t_1) &= \log_{10}(E_2) - \log_{10}(E_1), \\ a &= \frac{\log_{10}(E_2) - \log_{10}(E_1)}{t_2 - t_1}. \end{aligned} \quad (3.14)$$

This allows us to find a , the logarithm of the growth rate. We also want to relate the growth rate to the doubling time. To do this we first use that $\log(x) - \log(y) = \log(x/y)$ so that we get this equation for the growth rate:

$$a = \frac{\log_{10}(E_2/E_1)}{t_2 - t_1}. \quad (3.15)$$

Then we choose points such that E_2 it is twice as large as E_1 , i.e. $E_2 = 2E_1$. In that case, the time difference, $t_2 - t_1$, becomes the doubling time, which we will denote as d . If we insert this

into the final equation we get

$$\begin{aligned} a &= \frac{\log_{10}(2E_1/E_1)}{t_2 - t_1} \\ &= \frac{\log_{10}(2E_1/E_1)}{d} \\ &= \frac{\log_{10}(2)}{d}. \end{aligned} \quad (3.16)$$

We now express the doubling time d in terms of a . The doubling time of the bacterial population is given in terms of a as

$$d = \frac{\log_{10}(2)}{a}, \quad (3.17)$$

where a is the logarithm of the growth rate. Note that the unit of the doubling time is time, in our case minutes. We thus find the doubling time d for a bacterial population by finding the slope of the semilogarithmic graph. Since a is the logarithm of the growth rate, and we used \log_{10} to calculate it, we find the growth rate by taking ten and raise it to the power of a , or 10^a .

3.3.2 Calculating growth rate and doubling time from data

In the previous section we derived the relation between the growth rate and doubling time of a population. We repeat the relation here:

$$d = \frac{\log_{10}(2)}{a}, \quad (3.18)$$

where a is the logarithm of the growth rate, and d is the doubling time. The growth rate is directly related to the slope of the line in the exponential growth phase in Figure 3.5. There are many methods we can use to find this slope. Here we show a simple method, by choosing the first and last timepoints of the exponential growth phase. Looking at Figure 3.5 we see that these correspond to the timepoints 60 and 450 minutes, which are the elements with index 2 and 15 in E (the first two elements in E are measurements from the lag phase). Using (3.15), we then take the logarithm of these two numbers using the \log_{10} function, and divide their difference by the time difference. To calculate the growth rate, we then raise a to the power 10:

```
E_1 = E[2]
E_2 = E[15]
t_1 = t[2]
t_2 = t[15]

a = (log10(E_2) - log10(E_1)) / (t_2 - t_1)
print("Growth rate:", 10**a, "per minute")
```

Growth rate: 1.0353327430060728 per minute

To find the doubling time, we use Equation (3.17):

```
d = log10(2)/a
print("Doubling time:", d, "minutes")
```

```
| Doubling time: 19.96226875420508 minutes
```

As you see from the output, the doubling time for our dataset is approximately 20 minutes, which is very close to the generation time of *E. coli* in optimal conditions. Based on our calculations the growth rate is approximately 1.035, or slightly above 1. Note that the unit of the growth rate is per unit time, ($1/\text{time}$), in our case ($1/\text{minute}$). A growth rate of 1.035 means that each minute, the amount of bacteria is multiplied by 1.035 times the amount a minute earlier. This is why the growth rate as determined here is the *multiplicative growth rate*.

We can now check the relationship between the multiplicative growth rate and doubling time by considering that after 20 minutes, our population should have doubled in size. Let us do this for a hypothetical population of B bacteria. After one minute, our population will have multiplied by a factor equal to the multiplicative growth rate, i.e., it will be $B \times 1.035$. After two minutes, it will have multiplied with 1.035 again: $B \times 1.035 \times 1.035$ or $B \times 1.035^2$. If we continue this for t minutes, our population will have multiplied to $B \times 1.035^t$ its original size. After one doubling time, our population will have multiplied to $B \times 1.035^{20}$, which is $B \times 1.99$, in other words, very close to $B \times 2$.

The estimates of the multiplicative growth rate and doubling time above are obtained by a very simple method. Better (and more complicated) methods exist, but these are outside the scope of this book.

3.4 Exponential growth: A simple model for bacterial population growth

The goal of the rest of this chapter is to simulate all phases of the bacterial population growth. To create a simulation, we first need to make a *mathematical model* (most often just called a model) of what happens with the bacteria over time. Since our model describes changes (of the bacterial population) with time, it will be a *dynamic model*. The model is the set of rules that we make to describe the phenomenon, while a simulation is what we obtain when we apply those rules to a specific case. We create the model by writing mathematical equations based on what we know of bacterial population growth. For instance, we know that with a constant growth rate, the growth of the bacterial population is proportional to its size. This can be written as an equation, which can be turned into a *computational model* by studying the equation using programming. This is called to *implement* the model. The commands used in the program define a recipe, or *algorithm*, that is followed by the computer.

Mathematical and computational models are not restricted to bacterial population growth. Dynamic models can be created to study other systems, such as plant population growth, disease spread, the weather, or even the orbits of planets. Computational models are today widely used in many disciplines, including, but not limited to, mathematics, medicine, physics, chemistry, biology, engineering, and economy. Simulations are used to predict the future, such as in weather forecasting, or to perform virtual experiments. Among the benefits of using simulations instead

of real experiments include saved costs, saved time, and reduced risk. If the experiment is expensive or time consuming, we can try out many different configurations with simulations to see what works best before performing the real experiment with the best configuration. We can even simulate experiments that can not be performed at all in the real world.



Example of a simple model

In Chapter 1, we discussed the equation used to calculate the maximum heart rate, HR_{\max} :

$$HR_{\max} = 220 - \text{age}. \quad (3.19)$$

This is a model, although it is both simple and ignores many details. A more accurate model could include personal physiology and fitness, but the simple model is still useful.

When we create a model, it is often best to start out simple and then add additional details after we know the simple model works as intended. Building a model up piece by piece makes it easier to avoid mistakes. It also allows us to study the effects of each additional factor by comparing the new results to the results of the less detailed model. In this chapter, we start with simple population growth of the population of bacteria. We deliberately ignore many factors that would normally influence this growth, such as interactions with other species or the effect of temperature. It is a common misconception that models have to perfectly reproduce real data to be useful. In fact, simplified models are often the most useful ones because they are easier to understand, and it is easier to attribute the behavior of the model to a specific cause. As mentioned in Section 3.2.3, the bacterial population growth cycle consists of lag phase, exponential growth phase, stationary phase, death phase, and long-term stationary phase. The lag phase is the simplest to model, but not very interesting from a population dynamics view because the population size is more or less constant. We therefore skip this phase for now and return to it in the next chapter. In this chapter we model the exponential growth phase.

At the beginning of the chapter, we described how one bacterial cell divides into two daughter cells. The (average) time between the divisions of cells is called the generation time. This suggests a model where the population doubles after a fixed time. To make the model as simple as possible, we assume that there are no deaths. In this case, the doubling time is equal to the generation time. After each generation has passed, the bacterial population is twice as large. In the next chapter, we tackle the more realistic case of limited growth.

We use the microscopic view of a bacteria splitting in two as inspiration for making a macroscopic model of the entire population. When thinking about this process, it is perfectly fine to have the mental image of one bacteria splitting into two, which then split into four, and so on. If you try to include all the details at once, you will easily lose track of your thoughts, and you will not get very far. It is better to think about the simple picture until you have understood that, and then take a step back and reflect on what errors you might have made by choosing such a simple model, and that is exactly what we will do in this chapter.

Before implementing a model, it is always a good idea to state its *assumptions*:



Model assumptions for the exponential growth phase

We assume that:

- every bacterium is identical,
- every bacterium splits in two every generation, and
- there are no deaths.

A consequence of these assumptions is that the doubling time is equal to the generation time. For the rest of this chapter we use “doubling time” and “generation time” interchangeably.

In order to make it easier to use a computer program for our calculations, we need to work with discrete time steps. This means that we only perform calculations at certain times, and take a step between each point in time. The time between two consecutive calculations is called the *time step*.

We use the symbol N for the total number of time steps we want to take in our simulation including the initial time step, while the symbol n is used for the current time step. To represent the number of bacteria, we use the symbol E . The number of bacteria E at time step n is written as E_n . The subscript n means the value of E at time step n . The number of bacteria at the previous time step is E_{n-1} , two time steps in the past is E_{n-2} , and so on. In the same way, the number of bacteria in the next time step is E_{n+1} , the next after that is E_{n+2} , and so on (see Figure 3.10). Note that we start indexing from zero, meaning that the initial number of bacteria is E_0 , and the final population after N generations is E_{N-1} .

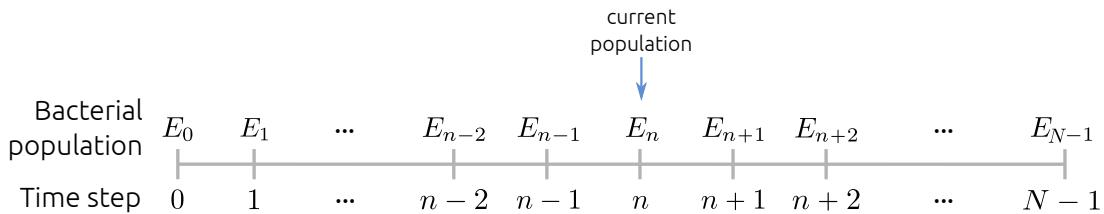


Figure 3.10: A timeline that shows the indexing we use on E .

If we were to define the time step to be equal to the generation time, the number of bacteria at time step n would be two times the number of bacteria at the time step $n - 1$. However, we usually work with more realistic time steps, such as days, hours or minutes. In the case of our bacterial population, we will use 1 minute as the time interval. We then have to consider how much our population grows at each time interval.

We know that every cell reproduces exactly once per generation, dividing into two cells. When considering intervals shorter than the generation time we need to assume that the timing of reproduction varies among individuals within the generation - not all cells divide at the same time. In other words, we assume that a certain constant fraction of the bacterial cells divide

each time step. Note that this means that in our model, the population growth is proportional to the population size.

We will start with the simple formulation that describes the growth of the population between two consecutive time steps. We can express the growth of the population from the previous time step $n - 1$ to the current time step n as $E_n - E_{n-1}$. This change should then be a fixed fraction of the population size at $n - 1$, which is E_{n-1} . We will call this fraction r . We thus calculate r as:

$$r = \frac{E_n - E_{n-1}}{E_{n-1}}. \quad (3.20)$$

r is here the *relative growth rate*, as it represents the mean number of new bacteria produced per bacterium in the time interval.

We are interested in the equation that tells us how to calculate the population size for the current time step given the population size of the previous time step, in other words, solve the equation for E_n . We do this by rearranging the terms:

$$E_n - E_{n-1} = rE_{n-1} \quad (3.21)$$

$$E_n = E_{n-1} + rE_{n-1}. \quad (3.22)$$

The final equation states that the size of the population at the current time step is composed of the size at the previous time step *plus* the fraction of the bacterial cells that divided in the time interval.

We can simplify the equation to:

$$E_n = (1 + r)E_{n-1}. \quad (3.23)$$

It makes sense that the value of r , the relative growth rate, is related to the value of a , the logarithm of the multiplicative growth rate, that we calculated in Section 3.3. To find this relationship, we need to consider Equation (3.15):

$$a = \frac{\log_{10}(E_2/E_1)}{t_2 - t_1}. \quad (3.24)$$

When we generalise this from timepoints 1 and 2 to $n - 1$ and n , we get:

$$a = \frac{\log_{10}(E_n/E_{n-1})}{t_n - t_{n-1}}. \quad (3.25)$$

Since we work with time steps of 1 minute, we know that $t_n - t_{n-1}$ is 1 minute. This simplifies the equation to:

$$a = \log_{10}(E_n/E_{n-1}). \quad (3.26)$$

To get rid of the logarithm on the right side, we use that $10^{\log_{10}(x)} = x$. We raise 10 to the power of each side of the equation, and get:

$$10^a = E_n / E_{n-1}. \quad (3.27)$$

Solving for E_n gives us:

$$E_n = 10^a \times E_{n-1}. \quad (3.28)$$

Comparing this equation to Equation (3.23), we see that

$$10^a = 1 + r \quad (3.29)$$

or

$$r = 10^a - 1 \quad (3.30)$$

This means that we can derive the value of the growth rate r from the data by calculating a as described in Section 3.3, raise 10 to the power of a and subtract 1. We also see that the relative growth rate is the multiplicative growth rate minus 1.

We derived the value of the 10^a for our dataset of population growth at 39°C in Section 3.3 to be 1.035. We will thus set $r = 1.035 - 1 = 0.035$.

We said earlier that after one minute, the amount of bacteria is multiplied by 1.035 times the amount a minute earlier. This means that the population increases by a fraction 0.035, or 3.5%, of its original size. This again illustrates the relationship between absolute and relative growth rate.

If we insert $n = 1$ into this equation, we see that it requires us to define a starting value. We need the number of bacteria in time step $n = 0$ to find the number of bacteria in time step $n = 1$, that is, we need to know E_0 before we can calculate E_1 :

$$E_1 = (1 + r) \times E_0. \quad (3.31)$$

This starting value is called the *initial condition*. Most models need an initial condition before we can begin calculations for the next time steps. From Section 3.3, we know that the exponential growth phase started at timepoints 60 minutes, at which point the population size was 10 042 bacteria. Thus, we use this as the initial condition in our model and set $E_0 = 10 042$.



Summary: Exponential growth model

The exponential growth model for the growth of *E. coli* can be summarized as:

$$E_n = (1 + r) \times E_{n-1}, \quad (3.32)$$

with the growth rate r

$$r = 0.035 \quad (3.33)$$

and with the initial condition

$$E_0 = 10\,042. \quad (3.34)$$

A summary of the different symbols we use in our model and their meaning are listed in the table below:

Symbol	Meaning
n	Current time step
N	Number of time steps
E	Number of bacteria
E_n	Number of bacteria at a given time step n
E_0	Initial condition, number of bacteria we start with
E_{N-1}	Number of bacteria at time step $N - 1$ (last time step)
r	Growth rate

We now have all the information needed to implement the model in Python and to perform the calculations of the population size over time. Remember that we calculated a doubling time of 20 minutes. If our model is correct, we should thus see the population size double around 20 minutes. Let us start by calculating the number of bacteria for the first five minutes. This means that we start with the original population, and let it grow for four minutes. In order to avoid mistakes, we use Python to help us with the calculations:

```
r = 0.035
E0 = 10042
print("E0:", E0)
E1 = (1 + r) * E0
print("E1:", E1)
E2 = (1 + r) * E1
print("E2:", E2)
E3 = (1 + r) * E2
print("E3:", E3)
E4 = (1 + r) * E3
print("E4:", E4)
```

```
E0: 10042
E1: 10393.47
E2: 10757.241449999998
E3: 11133.744900749996
```

E4: 11523.425972276245

We see that our population indeed is slowly growing between the time steps.

Note how the number of bacteria is not a whole number of bacteria, but has an increasing number of digits following the decimal point. This is caused by the fact that we multiply by 1 plus a number smaller than 1, which will give us fractional numbers. Biologically, it does not make sense to have a fraction of bacteria. Perhaps we should have rounded off the number of bacteria for each time step. However, even though the difference *per time step* would be small, after many time steps these differences accumulate and would make the final number wrong. So when we do the modeling, we choose to keep these numbers as fractions.

Population sizes are by definition whole numbers (integers), but they are usually modelled as continuous variables (floats), since this is a convenient assumption for modeling that usually does not lead to any problems.

We could continue our process to calculate the number of bacteria for all time steps up to 21 minutes, or any number of time steps for that matter, but that quickly becomes cumbersome. The next step is therefore to implement this model in a better way. First, making a new variable for each generation is unnecessary; we have already learned that lists can be used when we have large amounts of data. Second, we are doing repeated calculations and we should automate these. We thus invoke “the rule of three” and will learn how to automate repeating steps with *loops* in Python.



Rule of three

In programming, there is a rule of thumb called “the rule of three”. It states that if a task has to be repeated three times or more, we should automate it. This is for two reasons: First, repeating the task is time-consuming, and at some point we will save time by writing a program to do the task for us. Furthermore, when humans perform repeated boring tasks, we are prone to error. A computer never makes an error by accident by itself.

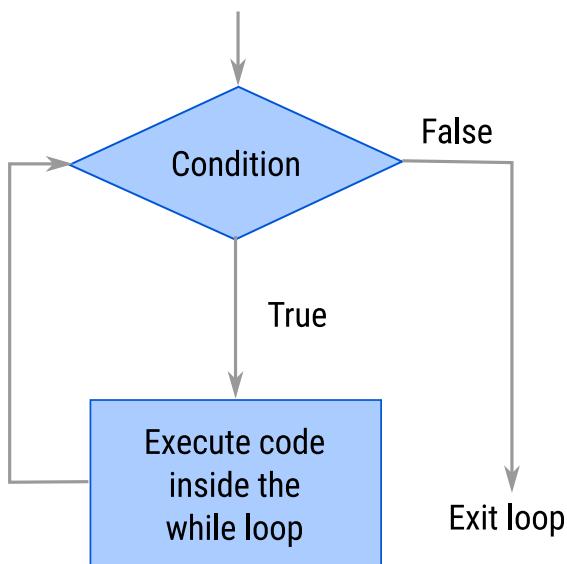
This rule also applies inside a program itself: If we copy the same lines to three or more places, we should find a more general way of writing the program. You will learn ways of doing this in later chapters.

3.5 While loops: Repeating a piece of code many times

When we have stored data in a list we often want to perform a specific task for each element in that list. In programming, this can be done with *loops*. A loop repeats one or more tasks and is an extremely useful concept in programming.

In Python there are two different variants: `while` loops and `for` loops. In this chapter, we address `while` loops, and we get back to `for` loops in Section 7.2.2.

A `while` loop repeats a set of statements as long as a specific condition is met (see Figure 3.11).

Figure 3.11: Flowchart of a `while` loop.

Let us illustrate `while` loops with an example. In this example, the loop continues as long as the value assigned to the variable `my_number` is smaller than 5.

```

1 my_number = 0
2 while my_number < 5:
3     print("Inside the loop, my_number is now", my_number)
4     my_number = my_number + 1
5
6 print("After the loop, my_number is", my_number)
  
```

```

Inside the loop, my_number is now 0
Inside the loop, my_number is now 1
Inside the loop, my_number is now 2
Inside the loop, my_number is now 3
Inside the loop, my_number is now 4
After the loop, my_number is 5
  
```

3.5.1 Understanding the `while` loop

To understand the flow of the above program we go through the code in detail. Note that, if you are reading this in a notebook, you can add line numbering to the cell by first selecting the cell, then choosing 'Toggle Line Numbers' from the 'View' menu.

- First, on line 1, we define the variable `my_number` and assign it the value 0.
- The `while` statement on line 2 includes the *condition* `my_number < 5`, and ends with a `:` (colon).

- The *indented code block* that follows this colon, lines 3 and 4, is run *as long as* the value of `my_number` is less than 5. The use of indented code blocks, also called *Indentation*, is the principle of starting every line with a specific number of *whitespace characters*. You will find that it is often used in Python to indicate code blocks that belong together. Whitespace characters are the invisible characters you create by pressing the **Space** or **Tab** keys on your keyboard.
- The indented code block forms the *body* of the while loop, and the statements in the body:
 - print out the current value of `my_number`
 - *increment* the value of `my_number` with 1: `my_number = my_number + 1`
- The body is executed several times, until the value of `my_number` becomes 5 and thus no longer is less than 5. At that point, the loop ends and Python continues on the first (unindented) line under the loop body, here line 6.
- After the loop ends, the value of `my_number` (which is 5) is printed on line 6.

This is the first time we see an indented block of code, which are (one or more) consecutive lines with the same indentation. We will see more statements in Python that end with a colon and are followed by one or more indented lines.

In Python, you can make an indented block using any number or combinations of white-spaces, but the convention used by programmers is to make an indent using four spaces. Each statement in a block must be indented in exactly the same way, i.e. have the exact same amount of whitespace in front of it. The Jupyter Notebook, but also other programs used to write Python code, give you this indentation automatically when you type the line of a `while` loop that ends with a colon, or press the Tab key. In many editors, you can also select an entire line and hit the Tab key to indent it by 4 spaces. To *unindent* it and remove 4 spaces, hit Shift+Tab.

The statement `my_number = my_number + 1` may seem counterintuitive if you are looking at it from a mathematical point of view: how can `my_number` be the same as `my_number + 1`? But remember, in Python, statements with `=` are assignments: what is on the right side of the `=` is evaluated first and the result is assigned to the variable on the left of it. In this case, the variable `my_number` is part of the calculation on the right side and the value it has been assigned at that point is used, before the result of the evaluation is assigned back to `my_number` itself. For example, if the value of `my_number` is 3, `my_number = my_number + 1` becomes `my_number = 3 + 1` and `my_number` becomes 4.

Finally the `while` statement (the line starting with `while`) contains a *condition*.

In our example the condition is that the value of `my_number` is less than 5. The `while` loop continues as long as this condition is true. This means we have to make sure we do something inside the body of the loop so that eventually, the condition in the `while` statement no longer is correct. Without this, you would get a loop that never ends. This is called an *endless loop*.

3.5.2 Common errors when writing while loops

There are several common pitfalls novice Python users encounter when they use `while` loops. In this section we go through a few of the most common ones.

Forgetting the colon. Many novice Python users forget the colon at the end of the `while` line. This colon marks the end of the `while` statement and the beginning of the indented block of statements inside the loop. If you forget this colon you get a *syntax error*. This is similar to a grammar error when writing English: Python does not understand what you mean. One example of such an error message is:

```
my_number = 0
while my_number < 5
    print("Inside the loop, my_number is now", my_number)
    my_number = my_number + 1
```

```
Input In [112]
    while my_number < 5
        ^
SyntaxError: expected ':'
```

This message tells you that there is a syntax error in your program and where it occurs, in this case is `line 2`. A mistake like this one is often straightforward to correct.

Forgetting the indentation. Another error you might encounter is the *indentation error*. It is very important that each statement in the `while`-block is indented and has the same indentation level. Everything that is not indented will not be part of the loop. For example, the code below has an indentation error since there are no indentation inside the `while` loop:

```
my_number = 0
while my_number < 5:
    print("Inside the loop, my_number is now", my_number)
    my_number = my_number + 1
```

```
Input In [113]
    print("Inside the loop, my_number is now", my_number)
    ^
IndentationError: expected an indented block after 'while' statement on line 2
```

The error tells us that we have forgotten to indent somewhere in our code, in this case on `line 3`.

Forgetting to define variables. We have to define all variables we want to use in the condition part of the `while` loop before we use them. We therefore set `my_number = 0` on the line before the `while` loop in our example. If we forget to do this we get an error:

```
while my_other_number < 5:
    print("Inside the loop, my_number is now", my_other_number)
```

```
my_other_number = my_other_number + 1
```

```
NameError                                 Traceback (most recent call last)
Input In [114], in <cell line: 1>()
----> 1 while my_other_number < 5:
      2     print("Inside the loop, my_number is now", my_other_number)
      3     my_other_number = my_other_number + 1

NameError: name 'my_other_number' is not defined
```

Endless loop. Finally, as mentioned, a common mistake is to forget the part inside the body of the loop that ensures that eventually, the condition following `while` no longer is true, and thus leads to an endless loop.

If you are in a notebook and you have created an endless loop, you see the code cell cell keeps running since there is a [*] symbol to the left of the cell. To stop the cell you can either press the black stop square button in the notebook toolbar or choose the Kernel menu option and then the interrupt option.

3.5.3 Using while loops to loop over elements in a list

We introduced `while` loops because we said we could use them to automate repeated tasks. One such task is doing something with each element in a list. The way this can be done is by writing a `while` loop that goes over all the indexes of our list, starting at 0 and ending with the last index. Remember, the last index is one less than the length of the list. Here is an example:

```
fruits = ["apple", "pear", "cherry", "berry"]

index = 0
while index < len(fruits):
    print(index, fruits[index])
    index = index + 1
```

```
0 apple
1 pear
2 cherry
3 berry
```

The variable `index` is initialized to 0, since we start indexing lists from 0 in Python. The condition tests whether `index` is less than the length of the list. In the example, `len(fruits)` equals 4 so `index` should not be greater than 3. The body prints the element of the list with the current `index`.

3.6 Implementing exponential growth

Let us return to our model and implement the calculations inside a `while` loop, instead of writing a line of code for each time step we want to simulate. We will first investigate if our model indeed gives a doubling of the number of bacteria after 20 minutes, the doubling time. We import the `pylab` package, set the number of time steps we need, and create an empty list that we will fill with elements. In the end, the list will contain N elements. The first element in the list is set to the known initial condition, $E_0 = 10042$. We create an empty list by typing two square brackets with nothing in-between. Then, we add the initial condition using the `.append()` function. This function is specific for lists, and adds a new element to the end. To add the value 10042 to the (empty) list `E`, we write `E.append(10042)`. Note that we use a dot (.) between the list name and the function name. This is because the `.append()` function only works with lists. It is another example of the type of functions that have a dot (.) between the variable we use it with and the function, similar to the `.copy()` function to make a copy of a list we saw in Section 2.3.3.

```
N = 20          # number of time steps (minutes)

E = []          # create empty list
E.append(10042) # initial condition
```

The list now contains one element:

```
print(E)
```

```
[10042]
```

Then we loop over each time step as a sequence of numbers using a `while` loop. We let our condition be `while n < N`, where we still have that $N = 20$ from the above code. This means that when n is 20, we have done 20 time steps already and n should not become bigger. Further, we start the loop from $n = 1$, which means that we skip 0 because we already know the initial condition, $E[0]$. We will also use the growth rate r we earlier determined to be 0.035.

```
r = 0.035 # growth rate

# Calculating number of bacteria for each time step
n = 1
while n < N:
    E_n = (1 + r) * E[n-1]
    E.append(E_n)
    n = n + 1
```

Inside the `while` loop we write the equation for the number of bacteria, $E_n = (1 + r) * E[n-1]$. We then append the result `E_n` to the list `E`. In the first pass of the loop, n is 1, and we use `E[0]` to calculate `E_n`, which is added to the list as the second element, also known as `E[1]`. The value of `E[1]` is therefore $(1 + r) * E[0]$. Finally, we increase n by 1.

There are no more statements in the loop block, so we proceed to the next iteration of the loop. In the second iteration, n is 2, and we calculate and add then next element to the list, which is now $E[2] = (1 + r) * E[1]$. Continuing in the same way, we calculate `E[3]` using `E[2]`,

and $E[4]$ using $E[3]$, and so on. After computing the number of bacteria in the 20th time step, which is when n has the value 20, we increase n one final time. Now, n has the value 21, which is not less than N , and we exit the loop.

Notice how similar the Python syntax $E_n = (1 + r) * E[n-1]$ is to the mathematical version of the equation (3.23): $E_n = (1 + r) \times E_{n-1}$. While E_n refers to the population at the n th time step, similarly, $E[n]$ is the n th element of the list E . This similarity makes it easier to implement our model, while keeping track of the mathematics that define it.

As a final step we print the calculated value for the number of bacteria after 20 minutes by using the fact that we can get the last element of the list E by using $E[-1]$:

```
print("Number of bacteria after", N, "minutes:", E[-1])
```

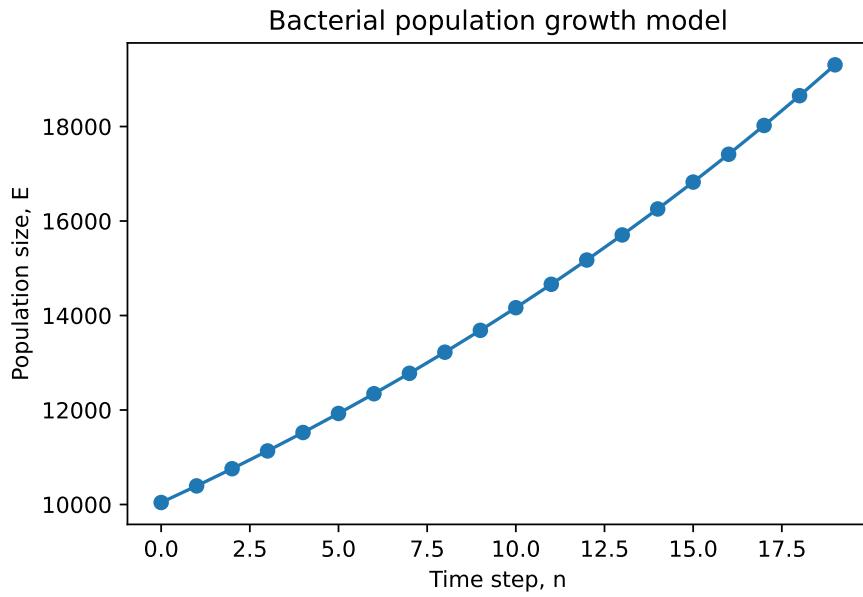
```
| Number of bacteria after 20 minutes: 19305.75822891207
```

Note that we could achieve the same by writing $E[19]$. However, if we later decide to run the model for a different number of minutes, using $E[-1]$ ensures that we always will get the number of bacteria after the last iteration of the end of the model without having to change this line.

The number of bacteria after 20 minutes, the doubling time, is almost double of that we started with. Although we did not precisely double the amount of bacteria, our model is still a good enough approximation as we will see soon. Our results so far above indicate that our new program using a `while` loop behaves as expected. We are now able to calculate the number of bacteria after any number of time steps (minutes), by increasing the number of time steps, N . You can test this for yourself by for example setting $N = 100$.

As we can see from the output of our program, the model we have developed predicts the growth of *E. coli* to be increasing rapidly. To get a better overview of the results, we want to plot the population size. Previously, we have plotted two lists against each other, but we can also make a plot of the population size against its indices (which corresponds to the number of time steps) by passing only the population size list to the `plot()` function. Remember that we need to import the `pylab` package first:

```
from pylab import *
plot(E, "o-")
xlabel("Time step, n")
ylabel("Population size, E")
title("Bacterial population growth model")
show()
```



This modelled bacterial population growth curve seems to be more exponential than linear, but we will need to see more timepoints to be sure.

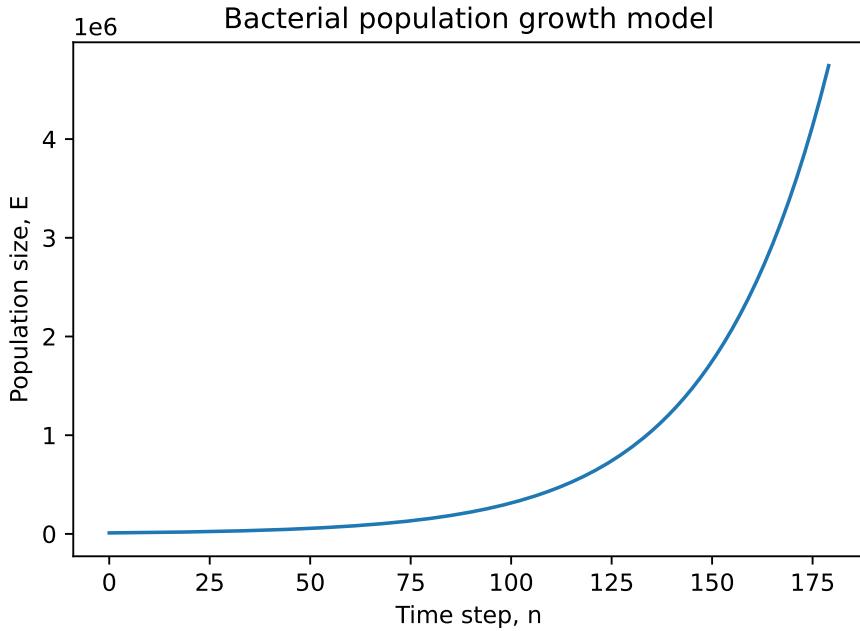
Let us now plot the modeled number of bacteria for the entire interval that we have measurements for. We decided to model the exponential phase which started with 10 042 bacteria. If we look back at the dataset, we see that this corresponds to the timepoint 60 minutes. The final timepoint is 240 minutes, so we should model 180 minutes (three hours).

We change the number of time steps to 180 minutes with $N = 180$ and run our program again. To make the plot more readable, we remove the datapoints and only plot a line, by changing `plot(E, "o-")` to `plot(E, "-")` (which is the same as `plot(E, "-")`). Our program than becomes:

```

1  from pylab import *
2  N = 180          # number of time steps (minutes)
3  r = 0.035 # growth rate
4
5  E = []          # create empty list
6  E.append(10042) # initial condition
7
8  # Calculating number of bacteria for each time step
9  n = 1
10 while n < N:
11     E_n = (1 + r) * E[n-1]
12     E.append(E_n)
13     n = n + 1
14
15 plot(E)
16 xlabel("Time step, n")
17 ylabel("Population size, E")
18 title("Bacterial population growth model")
19 show()

```



This bacterial population growth curve has the same shape as we observed in our measured data, which is promising. However, to see the actual differences we need to compare the modeled data to the measured data.

3.7 Adding time to our model

To be able to compare our model to our measured data, we must work with actual time that has passed, not time steps. The time at step n in our calculations is the time at the previous time step, t_{n-1} , plus the duration of the time step, one minute:

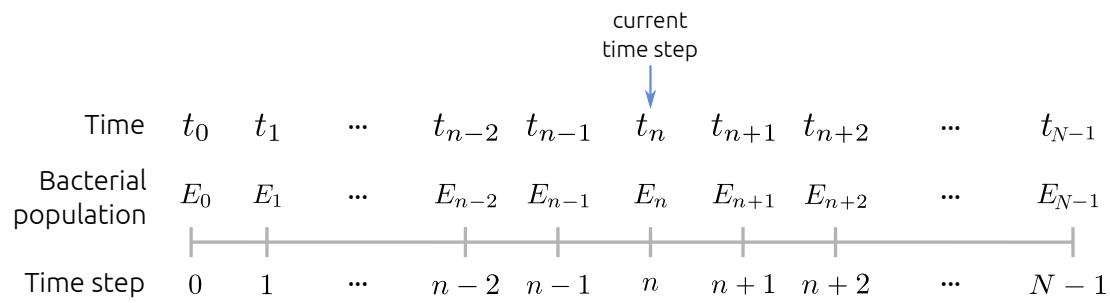
$$t_n = t_{n-1} + 1 \text{ (minute).} \quad (3.35)$$

As with the number of bacteria, E , the subscript n means the value of t at time step n . A timeline that shows this indexing is in Figure 3.12.

As we model the exponential growth phase, which starts at 60 minutes, we thus need to make sure that our model also starts at that timepoint. We thus set the initial condition for the time to be:

$$t_0 = 60 \text{ (minutes).} \quad (3.36)$$

The symbols we use are summarized in the following table:

Figure 3.12: A timeline that shows the indexing we use on t and E .

Symbol	Meaning
n	Current time step
N	Number of time steps
E	Number of bacteria
E_n	Number of bacteria at a given time step n
E_0	Initial condition, number of bacteria we start with
E_{N-1}	Number of bacteria at time step $N - 1$ (last time step)
t	Time
t_n	Time at a given time step n
t_0	Initial condition, time at the start of our simulation
r	Growth rate



Summary of the exponential growth model with time

The exponential growth model for *E. coli* with time can be summarized as:

$$E_n = (1 + r) \times E_{n-1}, \quad (3.37)$$

$$t_n = t_{n-1} + 1 \text{ (minute)}, \quad (3.38)$$

with initial conditions

$$E_0 = 10\,042, \quad (3.39)$$

$$t_0 = 60. \quad (3.40)$$

Let us implement time in our model, which is similar to how we implemented the calculations for E .

We want to again calculate the population size for the entire 4 hour (180 minutes) interval that we have measurements for. We change the number of time steps to 180 minutes with $N = 180$ and run our program again.

The initial time condition is $t[0] = 60$.

```
N = 180      # number of time steps (minutes)
r = 0.035 # growth rate

E = []
t = []

# Initial conditions
E.append(10042)  # initial number of bacteria
t.append(60)      # initial measurement time (in minutes)
```

In the `while` loop we calculate the n th time value, $t[n]$, to be the previous time value plus the time step, $t[n-1] + 1$.

```
# Calculating number of bacteria and time for each generations
n = 1
while n < N:
    E_n = (1 + r) * E[n-1]
    E.append(E_n)

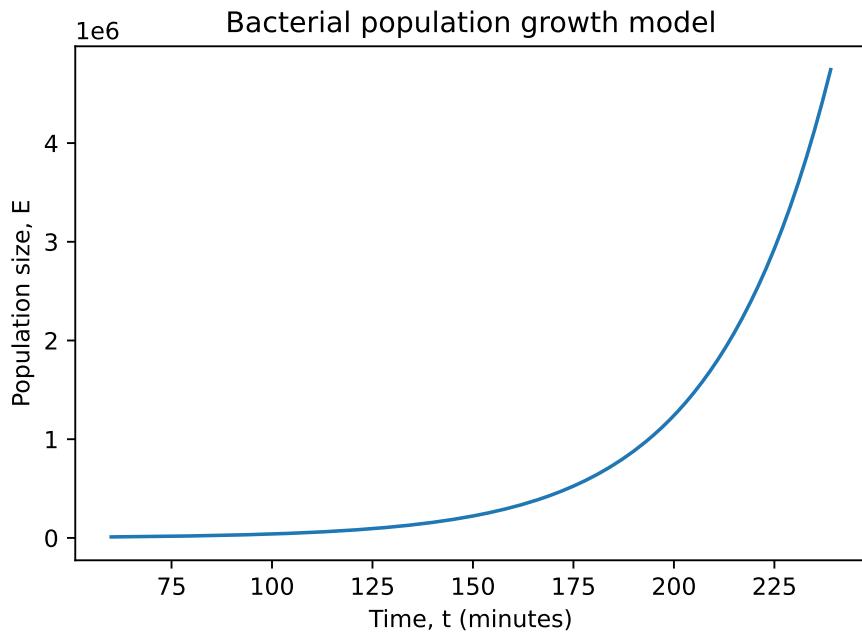
    t_n = t[n-1] +1
    t.append(t_n)

    n = n + 1
```

Finally, we plot the bacterial population against time with `plot(t, E)`, and add labels to each axis:

```
from pylab import *

plot(t, E)
xlabel("Time, t (minutes)")
ylabel("Population size, E")
title("Bacterial population growth model")
show()
```

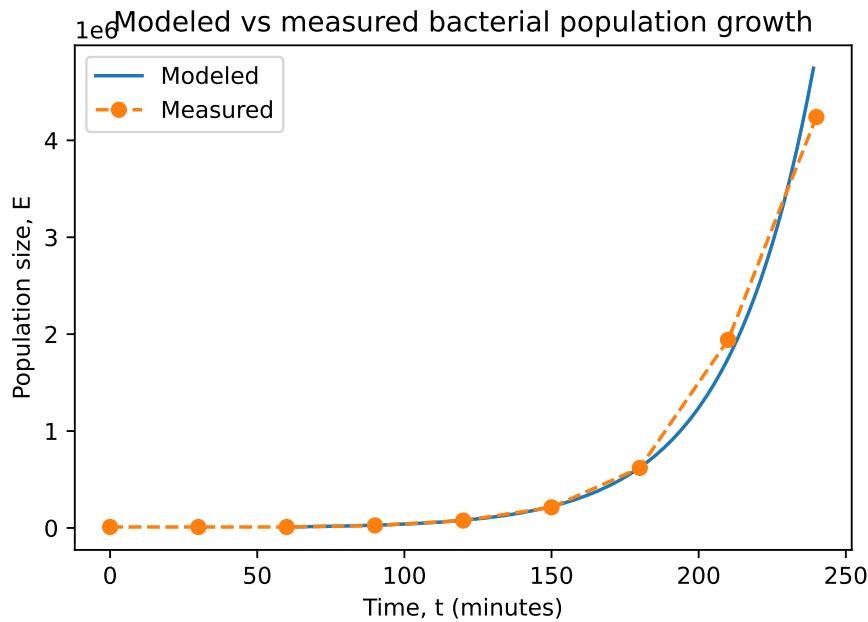


We now have a time value for each time step, which makes it possible to compare our model to experimental data. We compare the two by reading our experimental data from a .csv file, and plot the measured data and the modeled data in the same figure:

```
import pandas

# Reading in experimental values
data = pandas.read_csv("ecoli.csv")
t_measured = list(data["t"])
E_measured = list(data["E"])

# Plotting the modeled and measured data in the same plot
plot(t, E, label="Modeled")
plot(t_measured, E_measured, "--o", label="Measured")
xlabel("Time, t (minutes)")
ylabel("Population size, E")
title("Modeled vs measured bacterial population growth")
legend()
show()
```



We see that the modelled population growth follows the measured growth in the first part of the graph, then deviates a little bit at the end, with simulated numbers being slightly lower or higher than the measured data. This means we have a good enough approximation.

To summarize all we have done, we present the entire program. Note that we start with both `import` statements, rather than add them just before we need the imported code. Adding all necessary `import` statements at the beginning is common practice when writing Python programs.

```

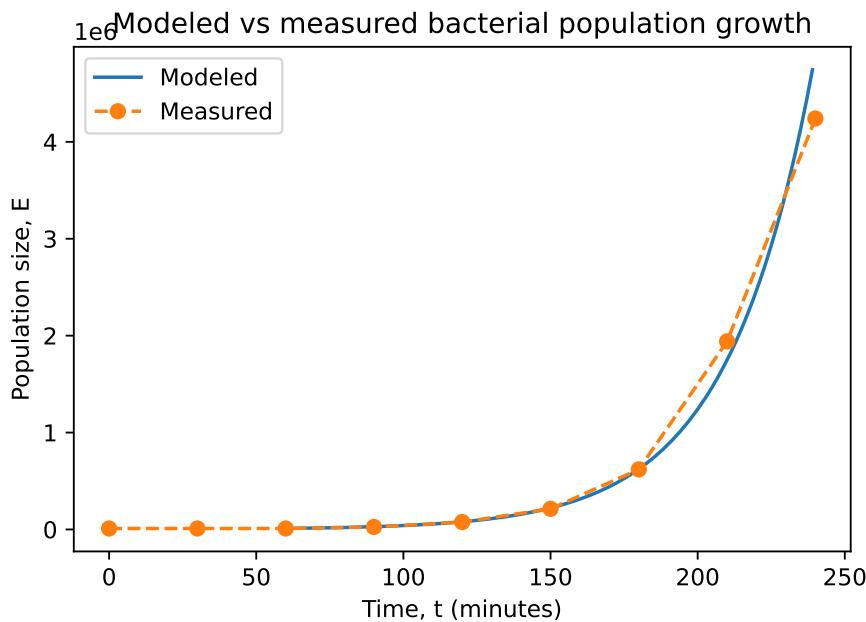
1  from pylab import *
2  import pandas
3
4  N = 180          # number of time steps
5  r = 0.035 # growth rate
6
7  E = []
8  t = []
9
10 # Initial conditions
11 E.append(10042)    # initial number of bacteria
12 t.append(60)        # initial measurement time (in minutes)
13
14 # Calculating number of bacteria and time for each generation
15 n = 1
16 while n < N:
17     E_n = (1 + r) * E[n-1]
18     E.append(E_n)
19
20     t_n = t[n-1] + 1
21     t.append(t_n)
22
23     n = n + 1

```

```

24
25 # Reading in experimental values
26 data = pandas.read_csv("ecoli.csv")
27 t_measured = list(data["t"])
28 E_measured = list(data["E"])
29
30 # Plotting the modeled and measured data in the same plot
31 plot(t, E, label="Modeled")
32 plot(t_measured, E_measured, "--o", label="Measured")
33 xlabel("Time, t (minutes)")
34 ylabel("Population size, E")
35 title("Modeled vs measured bacterial population growth")
36 legend()
37 show()

```



The bacterial population growth model is in accordance with our measured data. There are some differences, but the overall trend is very similar to experimental data.

3.8 Limitations of the exponential growth model

To find the limitations of our model, we should first look at our assumptions. We assumed in our model that there are no deaths, which is clearly wrong. How important is this? It turns out that the bacterial population doubles after a fixed time, even with death included. However, the doubling time is not exactly the same as the generation time. In reality, some bacteria die while others multiply (see Figure 3.13). The time it takes for the total population to double is still fixed, but it is longer than the generation time.

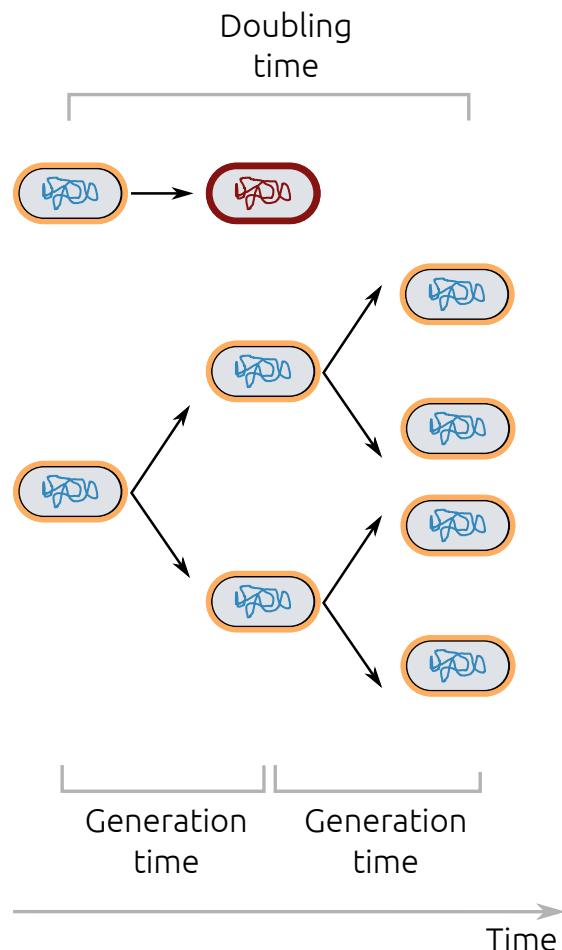


Figure 3.13: The difference between generation time and doubling time. Generation time is the (average) time between the divisions of cells, while the doubling time is the time it takes for the population to double. In reality, since some bacteria die (red), the doubling time is longer than the generation time. In the illustration above, the doubling time is twice the generation time.

Is exponential growth a good approximation? From our simulations, we have seen that it is a good approximation in the growth phase, but what happens if the exponential growth continues for a day or two? Let us simulate the bacterial population growth over two days. We need the number of time steps we must take in our simulation before two days have passed. The number of minutes in two days is:

$$2 \times 24 \times 60 \text{ minutes} = 2880 \text{ minutes.} \quad (3.41)$$

Thus, we simulate the bacterial population growth over 2880 minutes, i.e., set $N = 2880$, and have an initial population of $E_0 = 10042$ bacteria. If you do this, and print out the final number of bacteria, $E[-1]$, the output will be roughly 10^{47} .

This number is so large, it is difficult to get an idea of how many bacteria there actually are. Let us calculate the weight of this amount of bacteria. We only want to get a sense for the scale. The weight of an *E. coli* bacterium is around 10^{-15} kg, which means the total weight of all the bacteria is,

$$10^{47} \times 10^{-15} \text{ kg} = 10^{32} \text{ kg.} \quad (3.42)$$

After two days the total weight of the *E. coli* bacteria is approximately 10^{32} kg. Again, it is hard to understand how big this number is.

When we have numbers either very big or very small, it is useful to find something to compare the numbers to. In this case, the weight of the population is about 100 times the weight of the Sun! We have performed similar calculations for different time spans and found objects with similar mass to compare to the total weight of all the *E. coli* bacteria. The results are in the following table, note that these values are approximations:

Hours	Total mass of the <i>E. coli</i> equal
5	A fruit fly
11	An orange
13	Newborn human baby
20	The quaking aspen named Pando, largest living organism
25	Total mass of the human population
35	All of the oceans on Earth
40	Mass of Earth
46	Mass of the Sun
71	Mass of the entire observable universe

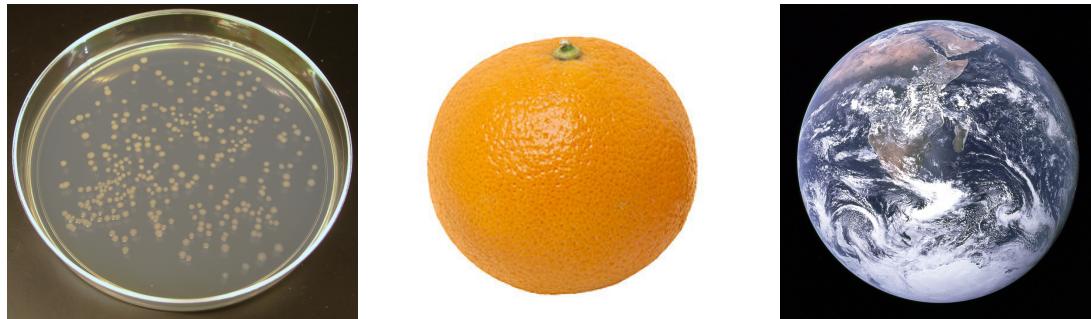


Figure 3.14: Bacterial population growth without limitation. Starting with a batch culture with 10 042 individuals, their total mass after 11 hours is close to the mass of an orange. After 40 hours the batch culture weighs close to the mass of the Earth. Left: [8], middle: [5], right: [2].

As seen from this table, the weight of the *E. coli* bacteria quickly gets enormous. After 40 hours the weight of the population is equal to the weight of the Earth, and after three days the *E. coli* bacteria weigh more than all the matter in the entire observable universe. Clearly exponential growth cannot continue indefinitely.

The reason why we get eternal exponential growth is because we assume that there are no factors that limit the population growth. This means the bacteria do not compete against each other, and there is always enough food in the environment. From 3.2.3, we know that the exponential growth phase is followed by a stationary phase, where the population growth is limited by factors such as nutrition, space, and waste products that inhibit further growth. The bacteria can no longer grow freely in this phase, and our assumption that there is no death is invalid. In the next chapter, we expand our model to also include the finite possible population size.

3.9 Difference equations: A mathematical look at what we have done

In this chapter, we have worked with problems where we want to find a sequence of numbers, where one number is defined in terms of the previous numbers, as for exponential growth:

$$E_n = (1 + r) \times E_{n-1}. \quad (3.43)$$

These kinds of equations are called *difference equations* or sometimes *recurrence relations*. We have purposefully avoided most of the mathematical terminology, as it distracts us from the biology. However, we want you to be aware of what you have actually learned in a broader context. If anyone ever asks if you have worked with difference equations, you can safely say you have. In this section, which can be skipped on a first reading of the book, we introduce the more general problem of difference equations, and some common terminology used to describe them.

3.9.1 A brief overview of difference equations

A difference equation is any equation that defines a sequence of numbers, where the each term in a sequence is given by the previous. This allows for a huge amount of different equations, with very different behaviors. The following are three examples of difference equations.

$$x_n = 2x_{n-1}, \quad (3.44)$$

$$x_n = \sqrt{x_{n-1}^2 + 1}, \quad (3.45)$$

$$x_n = \sin(x_{n-1}). \quad (3.46)$$

We need an initial condition in order to solve equations. Try to solve these with setting $x_0 = 1$, similar to how you solved the equation for the exponential growth earlier in Section 3.6. They all behave very differently!

The equations we have solved in this chapter, such as $E_n = (1 + r) \times E_{n-1}$, are called *first-order difference equations*, since they only depend on one previous term.

There are equations where the new element depends on the two previous elements, such as

$$x_n = x_{n-1} + x_{n-2}. \quad (3.47)$$

These equations, where the new term depends on the two previous terms are called *second-order difference equations*. For these equations we need two initial conditions in order to begin the loop. If we set $x_0 = 1$ and $x_1 = 1$, we calculate $x_2 = 2$, and the sequence goes

$$1, 1, 2, 3, 5, 8, 11, \dots \quad (3.48)$$

and so on.

In general, the order of the equation refers to the number of previous steps needed to calculate the current step. A difference equation where the new element in the series depends on the previous k elements is called an k -th order difference equation.

The reason we have all these terms to describe difference equations is that when working with them in a theoretical setting, it is possible to prove general features for all difference equation of a certain type. For example, it is possible to find a solution for some difference equations where you do not have to calculate all the intermediate terms. For our exponential growth difference equation, this solution is

$$E_n = (1 + r)^n E_0. \quad (3.49)$$

This type of solution is called a *closed-form* solution. The solution we have used, where we solve for one step at a time until we get to the step we want to calculate is called an *iterative* solution, we iterate to calculate each step.

It is reasonable to ask why we would go through all this work of using loops to iteratively solve our equation, when we can just find a closed-form solution. The answer is that a closed form solution does not always exist, while the iterative method works for any difference equation you will ever meet. It is also impossible to find a closed-form solution that could deal with all phases of bacterial growth, while our iterative method did that just fine.

3.10 Summary

In this chapter, you have learned to model the exponential phase in the bacterial population growth cycle, as examined in the previous chapter. In order to do this, you learned how `while` loops work in Python.

3.10.1 While-loops

A `while` loop repeats a set of statements as long as a specific condition is met:

```
a = 0
while a < 5:
    print("Inside the loop, a is now", a)
    a = a + 1
print("After the loop, a is", a)
```

```
Inside the loop, a is now 0
Inside the loop, a is now 1
Inside the loop, a is now 2
```

```

Inside the loop, a is now 3
Inside the loop, a is now 4
After the loop, a is 5
```

Here, the indented code is repeated as long as the condition is evaluated to **True**.

3.10.2 Mathematical symbols and their corresponding variable names

The different symbols we have used in this chapter are in the table below:

Symbol	Variable	Value	Meaning
n	<code>n</code>		Current time step
N	<code>N</code>		Number of time steps
E	<code>E</code>		Number of bacteria
E_n	<code>E[n]</code>		Number of bacteria at a given time step n
t	<code>t</code>		Time
t_n	<code>t[n]</code>		Time at a given time step n
E_0	<code>E[0]</code>	10042	Initial condition, number of bacteria we start with
E_{N-1}	<code>E[N-1]</code>		Number of bacteria at the final time step, $N-1$
r	<code>r</code>	0.035	Growth rate

3.10.3 Exponential growth

In the exponential growth model the bacteria grow without bounds as this model does not take limited resources into account.

Assumptions.

- Every bacterium is identical,
- every bacterium splits in two every generation, and
- there are no deaths.

Equation describing the model.

$$E_n = 2E_{n-1}, \quad (3.50)$$

$$t_n = t_{n-1} + 1 \text{ (minute).} \quad (3.51)$$

Implementation of the model. The core loop of this model is:

```

n = 1
while n < N:
    E[n] = 2*E[n-1]
    t[n] = t[n-1] + 1
    n = n + 1
```

3.10.4 Chapter glossary

Important terms introduced in this chapter are:

- **Asexual reproduction:** A mode of reproduction by which offspring arise from a single organism.
- **Binary fission:** A form of asexual reproduction, where two daughter cells arise from a single cell.
- **Eukaryote:** Cells with membrane bound nucleus, such as plant and animal cells.
- **Prokaryote:** Cells with their genetic material freely floating in the cytoplasm, such as bacteria.
- **Model organism:** A non-human organism that is extensively studied to understand particular biological phenomena.
- **Lag phase:** The phase of population growth where the population adapts to the environment, and there is no increase in the population.
- **Exponential growth phase:** The phase of population growth where we have a rapidly increasing (exponential) population growth.
- **Stationary phase:** The phase of population growth where the number of individuals that die is equal to the individuals that are born, so there is no net population growth.
- **Death phase:** The phase of population growth where there is (exponential) decay.
- **Long-term stationary phase:** The phase of population growth after the death phase where we have a long-term stable population.
- **Generation time:** The average age of mother cells that splits into two new daughter cells, for organisms that can reproduce more than once, the definition is more involved.
- **Doubling time:** The time it takes before a population is doubled in size.
- **Population growth rate:** How quickly a population grows, related to the doubling time.

Chapter 4

Models for limited bacterial population growth

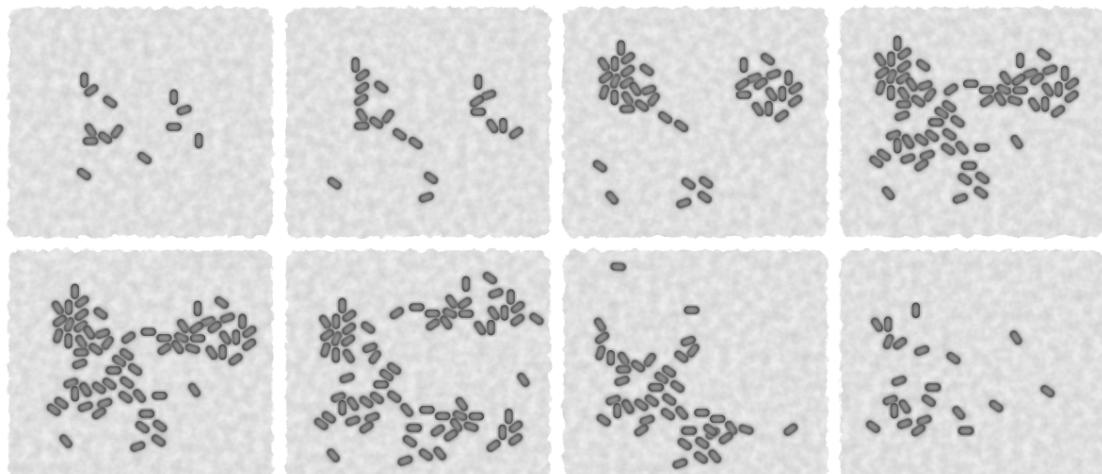


Figure 4.1: Bacterial population growth. In this chapter we simulate the limited population growth in a virtual experiment on a computer. The population grows exponentially at first. After some time the medium runs out of nutrients, and the population stops growing. Most bacteria die off, leaving a small stable population behind.

The exponential growth model shows how powerful programming is. Using only a few biological facts and less than 30 lines of Python code, we have created a model for how bacteria grow during the exponential growth phase and compared the model to experimental results.

The problem with the simple exponential growth model is that it grows without bounds. We saw in the previous chapter that this led to an unrealistic population size after only a few hours. This is because we did not include death of bacteria in that model. In Section 3.2.3, we saw that the population eventually reaches a stationary phase, and then begins to die off. This happens

because more bacteria die as the population grows larger, and the medium runs out of nutrients, as seen in Figure 4.1.

Our philosophy is to start with the simplest model possible and add additional details once the model has been implemented and tested. We are now ready to expand on the simple model for bacterial population growth. In this chapter we will make models which includes the stationary and death phases of the population growth cycle.



Learning outcomes

After working with this chapter you know how to create and implement the following models, how to compare them to data, as well as the limitations of each model:

- a logistic growth model,
- a model for the death phase, and
- a model for the death phase and long-term stationary phase.

We will use the programming concepts introduced in the previous chapters.

4.1 Logistic growth: A more detailed model

We saw in the previous chapter that exponential growth model allows the population to grow without bounds, which in the long run is unrealistic. In reality, any population that expands in this way eventually runs out of resources. This is what leads to the stationary phase of the *E. coli* population growth seen in Figure 3.3. We want to create a model that has this behavior.

In Chapter 3, we defined the exponential model for bacterial growth as:

$$E_n = (1 + r) \times E_{n-1}. \quad (4.1)$$

In this model, r , the growth rate, is the constant fraction of the bacterial cells that divide each time step.

4.1.1 Requirements of our new model

We use equation (4.1) as a starting point to find a more realistic model than the exponential growth model. When available resources in the environment are limited there is a maximum sustainable population size where the population use all resources available at any one time. In this section, we use the following assumptions:



Model assumptions for the logistic growth model

We assume that:

- there is exponential growth when there is a large amount of resources compared to the needs of the population, and
- there is a maximum sustainable population size.

This maximum sustainable population size is called the *carrying capacity*, and when we use it in equations, it has the symbol K . The carrying capacity varies from one environment to the next.

For *E. coli*, available nutrients and space, and amount of waste products in the environment limits the growth. From Figure 3.5 we see that the carrying capacity for our measured bacterial population is around $K = 8 \times 10^9$.

We want our more realistic model to have:

- exponential growth when the population is much smaller than the carrying capacity, as shown in Figure 4.2 (top), and
- no population growth if the population is equal to the carrying capacity, as shown in Figure 4.2 (bottom).



Mathematical approximation symbols

In the following derivations, we use some mathematical symbols indicating approximations, rather than exact mathematics:

- the symbol \ll means *much smaller than*,
- the symbol \gg means *much greater than*, and
- the symbol \approx means *approximately equal to*.

To be able to model realistic population growth, we must find an equation for the population growth that gives us the above behavior. If the population is much smaller than the carrying capacity, we write this mathematically as $E_{n-1} \ll K$ (see the box above). In this case, the population should grow exponentially:

$$E_n = (1 + r) \times E_{n-1}. \quad (4.2)$$

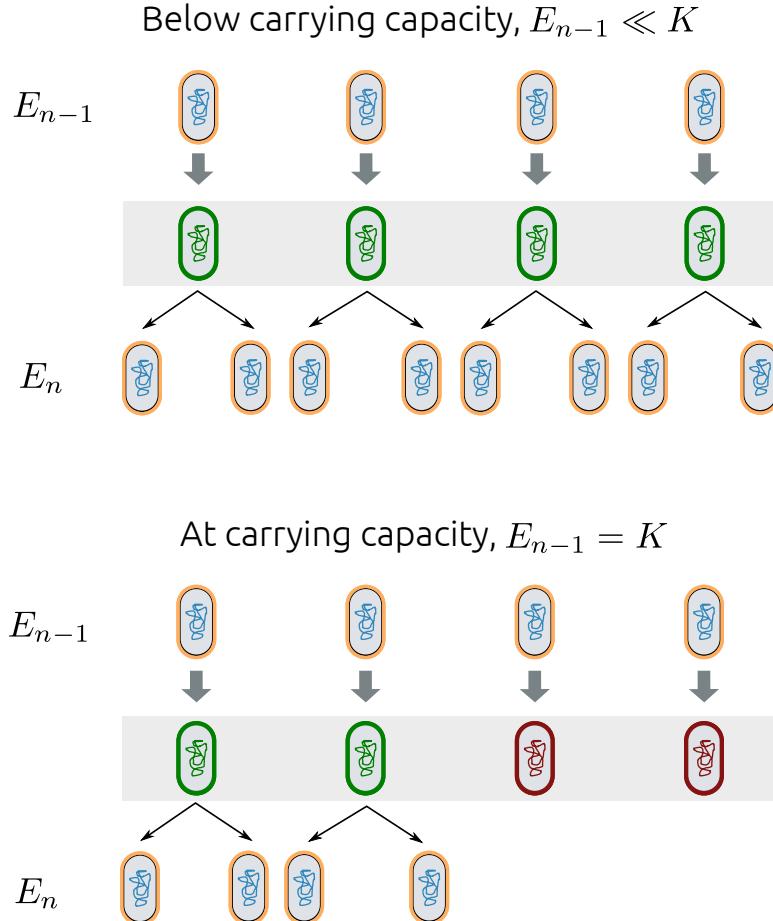


Figure 4.2: *Top:* When the number of bacteria is much lower than the carrying capacity, $E_{n-1} \ll K$, growth is exponential, with the population doubling each generation. *Bottom:* When the number of bacteria is at the carrying capacity, $E_{n-1} = K$, then the number of bacteria that undergo binary fission (green) is the same as the number of bacteria that die (red), and the net growth is zero.

When the population is at the carrying capacity, which is when $E_{n-1} = K$, there should be no change in the population size:

$$E_n = E_{n-1}. \quad (4.3)$$

This then means that:

$$(1 + r) = 1 \quad (4.4)$$

Or $r = 0$. This implies that we should make r , the growth rate, dependent on the relation between the population size E_n and the carrying capacity K .

4.1.2 The logistic growth equation

To summarize the above requirements, we need an equation for bacterial population growth that fulfills:

- when the population is much smaller than the carrying capacity ($E_{n-1} \ll K$), then $E_n = (1 + r) \times E_{n-1}$, and
- when the population is near to, or equal the carrying capacity ($E_{n-1} \approx K$), then $E_n = 1 \times E_{n-1}$.

Such an equation takes into account limited resources in the environment. Finding an equation that fulfill these requirements is not trivial and is outside the scope of this book. Instead, we give you one of the simpler ones, after first explaining how it was arrived at.

We start by looking at the relation between the population size E_n and the carrying capacity K for the requirements described above, and will consider the fraction $\frac{E_{n-1}}{K}$:

- when the population is much smaller than the carrying capacity ($E_{n-1} \ll K$), then

$$\frac{E_{n-1}}{K} \approx 0 \quad (4.5)$$

- when the population is near to, or equal the carrying capacity ($E_{n-1} \approx K$), then $\frac{E_{n-1}}{K} \approx 1$

$$(4.6)$$

Considering our requirements, it seems that using the fraction $\frac{E_{n-1}}{K}$ gives us the opposite of what we need: 0 when there is exponential growth and 1 near the carrying capacity. How can we then still use this fraction to construct an equation that fulfills our requirements? We can do this by, instead of applying the fraction itself, using

$$\left(1 - \frac{E_{n-1}}{K}\right). \quad (4.7)$$

and multiplying r by this term! Our equation then becomes:

$$E_n = \left(1 + r \left(1 - \frac{E_{n-1}}{K}\right)\right) \times E_{n-1}. \quad (4.8)$$

This equation is what is called a *logistic equation*.

Going back to our requirements, we see that they indeed are fulfilled. When the population is much smaller than the carrying capacity ($E_{n-1} \ll K$), then

$$\frac{E_{n-1}}{K} \approx 0, \quad (4.9)$$

thus

$$\left(1 - \frac{E_{n-1}}{K}\right) \approx 1, \quad (4.10)$$

and

$$E_n = (1 + r) \times E_{n-1}. \quad (4.11)$$

However, when the population is near, or equal to the carrying capacity ($E_{n-1} \approx K$), then

$$\frac{E_{n-1}}{K} \approx 1, \quad (4.12)$$

thus

$$\left(1 - \frac{E_{n-1}}{K}\right) \approx 0, \quad (4.13)$$

and

$$E_n = 1 \times E_{n-1}. \quad (4.14)$$

Let us verify that this equation fulfills our requirements. We set the carrying capacity to be 40 000 bacteria, and test two different values for the population at the previous time step, corresponding to our two requirements. We use `E_previous` as the name for E_{n-1} in this code. We calculate E_n for each of these different values:

```
K = 40000
r = 0.035

E_previous = 1000
r_adjust = (1 - E_previous/K) # multiplication factor
E_n = (1 + r * (1 - E_previous/K)) * E_previous
print("When E_previous =", E_previous, "r is multiplied by", r_adjust, "and thus E_n =", E_n)
```

When `E_previous = 1000` `r` is multiplied by 0.975 and thus `E_n = 1034.125`

- When `E_previous = 1000`, the population is much smaller than the carrying capacity, and `r` is multiplied by 0.975, which is almost 1.

```
E_previous = 40000
r_adjust = (1 - E_previous/K) # multiplication factor
E_n = (1 + r * (1 - E_previous/K)) * E_previous
print("When E_previous =", E_previous, "r is multiplied by", r_adjust, "and thus E_n =", E_n)
```

When `E_previous = 40000` `r` is multiplied by 0.0 and thus `E_n = 40000.0`

- When `E_previous = 40000`, the population is equal the carrying capacity, and `r` is multiplied by 0, such that there is no growth.

The equation we use here thus fulfills both requirements.

We keep the initial conditions and time calculations similar to the exponential growth model. This new model is called a logistic model or a logistic growth model.



Summary: The logistic growth model for bacterial populations

The logistic growth model for *E. coli*:

$$E_n = \left(1 + r \left(1 - \frac{E_{n-1}}{K}\right)\right) \times E_{n-1}, \quad (4.15)$$

$$t_n = t_{n-1} + 1 \text{ (minute)}, \quad (4.16)$$

with carrying capacity

$$K = 8 \times 10^9, \quad (4.17)$$

and initial conditions

$$E_0 = 10\,042, \quad (4.18)$$

$$t_0 = 60. \quad (4.19)$$



Doubling time and generation time

When working with the exponential model, we tend to use the terms “doubling time” and “generation time” interchangeably. However, with the logistic model these terms are no longer identical. As the population grows close to the carrying capacity, the *E. coli* bacteria will start to reproduce slower, therefore the generation time becomes longer. Furthermore, the system does not really have a fixed doubling time, as the population never grows past the carrying capacity.

In order to implement this model, we only need to change three lines in the exponential growth program we created in the previous chapter. We need to specify the variable K, and change the equation for calculating E[n]. We would also like to simulate a longer time period, since the logistic model also models the stationary phase in addition to the exponential growth phase.

From Figure 3.5 we see that the stationary phase lasts until 810 minutes, and with $t_0 = 60$ minutes, we thus set the number of time steps to N = 750.

The modified code looks like:

```

1  from pylab import *
2
3  N = 750          # number of time steps                      # New
4  r = 0.035        # growth rate
5  K = 8e9          # carrying capacity                         # New
6
7  E = []

```

```

8   t = []
9
10  # Initial conditions
11  E.append(10042)
12  t.append(60)      # minutes
13
14  # Calculating number of bacteria and time for each time step
15  n = 1
16  while n < N:
17      E_n = (1 + r * (1 - E[n-1]/K)) * E[n-1]           # New
18      E.append(E_n)
19
20      t_n = t[n-1] + 1
21      t.append(t_n)
22
23      n = n + 1
24
25  plot(t, E)
26  xlabel("Time, t (minutes)")
27  ylabel("Population size, E")
28  title("Bacterial population growth model")
29  show()

```

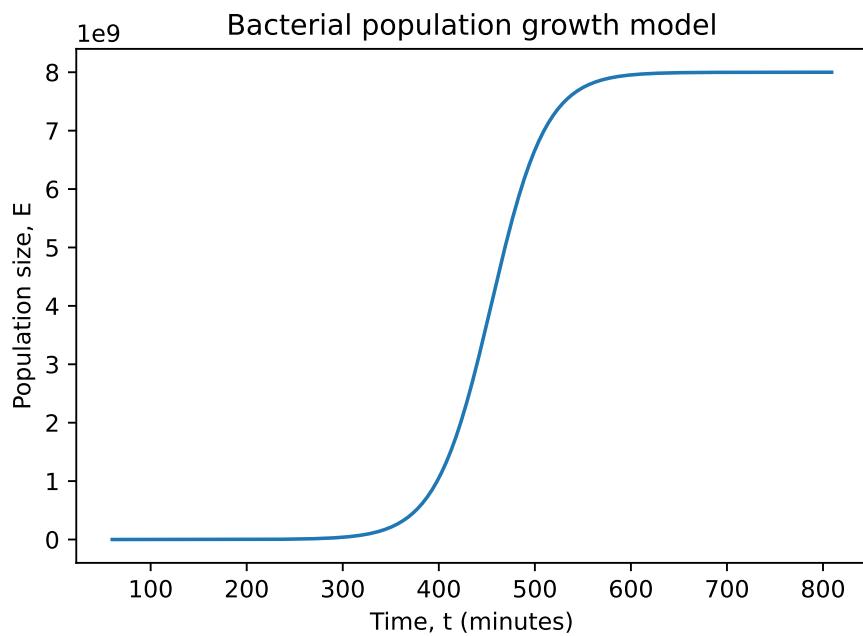


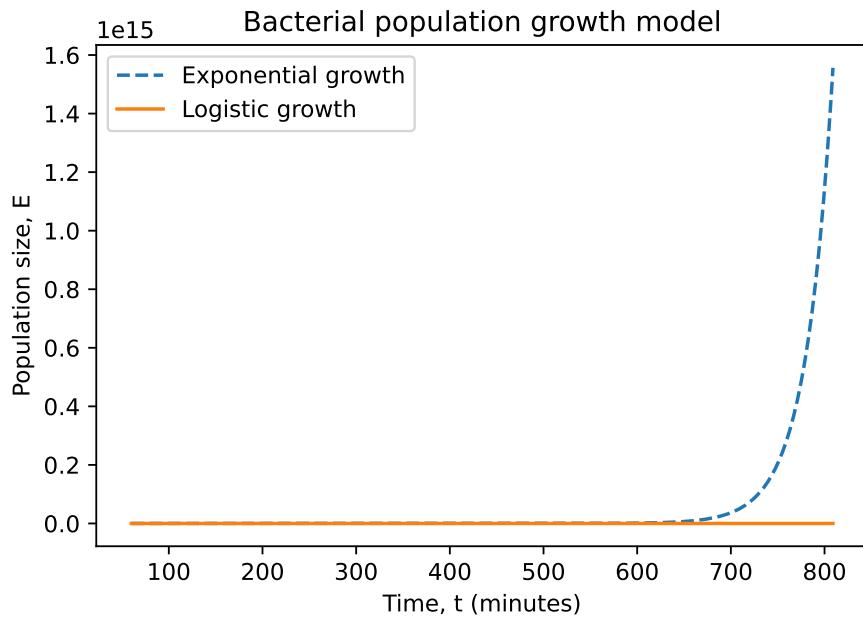
Figure 4.3: A plot showing the logistic growth model. The bacterial population growth slows down as the population approaches K .

As seen in Figure 4.3, in the beginning our new model behaves almost identical to the exponential growth model; we nearly double the bacterial population each time step. However, once we approach the carrying capacity the growth slows down until it stagnates. In order to highlight the differences, let us plot both models in the same plot.

4.1.3 A comparison of logistic and exponential growth

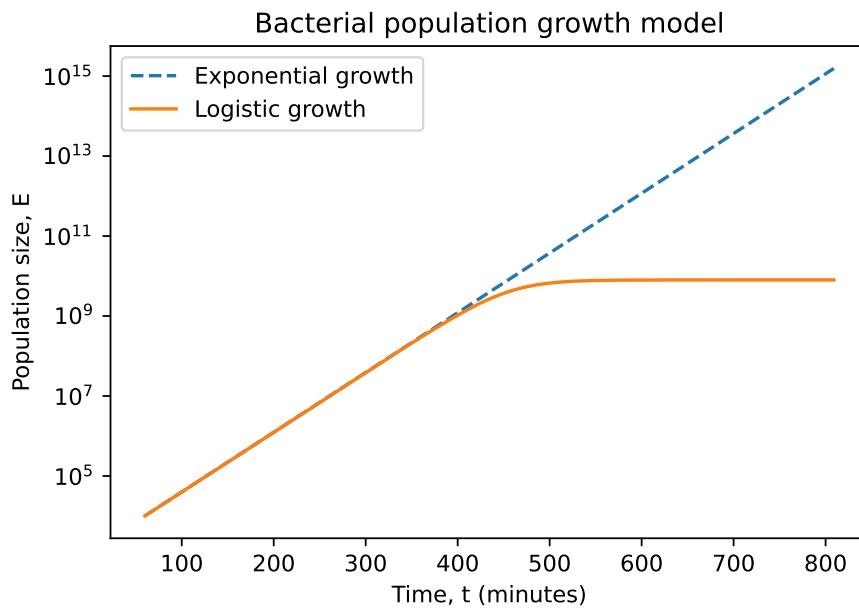
To compare the exponential growth model and the logistic growth model we calculate for both models inside the same loop. We use `E_exponential` for the exponential model and `E_logistic` for the logistic model and plot them in the same plot:

```
1 from pylab import *
2
3 N = 750          # number of time steps
4 r = 0.035        # growth rate
5 K = 8e9          # carrying capacity
6
7 E_exponential = []
8 E_logistic = []
9 t = []
10
11 # Initial conditions
12 E_exponential.append(10042)                      # New
13 E_logistic.append(10042)                          # New
14 t.append(60)          # minutes
15
16 # Calculating number of bacteria and time for each time step
17 n = 1
18 while n < N:
19     E_exponential_n = (1 + r) * E_exponential[n-1]
20     E_exponential.append(E_exponential_n)
21     E_logistic_n = (1 + r * (1 - E_logistic[n-1]/K)) * E_logistic[n-1]    # New
22     E_logistic.append(E_logistic_n)
23     t_n = t[n-1] + 1
24     t.append(t_n)
25     n = n + 1
26
27
28 plot(t, E_exponential, "--", label="Exponential growth")                  # New
29 plot(t, E_logistic, "-", label="Logistic growth")                         # New
30 xlabel("Time, t (minutes)")
31 ylabel("Population size, E")
32 title("Bacterial population growth model")
33 legend()
34 show()
```



The exponential growth dominates and the exponential population increases to very large values, so large that we are unable to see what happens with the logistic population. To be able to compare both high and low values we plot with a logarithmic y -axis:

```
plot(t, E_exponential, "--", label="Exponential growth")
plot(t, E_logistic, "-", label="Logistic growth")
xlabel("Time, t (minutes)")
ylabel("Population size, E")
title("Bacterial population growth model")
legend()
yscale("log")                                     # New
show()
```

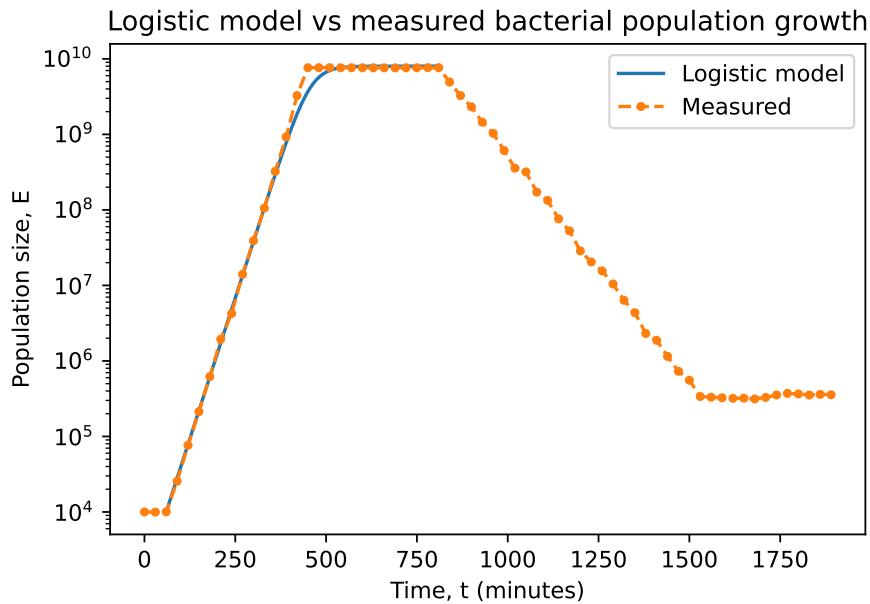


This plot makes it easier to compare our two models. Both behave similarly for small populations. However, as the population size gets close to the carrying capacity ($K = 8 \times 10^9$), the growth in the logistic model slows down, and eventually stops, while the exponential model continues to increase.

Let us compare the results from the logistic model to the measured data to verify that the logistic model is indeed better than the exponential model. We load the measured *E. coli* data from all phases of the population growth and plot it the same plot as the logistic model. We use a smaller marker so it becomes easier to compare the two lines in the plot.

```

1 import pandas
2
3 # Reading in experimental values
4 data = pandas.read_csv("ecoli_all.csv")
5 t_measured = list(data["t"])
6 E_measured = list(data["E"])
7
8 # Plotting the modeled and measured data in the same plot
9 plot(t, E_logistic, "-", label="Logistic model")
10 plot(t_measured, E_measured, "o-", label="Measured")
11 xlabel("Time, t (minutes)")
12 ylabel("Population size, E")
13 title("Logistic model vs measured bacterial population growth")
14 legend()
15 yscale("log")
16 show()
```



The result from the logistic model fits well with the measured data for both the exponential growth phase and the stationary phase, much better than exponential growth model. The logistic growth model avoids the unlimited growth of the exponential growth model, since the logistic growth takes into account that there are limited resources available in the environment. To use a logistic growth model we need to know the carrying capacity, which varies from environment to environment.

It is important to note that we rarely find perfect logistic growth in nature. The environment of a species varies with time, and can suffer catastrophes that set back the population growth. You will see an example of this in the next chapter. Some populations never even approach the carrying capacity because the growth is always set back by periodic catastrophes. This is especially the case for small, quickly reproducing animals like insects.

Another assumption we made was that the population grows most rapidly when there are few individuals. This is not the case for all populations, some have a harder time surviving when there are few individuals. For example, solitary animals, like the rhinoceros, might have problems finding a mate when there are too few individuals around.

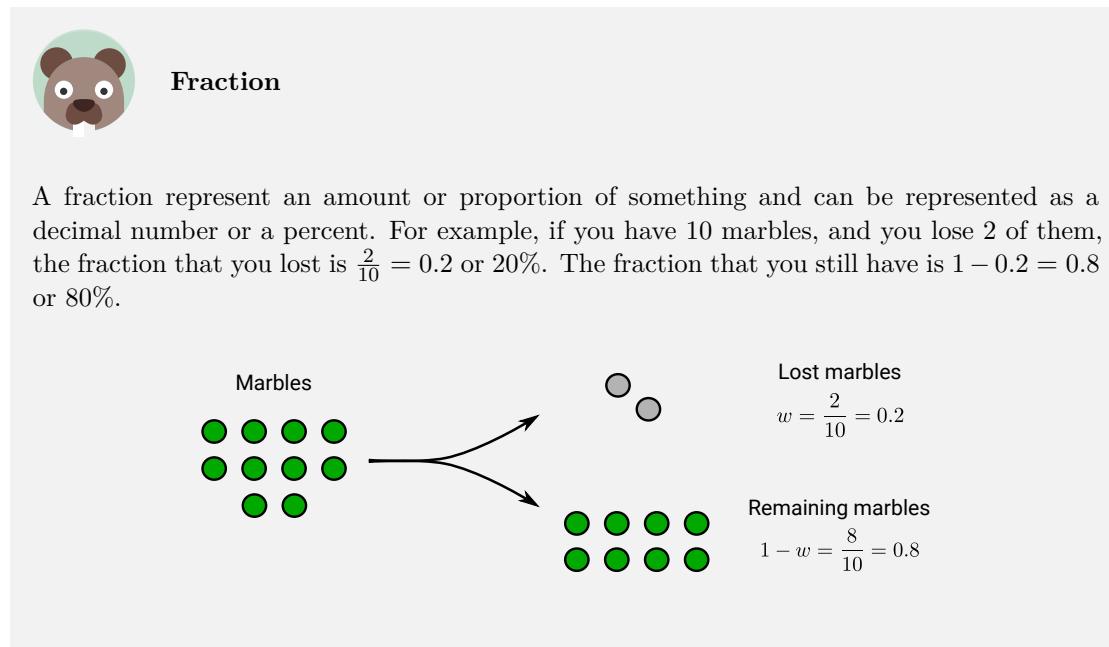
Another limitation of the logistic model is that it smoothly approaches the carrying capacity, but for most natural populations there will be some lag. Food might become limiting, but the birth of new individuals is not affected until one generation has passed. The population might then overshoot the carrying capacity before it falls down and stabilizes at the carrying capacity. The population might also fluctuate around the carrying capacity instead of smoothly approaching it. The logistic model is still a good model to use, but it is important to keep the above limitations in mind when using it.

4.2 Modeling the death phase

We now have a good model for the exponential growth phase and the stationary phase. Looking back to the complete *E. coli* cycle in Figure 3.5, there are another two distinct phases in the life span of an *E. coli* batch culture: the lag phase and the death phase. After the population has remained roughly constant for a while it enters the death phase and begins to drop quite fast, before it stabilizes at a much lower number of bacteria. One reason for this drop can be a lack of nutrients supplied to the batch culture. In this section we create a model for this death phase.

4.2.1 Simple model for the death phase

We start with a simple model for the death phase before we create a more realistic model. In the death phase, the number of bacteria is decreasing. This is not because the bacteria have stopped reproducing, but there are fewer births than deaths, which means the total number of bacteria decreases with time. We assume that each time step, the number of bacteria is reduced by a fixed fraction, D , which we call the *death rate*.



Let us say we start with 5 bacteria. After one time step, 3 have died, and the other 2 have reproduced, leaving 4 bacteria in total, see Figure 4.4. The fraction the bacteria is reduced by is $\frac{5-4}{5} = 0.2$ or 20%. This means that reduction in the number of bacteria varies, it is largest in the beginning and steadily falls with time.

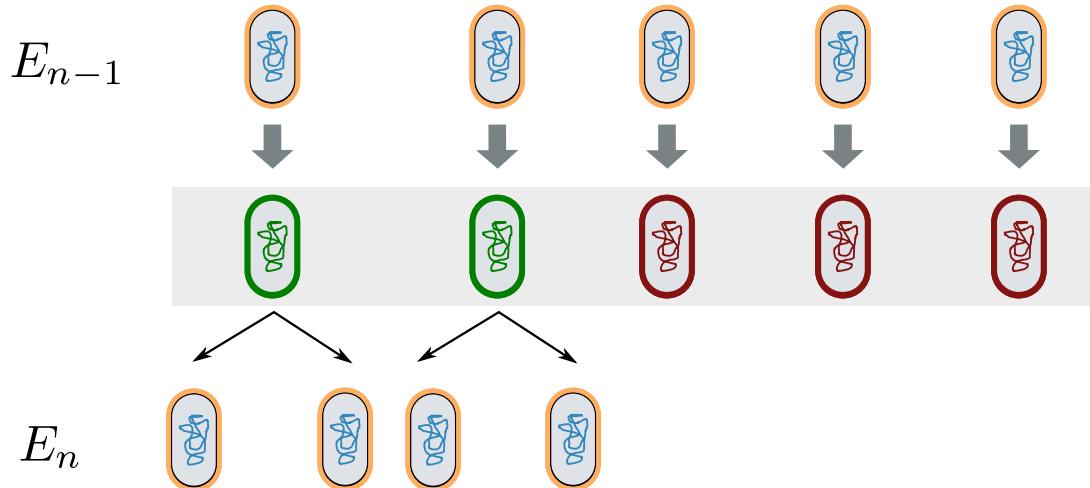


Figure 4.4: The bacteria is reduced by a fraction D , which is the sum of bacteria that are born and die.



Model assumptions for the simple death phase model

We assume that the number of bacteria is reduced by a fixed fraction each time step.

In other words, instead of a growth rate r , we need a death rate, which we will call D . As with the growth rate, the death rate is proportional to the population size, and we again assume that a certain constant fraction of the bacterial cells dies each time step.

Our simple model for the death phase then becomes:

$$E_n = E_{n-1} - D \times E_{n-1}. \quad (4.20)$$

This equation states that the size of the population at the current time step is composed of the size at the previous time step *minus* the fraction of the bacterial cells that died in the time interval. Such a model is called an *exponential decay* model. We simplify again to:

$$E_n = (1 - D)E_{n-1}. \quad (4.21)$$

It is reasonable to assume that the initial number of bacteria in the death phase is approximately at the carrying capacity $K = 8 \times 10^9$, as this is the number of bacteria in the stationary phase. From Figure 3.5 we get that the death phase starts at time $t_0 = 810$ minutes and ends at the last timepoint in our dataset, which is $t = 1890$ minutes. We call this model the *simple death model*.

We use $D = 0.014$ in our model.



Determining death rate

How did we arrive at this value for the death rate D ? For our population, we assume that 25% of the bacteria die each generation. This means that after 20 minutes, which is the generation time of our population, we are left with 75%, or 0.75 of the number of bacteria at the previous generation. In other words, for one generation we get:

$$E_{n+20} = 0.75E_n \quad (4.22)$$

From equation (4.21) we know that

$$E_{n+1} = (1 - D)E_n, \quad (4.23)$$

We now need an equation to calculate E_{n+20} from E_n . Let us first calculate E_{n+2} :

$$E_{n+2} = (1 - D)E_{n+1} \quad (4.24)$$

$$= (1 - D)(1 - D)E_n \quad (4.25)$$

$$= (1 - D)^2 E_n \quad (4.26)$$

$$(4.27)$$

Now we calculate E_{n+3} :

$$E_{n+3} = (1 - D)E_{n+2} \quad (4.28)$$

$$= (1 - D)(1 - D)^2 E_n \quad (4.29)$$

$$= (1 - D)^3 E_n \quad (4.30)$$

$$(4.31)$$

There is a pattern here, so that we now can obtain the equation for E_{20} :

$$E_{n+20} = (1 - D)^{20} E_n \quad (4.32)$$

Combining this with equation (4.22), this means that:

$$(1 - D)^{20} E_n = 0.75E_n \quad (4.33)$$

Dividing both sides by E_n gives us

$$(1 - D)^{20} = 0.75 \quad (4.34)$$

Now we can calculate D . To get rid of the power term on the left side, we first raise both sides to the power $\frac{1}{20}$, then solve for D :

$$(1 - D) = 0.75^{1/20}, \quad (4.35)$$

$$D = 1 - 0.75^{1/20}, \quad (4.36)$$

$$D = 1 - 0.986, \quad (4.37)$$

$$D = 0.014 \quad (4.38)$$

What we have done here is to derive the so-called *analytical* solution to the difference equation (4.21). Analytical solutions allow one to calculate the value for the model for any step from the initial condition(s), without the need to calculate all intermediate step. Analytical solutions are outside the scope of this book, but we give it here to show how we derived the value of D , and for illustrative purposes.



Summary: The simple death model

The simple death model for the growth of *E. coli*:

$$E_n = (1 - D)E_{n-1}, \quad (4.39)$$

$$t_n = t_{n-1} + 1 \text{ (minute)}, \quad (4.40)$$

with death rate,

$$D = 0.014, \quad (4.41)$$

and the initial conditions,

$$E_0 = 8 \times 10^9, \quad (4.42)$$

$$t_0 = 810. \quad (4.43)$$

We want to examine the population for the period from time $t_0 = 810$ minutes to $t_0 = 1890$ minutes, which is 1080 minutes in total. We thus set $N = 1080$, and plot the results in a semilogarithmic plot. We suggest you attempt to implement this model on your own as an exercise. For completeness we include an example of an implementation:

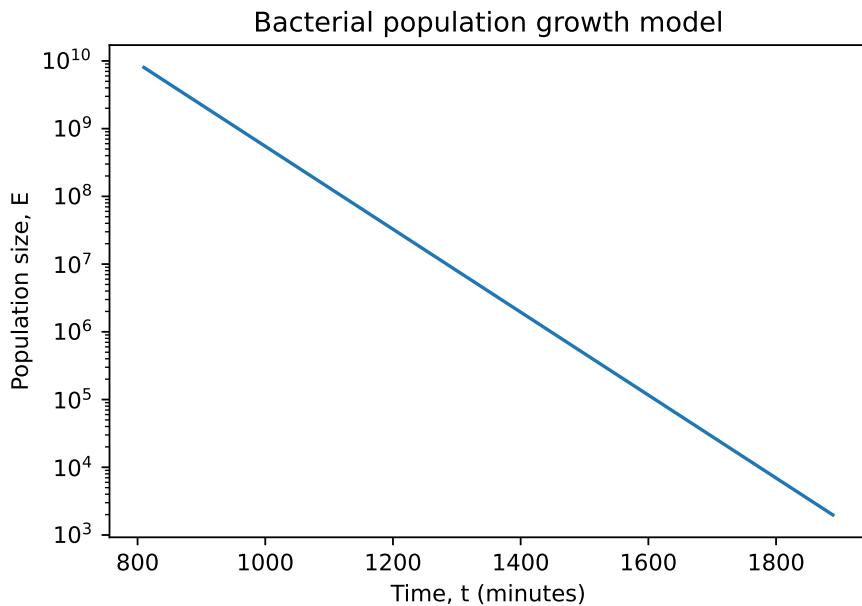
```

1  from pylab import *
2
3  N = 1080          # number of time steps
4  D = 0.014         # death rate
5
6  E = []
7  t = []
8
```

```

9 # Initial conditions
10 E.append(8e9)                                     # New
11 t.append(810)      # minutes                      # New
12
13 # Calculating number of bacteria and time for each time step
14 n = 1
15 while n < N:
16     E_n = (1 - D) * E[n-1]                         # New
17     E.append(E_n)
18     t_n = t[n-1] + 1
19     t.append(t_n)
20     n = n + 1
21
22 plot(t, E, "-")
23 xlabel("Time, t (minutes)")
24 ylabel("Population size, E")
25 title("Bacterial population growth model")
26yscale("log")
27 show()

```



This result seems reasonable, but let us compare it to the measured data to verify the simple death model. As before, we load the measured *E. coli* data from all phases of the population growth, and plot it the same plot as the results from the simple death model:

```

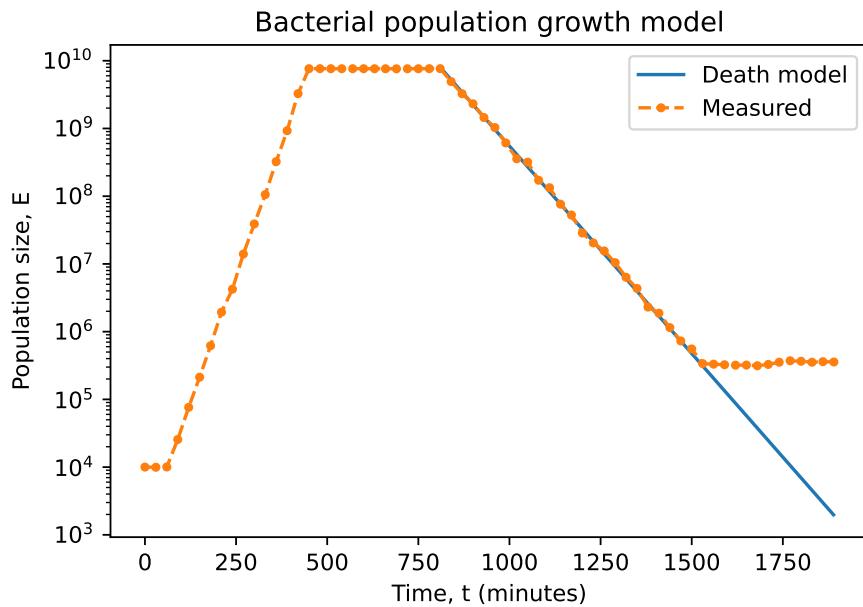
1 # Reading in experimental values
2 data = pandas.read_csv("ecoli_all.csv")
3 t_measured = list(data["t"])
4 E_measured = list(data["E"])
5
6 # Plotting the modeled and measured data in the same plot
7 plot(t, E, "-", label="Death model")
8 plot(t_measured, E_measured, ":", label="Measured")
9 xlabel("Time, t (minutes)")

```

```

10     ylabel("Population size, E")
11     title("Bacterial population growth model")
12     yscale("log")
13     legend()
14     show()

```



This simple death model works very well for the death phase of the bacterial population growth, but the decay never stops. From our measured data we observe that the decay should eventually end, and we should reach a new long-term stationary phase. We now introduce a model which allows us to simulate both the death phase and the long-term stationary phase. This is similar to how we introduced logistic growth to simulate both the exponential growth phase and the stationary phase.

4.2.2 Detailed model with long-term stationary phase

Our new model should have the same decay as the simple model while the population is larger than the final population size, but once we get closer to the final population size the decay should end.



Model assumptions for the advanced death model

For the advanced death model we assume that:

- there is a final stable population, where there is no change in the number of bacteria, and

- when the population is much larger than the final population, the number of bacteria is reduced by a fixed fraction each time step.

We call this final population size F . From our measured data we observe that the bacterial population stabilizes at approximately $F = 3 \times 10^5$ bacteria. Let us find a model that fulfills the above assumptions. We start from the equation for the simple death model:

$$E_n = (1 - D)E_{n-1}. \quad (4.44)$$

For the advanced model, we revert back to the version we started with:

$$E_n = E_{n-1} - D \times E_{n-1}. \quad (4.45)$$

The advanced model should have the same decay as in the simple death model when the population is much larger than the final population size. If the population is at the final population size, $E_{n-1} = F$, then we want there to be no change in the population size. As with the logistic growth model, there are several equations that achieve this and finding such an equation is complicated, so we simply state the equation here:

$$E_n = E_{n-1} - D(E_{n-1} - F). \quad (4.46)$$

Let us examine this expression to make sure it fulfills our requirements. When we are at the final population size we have $E_{n-1} = F$. Inserting this into Equation (4.46) gives us:

$$E_n = F - D(F - F). \quad (4.47)$$

and as $F - F = 0$, the result is:

$$E_n = F. \quad (4.48)$$

And we have no population growth, as we required. If we are far from final population size, such as when we start at the carrying capacity from the stationary phase, $E_{n-1} = 8 \times 10^9$, we note that the term:

$$-D(E_{n-1} - F) \quad (4.49)$$

becomes

$$-D(8 \times 10^9 - 3 \times 10^5). \quad (4.50)$$

or

$$-D(7.9997 \times 10^9). \quad (4.51)$$

But $7.9997 \times 10^9 \approx 8 \times 10^9 = E_{n-1}$. This is because at this stage, $E_{n-1} \gg F$. We can thus simplify the equation to

$$E_n = E_{n-1} - D \times E_{n-1}, \quad (4.52)$$

which we can rewrite to

$$E_n = (1 - D)E_{n-1}. \quad (4.53)$$

This is again our equation for the simple model. So our new model appears to fulfill our requirements. We call this new model the *advanced death model*.



Summary: The advanced death model

The advanced death model for the growth of *E. coli*:

$$E_n = E_{n-1} - D(E_{n-1} - F), \quad (4.54)$$

$$t_n = t_{n-1} + 1 \text{ (minute)}, \quad (4.55)$$

with death rate,

$$D = 0.014, \quad (4.56)$$

final population size,

$$F = 3 \times 10^5, \quad (4.57)$$

and the initial conditions,

$$E_0 = 8 \times 10^9, \quad (4.58)$$

$$t_0 = 810. \quad (4.59)$$

We implement this model as follows:

```

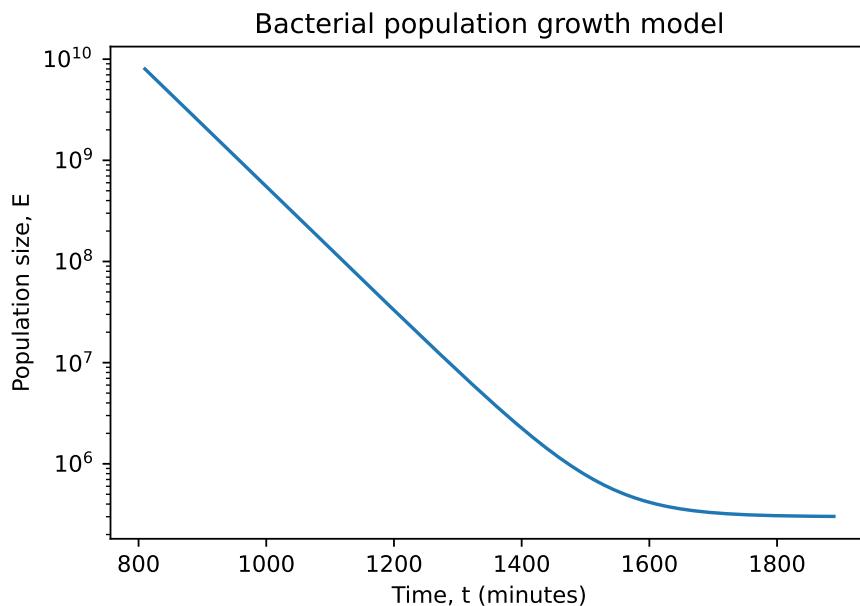
1  from pylab import *
2
3  N = 1080      # number of time steps
4  D = 0.014     # death rate
5  F = 3e5       # final population size           # New
6
7  E = []
8  t = []         # New
9
10 # Initial conditions
11 E.append(8e9)
12 t.append(810)   # minutes
13
14 # Calculating number of bacteria and time for each time step

```

```

15 n = 1
16 while n < N:
17     E_n = E[n-1] - D*(E[n-1] - F) # New
18     E.append(E_n)
19     t_n = t[n-1] + 1
20     t.append(t_n)
21     n = n + 1
22
23 plot(t, E, "-")
24 xlabel("Time, t (minutes)")
25 ylabel("Population size, E")
26 title("Bacterial population growth model")
27 yscale("log")
28 show()

```



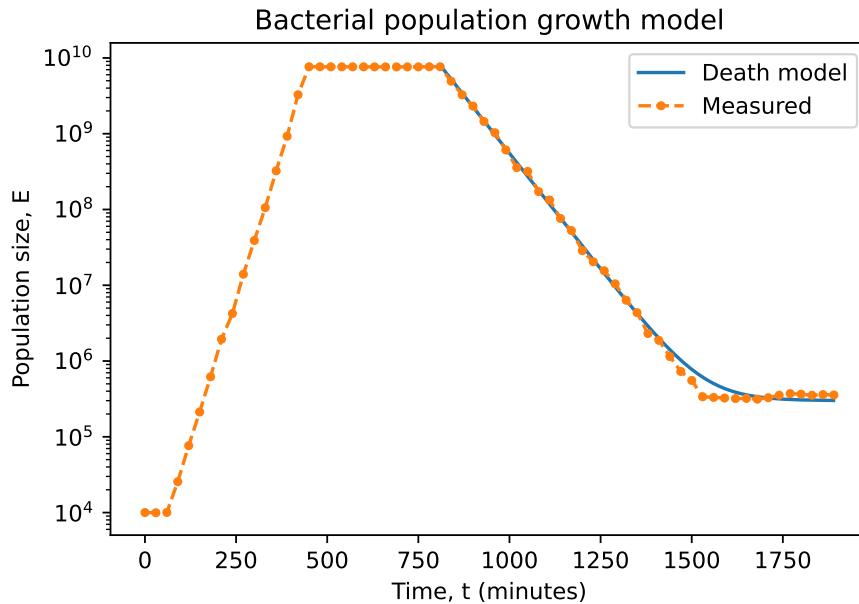
This result seems reasonable, because the population stabilizes at the final population size. Let us compare the advanced death model to the measured data. We load the measured *E. coli* data from all phases of the population growth and plot it the same plot as the results from the advanced death model:

```

1 import pandas
2
3 # Reading in experimental values
4 data = pandas.read_csv("ecoli_all.csv")
5 t_measured = list(data["t"])
6 E_measured = list(data["E"])
7
8 # Plotting the modeled and measured data in the same plot
9 plot(t, E, "-", label="Death model")
10 plot(t_measured, E_measured, "--.", label="Measured")
11 xlabel("Time, t (minutes)")
12 ylabel("Population size, E")
13 title("Bacterial population growth model")

```

```
14  yscale("log")
15  legend()
16  show()
```



In the next section, we deal with the lag phase, which we skipped initially. This will allow us to model the complete life cycle of the *E. coli* population.

4.3 Modeling the lag phase

In the lag phase, the number of bacteria is roughly constant. The only variable is how many time steps the lag phase lasts.



Model assumptions for the lag phase

For the lag phase we assume that the number of bacteria is constant.

From Figure 3.5, we see that the lag phase lasts for about 60 minutes, which is roughly to the fourth time step. Since there is no population change between the time steps in this phase, each time step has the same number of bacteria as the previous time step:

$$E_n = E_{n-1}. \quad (4.60)$$

To model the lag phase we loop through the first 60 minutes and set the number of bacteria equal to the number of bacteria in the previous time step:

```

1 from pylab import *
2
3 N = 60
4
5 E = []
6 t = []
7
8 E.append(10042)
9 t.append(0)      # minutes
10
11 n = 1
12 while n < N:
13     E_n = E[n-1]          # New
14     E.append(E_n)
15     t_n = t[n-1] + 1
16     t.append(t_n)
17     n = n + 1
18
19 print(E)

```

```
[10042, 10042, 10042, 10042, 10042, 10042, 10042, 10042, 10042, 10042, 10042, 10042,
 10042, 10042, 10042, 10042, 10042, 10042, 10042, 10042, 10042, 10042, 10042,
 10042, 10042, 10042, 10042, 10042, 10042, 10042, 10042, 10042, 10042, 10042,
 10042, 10042, 10042, 10042, 10042, 10042, 10042, 10042, 10042, 10042, 10042,
 10042, 10042, 10042, 10042, 10042, 10042, 10042, 10042]
```

This result is as we expected. Nothing happens with the population from one time step to the next.

4.4 Modeling all phases of the bacterial population growth

In this final section, we combine models from this chapter to simulate the entire *E. coli* life cycle. There is no single equation for the change in the number of bacteria, that allows us to model all phases of the bacterial population growth. Instead we use three of the models we have developed in this chapter. The first part of the population growth is modeled by the lag phase model, the second part by the logistic growth model, and the last part by the advanced death model. The easiest way to implement this using the programming tools we know is to use three separate loops, one for each phase. We begin by creating the basic structure of the program, and fill in the details afterwards:

```

N_lag = 60      # Time steps in the lag phase
N_log = 750     # Time steps in the logarithmic growth phase
N_death = 1080   # Time steps in the death phase

# Total number of time steps
N = N_lag + N_log + N_death

n = 1
while n < N_lag:

```

```

# perform lag phase modeling
n = n + 1

while n < N_lag + N_log:
    # perform logistic growth modeling
    n = n + 1

while n < N:
    # perform death phase modeling
    n = n + 1

```

Note that the logistic growth starts after the last lag phase time step and therefore lasts until time step $N_{\text{lag}} + N_{\text{log}}$. Similarly, the death phase modeling starts at time step $N_{\text{lag}} + N_{\text{log}}$ and lasts until the final time step N . The variables N_{lag} , N_{log} , N_{death} , K , D and F determine the behavior of the population growth and are properties of the environment the bacteria grow in. Finally, note how we use the variable n in all the `while` loops to keep track of the time steps.

Variables that influence how the model behaves, are called model parameters or just parameters. We can change these variables to simulate how the bacteria grow in another environment. To compare our full model to the measured bacterial population growth in the previous chapter, we find these numbers from Figure 3.5.

Variable	meaning	Value
N_{lag}	Time steps in the lag phase	4
N_{log}	Time steps in the logarithmic growth phase	36
N_{death}	Time steps in the death phase	53
N	Total number of time steps	$N_{\text{lag}} + N_{\text{log}} + N_{\text{death}}$
K	Carrying capacity	8×10^9
r	Growth rate	0.035
D	Death rate	0.014
F	Final population size	3×10^5
$E[0]$	Initial condition	10042

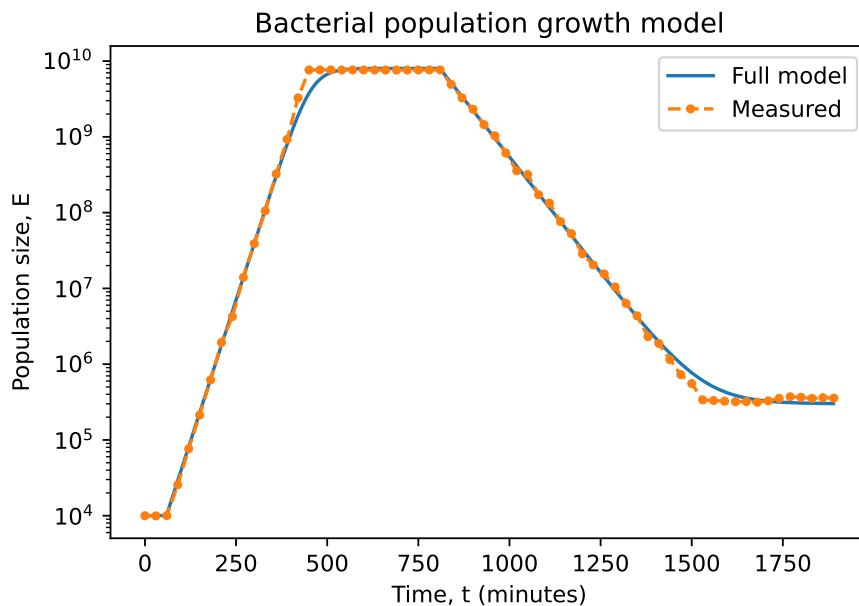
We use the values from the above table and plot the final result along with the measured data:

```

1  from pylab import *
2  import pandas
3
4  N_lag = 60      # Time steps in the lag phase
5  N_log = 750     # Time steps in the logarithmic growth phase
6  N_death = 1080  # Time steps in the death phase
7
8  # Total number of time steps
9  N = N_lag + N_log + N_death
10
11 K = 8e9          # carrying capacity
12 r = 0.035        # growth rate
13 D = 0.014        # death rate
14 F = 3e5          # final population size
15
16 E = []
17 t = []

```

```
18 E.append(10042)
19 t.append(0)      # minutes
20
21 # perform lag phase modeling
22 n = 1
23 while n < N_lag:
24     E_n = E[n-1]
25     E.append(E_n)
26     t_n = t[n-1] + 1
27     t.append(t_n)
28     n = n + 1
29
30 # perform logistic growth modeling
31 while n < N_lag + N_log:
32     E_n = (1 + r * (1 - E[n-1]/K)) * E[n-1]
33     E.append(E_n)
34     t_n = t[n-1] + 1
35     t.append(t_n)
36     n = n + 1
37
38 # perform death phase modeling
39 while n < N:
40     E_n = E[n-1] - D*(E[n-1] - F)
41     E.append(E_n)
42     t_n = t[n-1] + 1
43     t.append(t_n)
44     n = n + 1
45
46 # Reading in experimental values
47 data = pandas.read_csv("ecoli_all.csv")
48 t_measured = list(data["t"])
49 E_measured = list(data["E"])
50
51 # Plotting the modeled and measured data in the same plot
52 plot(t, E, "-", label="Full model")
53 plot(t_measured, E_measured, "--.", label="Measured")
54 xlabel("Time, t (minutes)")
55 ylabel("Population size, E")
56 title("Bacterial population growth model")
57 yscale("log")
58 legend()
59 show()
```



We have now modeled the entire bacterial population growth cycle. The result is not perfect, but quite good considering how simple our model is, relatively speaking. To get a feeling for how the different parameters affect this complete model you should vary the parameters and play around with them to see how that affects the final result. It is important to know that the modeling we have performed in this chapter is not specific to bacterial growth, and the same models can be used on many other types of population dynamics. The methods you have learned can also be extended and adapted to many new systems. This is what we are going to do in the next chapter, where we use the same methods to model plant growth, which has a slightly different behavior than the bacterial population growth studied here.

4.5 Summary

In this chapter you have learned to model all phases of the bacterial population growth cycle.

4.5.1 Mathematical symbols and their corresponding variable names

The different symbols we have used in this chapter are in the table below:

Symbol	Variable	Meaning
r	r	Growth rate
n	n	Current time step
N	N	Number of time steps
E	E	Number of bacteria
E_n	$E[n]$	Number of bacteria at a given time step n
t	t	Time
t_n	$t[n]$	Time at a given time step n
E_0	$E[0]$	Initial condition, number of bacteria we start with
K	K	Carrying capacity
D	D	Death rate
F	F	Final population size

4.5.2 Lag phase

The lag phase is the first phase where the bacteria adapt to a new environment before they start to reproduce.

Assumptions. The population does not change.

Equation describing the model.

$$E_n = E_{n-1}, \quad (4.61)$$

$$t_n = t_{n-1} + 1 \text{ (minute)}. \quad (4.62)$$

Implementation of the model. The core loop of this model is:

```

n = 1
while n < N:
    E_n = E[n-1]
    E.append(E_n)
    t_n = t[n-1] + 1
    t.append(t_n)
    n = n + 1

```

4.5.3 Logistic growth

This model takes into account that there are limited resources in the environment. It models both the exponential growth phase and the stationary phase.

Assumptions. We assume that:

- there is exponential growth when there is a large amount of resources compared to the needs of the population, and

- there is a maximum sustainable population size.

Equation describing the model.

$$E_n = \left(1 + r \left(1 - \frac{E_{n-1}}{K}\right)\right) \times E_{n-1}, \quad (4.63)$$

$$t_n = t_{n-1} + 1 \text{ (minute)}, \quad (4.64)$$

Implementation of the model. The core loop of this model is:

```
n = 1
while n < N:
    E_n = (1 + r * (1 - E[n-1]/K)) * E[n-1]
    E.append(E_n)
    t_n = t[n-1] + 1
    t.append(t_n)
    n = n + 1
```

4.5.4 Simple death phase

A fraction of the bacteria die each time step. Does not take into account that the bacterial population growth reaches a new stationary phase after a while and instead continues to decline.

Assumptions. We assume that the number of bacteria is reduced by a fixed fraction each time step.

Equation describing the model.

$$E_n = (1 - D)E_{n-1}, \quad (4.65)$$

$$t_n = t_{n-1} + 1 \text{ (minute)}. \quad (4.66)$$

Implementation of the model. The core loop of this model is:

```
n = 1
while n < N:
    E_n = (1 - D) * E[n-1]
    E.append(E_n)
    t_n = t[n-1] + 1
    t.append(t_n)
    n = n + 1
```

4.5.5 Advanced death phase

This model takes into account that the population reaches a long-term stationary phase.

Assumptions. We assume that:

- there is a final stable population, where there is no change in the number of bacteria, and
- when the population is much larger than the final population, the number of bacteria is reduced by a fixed fraction each time step.

Equation describing the model.

$$E_n = E_{n-1} - D(E_{n-1} - F), \quad (4.67)$$

$$t_n = t_{n-1} + 1 \text{ (minute).} \quad (4.68)$$

Implementation of the model. The core loop of this model is:

```
n = 1
while n < N:
    E_n = E[n-1] - D*(E[n-1] - F)
    E.append(E_n)
    t_n = t[n-1] + 1
    t.append(t_n)
    n = n + 1
```

4.5.6 Complete model of the bacterial population growth

A complete model consists of the lag phase model, the logistic growth model, and the advanced death phase model. The three core loops of this complete model are:

```
N_lag = 60      # Time steps in the lag phase
N_log = 750     # Time steps in the logarithmic growth phase
N_death = 1080   # Time steps in the death phase

# Total number of time steps
N = N_lag + N_log + N_death

K = 8e9          # carrying capacity
r = 0.035        # growth rate
D = 0.014        # death rate
F = 3e5          # final population size

E = []
t = []

E.append(10042)
t.append(0)       # minutes

# perform lag phase modeling
n = 1
while n < N_lag:
    E_n = E[n-1]
    E.append(E_n)
    t_n = t[n-1] + 1
    t.append(t_n)
    n = n + 1
```

```
# perform logistic growth modeling
while n < N_lag + N_log:
    E_n = (1 + r * (1 - E[n-1]/K)) * E[n-1]
    E.append(E_n)
    t_n = t[n-1] + 1
    t.append(t_n)
    n = n + 1

# perform death phase modeling
while n < N:
    E_n = E[n-1] - D*(E[n-1] - F)
    E.append(E_n)
    t_n = t[n-1] + 1
    t.append(t_n)
    n = n + 1
```

Chapter 5

Modeling plant population growth



Figure 5.1: Peas are an annual plant [10].

Farming is important for sustaining and enhancing human life. At the same time, farming and exploitation of forests have a strong impact on the environment. Many environmental projects therefore aim to balance the natural resources needed by future generations, while keeping a level of production that meets the needs of the present. Mathematical models are often used to obtain this balance. Such models give us the ability to make predictions about the future, based on

the assumptions and parameters in the model. The parameters describe various environmental factors, physiological factors, or both, and depend on the level of detail in the model.

In this chapter, we use the same techniques as in the previous chapter, this time to model the growth of a plant population. In particular, we study the growth of the pea plant *Pisum sativum*, which is an *annual plant*, meaning it completes its life cycle in one year and then dies. Pea plants start to grow from a seed once the temperature, water, and light conditions are right. They grow during spring and summer, and once autumn arrives, spread their seeds before they die. During winter some of the seeds may die due to environmental factors such as weather or being eaten by animals. Among those who survive the winter, a certain fraction *germinates*, that is, grows into a new plant. These give rise to a new generation of plants and the cycle starts over.

In order to survive as a species, it is critical that a sufficiently large group of the seeds survive the winter and germinate from generation to generation. Note that years and generations are used interchangeably in this chapter as each generation lasts one year.

We will describe the dynamics of such an annual plant using a computational model. We start out with a simple model and extend this to explore the effects of biological and environmental factors.



Learning outcomes

After working with this chapter, you know how to create and implement the following models and examine them analytically:

- the growth of an annual plant, and
- the growth of an annual plant, taking survival of seeds during winter into consideration.

The programming concepts we introduce in this chapter are the `range()` function and subplots.

5.1 One-year model for plant population growth

We want to formulate a model for how the pea population grows over time based on the life cycle of these plants. In particular, we consider two different cases. First, we assume that seeds only survive one winter, but later in this chapter we study the case where seeds can survive two winters. We call the first case the *one-year model* and this model has the following assumptions:



Model assumptions for the one-year model

We assume that:

- all plants are identical,
- seeds can only survive one winter,
- the fraction of seeds that survive a winter is constant, and
- the fraction of seeds that germinate is constant.

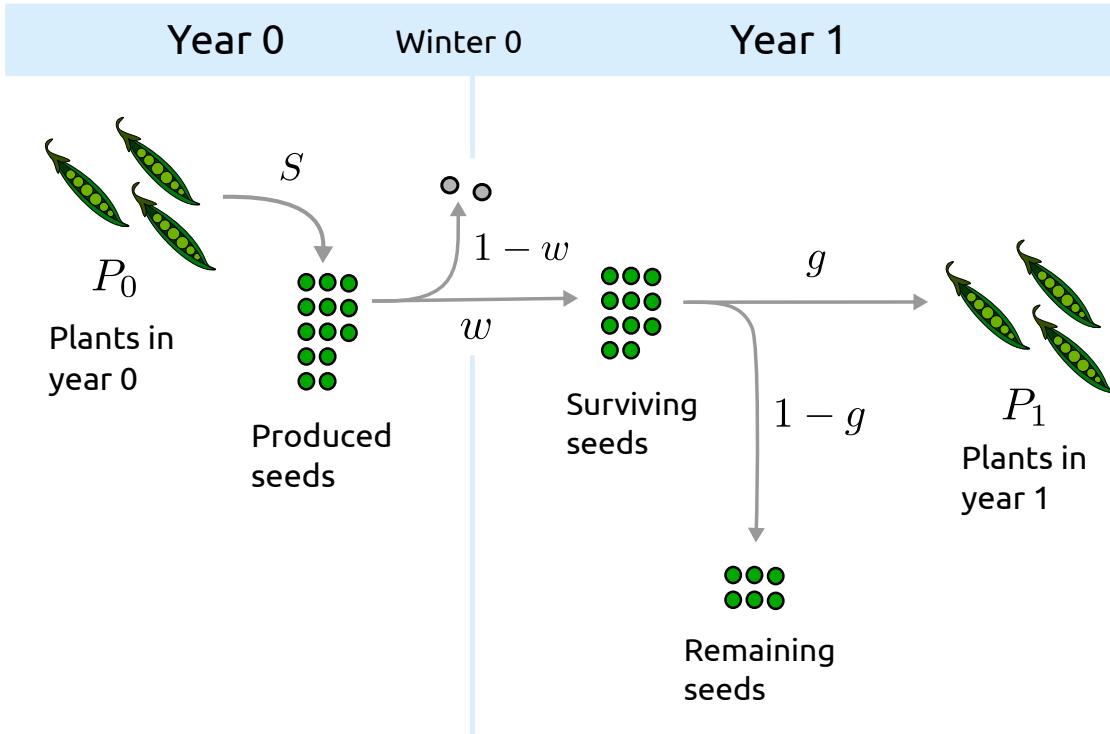


Figure 5.2: Overview of the steps in the model for annual growth of pea plants shown from year 0 to year 1. At year 0, the number of pea plants is P_0 . Each of these plants produce S seeds. During the winter only a fraction w of the produced seeds survive, while the rest die due to environmental factors. Among those seeds that survive the winter, a certain fraction g germinate in the following spring to give rise to a new population of plants P_1 . Note that we can use the fraction w of seeds that survive a winter to find the fraction of seeds that die during the winter as $1 - w$. The same is true for the germination rate.

Figure 5.2 shows the steps involved in calculating the production of plants from year 0 to year 1. The steps are equivalent for the following years. We introduce the terms in this figure in the following sections and have listed them here for reference:

Symbol	Meaning
n	Generation (year)
N	Number of generations (years)
P	Number of pea plants
P_n	Number of pea plants in year n
P_0	Number of pea plants in year 0
P_1	Number of pea plants in year 1
w	Fraction of seeds that survive the winter (survival rate)
g	Fraction of seeds that germinate (germination rate)
S	Number of seeds each pea plant produces

Use this table and figure as a reference while reading to keep track of the different terms.

5.1.1 Implementing the first year

Our goal is to create a model for the number of plants P_n in year n . To begin with, we calculate the number of plants after one year, P_1 . Then we generalize this calculation to obtain the number of plants in year n from the number in year n_1 , and step through each year by using a `while` loop. This allows us to find the number of plants P_n for all years.

We begin by importing the `pylab` package and set up a list `P` that will contain the number of plants for each year. Here, we want to calculate the population over $N = 10$ years including the first year (which is year 0):

```
from pylab import *
N = 10
P = []
```

Next, we set up a list `t` that contains the years. For this, we use the `range()` function. Using `range(N)` produces a range of numbers from zero up to, but not including, N :

```
t = range(N)
```

If we try to print the content of `t`, however, we do not get the numbers in this range, but Python does let us know what the range of the numbers is:

```
print(t)
```

```
| range(0, 10)
```

For plotting, this does not matter and we can use `t` directly. However, if we are interested in the numbers assigned to `t` as a list, we first need to convert it using the `list()` function:

```
print(list(t))
```

```
| [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In this section, we use some reasonable numbers for the initial number of plants, survival rate, and so on, but in Section 5.1.6, we go into detail on how these parameters can be calculated from experiments. We assume that the number of pea plants in the first year is $P_0 = 20$:

```
P.append(20)
```

We multiply this by the number of seeds produced by each plant, $S = 50$, to find the total number of seeds produced:

$$\text{produced seeds} = SP_0. \quad (5.1)$$

We express this in code as:

```
S = 50
produced_seeds = S * P[0]
print(produced_seeds)
```

```
1000
```

In this chapter, we combine more descriptive variable names, such as `produced_seeds`, with names that lie close to the mathematics, such as `S`, to make it easier to understand each line of code.

Only a fraction $w = 0.25$ of these seeds survive the winter, while the rest die due to disease, weather, or being eaten by animals. Thus, the number of seeds after one winter is:

$$\text{surviving seeds} = w \times \text{produced seeds}. \quad (5.2)$$

We express this in code as:

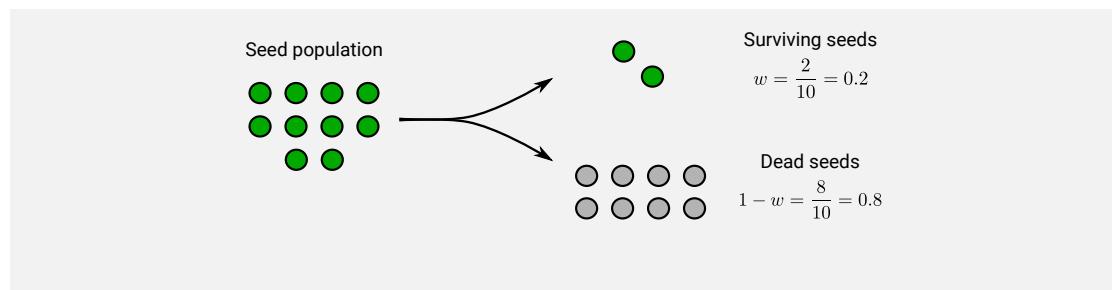
```
w = 0.25
surviving_seeds = w * produced_seeds
print(surviving_seeds)
```

```
250.0
```



Reminder: Fractions

A fraction represent an amount or proportion of something and can be represented as a decimal number or a percent. For example if 2 of 10 seeds survive a winter, the fraction of seeds that survive (survival rate) is $\frac{2}{10} = 0.2$ or 20%. The fraction of seeds that die during the winter is $1 - 0.2 = 0.8$ or 80%.



After the winter, the surviving seeds can germinate and grow into mature plants. However, only a fraction $g = 0.16$ of the surviving seeds germinate, while the rest stay dormant. Some might not have the right conditions to sprout, being buried too deep or lacking water.

The seeds that germinate grow into mature plants, giving rise to a new generation of pea plants. The number of plants next year is:

$$P_1 = g \times \text{surviving seeds.} \quad (5.3)$$

In code, we find the actual number of plants:

```
g = 0.16
P_1 = g * surviving_seeds
print("number of plants after one year:", P_1)
```

```
| number of plants after one year: 40.0
```

These plants survive until fall, where they each produce a number of seeds before they die. This cycle starts again after the following winter.

The complete program containing all the steps is:

```
from pylab import *
N = 10      # Number of generations
S = 50      # seed produced per plant
w = 0.25    # survival rate
g = 0.16    # germination rate

P = []
t = range(N)

P.append(20)
produced_seeds = S * P[0]
surviving_seeds = w * produced_seeds

P_1 = g * surviving_seeds
P.append(P_1)

print(P)
```

```
| [20, 40.0]
```

As you can see from the output above, we have the number of plants for the first two years. Note how the list contains first a number of type integer, then one of type float. This is because when we multiply numbers in Python, and at least one of these is of type float, then the result of the calculation becomes a float.

To calculate the population size for the remaining years we need to repeat the steps above, first by using P_1 to calculate P_2 , then use P_2 to calculate P_3 , and so on.

5.1.2 Implementing the following years

It is cumbersome to manually perform the above calculation year by year. We therefore again use a `while` loop to calculate the number of plants in the following years. For each generation, we calculate the number of pea plants by:

- multiplying the number of plants from last year, P_{n-1} , by the number of seeds per plant,

$$\text{produced seeds} = SP_{n-1}, \quad (5.4)$$

- multiplying the number of produced seeds by the fraction of seeds that survive the winter,

$$\text{surviving seeds} = w \times \text{produced seeds}, \quad (5.5)$$

- and multiplying the number of surviving seeds by the fraction of seeds that germinate,

$$P_n = g \times \text{surviving seeds}. \quad (5.6)$$

We combine all these steps in a loop over year 1 to N , because we already know the population size in the first generation, P_0 . Note how we set up the variable containing the years t before the loop, and do not add anything to it inside the loop. By using `t = range(N)` we have already filled t with everything we need. In the end, we plot and print the results:

```
from pylab import *

N = 10    # Number of generations
S = 50    # seed produced per plant
w = 0.25  # survival rate
g = 0.16  # germination rate

P = []
P.append(20)

t = range(N)

n = 1
while n < N:
    produced_seeds = S * P[n-1]
    surviving_seeds = w * produced_seeds
    P_n = g * surviving_seeds
    P.append(P_n)
    n = n + 1

plot(t, P, "-o")
xlabel("Generations, t")
ylabel("Number of pea plants, P")
title("Simulated plant population")
```

```
show()
print(P)
```

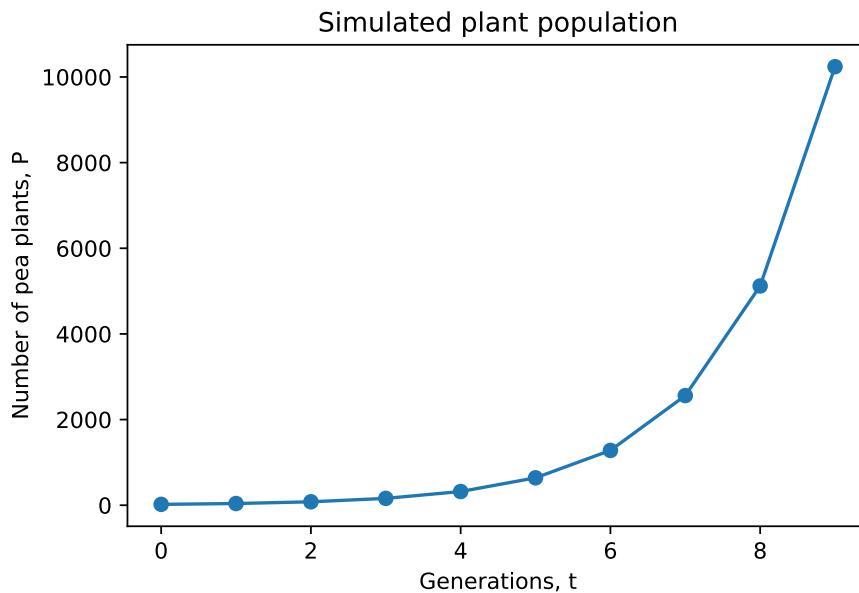


Figure 5.3: Pea plant population growth over multiple years.

```
[20, 40.0, 80.0, 160.0, 320.0, 640.0, 1280.0, 2560.0, 5120.0, 10240.0]
```

As you see from the figure above, the model predicts an increasing growth, similar to the exponential growth phase in the bacterial population growth. When we combine the terms in this model into one equation it is easier to see why we get this behavior.



Summary: The one-year model

In the one-year model, g is the germination rate, w is the survival rate, and S is the number of seeds per plant.

For each generation, we calculate the number of pea plants by:

- multiplying the number of plants from last year, P_{n-1} , by the number of seeds per plant,

$$\text{produced seeds} = SP_{n-1}, \quad (5.7)$$

- multiplying the number of produced seeds by the fraction of seeds that survive the winter,

$$\text{surviving seeds} = w \times \text{produced seeds}, \quad (5.8)$$

- and multiplying the number of surviving seeds by the fraction of seeds that germinate,

$$P_n = g \times \text{surviving seeds}. \quad (5.9)$$

The model parameters we use are:

$$\begin{aligned} P_0 &= 20, \\ g &= 0.16, \\ w &= 0.25, \\ S &= 50. \end{aligned}$$

5.1.3 Expressing the model as a difference equation

In this chapter, we have more terms than in the simple exponential bacterial population growth model and therefore use longer variable names in the code to keep track of the intermediate steps. However, it is sometimes useful to combine all these steps together in a single equation to get a better overview. To find the number of plants P_n in a single equation, we could multiply the number of seeds produced per plant S , the winter survival rate w , and the germination rate g with the number of plants P_{n-1} from last year:

$$P_n = gwSP_{n-1}. \quad (5.10)$$

This puts the solution in a form that shows that we are working with a difference equation, as in the previous chapters on bacterial modeling.

In programming, you sometimes have to choose between doing multiple calculations on a single line, or step by step. It is a balance between explicitly including all details and keeping the code short. In the end, the goal is to make the code easy to read. Having too many lines of code with long variable names makes it hard to get an overview, while having everything on one line makes it complicated.

In this chapter, we have chosen a mix of short variable names for the most important things, and long variable names for the intermediate steps. Keep this in mind while reading the rest of the chapter and think about whether you prefer shorter names and more calculations on one line, or if you prefer longer names and more intermediate steps.

Implementing Equation (5.10) gives us the same result as we get by writing it in multiple steps:

```
P = []
t = range(N)

P.append(20)
n = 1
while n < N:
    P_n = g*w*S*P[n-1]
    P.append(P_n)
```

```
n = n + 1
print(P)
[20, 40.0, 80.0, 160.0, 320.0, 640.0, 1280.0, 2560.0, 5120.0, 10240.0]
```

Looking at Equation (5.10) it is easier to understand why we get the same exponential growth as in the exponential bacterial population growth model. We multiply a number (gwS) with the previous number of plants (P_{n-1}) to get the current number of plants. With current model parameters this number is

$$gwS = 0.16 \times 0.25 \times 50 = 2. \quad (5.11)$$

This product is multiplied with the current population size to get the corresponding size for the next generation.

$$P_n = 2P_{n-1}. \quad (5.12)$$

This is the same equation as the equation for the exponential bacterial population growth model, $E_n = 2E_{n-1}$. This means the plant population doubles each generation, as seen from the model results. Note that the two models are equal only with the current set of parameters.

5.1.4 Examining low survival rate

Let us examine what happens if the winters are much colder and only 8% of the seeds survive each winter, instead of the 25% which originally survived. In order to do that, we need to update the value of `w` and recalculate the population growth:

```
w = 0.08 # fraction of the seeds that survive the winter      # New
P = []
t = range(N)

P.append(20)
n = 1
while n < N:
    produced_seeds = S * P[n-1]
    surviving_seeds = w * produced_seeds
    P_n = g * surviving_seeds
    P.append(P_n)
    n = n + 1

plot(t, P, "-o")
xlabel("Generations, t")
ylabel("Number of pea plants, P")
title("Plant population, low survival rate")
show()
```

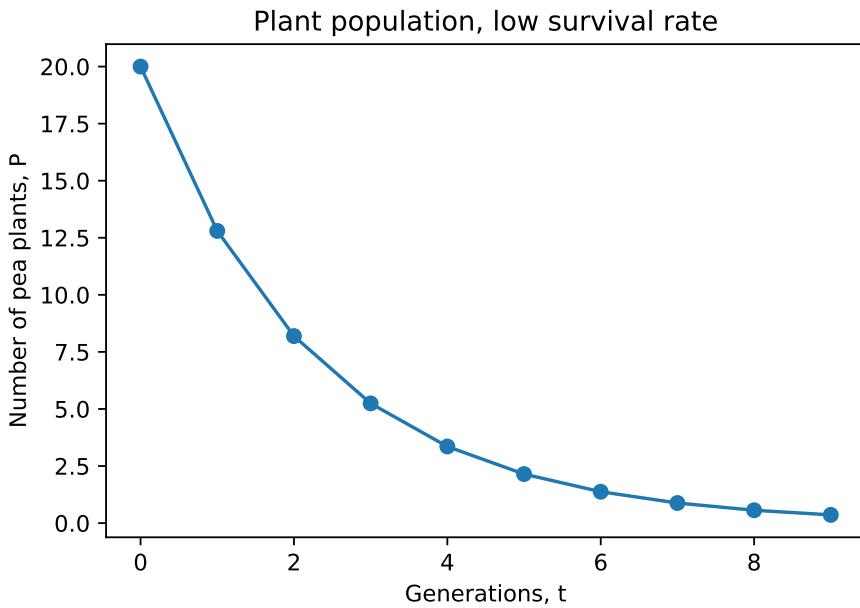


Figure 5.4: Pea plant population growth with lower survival rate.

The predicted plant population is very different from the previous case. The current survival rate is so low that the number of plants decreases with time. More seeds die through the winter and there are fewer seeds to germinate. The population is no longer able to sustain itself and we observe an exponential decay in the number of plants.

The decrease could also have been predicted from the product

$$gwS = 0.16 \times 0.08 \times 50 = 0.64. \quad (5.13)$$

This means that in each generation the size of the population is 64% of the previous generation. The population therefore decreases (a case of exponential decay) and dies out within the span of 10 years.

This result also highlights a problem with our model, namely that we end up with non-integer number of plants:

```
print(P)
```

```
[20, 12.8, 8.192, 5.24288, 3.355443200000003, 2.147483648, 1.3743895347200001, 0.8796093022208001,
 0.5629499534213122, 0.36028797018963976]
```

This does not make any sense, because we cannot have a fraction of a plant. One solution to this problem is to round all the numbers to integers using the `round()` function. However, we simply ignore this problem in this chapter to keep the code as simple as possible.

5.1.5 Examining different survival and germination rates

Depending on our parameters, we either observed exponential growth or exponential decay. Let us visualize how the growth depends on the parameters g and w . We only describe how this is done, because the code to visualize this is a bit too complicated to explain at this point.

We choose different values between 0 and 1 for both the germination rate and the survival rate. These values are the x - and y -axes in our plot. We then simulate the plant growth for any one combination of germination rate and survival rate and see if the population size increases or decreases. The plot is colored depending on whether we have growth or decay. If the population increases, the color is green. If the population decreases, the color is white. The result is shown in Figure 5.5.

This type of plot is called a *phase diagram*. To see if a set of parameters g and w gives an increase or a decrease in the population, find the point in the plot that corresponds to these values, and check if the color is green or white. If the point is green, using that germination rate and survival rate in our models gives us an increase in the final population, and we get exponential growth. If the point is white, the final population is smaller than the initial population and we get an exponential decay.

For example, if we have a survival rate $w = 0.4$ and a germination rate $g = 0.8$ we have a green point, and the final plant population is greater than the initial population. Using phase diagrams is a very efficient way of examining a model, and helps us to learn more on how the model behaves.

Alternatively, we can predict an increase or decrease by calculating gwS :

- If $gwS > 1$, the plant population *increases* over successive generations.
- If $gwS = 1$, the plant population *does not change*.
- If $gwS < 1$, the plant population *decreases* over successive generations.

Examples of these three scenarios are illustrated in Figure 5.6. In the first case with $gwS = 2$, the population doubles in every generation, while with $gwS = 0.5$ the population is reduced by 50% in every generation. When $gwS = 1$, the population neither increases or decreases.

5.1.6 Estimating model parameters

By doing experiments, we can measure the actual number of seeds each plant produces S , the winter survival rate w , and the germination rate g for a certain environment. These experiments are quite simple, they just take some time to perform. Here, we outline the calculations and measurements you have to perform to experimentally find the parameters.

Seeds per plant. We measure S by counting the total number of seeds and divide by the number of plants:

$$S = \frac{\text{number of seeds produced}}{\text{number of plants}}. \quad (5.14)$$

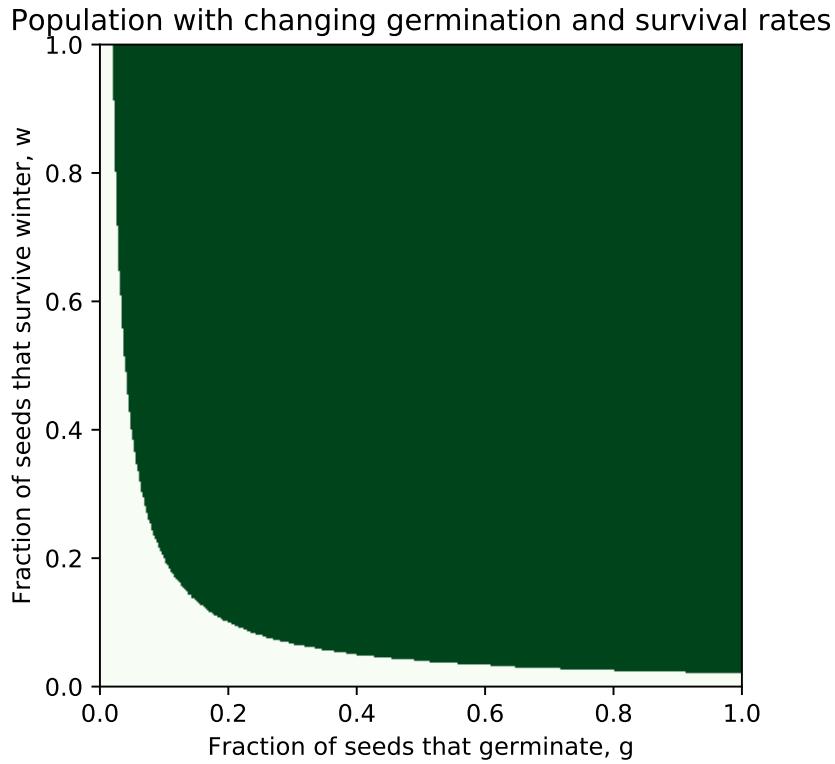


Figure 5.5: Effect of different survival and germination rates on the final population size. Green regions are where the parameter combination g , w , and $S = 50$ leads to exponential growth, while white regions are where the population decreases over successive generations.

Germination rate. To measure g , we plant a number of seeds in the spring and count the number of plants that germinate:

$$g = \frac{\text{number of seeds that germinated}}{\text{number of seeds planted in the spring}}. \quad (5.15)$$

Survival rate. To measure w , we plant a number of seeds before the winter starts and measure the number of plants that germinate. This fraction corresponds to plants that both survived the winter and germinated, and is therefore equal to the product of the survival rate w and the germination rate g :

$$wg = \frac{\text{number of seeds that germinated}}{\text{number of seeds planted before winter}}. \quad (5.16)$$

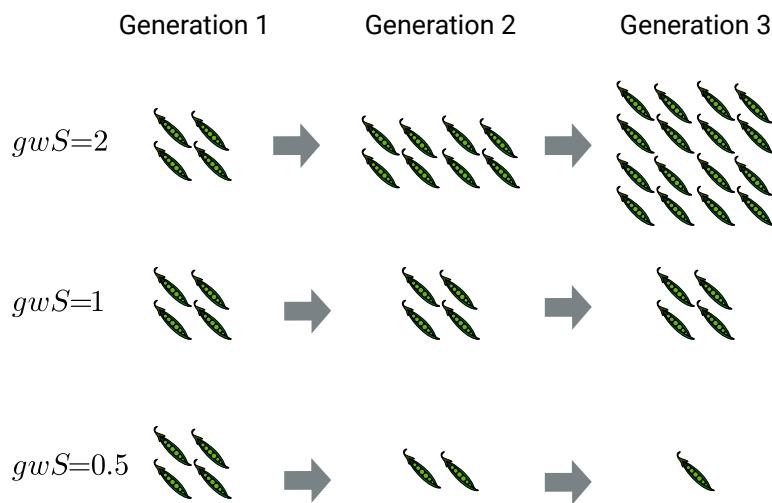


Figure 5.6: Different scenarios for annual plant population growth based on the magnitude of the product of seeds per plant (S), survival rate (w), and germination rate (g).

When we know the germination rate we calculate the winter survival rate:

$$w = \frac{\text{number of seeds that germinated}}{g \times \text{number of seeds planted before winter}}. \quad (5.17)$$

5.2 Variable germination rate

We have so far assumed that the fraction of seeds that germinate each year and survive each winter is constant. However, some years are harder on the seeds than others. Factors that can keep the seeds from germinating include drought and high temperature, cold springs, extremely wet environments, lack of sunlight, and lack of oxygen. In this chapter, we extend our model to include such effects by including a varying germination rate. Our model assumptions are then:



Model assumptions for the one-year model with variable germination

We assume that:

- all plants are identical,
- seeds can only survive one winter, and
- the fraction of seeds that survive a winter is constant.

We use a list with N items to store the germination rate for each year. The germination rate for a given year can then be picked from this list with $g[n]$ when we calculate the number of pea plants. In the first version of our program, we use the same value for each year as we used in the previous section. We do this to verify that we obtain the same result and do not introduce any bugs by using a list instead of a constant value:

```
from pylab import *

N = 10
S = 50
w = 0.25
g = [0.16, 0.16, 0.16, 0.16, 0.16, 0.16, 0.16, 0.16, 0.16, 0.16]      # New
P = []
P.append(20)

n = 1
while n < N:
    produced_seeds = S * P[n-1]
    surviving_seeds = w * produced_seeds
    P_n = g[n] * surviving_seeds
    P.append(P_n)
    n = n + 1

print(P)
```

```
[20, 40.0, 80.0, 160.0, 320.0, 640.0, 1280.0, 2560.0, 5120.0, 10240.0]
```

This produces the same result as before, and we most likely did not introduce any errors in our code by using a list.

What would happen if the plants experienced a lack of sunlight in year 3? Let us assume that it causes the germination rate to fall to 2% and plot the result:

```
g = [0.16, 0.16, 0.16, 0.02, 0.16, 0.16, 0.16, 0.16, 0.16, 0.16]      # New
#           ^ Changed
P = []
P.append(20)

n = 1
while n < N:
    produced_seeds = S * P[n-1]
    surviving_seeds = w * produced_seeds
    P_n = g[n] * surviving_seeds
    P.append(P_n)
    n = n + 1

plot(t, P, "-o")
xlabel("Generations, t")
ylabel("Number of pea plants, P")
title("Plant population, reduced sunlight")
show()
```

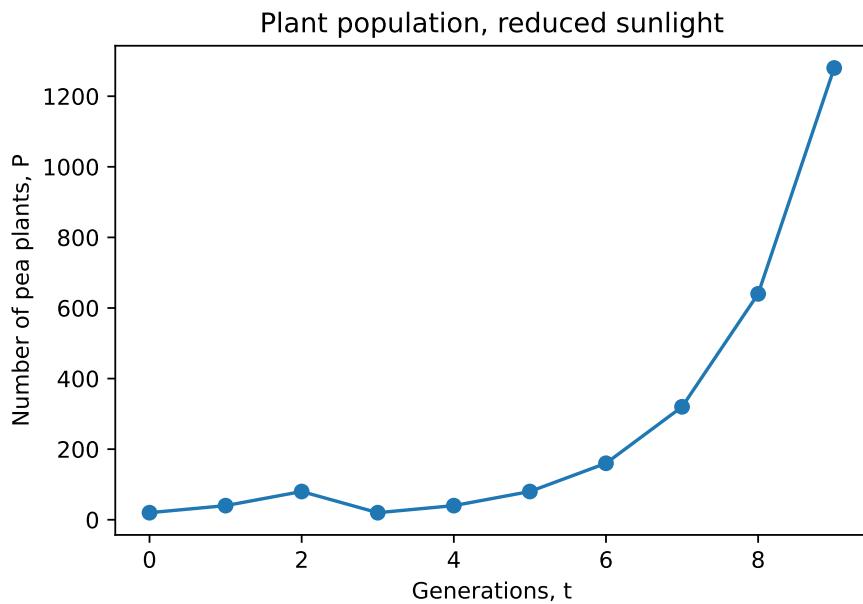


Figure 5.7: Pea plant growth with one year of reduced sunlight.

The figure above shows that this one rough year with little sunlight sets back the number of pea plants almost to the initial number of plants, and the exponential growth starts over from this value. Within the 10 years we simulate, one such setback reduces the final number from around 10,000 plants to just over 1,200 plants. This illustrates that even one bad summer can have a long-lasting effect on the pea population.

What about many years of drought, taking us from normal germination rate all the way down to zero in year 5 and then slowly recovering back to normal germination rate? Let us write a list of germination rates going from 0.16, all the way down to zero and back up to 0.16, and see how the plant population develops:

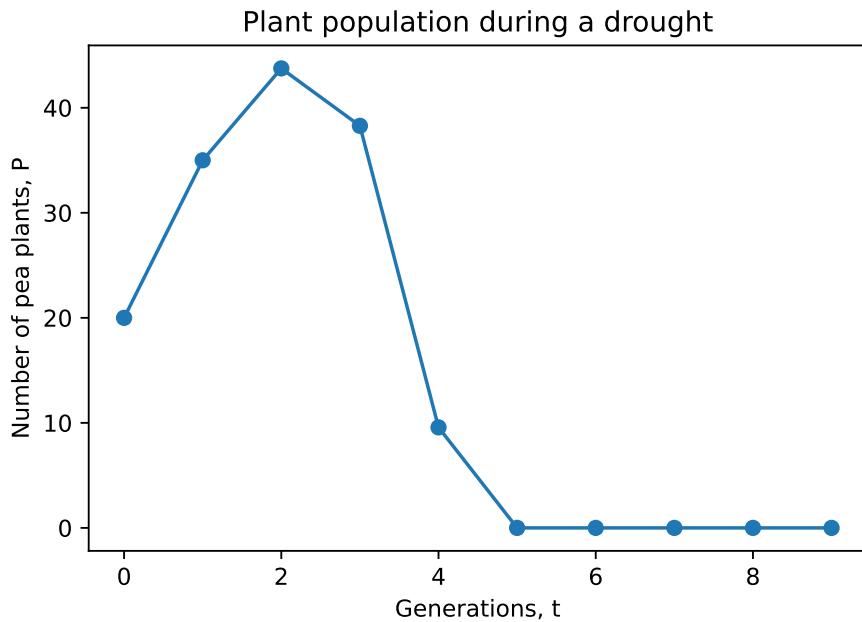
```

g = [0.16, 0.14, 0.10, 0.07, 0.02, 0.0, 0.08, 0.12, 0.15, 0.16]           # New
P = []
P.append(20)

n = 1
while n < N:
    produced_seeds = S * P[n-1]
    surviving_seeds = w * produced_seeds
    P_n = g[n] * surviving_seeds
    P.append(P_n)
    n = n + 1

plot(t, P, "-o")
xlabel("Generations, t")
ylabel("Number of pea plants, P")
title("Plant population during a drought")
show()

```



The plant population initially grows before the drought increases in severity and starts reducing the number of plants year by year. Eventually, no seeds germinate in the year with zero germination rate, which kills the entire population and results in no plants in the following years.

Models like this can make predictions about the number of pea plants that will grow in the coming years, but it requires that we know the germination rate for each year. This, in turn, means we need to predict the severity of the coming winters and possible droughts, and that is the hard problem of weather forecasting which meteorologists are trying to solve.

A model is only as good as the assumptions we put into it, but it can still be useful. Even though we cannot reliably predict exactly how many pea plants we will have in ten years, we can collect data on germination rates from previous years and make an estimate for the coming years. We can, for instance, answer questions like: “How many pea plants can we expect if we have multiple years of wet summers?” The answer can be used to prepare ourselves for different scenarios that might play out.

To improve our model, we should also consider other variables that may be varying from year to year. For instance, we still assume the winter survival rates are constant over time. The severity of winters vary, and we could implement varying survival rates in our model similarly to how we implemented the varying germination rate above.

We will now expand our one-year model in the next section to make it more realistic. In order to compare the new model with one-year model, we store the results from the calculations of the number of plants with variable generation rates in a new list called `P_one_var`:

```
P_one_var = P.copy()
```

For this, we need to use the `.copy()` function we saw in Section 2.3.3 to make a copy of the elements of the list, rather than a new variable that refers to the same list.

5.3 Two-year model for plant population growth

In the previous section, we created a simple model for the growth of pea plants. We assumed that no seeds survived more than a year, but in reality, seeds can survive for many years. This is a survival strategy that can keep the population alive through long periods of drought, such as the one which killed our entire plant population in the previous section.

In this section, we expand the model to include seeds that survive two years. We call this new model “the two-year model”. We test this model with a constant germination rate, before we introduce the same drought as modeled in the previous section. We will ask the question: will the plant population survive the drought period we modeled in the previous section, when it includes seeds that survive two years?

The following are the assumptions in this model:



Model assumptions for the two-year model

We assume that:

- all plants are identical,
- seeds can survive two winters, and
- the fraction of seeds that survive a winter is constant.

The symbols used in this model are shown in the following table:

Symbol	Description
n	Generation/year
N	Number of generations/years
P_n	Number of pea plants in year n
w	Fraction of seeds that survive the winter
g_n	Fraction of seeds that germinate in year n
S	Number of seeds each pea plant produces

5.3.1 Adding two-year-old seeds to the model

Let us begin by defining the variables from the previous section:

```
from pylab import *
N = 10
```

```
S = 50
w = 0.25
g = [0.16, 0.16, 0.16, 0.16, 0.16, 0.16, 0.16, 0.16, 0.16]
P = []
P.append(20)
```

When seeds can survive for two years P_n becomes the number of pea plants that germinate from one-year-old seeds, $P_{n, \text{one-year-old}}$, plus the number of pea plants that germinate from two-year-old seeds, $P_{n, \text{two-year-old}}$:

$$P_n = P_{n, \text{one-year-old}} + P_{n, \text{two-year-old}}. \quad (5.18)$$

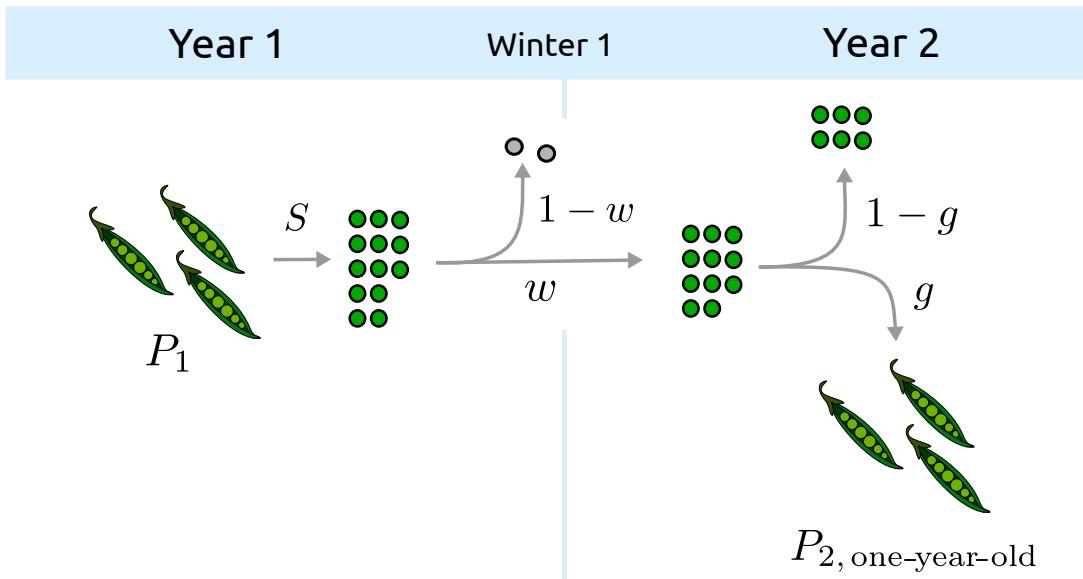


Figure 5.8: The contribution from one-year-old seeds is the same as in the previous sections. This is the same process as in figure 5.2, but specified from year 1 to year 2.

We build up the loop piece by piece to introduce the two contributions. The contribution from one-year-old seeds is calculated the same way as in the previous sections. Figure 5.8 shows this for year 1 to year 2. This time we store the number of pea plants from one-year-old seeds in a variable called `plants_1_year`. We add this variable to `P[n]`, and add a comment that we will include `plants_2_year` later. This comment starts with `TODO`, which is a common way to indicate to both yourself and others that changes are going to be made in the code later.

```
P = []
P.append(20)
n = 1
while n < N:
    produced_seeds = S * P[n-1]
    surviving_seeds = w * produced_seeds
    plants_1_year = g[n] * surviving_seeds
```

```

P_n = plants_1_year
P.append(P_n)

n = n + 1

print(P)

```

```
[20, 40.0, 80.0, 160.0, 320.0, 640.0, 1280.0, 2560.0, 5120.0, 10240.0]
```

To find the contribution from the two-year-old seeds, we need to know the number of one-year-old seeds that did not germinate and remained in the ground. We therefore want to store the number of seeds that remained in the ground in a list called `remaining_seeds`, with 0 as initial value:

```

remaining_seeds = []
remaining_seeds.append(0)

```

The first year there are zero remaining seeds. The number of one-year-old seeds that did not germinate is:

$$\text{remaining seeds} = (1 - g_n) \times \text{surviving seeds}. \quad (5.19)$$

We append this result to the `remaining_seeds` list. Finally, we print it to check the result:

```

P = []
P.append(20)
remaining_seeds = []
remaining_seeds.append(0)

n = 1
while n < N:
    produced_seeds = S * P[n-1]
    surviving_seeds = w * produced_seeds
    plants_1_year = g[n] * surviving_seeds

    remaining_seeds_n = (1 - g[n]) * surviving_seeds      # New
    remaining_seeds.append(remaining_seeds_n)              # New

    P_n = plants_1_year                                    # TODO: add plants_2_year
    P.append(P_n)

    n = n + 1

print(remaining_seeds)

```

```
[0, 210.0, 420.0, 840.0, 1680.0, 3360.0, 6720.0, 13440.0, 26880.0, 53760.0]
```

We now use `remaining_seeds` to calculate the number of plants that germinate from these seeds. Of these seeds, only a fraction w are able to survive the second winter, as shown in Figure 5.9.

The number of seeds we are left with after the second winter is

$$\text{seeds after the second winter} = w \times \text{remaining seeds}. \quad (5.20)$$

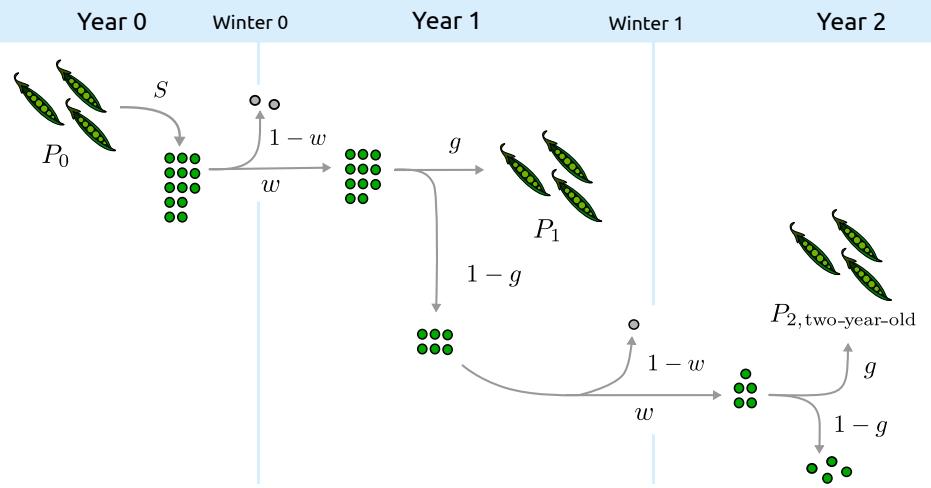


Figure 5.9: Only a fraction of the two-year-old seeds survive the second winter and are able to germinate. This gives us the number of plants produced from two-year-old seeds.

Then spring comes around again and a fraction g_n of these seeds germinate to become the pea plants from two-year-old seeds, $P_{n,\text{two-year-old}}$, shown in Figure 5.9.

$$P_{n,\text{two-year-old}} = g_n \times \text{surviving seeds after second winter.} \quad (5.21)$$

This term needs to be added to the number of plants from one-year-old seeds. We show this in Figure 5.10, where we sum the number of plants from one- and two-year-old seeds to find P_2 . Note that this figure only shows the first three years. Every following year is similar to year 2, because seeds only survive for two years. If we included even older seeds, this figure would have to be expanded.

We implement these calculations and plot the result in the following code:

```

P = []
P.append(20)
remaining_seeds = []
remaining_seeds.append(0)

n = 1
while n < N:
    produced_seeds = S * P[n-1]
    surviving_seeds = w * produced_seeds
    plants_1_year = g[n] * surviving_seeds

    remaining_seeds_n = (1 - g[n]) * surviving_seeds
    remaining_seeds.append(remaining_seeds_n)

    surviving_seeds_2_year = w * remaining_seeds[n-1]
    plants_2_year = g[n] * surviving_seeds_2_year
    # New
    # New

    P_n = plants_1_year + plants_2_year
    P.append(P_n)

```

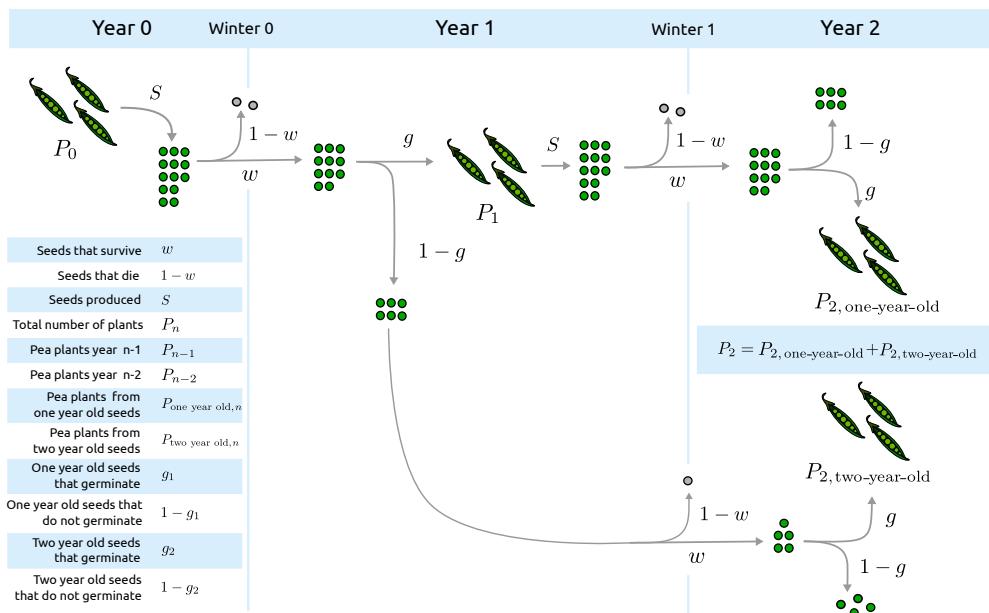


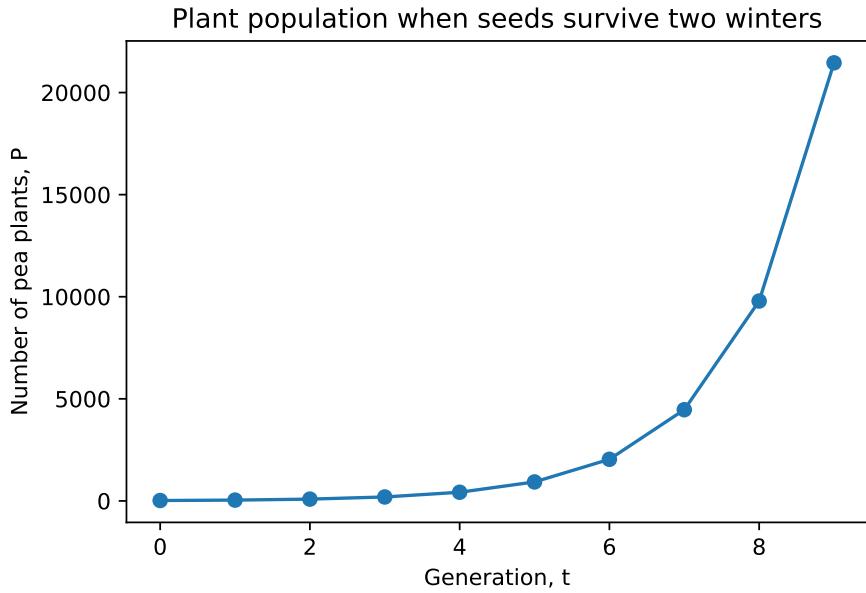
Figure 5.10: The complete two-year model shown from year 0 to year 2. The later years are similar to year 2 in that they get a contribution from both one- and two-year-old seeds.

```

n = n + 1

plot(t, P, "-o")
xlabel("Generation, t")
ylabel("Number of pea plants, P")
title("Plant population when seeds survive two winters")
show()

```



The above figure is similar to Figure 5.3, where we had exponential growth in the one-year model. However, we see that we end up with a different number of plants, closer to 20,000 plants in comparison to about 10,000 plants in the one-year model. The effect is not surprising, as the seeds have an additional chance to germinate so we should expect to get more pea plants. More interesting things happen when we reintroduce the drought from the previous section.



Summary: The two-year model

For each generation, we calculate the number of pea plants as

$$P_n = P_{n, \text{one-year-old}} + P_{n, \text{two-year-old}}, \quad (5.22)$$

where the one-year-old seeds are found by using the one-year model.

Further, we do the following:

- find number of remaining seeds that did not germinate,

$$\text{remaining seeds} = (1 - g_n) \times \text{surviving seeds}, \quad (5.23)$$

- multiply the number of remaining seeds by the fraction of seeds that survive the second winter,

$$\text{seeds after the second winter} = w \times \text{remaining seeds}, \quad (5.24)$$

- and multiply the number of seeds that survived the second winter with the fraction of seeds that germinate after two years,

$$P_{n, \text{two-year-old}} = g_n \times \text{surviving seeds after second winter.} \quad (5.25)$$

The model parameters we use, are

$$\begin{aligned} P_0 &= 20, \\ g_n &= 0.16, \\ w &= 0.25, \\ S &= 50. \end{aligned}$$

5.3.2 Variable germination in the two-year model

The drought we added in the one-year model took us down to zero germination rate in year five. After the drought, the germination rate slowly improved, but in the one-year model this was too late, because the entire plant population was already dead. What happens if seeds can survive two years?

We use the same germination rates as for the one-year model with drought, and plot the result. We will also add the results from the one-year model we developed earlier and stored in the variable `P_one_var`.

```

g = [0.16, 0.14, 0.10, 0.07, 0.02, 0.0, 0.08, 0.12, 0.15, 0.16] # New
P = []
P.append(20)
remaining_seeds = []
remaining_seeds.append(0)

n = 1
while n < N:
    produced_seeds = S * P[n-1]
    surviving_seeds = w * produced_seeds
    plants_1_year = g[n] * surviving_seeds

    remaining_seeds_n = (1 - g[n]) * surviving_seeds
    remaining_seeds.append(remaining_seeds_n)

    surviving_seeds_2 = w * remaining_seeds[n-1]
    plants_2_year = g[n] * surviving_seeds_2

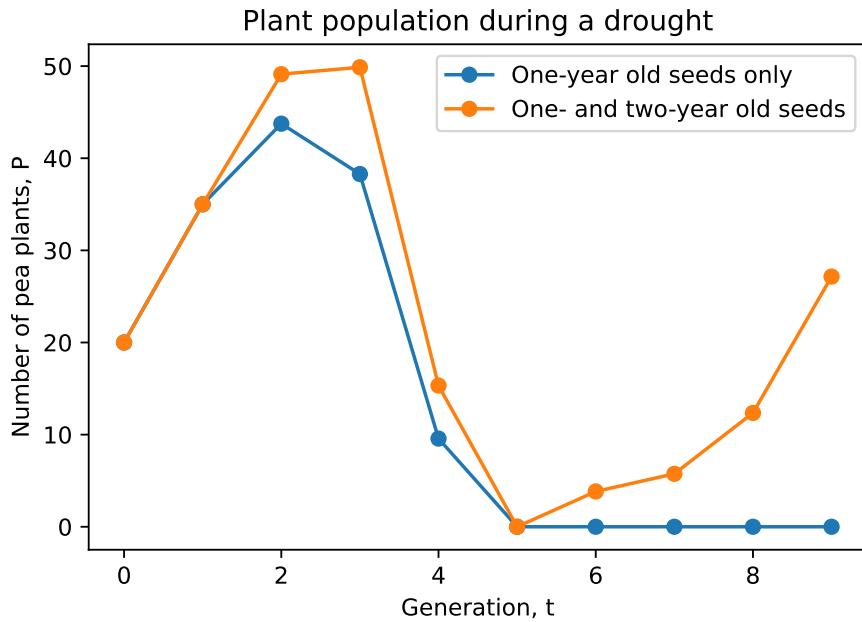
    P_n = plants_1_year + plants_2_year
    P.append(P_n)

    n = n + 1

plot(t, P_one_var, "-o", label = "One-year old seeds only")
plot(t, P, "-o", label = "One- and two-year old seeds")
xlabel("Generation, t")
ylabel("Number of pea plants, P")

```

```
title("Plant population during a drought")
legend()
show()
```



For the first 5 years, the results of the two-year model look similar to the results found in the one-year model, except with a slightly higher number of plants. However, the really interesting change occurs after the year with zero germination rate: the plants start to grow again, even though no seeds germinated in year five.

5.3.3 Subplots: Showing two plots in the same figure

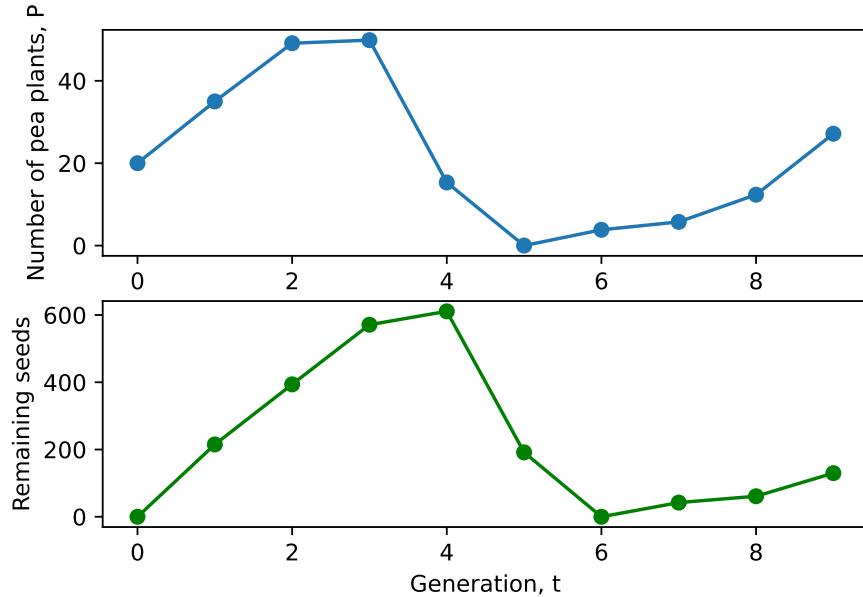
We more easily see how the growth starts again by plotting the number of remaining seeds along with the number of plants. However, because the ranges of those numbers are rather different, adding them to the same plot would difficult to interpret (we encourage you to try this out). It would be more informative to create a figure with each line in a separate plot, one below the other. This is done by calling `subplot()` before each plot. The `subplot()` function creates a table of plots. It takes three arguments: the number of rows, the number of columns, the current plot index. In our case, we plot two rows and one column, so we need to call `subplot(2, 1, 1)` for the first plot and `subplot(2, 1, 2)` for the second. We restrict the plots to the two-year model:

```
subplot(2, 1, 1)
plot(t, P, "-o")
xlabel("Generation, t")
ylabel("Number of pea plants, P")

subplot(2, 1, 2)
plot(t, remaining_seeds, "g-o")
```

```
xlabel("Generation, t")
ylabel("Remaining seeds")

show()
```



In this figure we see both the population size and remaining seeds over time.

- In year 5, we have the following situation:
 - the germination rate is zero,
 - thus the number of plants is zero, and so
 - all the one-year-old seeds from year 4 remain.
- In year 6:
 - some of the now two-year-old seeds (from year 4) survive the winter and germinate to give rise to a new generation of plants, however
 - since no plants were produced in year 5, there are zero one-year-old seeds, so the number of remaining seeds is zero.
- Then in year 7:
 - some one-year-old seeds (from year 6) survive the winter and germinate, but
 - since there were zero remaining seeds in year 6, there are zero two-year-old seeds that can germinate.

In the following years, we get a contribution from both one- and two-year-old seeds. With seeds that survive for more than one winter the pea plant population is able to survive the drought.

Many plants have evolved seeds that can survive for several years to get more chances to germinate. In nature, seeds can survive for more than two years, in order to germinate when the conditions are sufficient. However, in our model we ignore seeds that survive for more than two years.

Feel free to play around with the germination rates to see what happens if you have two consecutive years with zero germination. Will our plant population survive in this case or would we have to expand our model to include seeds that can survive for more than two years? In general, we can create a multi-year model by introducing a second loop that goes over all previous years and tries to germinate any remaining seeds. This is a bit complicated, we therefore do not introduce such a model in this chapter.

5.3.4 Expressing the two-year model as a difference equation

As with the one-year model, we can also write the two-year model more compactly as a difference equation. First, P_n can be written as

$$\begin{aligned} P_n &= P_{n, \text{one-year-old}} + P_{n, \text{two-year-old}} \\ &= g_n w S P_{n-1} + g_n w R_{n-1}, \end{aligned} \tag{5.26}$$

where R_n is defined by the number of remaining seeds from the previous generation:

$$R_n = (1 - g_n) w S P_{n-1}. \tag{5.27}$$

You can follow the “flow” in Figure 5.10 to see how we get the different terms. We now have two equations that define our model. Together they are known as a *system of coupled first-order difference equations*. They are “coupled” because they both depend on each other, and they are “first-order” because they depend only on the previous generation. Together they make a “system”.

We could go even further and write everything on one line by inserting Equation (5.27) with $n - 1$,

$$R_{n-1} = (1 - g_{n-1}) w S P_{n-2}, \tag{5.28}$$

into Equation (5.26):

$$P_n = g_n w S P_{n-1} + g_n w (1 - g_{n-1}) w S P_{n-2}. \tag{5.29}$$

This equation shows how P_n is actually defined by the number of plants two generations ago, P_{n-2} , which is because seeds can survive two winters. This dependency makes it a *second-order difference equation*. The equation is no longer “coupled” or a “system”, because it is only one equation. Each term in equation Equation (5.29) is explained in Figure 5.11.

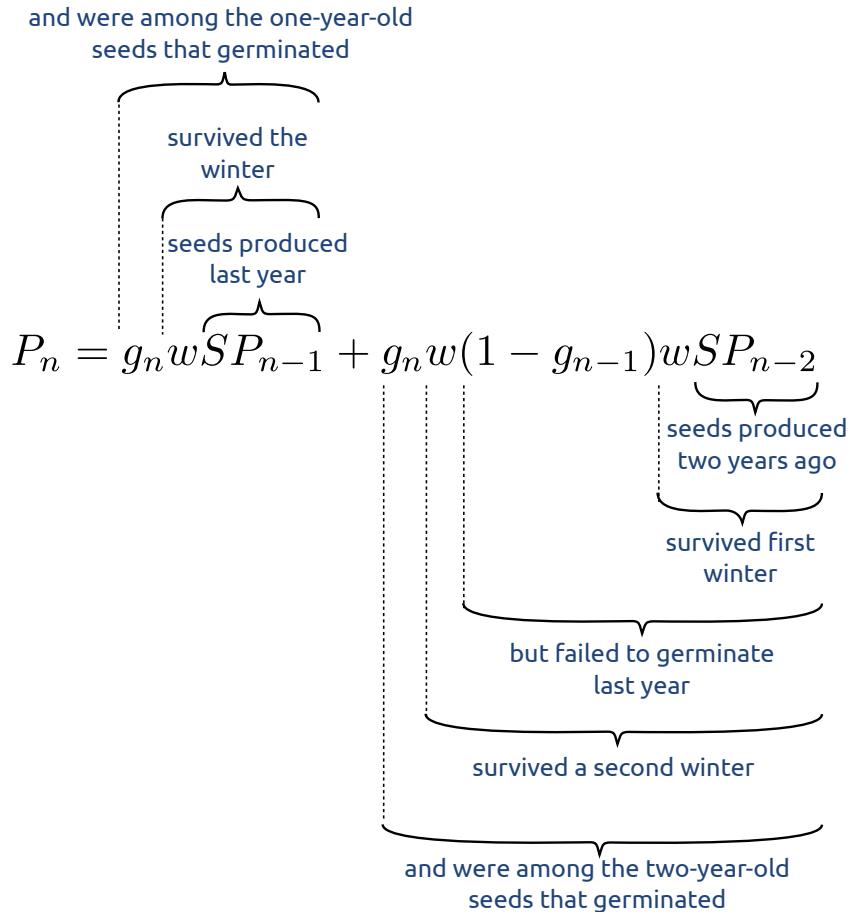


Figure 5.11: Explanation of terms in Equation (5.29). Adapted from chapter 1 in [7].

5.4 Summary

In this chapter, we modeled the growth of pea plants. We started out with a simple model and extended it to explore the effects of environmental factors. The first model included only one-year-old seeds. The model was expanded with a variable germination rate, where one year had zero germination. This resulted in the plant population dying out. Then we expanded the model with two-year-old seeds. This gave the seeds a second chance to germinate the year after, which in turn kept the plant population alive because they could “skip” the year with zero germination rate. In each model, there is a set of variables that describe the environment. To model the pea plant population growth in a given environment, we need the specific values for these variables. By changing the values of the variables, we see their effect on the plant population growth.

5.4.1 Mathematical symbols and their corresponding variable names

The different symbols we have used in this chapter are in the table below:

Symbol	Variable	Meaning
n	n	Generation/year
N	N	Number of time steps
P	P	Number of pea plants
P_n	$P[n]$	Number of pea plants in year n
P_0	$P[0]$	Initial condition
w	w	Fraction of seeds that survive the winter (survival rate)
g	g	Fraction of seeds that germinate (germination rate)
S	S	Number of seeds each pea plant produces

5.4.2 One-year model of annual plant population growth

Assumptions. We assume that:

- all plants are identical,
- seeds can only survive one winter,
- the fraction of seeds that survive a winter is constant, and
- the fraction of seeds that germinate is constant.

The one-year model. For each generation, we calculate the number of pea plants by:

- multiplying the number of plants from last year, P_{n-1} , by the number of seeds per plant,

$$\text{produced seeds} = SP_{n-1}, \quad (5.30)$$

- multiplying the number of produced seeds by the fraction of seeds that survive the winter,

$$\text{surviving seeds} = w \times \text{produced seeds}, \quad (5.31)$$

- and multiplying the number of surviving seeds by the fraction of seeds that germinate,

$$P_n = g \times \text{surviving seeds}. \quad (5.32)$$

Implementation of the one-year model. The core loop of this model is:

```

n = 1
while n < N:
    produced_seeds = S * P[n-1]
    surviving_seeds = w * produced_seeds
    P_n = g * surviving_seeds
    P.append(P_n)
    n = n + 1

```

Equation describing the one-year model.

$$P_n = gwSP_{n-1}. \quad (5.33)$$

5.4.3 Two-year model of annual plant population growth

Assumptions. We assume that:

- all plants are identical,
- seeds can survive two winters, and
- the fraction of seeds that survive a winter is constant.

The two-year model. For each generation, we calculate the number of pea plants as

$$P_n = P_{n, \text{one-year-old}} + P_{n, \text{two-year-old}}, \quad (5.34)$$

where the one-year-old seeds are found by using the one-year model. Further, we do the following:

- find number of remaining seeds that did not germinate,

$$\text{remaining seeds} = (1 - g_n) \times \text{surviving seeds}, \quad (5.35)$$

- multiply the number of remaining seeds by the fraction of seeds that survive the second winter,

$$\text{seeds after the second winter} = w \times \text{remaining seeds}, \quad (5.36)$$

- and multiply the number of seeds that survived the second winter with the fraction of seeds that germinate after two years,

$$P_{n, \text{second year seeds}} = g_n \times \text{surviving seeds after second winter}. \quad (5.37)$$

Implementation of the two-year model. The core loop of this model is:

```
P = []
P.append(20)
remaining_seeds = []
remaining_seeds.append(0)

n = 1
while n < N:
    produced_seeds = S * P[n-1]
    surviving_seeds = w * produced_seeds
    plants_1_year = g[n] * surviving_seeds

    remaining_seeds_n = (1 - g[n]) * surviving_seeds
    remaining_seeds.append(remaining_seeds_n)
```

```

surviving_seeds_2_year = w * remaining_seeds[n-1]
plants_2_year = g[n] * surviving_seeds_2_year

P_n = plants_1_year + plants_2_year
P.append(P_n)

n = n + 1

```

Equation describing the model. We can describe the model using a system of coupled first-order difference equations,

$$P_n = g_n w S P_{n-1} + g_n w R_{n-1}, \quad (5.38)$$

$$R_n = (1 - g_n) w S P_{n-1}, \quad (5.39)$$

or as a second-order difference equation,

$$P_n = g_n w S P_{n-1} + g_n w (1 - g_{n-1}) w S P_{n-2}. \quad (5.40)$$

5.4.4 Range

Using `range(N)` produces a range of numbers from zero up to, but not including, `N`. We use this function to set up a variable containing the years in the model:

```
t = range(N)
```

If we are interested in the numbers assigned to `t` as a list, we first need to convert it using the `list()` function:

```
print(list(t))
```

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

5.4.5 Subplots

Subplots are multiple plots in the same figure. We create them by calling `subplot()` before each plot. The `subplot()` command takes three arguments: the number of rows, the number of columns, and the current plot index. In our case, we plot two rows and one column, so we need to call `subplot(2, 1, 1)` for the first plot and `subplot(2, 1, 2)` for the second.

```

subplot(2, 1, 1)
plot(t, P, "-o")
xlabel("Generation, t")
ylabel("Number of pea plants, P")

subplot(2, 1, 2)
plot(t, remaining_seeds, "g-o")
xlabel("Generation, t")

```

```
ylabel("Remaining seeds")  
show()
```

Chapter 6

Modeling inheritance

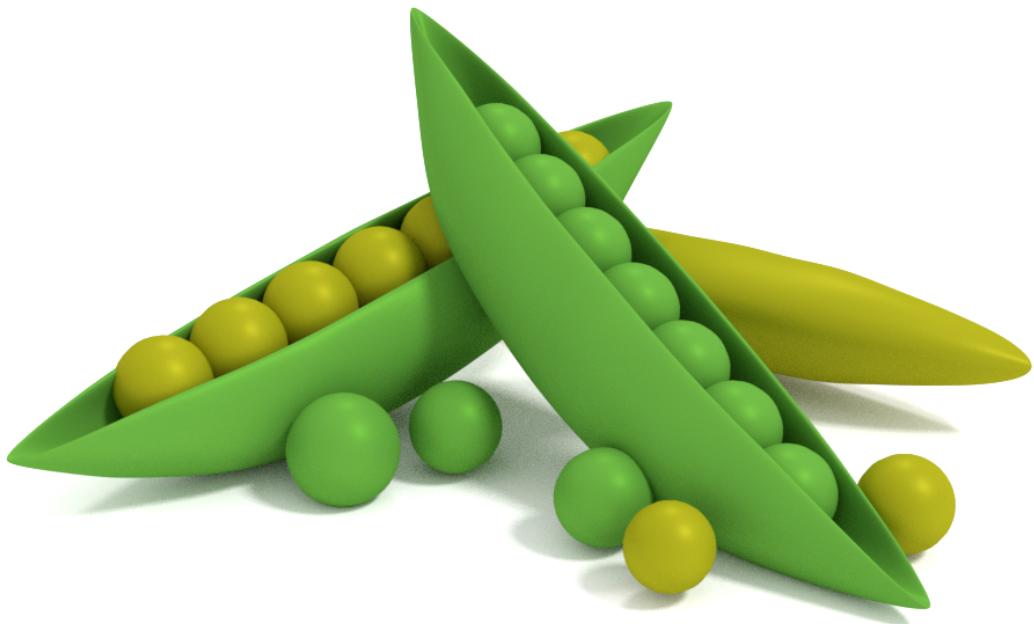


Figure 6.1: Mendel was curious about why some pea plants had different characteristics, such as differences in pod shape and seed color. By breeding garden peas in carefully planned experiments, he discovered the basic principles of heredity.

When describing a person's appearance, we might refer to *traits* from both sides of their family, such as: "She has her mother's hair and her grandfather's eyes". The fact that physical traits are carried over from one generation to the next was known already in ancient Greece, but the exact rules of inheritance remained a mystery for a long time. It would take the hard work and brilliant insight of a Moravian monk to shed light on the rules of heredity.

Gregor Mendel studied at the university of Vienna from 1851 to 1853, with the goal of improving his formal education in order to become a certified high school teacher. During this period he was inspired by his professors to study the variations of plant offspring and to use mathematics to describe his findings. Back at his monastery, he began his now famous investigation of the hereditary properties of garden peas, *Pisum sativum*. Between 1856 and 1863, he cultivated over 28,000 plants. The discoveries he made is the topic of this chapter.

In the first part of this book we worked with population dynamics. We analyzed and modeled how the number of individuals in a population changes over time. We assumed that all individuals were identical, and that their properties, or traits, did not change from one generation to the next. In this chapter, we will look at how the traits of parents are passed to the next generation.

We use Mendel's experiments to show how there is an element of randomness when it comes to inheritance. In order to better understand Mendel's so-called laws of inheritance, we will implement his model for inheritance, which involves random choices in Python, so we can perform experiments to examine Mendel's experimental observations. This shows you a different method for modeling nature in your computer, in contrast to the modeling methods used in the last two chapters.



Learning outcomes

After working with this chapter, you will:

- know the underlying assumptions in Mendel's model for inheritance,
- know how to create Punnett squares both by hand and with Python, and
- know how to model random inheritance of traits.

The programming concepts we introduce in this chapter are:

- random selection,
- functions,
- boolean statements, and
- `if` tests.

6.1 Mendelian genetics

There are several reasons why Mendel chose to work on garden peas, but one of the more important reasons is that garden peas come in many easily distinguishable varieties. They vary in features such as flower color, seed color, seed shape, pod color, and pod shape (see Figure 6.2). Such a heritable feature is called an inherited *character*, and the different variants of such characters are called *traits*. Another reason for using garden peas is that pea plants are

able to both cross-pollinate, i.e., be fertilized by other plants, *and* self-pollinate, i.e. self-fertilize. Finally, they give a large number of offspring and have a relatively short generation time. Mendel ensured complete control over the family tree of the pea plants, by manually cross-pollinating them.

Character	Stem length (tall or short)	Flower color (purple or white)	Seed color (yellow or green)	Seed shape (round or wrinkled)	Pod color (yellow or green)	Pod shape (inflated or constricted)
Traits						

Figure 6.2: Some different garden pea traits.

Mendel started out with plants that were *true breeding* for the characters he studied, that is, the offspring has the same traits when the plants self-pollinate. From these plants he bred a new generation of offspring and allowed them to self-pollinate. Mendel noticed that when he bred true breeding violet-flowered pea plants with true breeding white-flowered pea plants, the resulting offspring would always have violet flowers. However, when he allowed the resulting offspring to self-pollinate, around $1/4$ of the third generation would have white flowers and the rest had violet flowers, illustrated in Figure 6.3. The white color had seemingly disappeared in the offspring of the true breeding parents, but it reappeared after self-pollination. He observed the same phenomenon with several other traits. When performing the experiments for yellow and green seeds, the second generation would be all yellow seeds, while in the third generation, $1/4$ of the seeds would once again be green.

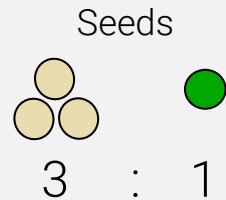
When working with inheritance, it is common to express these numbers as *ratios* rather than fractions.



Ratio

A ratio describes the quantitative relation between two amounts. For example, if $3/4$ of a number of seeds are yellow and the rest ($1/4$) are green, the ratio between them is 3:1

(pronounced “three to one”), meaning that there are three times more yellow seeds than green seeds. The notation using a colon to indicate the ratio between the two amounts is used throughout the rest of this chapter.



Mendel's observations were surprising at the time. They could not be explained by the then dominant theory of inheritance, which suggested that the traits of the offspring is the average of their parents, the so-called *blending hypothesis*. There are many problems with this hypothesis; it does not explain why siblings are not identical, and it suggests that over time, all genetic diversity would disappear as every next generation would be more alike. The blending hypothesis was ultimately dismissed when Mendel's model of inheritance, which he presented in 1865, became widely accepted after his death.

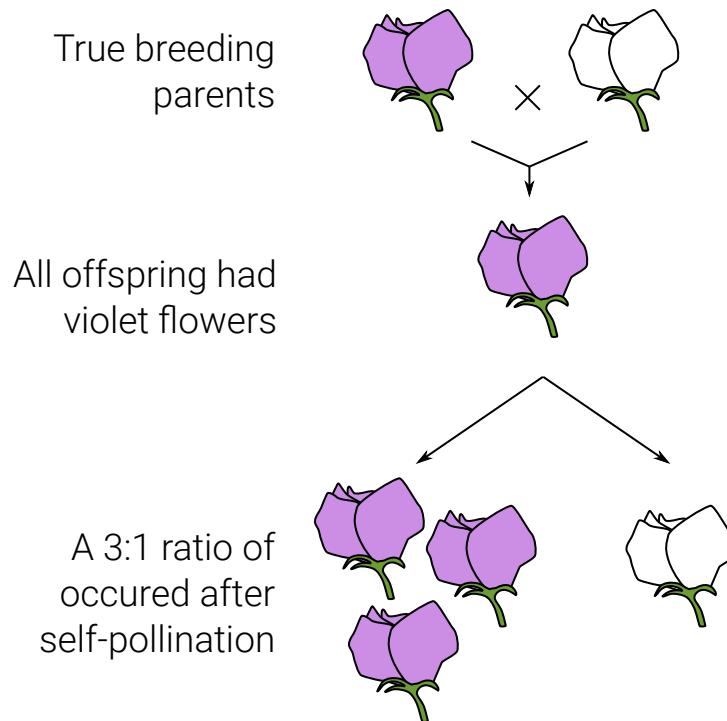


Figure 6.3: Mendel observed that breeding violet-flowered pea plants with white-flowered pea plants, results in offspring with violet flowers. Self-pollination of the offspring results in a 3:1 ratio of violet and white flowers.

6.1.1 Mendel's model of traits and genes

Mendel formulated a model to explain his observations. We will use modern terms to describe his model, rather than the terms Mendel used himself. Firstly, he proposed that the inherited trait was carried by a *gene* (Mendel called it a heritable factor), and the genes existed in two different versions. In modern terminology, we refer to different versions of a gene as different *alleles*. In Mendel's experiment there was one version of a gene that specified a violet flower color, and another of that gene that specified a white color. Secondly, he proposed that each organism inherits two versions of the gene, one from each parent. Thirdly, if the two alleles are different, one is *dominant* and always overrules the other, which is *recessive*. This explains why a cross-pollination of white and violet flowers produced flowers with only one color. Apparently, the violet flower version of the gene is dominant. Finally, Mendel suggested that alleles are separated during the formation of gametes (egg and sperm cells) such that each gamete only receives one allele, this is Mendel's *law of segregation*. Importantly, this means that the allele that is passed on to a particular offspring by a parent is chosen at random. Note that for true-breeding plants, this means that the traits they pass on to the next generation unchanged, must have identical alleles.



Summary: Mendel's model

1. An inherited trait is carried by a gene, and the genes exist in two alleles.
2. If the inherited alleles are different, one is dominant and overrules the other, which is recessive.
3. Each organism inherits two alleles, one from each parent.
4. Inherited alleles are chosen at random.

6.1.2 Using a Punnett square to determine the offspring of two parents

Let us use Mendel's model to explain the outcome of his cross-pollination experiment. First we introduce some standard notation. We denote the violet flower allele by B and the white flower allele by b. It is common to denote the dominant trait by a capital letter and the recessive trait by a lower-case letter.

According to Mendel's rules, a true breeding plant must have equal alleles, either BB or bb. Since the initial plants were true breeding, the violet flowered plants had the allele combination BB and the white flowered plants had the combination bb. We call this generation the P generation (parental generation).

Following Mendel's model, we pick one random allele from each of the plants. With the parent with violet flowers having the allele combination BB, we will always pick the B allele from this parent. Likewise, we will always pick the b allele from the parent with white flowers. In this case

the resulting combination is thus always Bb , and all offspring are violet since B is the dominant trait. We refer to the first generation of offspring as the F_1 generation (first filial generation, where filial means "son" in Latin). The subscript is used similarly to the subscript in the previous chapters and denotes the generation.

We call the third generation of pea plants F_2 (the second offspring generation). The parents of this generation are the F_1 generation, which all have the allele combination Bb . When we let these flowers self-pollinate (picking two alleles from each flower at random), we get several different possible outcomes; BB , Bb , and bb . To show all possible combinations it is common to represent the experiment by a 2 by 2 table. This is called a *Punnett square*, and is shown in Figure 6.4. This allows us to quickly see the possible outcomes and their probability.

In a Punnett square, we list the *paternal* alleles (from the sperm cell) above the top row, and the *maternal* alleles (from the egg cell) to the left of the leftmost column. We fill in the elements by combining one maternal allele and one paternal allele. From Figure 6.4 we see that in the F_1 generation, all outcomes are the same. In the F_2 generation, on the other hand, $1/4$ of the offspring have white flowers. These results, which are based on a random selection of genes, match the experimental observations made by Mendel.

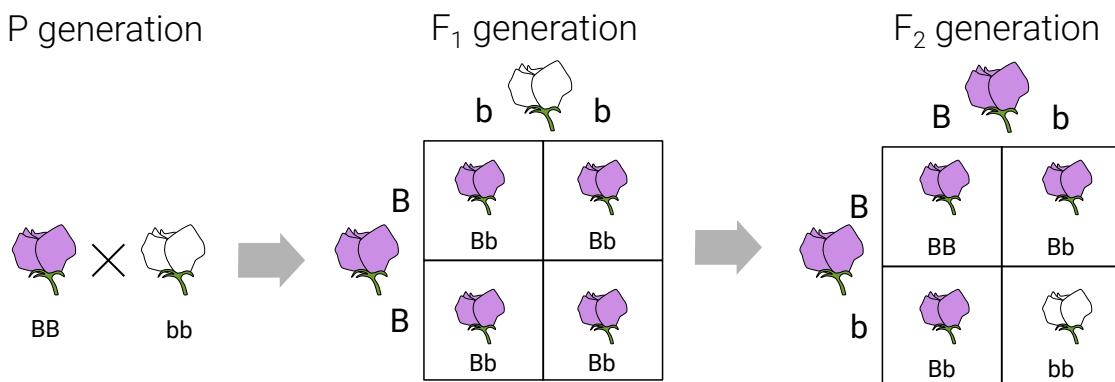


Figure 6.4: Illustrating Mendel's model for inheritance using Punnett squares. The violet-flower allele (B) is dominant, and the white-flower allele (b) is recessive. The flowers of the F_1 generation are all violet, but a 3:1 ratio occurs in the F_2 generation.

The Punnett square is a useful tool for visualizing the possible offspring. From our example, we see that two pea plants may have different allele combinations, even though they look identical. For this reason, it is useful to separate the plant's allele combination from its physical features.

We refer to the genetic combination of the plant, its set of alleles, as its *genotype*, and the physical properties, its traits, as its *phenotype*. In the setting of garden peas, the flower color (violet or white) is a phenotype, while the allele combination that controls the flower color (any combination of alleles B or b) is the genotype. Furthermore, we say that when an organism has a pair of identical alleles (e.g. BB) it is *homozygous* for this gene, while it is called *heterozygous* if the alleles are different (e.g. Bb).

6.1.3 Implementing a Punnett square

Mendel's rules of inheritance are well suited for implementation as a computer program, which is the goal of this chapter. To get started, let us create a Punnett square in Python for the second generation (F_2) in figure 6.4. We store the allele combination of the flowers in lists with two elements:

```
flower_1 = ["B", "b"] # left flower
flower_2 = ["B", "b"] # top flower
```

We find all possible combinations of offspring when crossbreeding a violet and a white flower by combining their alleles. We go through the alleles for the pure violet flower, and we combine each of the two alleles in the pure violet flower with the two alleles in the pure white flower. Doing this by hand we generate the following combinations:

- the first allele "B" of the left flower with the first allele "B" of the top flower, which gives "BB".
- the first allele "B" of the left flower with the second allele "b" of the top flower, which gives "Bb".
- the second allele "b" of the left flower with the first allele of the top flower, which gives "bB", which we write as Bb.
- the second allele "b" of the left flower with the second allele "b" of the top flower, which gives "bb".

In other words, we combine the alleles of each *row* in the Punnett square (which represents the alleles of the left flower), with the alleles of each *column* in the Punnett square (which represents the alleles of the right flower).

Instead of doing this by hand we can use a `while` loop for the alleles of the left flower, by going over each row in the Punnett square:

```
print("F2 generation:")

# while loop for flower_1 (left flower)
row_index = 0
while row_index < len(flower_1):
    allele_1 = flower_1[row_index]

    # first allele of flower_2 (right flower)
    allele_2 = "B"
    print("Possible offspring:", allele_1, allele_2)
    # second allele of flower_2 (right flower)
    allele_2 = "b"

    print("Possible offspring:", allele_1, allele_2)

    row_index = row_index + 1
```

```
F2 generation:
Possible offspring: B B
Possible offspring: B b
```

```
Possible offspring: B B  
Possible offspring: b b
```

Here we still create parts of the combinations by hand, namely the ones for the alleles of the right flower. We can avoid this if we use another `while` loop for the right flower *inside* the `while` loop for the left flower. This second loop should go over the columns of the Punnett square:

```
print("F2 generation:")  
  
# outer while loop for flower_1 (left flower)  
row_index = 0  
while row_index < len(flower_1):  
    # inner while loop for flower_2 (right flower)  
    column_index = 0  
    while column_index < len(flower_2):  
  
        # allele for flower_1 (left flower)  
        allele_1 = flower_1[row_index]  
        # allele for flower_2 (right flower)  
        allele_2 = flower_2[column_index]  
  
        print("Possible offspring:", allele_1, allele_2)  
  
        column_index = column_index + 1  
  
    row_index = row_index + 1
```

```
F2 generation:  
Possible offspring: B B  
Possible offspring: B b  
Possible offspring: b B  
Possible offspring: b b
```

This introduces something new: a loop inside a loop. This is a *nested while loop*, sometimes called a *double while loop*. Let us show another example of a nested `while` loop, where it is easier to see what happens:

```
colors = ["red", "blue", "green"]  
items = ["book", "car", "pill", "flag"]  
  
# outer while loop  
index_outer = 0  
while index_outer < len(colors):  
    # inner while loop  
    index_inner = 0  
    while index_inner < len(items):  
        print(colors[index_outer], items[index_inner])  
        index_inner = index_inner + 1  
  
    print("inner loop complete!")  
    index_outer = index_outer + 1  
  
print("outer loop complete!")
```

```
red book
```

```
red car
red pill
red flag
inner loop complete!
blue book
blue car
blue pill
blue flag
inner loop complete!
green book
green car
green pill
green flag
inner loop complete!
outer loop complete!
```

Let us break down the program and the output, and discuss each part separately. First, notice the syntax for the double `while` loops. We already know the syntax of a normal `while` loop, where everything inside the loop must be indented. This also applies to the inner `while` loop, which means everything inside the inner loop must be indented twice. In this example, the statement `print(colors[index_outer], items[index_inner])` is inside both loops.

The `index_outer` variable is 0 in the first iteration of the outer loop and thus `colors[index_outer]` becomes `red`. For each pass of the outer loop we perform all iterations of the inner loop, which goes over the indexes 0, 1, 2, 3 and 4, and thus `items[index_inner]` becomes `book`, `car`, `pill`, and `flag`:

```
red book
red car
red pill
red flag
inner loop complete!
```

We are then finished with the inner loop and continue with the next iteration of the outer loop, `outer_index` now becomes 1 and `colors[index_outer]` now becomes `blue`. We once again perform all iterations of the inner loop and print out the letters.

```
blue book
blue car
blue pill
blue flag
inner loop complete!
```

Then we perform the last iteration of the outer loop, before we are finished with the outer loop and exit it, printing `outer loop complete!`.

```
green book
green car
green pill
green flag
inner loop complete!
outer loop complete!
```

In the end, we have printed all combinations of the color and items in each list, which is the purpose of a double `while` loop. It gives us a way to combine every element in the first list with every element in the second list.

We want to organize the results of the above program in a Punnett square, which is similar to a table. The `pandas.DataFrame()` function generates a table with labeled rows and columns. To use this function, we first import the `pandas` package and define the two flower alleles:

```
import pandas

flower_1 = ["B", "b"]
flower_2 = ["B", "b"]
```

`pandas.DataFrame()` takes the arguments `index` and `columns`. These are the labels for the rows and columns, and are set to `flower_1` and `flower_2` above. Because each list has two elements, this results in a two-by-two table:

```
punnett_square = pandas.DataFrame(index=flower_1, columns=flower_2)
print(punnett_square)
```

	B	b
B	NaN	NaN
b	NaN	NaN

When printing the `DataFrame()` we see that it is nicely formatted. An empty `DataFrame()` has `NaN` as each of its elements. `NaN` stands for “not a number”, which is a common way to denote that nothing has been assigned to the element.

To fill the upper left element of the Punnett square with the Bb combination, we write:

```
punnett_square.iloc[0, 0] = "BB"

print(punnett_square)
```

	B	b
B	BB	NaN
b	NaN	NaN

The operator `iloc[row, column]` takes in a row index as its first element, and its column index as the second element. In all standard Python libraries, tables are indexed this way, called *row-by-column* (see Figure 6.5). Note that `iloc[]` uses square brackets, and not regular parentheses.

We see that one element has been entered, while the rest is `NaN`. We fill out the rest of the elements of the Punnett square in the same way:

```
punnett_square.iloc[0, 1] = "Bb"
punnett_square.iloc[1, 0] = "Bb"
punnett_square.iloc[1, 1] = "bb"

print(punnett_square)
```

		Columns			
		0	1	2	
		0	[0, 0]	[0, 1]	[0, 2]
		1	[1, 0]	[1, 1]	[1, 2]
		2	[2, 0]	[2, 1]	[2, 2]

Figure 6.5: Row-by-column indexing. The numbers inside each square denote the index of a specific element in the table.

```
B   b
B  BB  Bb
b  Bb  bb
```

Instead of writing the Punnett squares by hand, we can generate all combinations with a double `while` loop. We create the `DataFrame()` by setting the index and columns, but we do not pass it any data yet. We access each location in `DataFrame()` using two indices (one for row and one for column) corresponding to the current location in the double `while` loop, one loops over all rows and the other loops over all columns. We will use the index of the outer loop (the `flower_1` alleles) for the row indexes, and the index of the inner loop (the `flower_2` alleles) for the column indexes.

The code for that uses a double `while` loop to generate all allele combinations, store them in a `DataFrame()`, and finally print the `DataFrame()` then becomes:

```
import pandas

flower_1 = ["B", "b"]
flower_2 = ["B", "b"]

punnett_square = pandas.DataFrame(index=flower_1, columns=flower_2)

print("F2 generation")

# outer while loop for flower_1
```

```
row_index = 0
while row_index < len(flower_1):
    # inner while loop for flower_2
    column_index = 0
    while column_index < len(flower_2):
        allele_1 = flower_1[row_index]
        allele_2 = flower_2[column_index]
        punnett_square.iloc[row_index, column_index] = allele_1 + allele_2

        column_index = column_index + 1
    row_index = row_index + 1

print(punnett_square)
```

F2 generation
B b
B BB Bb
b bB bb

We use `allele_1 + allele_2` to get the allele combination. `allele_1` and `allele_2` are strings, and when we use the plus sign between two strings, they are concatenated.

```
print("B" + "b")
```

Bb

Note that the order of the alleles does not matter in nature, therefore the two combinations Bb and bB are the same.

In the Punnett square we generated, we get four possible outcomes: BB, Bb, bB and bb. Each of the outcomes are just as likely as the others, each offspring has a 25% chance (one in four) for ending up in each of the combinations. Since the B variation is dominant, BB, Bb, and bB produce a violet-flowered pea plant. The last combination, bb, produces a white-flowered pea plant. This accounts for the 3:1 ratio that Mendel observed in his experiment.

We could create the next generation by picking two of these offspring, and see what their children would look like. Doing this would be copying code we already have written, with only minor adjustments, which is not ideal. We want to do this for multiple generations and possibly many different combinations, so this is a good time to learn how to write code that can be reused.

6.2 Functions: Reusing code

Code that will be used multiple times should be put into *functions*. At the end of Section 3.4, we mentioned the rule of thumb called “the rule of three”, and it applies here as well. Think of a function as a machine that, when given some input, based on some instructions, produces some output (see Figure 6.6). By giving the function different types of input, this machine can be used to produce different outputs based on the same set of instructions. `float()` is an example of a function; it takes a string or number as input, converts the input to a float, and returns that float. We have used functions since the first chapter of this book, but an important aspect of programming is that you can write such functions yourself.

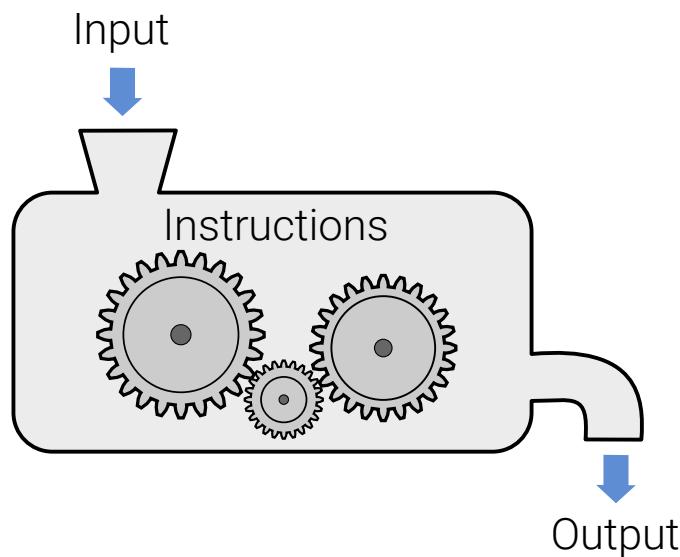


Figure 6.6: A function is like a machine. You give the function a set of inputs, and it produces some output based on the instructions in the function.

6.2.1 Creating functions

To make the concept of a function as clear and concise as possible, we begin with a simple example. Afterwards, we break down the individual pieces that go into it. Here is the Python code to create a function called `add_one`:

```
def add_one(some_value):
    new_value = some_value + 1
    return new_value
```

To define a function in Python, we use the `def` keyword. After `def` comes the name of the function, a set of parentheses and a colon. Inside the parentheses are the *parameters* of the function: variables that will be assigned its input. After the colon, anything that is indented becomes part of the function.

If we want the function to give an output, we need to *return* it by using the `return` keyword. Whatever is listed after `return` is sent out of the function and can be used by the program outside the function.

We will now use the function by *calling* it. This is done by writing the name of the function followed by a set of parentheses. Inside the parentheses are the *arguments* of the function, the input that will be assigned to the parameters described above. The `add_one()` function takes one argument, which we thus put inside the parentheses. We choose to assign what is returned from the function, its output, to a new variable `my_number`:

```
my_number = add_one(2)
print(my_number)
```

| 3

We see that the input to the function, the number 2, results in an output that is one number higher, 3. Looking at the instructions inside the function, this is what was intended. Let us give one more example:

```
my_number = add_one(-6)
print(my_number)
```

| -5

Since $-6 + 1 = -5$ we get -5 as output of the function.

A function can have multiple inputs, as you have seen with the `plot()` function. The inputs are separated by commas, and we can use the arguments inside the function. The following function takes two values and adds them together and returns the result:

```
def add(a, b):
    result = a + b
    return result

print(add(2, 3))
```

| 5

Note that in these examples, we do not assign the output of the function to a new variable, but put the function call directly inside the `print()` statement.

Functions influence how we read programs to understand what they are doing. Up until this point, we could read what a program does by reading it line by line (with the caveat that the lines in `while` loops are read several times). However, when our program contains functions that we define ourselves, and call in different parts of the program, we would need to study their behaviour separately to understand what their role is in the program.

6.2.2 Improving code using functions

Let us go through some typical examples of cases where using functions improve your code. In the previous section the functions were extremely simple, to the point where they do not seem very useful. In this section, we provide some more realistic uses of functions.

In the programs you have written so far, you have undoubtedly made quite a few errors. This is normal, and even as you become an experienced programmer, you will find that you spend quite a bit of your time locating the errors in your code. Even when you write a piece of code right the first time, you will often find that an error sneaks its way in when you try to write the same piece of code again. For this reason, we would whenever possible like to avoid writing the same piece of code more than once.

Example 6.2.1. Use a function to avoid rewriting temperature conversion.

Let us go all the way back to Section 1.2, where we used Python to convert from Fahrenheit to Celsius. We do not want to write this code every single time we need to convert a temperature. We instead write a function to perform the conversion for us:

```
def fahrenheit_to_celsius(fahrenheit):
    celsius = (fahrenheit - 32)/1.8

    return celsius

fahrenheit = 10
celsius = fahrenheit_to_celsius(fahrenheit)

print(fahrenheit, " degrees Fahrenheit is equal to ", celsius, " degrees Celsius.")
```

```
| 10  degrees Fahrenheit is equal to  -12.22222222222221  degrees Celsius.
```

Now, we can use the code to easily convert any temperature, without having to write the conversion code every time.

A function does not have to return any values. The following is an example of a function that only creates a figure.

Example 6.2.2. *Use a function to make an E. coli plotting program easier to read.*

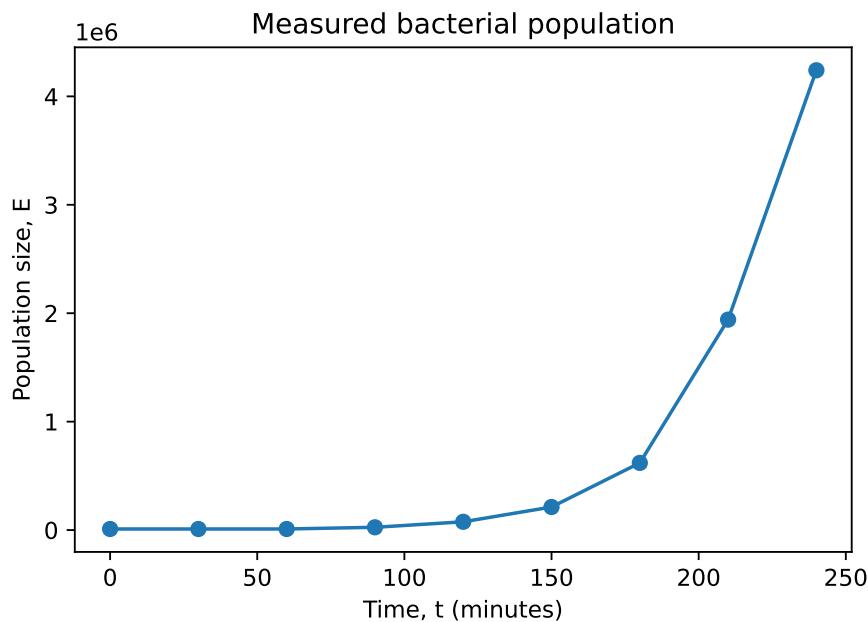
Let us go back to the earlier chapters on *E. coli*. We made several plots which were quite similar. We would write the same things on the *x*- and *y*-axes, make a title, and show the plot. All these commands can be put into a function, which makes the final program shorter and more readable. A function that performs these plot operations is:

```
from pylab import *
import pandas

def plot_ecoli(t, E):
    plot(t, E, "-o")
    xlabel("Time, t (minutes)")
    ylabel("Population size, E")
    title("Measured bacterial population")
    show()

data = pandas.read_csv("ecoli.csv")
t = list(data["t"])
E = list(data["E"])

plot_ecoli(t, E)
```



With this function we make it more clear what the lines that create the plot do. This example illustrates that a function does not have to return any values.

6.2.3 Creating a Punnett square function

We now make a function that generates a Punnett square from the allele combinations of two parents. It takes `parent_1` and `parent_2`, lists with parental alleles, as input, and returns the Punnett square:

```
def create_punnett_square(parent_1, parent_2):
    punnett_square = pandas.DataFrame(index=parent_1, columns=parent_2)
    row_index = 0
    while row_index < len(parent_1):
        column_index = 0
        while column_index < len(parent_1):
            punnett_square.iloc[row_index, column_index] = parent_1[row_index] +
                parent_2[column_index]
            column_index = column_index + 1
        row_index = row_index + 1
    return punnett_square
```

This greatly simplifies the creation of Punnett squares. Let us recreate the Punnett square for the F₁ generation:

```
print("F1 generation:")
punnett_square = create_punnett_square(["B", "B"], ["b", "b"])
print(punnett_square)
```

F1 generation:

	b	b
B	Bb	Bb
b	Bb	Bb

We continue by creating the Punnett square for the F₂ generation:

```
print("F2 generation:")
punnett_square = create_punnett_square(["B", "b"], ["B", "b"])
print(punnett_square)
```

	b	b
B	BB	Bb
b	bB	bb

As you see, we get the same results as before, but our code is cleaner and easier to read since we reuse the `create_punnett_square()` function for both generations.

6.2.4 Looking at several traits simultaneously

Mendel derived his rules of inheritance by following single traits over several generations. A natural extension is to see what happens if we follow two traits at the same time, such as the flower color and the seed color. The pea plant pods are either green or yellow, and either inflated or constricted, as shown in figure Figure 6.7. Green and inflated are the dominant traits.

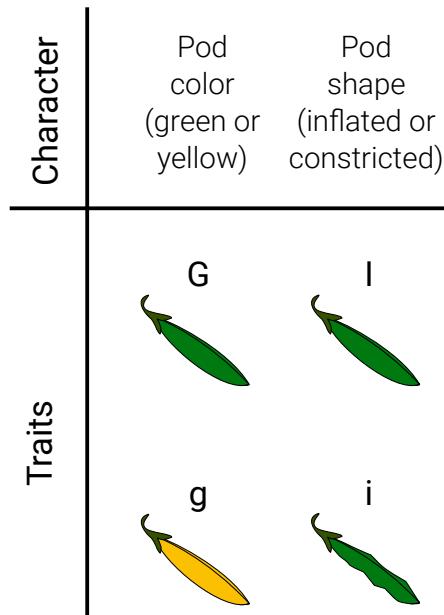


Figure 6.7: Two inherited traits of pea pods are color and shape. They can either be green (G) or yellow (g), and either inflated (I) or constricted (i).

We denote the green pod allele by G, and the yellow pod allele by g, the inflated pod allele by I, and the constricted pod allele by i. For pea plants with both traits, the dominant true breeding type is inflated and green pods (IIGG), while the recessive true breeding type is constricted and yellow pods (iigg). When crossbreeding, all the offspring are of type IiGg, which are inflated and green. What happens in the F₂ generation? There are two possibilities that occur in nature.

The first possibility is if the traits are not inherited independently, such that, for example, if the I trait is passed down, only the G trait can be passed down as well, and if the i trait is passed down, only the g trait is permissible.

We can use our function to show the resulting Punnett square by calling it with the possible allele combinations each parent can pass on to the next generation. In the case of connected traits, both parents can either pass on IG or ig. Using our function, we see that in this case, the Punnett square is a 2 by 2 grid:

```
punnett_square = create_punnett_square(["IG", "ig"], ["IG", "ig"])
print(punnett_square)
```

	IG	ig
IG	IGIG	IGig
ig	igIG	igig

The second possibility is that the traits are passed down independently of one another. In this case, the possible alleles combinations each parent can pass down to the next generation are IG, Ig, iG, or ig, which are the four different combinations of alleles that the parent has. A Punnett square listing all combinations is a 4 by 4 table, see Figure 6.8.

Our `create_punnett_square()` function uses a double `while` loop to go over the alleles of each parent. We can therefore create this extended Punnett square with 16 possible outcomes without any modification to our function. If we use the four possible allele combinations for each parent as arguments to our Punnett square function, we obtain a Punnett square as a 4 by 4 table:

```
parent_1 = ["IG", "Ig", "iG", "ig"]
parent_2 = ["IG", "Ig", "iG", "ig"]

punnett_square = create_punnett_square(parent_1, parent_2)
print(punnett_square)
```

	IG	Ig	iG	ig
IG	IGIG	IGIg	IGiG	IGig
Ig	IgIG	IgIg	IgiG	Igig
iG	iGIG	iGIg	iGiG	iGig
ig	igIG	igIg	igiG	igig

This highlights how useful functions are. Well-written functions can deal with cases we did not consider when we wrote the function.

In pea plants, the two traits are inherited independently and the 4x4 punnett square is the correct one. Whether two traits are inherited independently of each other or not varies from trait to trait, and from one organism to the next. The only way to determine this for a set of traits is to carry out crossbreeding experiments.

The Punnett square above contains all possible combinations from the crossbred generation. When the next plant generation is made, the alleles of the offspring are chosen randomly. We will model this random process in the next section.

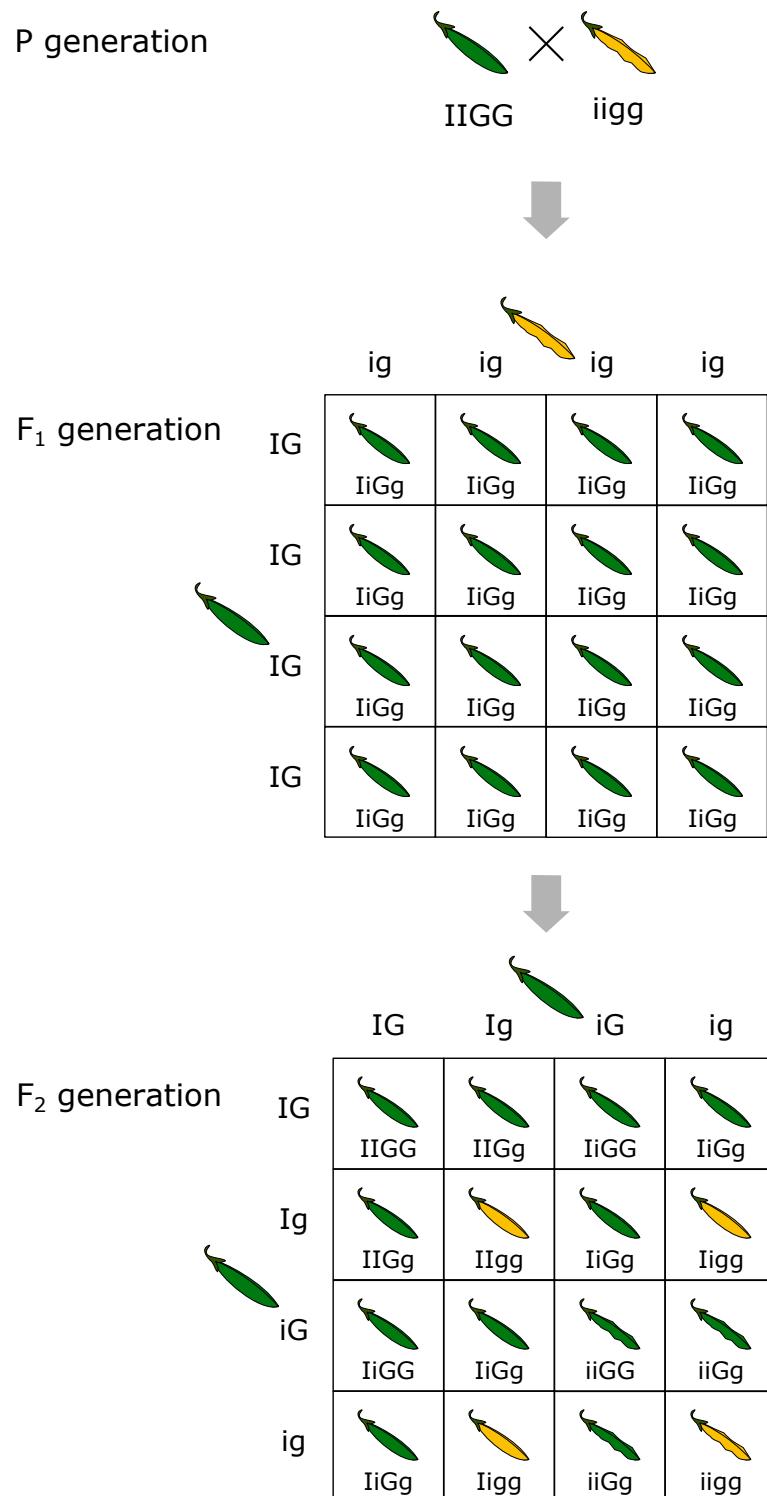


Figure 6.8: This diagram illustrates Mendel's model for inheritance when two traits are passed down independently.

6.3 Implementing Mendel's model

In the previous section, we chose two parents and saw all possible traits of the offspring they could produce. We also discussed traits for certain characters that are inherited independently, such that each offspring will become one of the possible combinations of traits. where the combination is selected at random. In this section, we show how you can do virtual experiments crossing pea plants on the computer by making choices with random outcomes. While Mendel had to plant and care for his pea plants over many years, we have the opportunity to blast through thousands of experiments within seconds.

To perform our virtual experiments, we use functions for making random choices from the `pylab` package. Selecting a random element from a list is done by calling the `choice()` function. In the following program, we define a list of colors and select a random color:

```
from pylab import *
colors = ["red", "green", "blue", "violet"]
color = choice(colors)

print(color)
```

| red

Try to run this program multiple times and notice how the output varies. This is because a color is chosen at random each time you run the program. We discuss randomness in computers later in this chapter.

6.3.1 Random pollination

Let us create a virtual experiment where we allow garden pea plants with the genotype Bb to self-pollinate. We begin by making a program that simulates a single generation. We restate Mendel's rules for inheritance to have them close at hand:

1. An inherited trait is carried by a gene, and the genes exist in two alleles.
2. If the inherited alleles are different, one is dominant and overrules the other, which is recessive.
3. Each organism inherits two alleles, one from each parent.
4. Inherited alleles are chosen at random.

First, we import the `pylab` package and define the genotypes of the parent generation. This implements the first of our rules:

```
from pylab import *
parent_1 = ["B", "b"]
parent_2 = ["B", "b"]
```

Next, we select one allele from each parent at random with `choice()`. This implements the second and fourth rules.

```
allele_1 = choice(parent_1)
allele_2 = choice(parent_2)
genotype = [allele_1, allele_2]

print("Inherited genotype:", genotype)
```

```
| Inherited genotype: ['b', 'B']
```

If you try running this code multiple times, you see that the result may change from one run to the next.

6.3.2 If tests: Taking different branches in code

The next step is to find the phenotype based on the randomly selected genotype. In order to do this, we need *if tests*. This is a concept that allows us to run different code blocks depending on the values of our variables. If a certain condition is met, we do something, and if the condition is not met, we skip it. This is often called *branching* because the computer runs different “branches” of the code depending on the condition (see Figure 6.9, left).

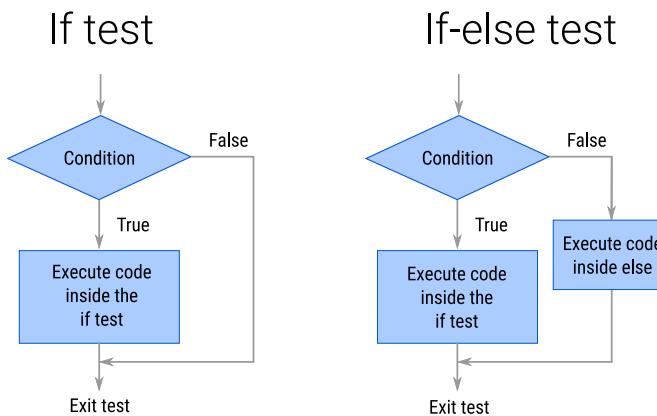


Figure 6.9: Flowchart of an `if` test (left) and `if-else` test (right).

We illustrate the use of an `if` test, by making a simple example. We create a variable:

```
my_number = 0
```

The following `if` test prints a statement if `my_number` is less than five, which is true in this case:

```
if my_number < 5: # condition
    # code block to run if condition is true
    print("Yes, my_number is less than five!")
```

```
| Yes, my_number is less than five!
```

Note how the line with the `if` statement ends in a colon, and thus the next lines are indented. It is these indented lines which only are executed when the `if` test evaluates to true. Similarly, the following `if` test checks if `my_number` is equal five, which is false. Now, the indented code block following the line with `if` is skipped, as illustrated in Figure 6.9, so nothing is printed to screen.

```
if my_number == 5:  
    print("Yes, my_number is equal to five!")
```

If you compare the `if` test flowchart in Figure 6.9 with the flowchart for `while` loops in Figure 3.11 you will see they both contain a conditional statement: the test in the `while` loop determines whether the code in the loop body is executed, and the test in the `if` test determines whether some code is executed at all.

The above `if` tests gets more interesting if we use the `choice()` function to pick between two numbers and test if the result is less than five, or equal to five:

```
my_numbers = [0, 5]  
my_number = choice(my_numbers)  
  
if my_number < 5:  
    print("Yes, my_number is less than five!")  
  
if my_number == 5:  
    print("Yes, my_number is equal to five!")  
  
print("my_number was", my_number)
```

```
| Yes, my_number is less than five!  
my_number was 0
```

Try running the above code a couple of times and see what you get. The output of this function will vary depending on the chosen color. Also, notice how the `print` statement which just prints the number is always run. This is because it is outside both `if` tests.

6.3.3 Comparison operators

The line with the `if` statement contains a condition, similar to the `while` statements in Section 3.5. A condition is a test that evaluates to either `True` or `False`. `True` or `False` are so-called *boolean* values. Boolean values are always either `True` or `False`, and are widely used in programming. In our example we test whether the value of `my_number` is less than 5. As long as the test evaluates to `True`, the code inside the `if` test is executed.

The `<` character in the conditional is called a *comparison operator*. Comparison operators compare expressions on both sides of the operator. Here is a list of the most commonly used comparisons operators in Python:

Syntax	Description
<code>a == b</code>	<code>a</code> is equal to <code>b</code>
<code>a != b</code>	<code>a</code> is not equal to <code>b</code>
<code>a < b</code>	<code>a</code> is less than <code>b</code>
<code>a > b</code>	<code>a</code> is greater than <code>b</code>
<code>a <= b</code>	<code>a</code> is less than or equal to <code>b</code>
<code>a >= b</code>	<code>a</code> is greater than or equal to <code>b</code>
<code>a in b</code>	<code>a</code> is an element in the list <code>b</code>

Note that we use *two* equal signs (`==`) instead of one to test whether two things are equal. This is because `==` is a *comparison operator*, while `=` is an *assignment operator*. Expressions that use comparison operators are called *boolean expressions*. `!=` is the comparison operator, while `a != b` is a boolean expression. The `in` operator tests if an element exists in a list:

```
colors = ["blue", "violet", "green", "red"]

if "red" in colors:
    print("The red color is in the list.")
```

The red color is in the list.

The keyword `not` is inserted in front of a boolean expression to change the value from `True` to `False`, or from `False` to `True`. We can for instance check if "red" is *not* in the list of colors:

```
colors = ["blue", "violet", "green"]

if not "red" in colors:
    print("The red color is not in the list.")
```

The red color is not in the list.

We are also able to use `if` tests to compare lists. The following code checks the genotype, and sets the phenotype to violet if the genotype is BB, Bb, or bB, and sets the phenotype to white if the genotype is bb:

```
parent_1 = ["B", "b"]
parent_2 = ["B", "b"]

allele_1 = choice(parent_1)
allele_2 = choice(parent_2)

genotype = [allele_1, allele_2]

print("The genotype is:", genotype)

if genotype == ["B", "B"]:
    phenotype = "violet"

if genotype == ["B", "b"]:
    phenotype = "violet"
```

```

if genotype == ["b", "B"]:
    phenotype = "violet"

if genotype == ["b", "b"]:
    phenotype = "white"

print("The phenotype is", phenotype)

```

The genotype is: ['b', 'B']
 The phenotype is violet

Each time you run this code, a combination of alleles from the two parents are chosen and the given genotype and phenotype are printed. This implements our third rule for inheritance.

We remember from Section 3.5, that `while` loops contain a conditional statement similar to `if` tests. This means that all the above comparison operators can be used with `while` loops as well. For example, the following `while` loop starts at 8, and counts down.

```

my_number = 8
while my_number > 5:
    print("Inside the loop, my_number is now", my_number)
    my_number = my_number - 1

print("After the loop, my_number is", my_number)

```

Inside the loop, my_number is now 8
 Inside the loop, my_number is now 7
 Inside the loop, my_number is now 6
 After the loop, my_number is 5

Similarly, `while` loops can be constructed using any of the above mentioned comparison operators.

6.3.4 If-else-tests: Taking an alternative branch

The `else` keyword is used to run a block of code only when the `if` condition is evaluated as `False`, see Figure 6.9 (right). We take an alternative branch to the one taken if the condition is true:

```

color = "blue"

if color == "red":
    print("The color is red!")
else:
    print("The color is something else!")

```

The color is something else!

Let us use `if-else` tests together with the `in` keyword to simplify the code that found the phenotype of a given genotype. We know that the allele B is dominant. If B is present in the genotype, the phenotype is a violet flower, else the phenotype is a white flower:

```
genotype = ["b", "B"]

if "B" in genotype:
    phenotype = "violet"
else:
    phenotype = "white"

print("The phenotype is", phenotype)
```

The phenotype is violet

In this case, we get the phenotype `violet` because there is an uppercase "B" in the `genotype` list.

If there is no "B" in the genotype list the phenotype is that of a white flower:

```
genotype = ["b", "b"]

if "B" in genotype:
    phenotype = "violet"
else:
    phenotype = "white"

print("The phenotype is", phenotype)
```

The phenotype is white

The other possible cases are `["B", "b"]` or `["B", "B"]`, both of which have an uppercase "B" in them. Their phenotype is therefore violet flowers. You can modify the above code with these other cases to verify that the code works correctly.

6.3.5 A complete virtual experiment

We are now able to implement Mendel's model for inheritance to test if it explains the result he obtained from his experiments. Mendel let true breeding violet and white flowered pea plants cross-pollinate before letting the resulting offspring self-pollinate. Finally, he counted the number of pea plants with white or violet flowers and found a 3:1 ratio of violet to white flowers.

To perform Mendel's experiment on our computer we need to find the genotype of the offspring from different parents over two generations. This is a good place to make a function we can reuse for each of the generations:

```
def create_offspring(parent_1, parent_2):
    allele_1 = choice(parent_1)
    allele_2 = choice(parent_2)

    genotype = [allele_1, allele_2]

    return genotype
```

This function creates one of the possible genotypes of an offspring based on the genotypes of the parents, following Mendel's rules for inheritance.

In his experiment, Mendel started with true breeding pea plants, with white flowers (bb) and violet flowers (BB):

```
# True bred pea plants
P_generation_violet = ['B', 'B']
P_generation_white = ['b', 'b']
```

He then cross-pollinated the pea plants to create the F₁ generation:

```
# Cross-pollinate
F1_generation = create_offspring(P_generation_violet, P_generation_white)
print(F1_generation)
```

```
['B', 'b']
```

In this case, the output will be the same every time. Afterwards, he self-pollinated the F₁ generation offspring to create the F₂ generation:

```
# Self-pollinate
F2_generation = create_offspring(F1_generation, F1_generation)
print(F2_generation)
```

```
['B', 'B']
```

The output of the last code block varies if you run it multiple times, because of the call to `choice()` inside the `create_offspring()` function.

Mendel counted the number of pea plants with white and violet flowers. In order to do the same, we check if the genotype of the F₂ generation gives violet or white flowers, and increase a corresponding count:

```
violet_flowers_count = 0
white_flowers_count = 0

# Find the phenotype and increase correct count
if 'B' in F2_generation:
    violet_flowers_count = violet_flowers_count + 1
else:
    white_flowers_count = white_flowers_count + 1

print("Number of white flowers:", white_flowers_count)
print("Number of violet flowers:", violet_flowers_count)
```

```
Number of white flowers: 0
Number of violet flowers: 1
```

Mendel did the above experiment for 28,000 pea plants, and we would like to do the same. To do this, we use a `while` loop and put the above experiment inside the loop body. The counting

variables `white_flowers_count` and `violet_flowers_count` need to be defined before the `while` loop to make sure they start at zero before we start the experiments:

```
white_flowers_count = 0
violet_flowers_count = 0

i = 0
while i < 28000:
    # True bred flowers
    P_generation_violet = ['B', 'B']
    P_generation_white = ['b', 'b']

    # Cross-pollinate
    F1_generation = create_offspring(P_generation_violet, P_generation_white)

    # Self-pollinate
    F2_generation = create_offspring(F1_generation, F1_generation)

    # Find the phenotype and increase correct count
    if 'B' in F2_generation:
        violet_flowers_count = violet_flowers_count + 1
    else:
        white_flowers_count = white_flowers_count + 1

    i = i + 1
```

Once we are finished with all repetitions of the experiment, we calculate the ratio of the number of pea plants with violet flowers to the number of pea plants with white flowers, and print the result:

```
ratio = violet_flowers_count/white_flowers_count

print("The result for the F2 was", violet_flowers_count,
      "violet flowers and", white_flowers_count, "white flowers.")
print("The ratio is", ratio, "to 1")
```

```
The result for the F2 was 20901 violet flowers and 7099 white flowers.
The ratio is 2.9442174954218903 to 1
```

When running this code several times, the resulting ratio will be similar, but probably not identical, to the one above. Since we are working with random selections, each run of the experiment gives a slightly different result. The chance of getting an exact ratio of 3:1 is small, but if we increase the number of experiments, we expect the result to approach this value more and more.

6.3.6 What do our results mean for Mendel's model?

Performing this experiment took Mendel many years, while we are able to do the experiment in a few seconds. Our implementation of Mendel's model for inheritance reproduces the 3:1 ratio that Mendel observed. We can therefore say that our implementation of Mendel's model leads to the same outcome as his experiments. However, an important distinction to keep in mind is that this does not necessarily mean that Mendel's model is correct. Our computational implementation

is only able to confirm the experimental results. We could also use our implementation to test or predict the outcome of other crosses. These results can then be tested by, or compared to the results of, real experiments. Using a computational model like this, which gives very fast results of virtual experiments, can thus be used in conjunction with real experiments to help build or improve models of (biological) reality.

The characters Mendel studied were those that have a simple genetic basis, and most were controlled by one gene present in two alleles. Genetics research done after Mendel has shown that there are many more characters for which the genetic basis is much more complex, and depends on many more genes working together to obtain the trait. The 'one gene - one character' model thus has limited applicability. Mendel's work still turned out to be very valuable, as some aspects of his model, such as the random selection of alleles in gametes, also hold for more complex characteristics.

6.3.7 A note on randomness in computers

Because we are using a computer, we are not working with truly random choices. When we say we use the computer to draw randomly from a list, we are being somewhat inaccurate. Nothing that happens in your program is truly random. However, random functions have properties which make their behavior look random. There is no obvious correlation between one choice and the next, but if we let the random generator draw choices for long enough, the sequence will eventually start to repeat. This is not something we need to worry about, because we can draw about 10^{6000} choices before this happens (that is 1 followed by 6000 zeros).

Sometimes, however, we want to make sure that the same "random" choices are made each time the code is run. To achieve this, we set the *seed* of the computer's *random number generator*. With the same seed, the program makes the same random choices each time it is run:

```
seed(42)
colors = ["blue", "green", "yellow"]
color = choice(colors)
print(color)
```

```
| yellow
```

The above code prints "yellow" each time.

The reason the `choice()` function gives varying results by default is because the seed is different each time the program is started. Only by setting it manually, we get the same results.

Properties of the random number generator may become important if you are doing large scale simulations. These are properties such as how many different numbers can be drawn without correlation, and how long the repeating sequence is. In this text, we are more interested in the core concepts of random numbers, but we urge the reader to investigate random numbers in more detail if you will be doing numerical experiments in a scientific project.

6.4 More on functions

We end this chapter by looking at advanced uses and features of functions, as well as some common errors you might encounter.

6.4.1 Default function values

Function arguments can have *default values*. This allows us to not always write all the arguments. An example that we have already used extensively is the `plot()` function, where the user can choose not to give a line color as an argument, but instead use the default color. In order to create a function with default argument values, we use the following syntax:

```
def add(x, y=0, z=0):
    print("x =", x, ", y =", y, ", z =", z)
    return x + y + z
```

This function takes up to three arguments as input, and returns the sum of the numbers. The `print()` statement is just there to help us understand what is happening. This function can be called in several ways. We can call it with only the first argument set:

```
print(add(1))
```

```
x = 1 , y = 0 , z = 0
1
```

We can call it as if the function had no default values:

```
print(add(1, 2, 3))
```

```
x = 1 , y = 2 , z = 3
6
```

We can explicitly set the value of each argument, even in a different order than in the function definition:

```
print(add(1, z=3, y=2))
```

```
x = 1 , y = 2 , z = 3
6
```

We can also choose to omit only the second variable only, as long as we explicitly state the name of the third argument, `z`:

```
print(add(1, z=2))
```

```
x = 1 , y = 0 , z = 2
3
```

Arguments with default values have to come after arguments without default values. This means we cannot write:

```
print(add(z=2, 1))
```

```
Input In [220]
    print(add(z=2, 1))
               ^
SyntaxError: positional argument follows keyword argument
```

6.4.2 Global and local variables

Variables defined outside of any function are *global*. Variables defined inside a function are *local*, and not accessible outside the function:

```
def my_function():
    inside = 1 # this is a local variable

my_function()
print(inside) # variable is not defined outside the function
```

```
NameError                                 Traceback (most recent call last)
Input In [221], in <cell line: 5>()
      2     inside = 1 # this is a local variable
      3
      4 my_function()
----> 5 print(inside)

NameError: name 'inside' is not defined
```

Variables defined inside a function *only* exist inside that function. The error message tells us that the variable `inside` is not defined, because we are outside the function when calling `print()`.

To access variables that are defined outside a function, inside a function, you should pass its value to the function as an argument:

```
def my_function(value):
    print(value)

outside = 2
my_function(outside)
```

2

Remember that the name of the parameter that will be assigned the value of the argument does not have to be the same as the variable name outside the function.

Be aware that a variable whose value you pass to a function does not necessarily change if you modify it inside the function:

```
def my_function(value):
    value = 5
    print("Value inside:", value)

outside = 2
my_function(outside)
print("Value outside:", outside)
```

```
| Value inside: 5
| Value outside: 2
```

However, whether the variable changes outside or not depends on the type. If you pass a list to a function and add an element to the list inside the function, the change is also visible outside the function:

```
def my_function(value):
    value.append(5)
    print("Value inside:", value)

outside = [1, 2, 3]
my_function(outside)
print("Value outside:", outside)
```

```
| Value inside: [1, 2, 3, 5]
| Value outside: [1, 2, 3, 5]
```

You should avoid modifying arguments inside functions, because the argument might be changed outside the function as well.

6.4.3 Common errors in functions

Here we list some of the errors you will come across when writing and using your own functions. As with 'while' loops, it is easy to forget the colon at the end of the function definition. Forgetting the colon gives the following error:

```
def hello_world()
    print("Hello, world!")

hello_world()
```

```
| Input In [225]
|     def hello_world()
|         ^
SyntaxError: expected ':'
```

Another common error is wrong indentation in the function. Here, the indentation level of the return statement is too high:

```
def my_function(a):
    a = 5
```

```
print(a)
    return a

a = 1
a = my_function(a)
print(a)
```

```
Input In [226]
    return a
^
IndentationError: unexpected indent
```

It is also easy to forget a comma between function arguments. This is shown below, with a comma missing between the `y` and `z` arguments:

```
#           V---- Error: missing comma
def add(x, y=0 z=0):
    print("x =", x, ", y =", y, ", z =", z)
    return x + y + z
```

```
Input In [227]
def add(x, y=0 z=0):
^
SyntaxError: invalid syntax. Perhaps you forgot a comma?
```

6.4.4 Docstrings

A good practice is to always write a *docstring* whenever you implement a function. A docstring is a string with information about the function, simply describing what the function does. A normal string can be used, but most often we need more text than a single line to adequately describe what the function does. It is therefore most common to use multi-line strings. Multi-line strings use """ (triple quotes) at the start and end, and can go over several lines. Other than this, multi-line strings behave exactly like normal strings. When you use ? after a function to get more information about it, it is the docstring that is shown. An example of a function with a docstring using multi-line strings is shown below:

```
def my_function(some_value):
    """
    An example function that takes an input,
    and returns the input unchanged.
    """
    return some_value
```

6.5 Summary

In this chapter, we have introduced the laws of Mendelian inheritance by describing the experiments performed by Mendel. We introduced several key terms which are used in later chapters. Below follows a quick reference guide for these terms.

6.5.1 Biology glossary

Gene: Mendel's gene concept was that of an heritable factor. Today a gene has a much more technical definition.

Allele: Different variations of a gene (violet vs. white flowers) are called different alleles.

Dominant: Mendel found out that in many cases one allele always overrules the other. The overruling one is called dominant.

Recessive: The non-dominant allele; the allele that is always overruled by the other.

Heterozygous: An organism carrying two different alleles of a gene, is said to be heterozygous for that gene.

Homozygous: An organism carrying only one allele of a gene, is said to be homozygous for that gene.

Phenotype: A physical trait of an organism, such as flower color.

Genotype: The allele combination of an organism.

6.5.2 Mendel's model

1. An inherited trait is carried by a gene, and the genes exist in two alleles.
2. If the inherited alleles are different, one is dominant and overrules the other, which is recessive.
3. Each organism inherits two alleles, one from each parent.
4. Inherited alleles are chosen at random.

6.5.3 Functions

Reusable code can be put into *functions*. A function is given input that are assigned to *parameters* and gives output using a *return* statement:

```
def add_one(some_value):
    new_value = some_value + 1
    return new_value
```

Using the function involves writing the function name followed by parentheses, and any *arguments* for it inside the parentheses:

```
a = my_function(2)
print(a)
```

Docstring. A docstring is a single or multi-line string describing what the function does:

```
def my_function(some_value):
    """
    My awesome function that
    takes in the input, and returns it.
    """
    return some_value
```

Multiple inputs. A function can receive multiple arguments separated by commas. The following function takes two values, and returns the sum:

```
def add(a, b):
    """
    Returns the sum of the inputs.
    """
    return a + b

print(add(2, 3))
```

5

Default function values. A function can have *default values* that are given along with the arguments:

```
def add(x, y=0, z=0):
    print("x =", x, ", y =", y, ", z =", z)
    return x+y+z

print(add(1, 3))
print(add(1, z=2))
```

```
x = 1 , y = 3 , z = 0
4
x = 1 , y = 0 , z = 2
3
```

Global and local variables. Variables defined inside a function are not available outside the function:

```
def my_function():
    inside = 1 # local variable
    return inside

outside = my_function()
print(outside)
```

NameError

Traceback (most recent call last)

```

Input In [234], in <cell line: 6>()
      3     return inside
      5 outside = my_function()
----> 6 print(inside)

NameError: name 'inside' is not defined

```

Boolean values. Boolean values (`True` and `False`) represent truth values of logic.

Comparison operators. Comparison operators compare expressions on both sides of the operator and return `True` or `False`. A list of the most commonly used operators in Python is listed below:

Code	Meaning
<code>a == b</code>	<code>a</code> is equal to <code>b</code>
<code>a != b</code>	<code>a</code> is not equal to <code>b</code>
<code>a < b</code>	<code>a</code> is less than <code>b</code>
<code>a > b</code>	<code>a</code> is greater than <code>b</code>
<code>a <= b</code>	<code>a</code> is less than or equal to <code>b</code>
<code>a >= b</code>	<code>a</code> is greater than or equal to <code>b</code>
<code>a in b</code>	<code>a</code> is an element in the list <code>b</code>

The `in` operator tests if `a` is in the list `b`. This can also be used with other list-like structures. The keyword `not` can be inserted in front of a boolean expression to change the value from `True` to `False`, or from `False` to `True`.

6.5.4 If-tests

`if` tests allows us to run different code depending on certain conditions, for example the values of some of our variables. If a certain condition is met, i.e. evaluates to `True`, we do something, and if the condition is not met, i.e. evaluates to `False`, we skip it. The general structure of an `if` test is:

```

color = "red"

if color == "red":
    print("The color is red!")

```

The color is red!

If-else-tests. By adding an `else` after the block of code in the `if` test, we can tell the program what to do if the `if` condition does not evaluate to `True`. An example is shown below:

```

genotype = ["b", "B"]

if "B" in genotype:
    phenotype = "violet"

```

```
else:  
    phenotype = "white"  
  
print("The phenotype is", phenotype)
```

The phenotype is violet

6.5.5 Random choice in Python

We introduced the `choice()` function that picks one element at random from a list:

```
from pylab import *  
  
parent_1 = ['B', 'b']  
parent_2 = ['B', 'b']  
  
allele_1 = choice(parent_1)  
allele_2 = choice(parent_2)  
genotype = [allele_1, allele_2]  
  
print("Inherited genotype:", genotype)
```

Inherited genotype: ['b', 'B']

Chapter 7

DNA sequence analysis

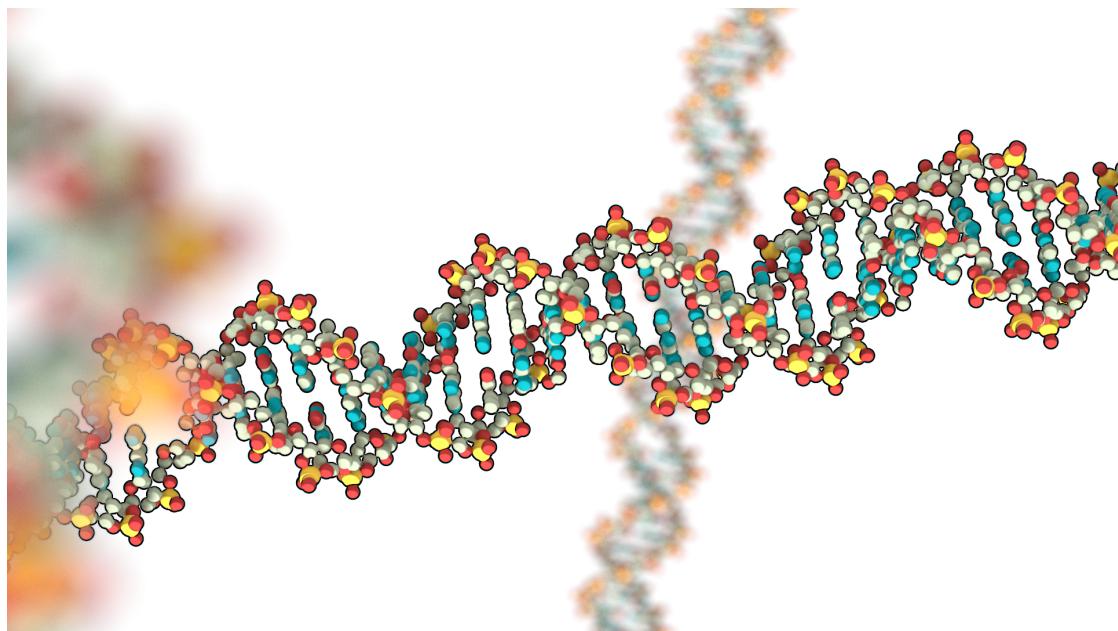


Figure 7.1: DNA, here depicted as a 'ball-and-stick' molecular model, contains information essential for the functioning of a living organism.

In the previous chapter, we discussed the basic principles of heredity based on discoveries made by Mendel. He used the term "heritable factor" for features that can be inherited from one generation to the next. Today, we know that these heritable factors reside in the DNA. DNA is present in all living organisms, and contains the genetic information inherited from parent to offspring. Certain regions of the DNA contain the genes, which provide the information for making specific proteins. Proteins, consisting of large chains of amino acids, perform many specialized tasks, such as transporting oxygen in the bloodstream, increasing the speed of specific

chemical reactions, and transmitting signals between cells. Some proteins are the building blocks for tissue such as muscles.

Biologists have been studying DNA for years as it proved extremely useful for a wide range of different areas. Thanks to DNA research we are able to detect and treat a wide range of inherited diseases that previously were untreatable, as well as predicting the onset of these diseases. One such disease, called *sickle-cell anemia*, causes the red blood cells of the body to take on a different shape; they become sickle-shaped. This non-standard shape causes blood clots, which can be deadly. In this chapter we will learn useful tools for analyzing DNA, which we apply to detect sickle-cell anemia in the next chapter.

In recent years, it has become possible to sequence the complete DNA of an organism, a process called *whole genome sequencing*. One common analysis, which you will learn in this chapter, is to measure the *GC-content* of an entire genome. Without going into details at this point, GC-content analysis can help us discover properties of organisms, such as what temperatures the organism is adapted to. Such techniques give amazing opportunities for understanding the DNA that makes up all organisms, but it also makes analyzing the data more difficult, because of the huge amounts of data involved. This contributes to why DNA research is one of the fields of biology that makes the most use of programming. DNA research yields large amounts of data and much of the data requires computational tools to handle, analyze, and understand the massive amounts of information it contains.

In this chapter, we discuss the structure of DNA and how proteins are made from the information embedded in the DNA. We begin by introducing DNA in more detail, and then introduce basic ways of reading and analyzing DNA data in Python. We finish this chapter by writing a Python program which translates the information contained in DNA into proteins.



Learning outcomes

After this chapter, you will know:

- the structure of DNA at different scales,
- how to count nucleotides in a DNA sequence,
- what GC/AT content means,
- how DNA is transcribed and translated to a protein, and
- how to translate a mRNA sequence to a protein.

The programming concepts we introduce in this chapter are:

- dictionaries,
- `elif` tests,
- `while` loops, and
- the `and` and `or` logical operators.

7.1 The structure of DNA

Before we start analyzing DNA, we give a brief overview of the structure of DNA. In this section we introduce the key terms needed to understand how the genetic material is stored. If you are interested in learning additional details about DNA, we recommend [11].

In prokaryotes, such as *Escherichia coli*, the DNA floats freely around the interior of the cell. In eukaryotes, such as humans, there is a central nucleus, surrounded by a membrane, see Figure 7.2. Inside the nucleus, there are structures called *chromosomes*, which contain the DNA, tightly packed together. Chromosomes come in pairs and humans have 23 pairs of chromosomes. The complete genetic material of an organism is called the *genome*.

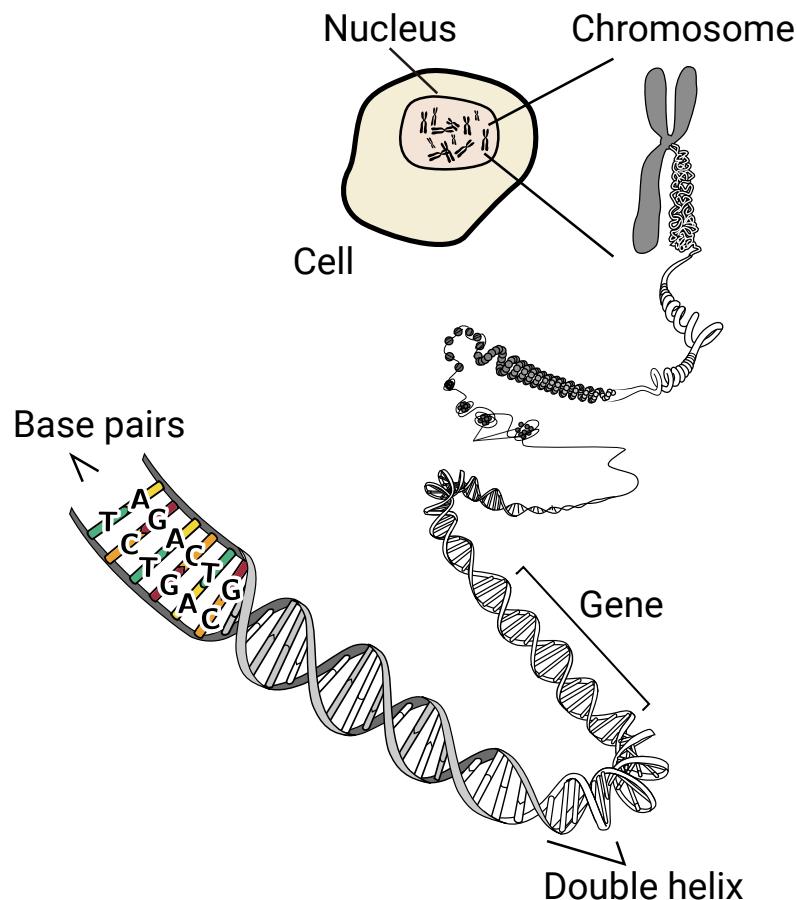


Figure 7.2: The structural features of DNA at different levels of complexity. Adapted from [3].

All cells in the human body have 23 pairs of chromosomes, with the exception of sperm and egg cells, that only have one of the chromosomes of each pair. When an egg cell is fertilized, i.e.,

when the sperm and egg cell merge, the result is again a complete set of 23 chromosome pairs. This means that the genetic material of the offspring is an equal mix between contributions from both parents: half the material comes from the mother, and half the material from the father. Knowing this helps understand the biological basis for the rules of inheritance found by Mendel we encountered in Section 6.1.1.

For a specific trait, the alleles are genetic variants, and when they are different they lie on the different chromosomes in a pair; one on the chromosome from mother and the other on the chromosome from the father.

When egg and sperm cells are formed, they receive one of the parental chromosomes at random. This results in each individual egg or sperm cell being a unique combination of chromosomes from the father and mother. This also relates to Mendel's rule that inherited alleles are chosen at random.

The individual chromosomes are made up of DNA (*deoxyribonucleic acid*) molecules. These are organized in two *strands* that wind around each other in a *double helix* structure, see Figure 7.2. If we zoom in on one of these strands, we see that it is made up of building chemical blocks called *nucleotides*. These are the primary building blocks of DNA. There are four different types of nucleotides, which also are called *bases*: adenine (A), guanine (G), thymine (T), and cytosine (C). We can thus represent the DNA molecule as a string of characters A, C, G and T. This representation is also called the *DNA sequence*.

A central feature of DNA organisation is the consistent pairing of the DNA bases. An adenine base in one strand is always paired with a thymine base in the opposite strand, and a guanine base is always paired with a cytosine. As a consequence, there are equal amounts of A and T, and equal amounts of G and C in a DNA double helix. This is known as the *AT/GC rule*. Note that this rule only applies to the DNA double helix, not single strands, like the ones you will encounter later in this chapter.

One of the most important features of DNA is that it contains *genes*. A *gene* is a region of DNA that encodes for a specific protein. Proteins consist of the molecular building blocks called *amino acids*. The amino acid composition of proteins is encoded in the DNA, by the order of nucleotides within the gene. More specifically, three subsequent nucleotides, called a *triplet*, code for a specific amino acid. For example, the DNA sequence GGT codes for the amino acid glycine, while CAA codes for glutamine. The sequence of triplets determines the order of amino acids in a protein, and thereby which protein that is encoded.

Technological advances have made it possible to sequence very large DNA sequences, such as entire genomes. *DNA sequencing* means to determine the precise order of nucleotides within a DNA strand, in other words, determining the DNA sequence. Determining the DNA sequence of one or more organisms is a first step towards studying genome organization, growth and development, gene functions, evolution and much more. Over the past years and decades, researchers have sequenced the genomes of many different individuals and species. In the table below you can see the number of base pairs in units of million base pairs (Mb) in the genome of a few different organisms.

Organism	Genome size
<i>Haemophilus Influenzae</i> (bacterium)	1.8 Mb
<i>Saccharomyces Cerevisiae</i> (yeast)	12 Mb
<i>Arabidopsis Thaliana</i> (plant)	135 Mb
<i>Caenorhabditis Elegans</i> (nematode)	100 Mb
<i>Homo sapiens</i> (human)	3200 Mb

The human genome consists of around 3,200,000,000, or 3200×10^6 , base pairs. Assuming we can read one letter each second, the time it would take us to read the entire human genome is

$$\frac{3200000000 \text{ letters}}{1 \text{ letters/s}} = 3200000000 \text{ s} = 101.4 \text{ years.} \quad (7.1)$$

This shows that it is not feasible to search through the human DNA by hand. It would take more than a century just to read through our DNA once, not even taking into consideration that we often want to find patterns in our DNA. Luckily, we can make computer programs that do this for us.

As noted before, any DNA molecule can be represented by a string of the characters A, C, G and T. This representation is ideal for processing DNA information with computers, and with a programming language like Python. In this chapter, we will demonstrate this by building a Python representation of the central biological processes that use the genetic information stored in a gene to build a protein. In the cell, this involves many steps and large complexes of enzymes. In the computer, we will see that we can represent these processes with relatively few lines of Python code.

But we will first start with a simple analysis of the composition of DNA in terms of its nucleotides.

7.2 Counting nucleotides

One of the simplest DNA analyses we can perform is to calculate the percentage of each nucleotide. This percentage varies from species to species, and knowing the frequency of each nucleotide may help us determine which species an unknown DNA sample comes from. The following table shows the percentage of each nucleotide in the genome from several organisms. Note that we have different percentages for A and T, and for G and C since this result is calculated for only one of the strands of DNA. The AT/GC rule is for double stranded DNA, and is not valid in this case.

Organism	Adenine	Thymine	Guanine	Cytosine
<i>E. coli</i>	26.0	23.9	24.9	25.2
Yeast	31.7	32.6	18.3	17.4
Turtle	28.7	27.9	22.0	21.3
Salmon	29.7	29.1	20.8	20.4
Chicken	28.0	28.4	22.0	21.6
Human	30.3	30.3	19.5	19.9

7.2.1 A simple program for counting nucleotides

Let us use Python to find the percentages of adenine, thymine, guanine, and cytosine in parts of the *E. coli* genome. First, we need to represent DNA sequences in Python. To do this, we are going to ignore that DNA is double stranded, and only represent one of the strands. This allows us to use strings in Python. We later work on large sequences by reading data from files, but for now we only examine a small part, 50 nucleotides, of the *E. coli* genome. We will represent this DNA sequence in Python with a variable called `e_coli_dna` of type string:

```
e_coli_dna = "AGCTTTCATTCTGACAGCAACGGCAATATGTCTCTGTGGATTAAT"
```

We want to find how many As this string contains. A manual count gives 13 occurrences of A. Counting A's with the computer is very similar to how we count by hand. How do we count the number of A's by hand?

- We start at the first character.
- If it is an A we increase the count of the number of A's we have found by one.
- We then go to the next character and repeat the process.

Let us implement this counting algorithm in Python. First we need to loop through the `e_coli_dna` character by character (nucleotide by nucleotide) and if the character is A, then we increase the number of A's by one. We assign the number of A's we have found to a variable, `A_count`, which we initially assign zero and is incremented by one each time we encounter an A. One thing to note is that we must compare with "A" and not "a", as strings in Python are case sensitive.

To access each nucleotide in the `e_coli_dna` string, we will make use of the fact that *string slicing* works the same way as list slicing, which we discussed in Section 2.3.3. This is because in Python, strings are considered as collections of characters - just as lists are collections of elements. Each character in a string has an index and can be accessed using that index. And as with list slicing, the first element has index 0. For example, to access the first character of our `e_coli_dna` string, we can do:

```
e_coli_dna = "AGCTTTCATTCTGACAGCAACGGCAATATGTCTCTGTGGATTAAT"  
print(e_coli_dna[0])
```

A

To access the last one we can use the negative index `-1`:

```
print(e_coli_dna[-1])
```

T

And to access the TTTT subsequence, which are the 4th to 7th element (including) we use:

```
print(e_coli_dna[3:7])
```

TTTT

Remember, with slices, the last index is not included.

To go over each nucleotide in our `e_coli_dna` sequence, we can now use a `while` loop to go over a set of indexes starting from 0 and ending with the last one (which is one less than the length of the sequence):

```
e_coli_dna = "AGCTTTTCATTCTGACAGCAACGGGCAATATGTCTCTGTGTGGATTAAT"
A_count = 0
index = 0
while index < len(e_coli_dna):
    nucleotide = e_coli_dna[index]
    if nucleotide == "A":
        A_count = A_count + 1
    index = index + 1
print(A_count)
```

13

We get the same number as when we counted by hand, which verifies that our program is correct.

7.2.2 Using a `for` loop to count nucleotides

When we introduced loops in Section 3.5, we mentioned that there are two different ways to write loops: `while` loops and `for` loops. In this chapter, we will introduce `for` loops.

So far we have used `while` to loop over the indexes so we could access each element of a list, as in

```
fruits = ["apple", "pear", "cherry", "berry"]
index = 0
while index < len(fruits):
    print(index, fruits[index])
    index = index + 1
```

0 apple
1 pear
2 cherry
3 berry

With a `for` loop, we can simplify this by writing:

```
for index in [0, 1, 2, 3]:
    print(index)
```

0
1
2
3

Here, the variable `index` is assigned each value of the list `[0, 1, 2, 3]` one by one. You could read the first line as 'for each element in the list `[0, 1, 2, 3]`'. This means that we can loop over our `fruits` list like this:

```
fruits = ["apple", "pear", "cherry", "berry"]
for index in [0, 1, 2, 3]:
    print(index, fruits[index])
```

```
0 apple
1 pear
2 cherry
3 berry
```

More general, a `for` loop assigns each element in a *collection* to a loop variable, and we can use this loop variable in the body of the loop. This means we can skip using the index entirely:

```
fruits = ["apple", "pear", "cherry", "berry"]
for fruit in fruits:
    print(fruit)
```

```
apple
pear
cherry
berry
```

Here, `fruit` is a variable name that each element in the `fruits` list is assigned to in order. In the first iteration of the `for` loop, `fruit` has the value `apple` (i.e. `fruits[0]`), in the second iteration, `fruit` has the value `pear` (i.e., `fruits[1]`), and so on. In each next iteration of the execution of the loop body, the next element in the list is printed to the screen. You will often see the list name as a plural noun and the loop variable as the corresponding singular noun, just like we used `fruits` and `fruit` in the above example.

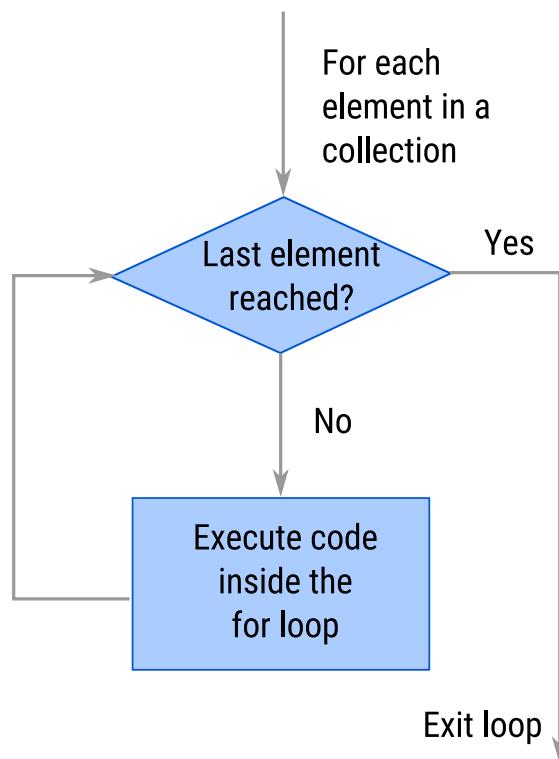
The colon marks the end of the `for` statement. As with `while` loops and `if` tests, the code block that follows is indented. This block is the body of the for loop, analogous to the indented body of a `while` loop.

The order of operations in a `for` loop is shown as a flowchart in Figure 7.3. As you can see, until the last element in the sequence is reached, the code inside the loop is executed. When the last element is reached, we exit the loop.

A `for` loop repeats a block of code statements for each element in a collection. Lists are an example of such collections, but `for` loops are not limited to loop over only those. You can also use `for` loop to iterate over each character in a string, which as mentioned above is also a collection (of characters).

```
word = "Hello"
for character in word:
    print(character)
```

```
H
e
```

Figure 7.3: Operation of a `for` loop.

```

l
l
o

```

Note that although we use `word` as the string (collection) name and `character` as loop variable name, because they describe well what we do, we could use any valid variable name instead. The program below gives the same result, although the variable names do not really make sense:

```

cat = "Hello"
for dog in cat:
    print(dog)
  
```

```

H
e
l
l
o

```

Going back to our example of counting the A's in our small part, of the *E. coli* genome, we did this with a `while` loop like this:

```
e_coli_dna = "AGCTTTCAATTCTGACAGCAACGGCAATATGTCTCTGTGTGGATTAAT"
A_count = 0
index = 0
while index < len(e_coli_dna):
    nucleotide = e_coli_dna[index]
    if nucleotide == "A":
        A_count = A_count + 1
    index = index + 1
print(A_count)
```

13

The corresponding program using a `for` loop looks like this:

```
e_coli_dna = "AGCTTTCAATTCTGACAGCAACGGCAATATGTCTCTGTGTGGATTAAT"
A_count = 0
for nucleotide in e_coli_dna:
    if nucleotide == "A":
        A_count = A_count + 1

print(A_count)
```

13

7.2.3 Shorthands for common variable operations

Incrementing a value is often done in computer programs. We have already used it for all the `while` loops we have written. There exists a shorthand syntax for doing incrementing, and similar types of operations.

Code	Equivalent code
<code>n += 1</code>	<code>n = n + 1</code>
<code>n -= 1</code>	<code>n = n - 1</code>
<code>n *= 1</code>	<code>n = n*1</code>
<code>n /= 1</code>	<code>n = n/1</code>

Applying this shorthand to our program means changing `A_count = A_count + 1` to `A_count += 1`.

To calculate the percentage of adenine, we divide the number of adenine occurrences with the total number of nucleotides in `e_coli_dna` and multiply with 100. Similarly to lists, we get the number of elements in a string with the `len()` function and the adenine percentage is be calculated as:

```
A_percentage = 100 * A_count / len(e_coli_dna)
```

The complete program is:

```
1 e_coli_dna = "AGCTTTCAATTCTGACAGCAACGGCAATATGTCTCTGTGTGGATTAAT"
2
3 A_count = 0
4 for nucleotide in e_coli_dna:
```

```

5     if nucleotide == "A":
6         A_count += 1                         # New
7
8     A_percentage = 100 * A_count / len(e_coli_dna)    # New
9     print("The adenine percentage is", A_percentage, "%") # New

```

The adenine percentage is 26.0 %

This percentage is the same as we find in the previous table, even though the numbers in the table are from the entire genome while we examine only a fragment of the *E. coli* DNA.

7.2.4 If-elif-else-tests to count all four nucleotides

Let us expand our code to count thymine, cytosine, and guanine, as well as adenine. This could be done by adding three more `if` tests to check if `nucleotide` is equal to "T", "C", or "G", respectively. It is however more common in such cases to use `elif`.

An `if-elif-else` test does something in one case, something else in another case, and something else for all other cases, see Figure 7.4. You could say that `elif` stands for “else if”. This becomes very useful when multiple options are available and you want to make different choices based on the different options.

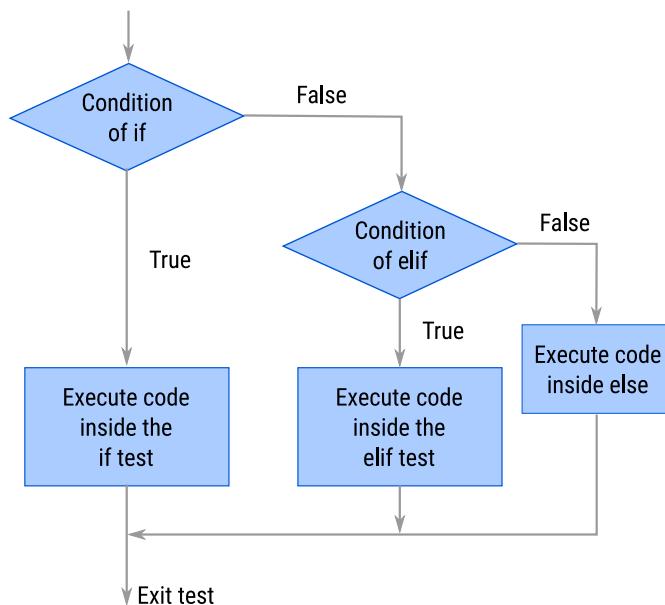


Figure 7.4: Flowchart of an `if-elif-else` test.

The difference between using two `if` tests after another and a `if-elif` test is that the `elif` is only evaluated if the first `if` statement is evaluated as false.

```
A_count = 13
```

```
if A_count > 2:  
    print("A_count is greater than two")  
elif A_count > 1:  
    print("A_count is greater than one")
```

A_count is greater than two

In the case of two `if` tests, both tests are evaluated no matter the result of the first.

```
A_count = 13  
  
if A_count > 2:  
    print("A_count is greater than two")  
if A_count > 1:  
    print("A_count is greater than one")
```

A_count is greater than two
A_count is greater than one

For our nucleotide counting program, we need to test if the nucleotide is either A, T, G or C, and if the nucleotide is something else, we should print a warning to screen. We achieve this by using multiple `elif` statements in one `if-elif-else` test. Lastly we calculate the percentage of each nucleotide in the DNA fragment.

```
1 e_coli_dna = "AGCTTTCAATTCTGACAGAACGGCAATATGTCTCTGTGATTAAAT"  
2  
3 A_count = 0  
4 T_count = 0  
5 G_count = 0  
6 C_count = 0  
7  
8 for nucleotide in e_coli_dna:  
9     if nucleotide == "A":  
10         A_count += 1  
11     elif nucleotide == "T":  
12         T_count += 1  
13     elif nucleotide == "G":  
14         G_count += 1  
15     elif nucleotide == "C":  
16         C_count += 1  
17     else:  
18         print("could not recognize the nucleotide")  
19  
20  
21 A_percentage = 100 * A_count / len(e_coli_dna)  
22 T_percentage = 100 * T_count / len(e_coli_dna)  
23 G_percentage = 100 * G_count / len(e_coli_dna)  
24 C_percentage = 100 * C_count / len(e_coli_dna)  
25  
26 print("The adenine percentage is", A_percentage, "%")  
27 print("The thymine percentage is", T_percentage, "%")  
28 print("The cytosine percentage is", C_percentage, "%")  
29 print("The guanine percentage is", G_percentage, "%")
```

```
The adenine percentage is 26.0 %
The thymine percentage is 34.0 %
The cytosine percentage is 18.0 %
The guanine percentage is 22.0 %
```

These results do not match the percentages in the previous table, but that should not be a surprise. The percentages in the table are from the entire *E. coli* genome, while we only examine 50 nucleotides.

The above code repeats almost equal code lines for each nucleotide. We are lazy and would like to do as little work as possible, so the above solution is far from ideal. We could instead use a `for` loop together with two lists, one to store our nucleotides, and one to store our counts:

```
nucleotides = ["A", "T", "G", "C"]
nucleotides_counts = [0, 0, 0, 0]
```

The problem with this solution is that we have to keep track of which count belongs to which nucleotide. Instead, we can use *dictionaries*, which are made for keeping track of pairs of names and values.

7.2.5 Dictionaries: Storing nucleotide counts

Roughly speaking, dictionaries are like lists, but their indices are not necessarily numbers. Their indexes are called *keys* and can be strings, numbers, and many other types. For each key, the dictionary holds a *value*, which is why each element in the dictionary is often called a *key-value pair*. A dictionary is created by using {} (curly brackets):

```
nucleotides_count = {"A": 0, "T": 0, "G": 0, "C": 0}
```

In this example, the nucleotides (A, T, G, and C) are the *keys*, while the 0s are the *values*.

Dictionaries with many elements are often written over multiple lines to make them easier to read. The following dictionary is the same as the above:

```
nucleotides_count = {
    "A": 0,
    "T": 0,
    "G": 0,
    "C": 0
}
```

Printing the dictionary `nucleotides_count` gives

```
print(nucleotides_count)
```

```
{'A': 0, 'T': 0, 'G': 0, 'C': 0}
```

Note that, in earlier versions of Python, the keys are not necessarily printed in the same order as they are defined.

We retrieve the value of a specific key by putting the key in square brackets:

```
print(nucleotides_count["G"])
```

```
| 0
```

This looks similar to list slicing, but instead of having the index number inside the square brackets, we use the key to retrieve the corresponding element.

To change a value for an existing key-value pair in the dictionary, we write:

```
nucleotides_count["G"] = 1
print(nucleotides_count["G"])
```

```
| 1
```

We add an additional key-value pair by assigning a value to a key that does not already exist:

```
nucleotides_count["N"] = 0
print(nucleotides_count)
```

```
| {'A': 0, 'T': 0, 'G': 1, 'C': 0, 'N': 0}
```

The keys and values can be extracted from a dictionary:

```
print(nucleotides_count.keys())
```

```
| dict_keys(['A', 'T', 'G', 'C', 'N'])
```

```
print(nucleotides_count.values())
```

```
| dict_values([0, 0, 1, 0, 0])
```

Both `keys()` and `values()` return a view to the dictionary. This means that if you modify the original dictionary, the contents of the view also changes:

```
key_view = nucleotides_count.keys()
nucleotides_count["X"] = 0
print(nucleotides_count)
```

```
| {'A': 0, 'T': 0, 'G': 1, 'C': 0, 'N': 0, 'X': 0}
```

```
print(key_view)
```

```
| dict_keys(['A', 'T', 'G', 'C', 'N', 'X'])
```

We can convert the dictionary view to a list simply by:

```
nucleotides = list(nucleotides_count.keys())
print(nucleotides)
```

```
| ['A', 'T', 'G', 'C', 'N', 'X']
```

Let us use a dictionary to count the percentage of each nucleotide. We know there are four nucleotides in `e_coli_dna`, and create a dictionary with the four nucleotides as keys with corresponding counts as values, all initially set to zero.

```
e_coli_dna = "AGCTTTCTATTCTGACAGCAACGGCAATATGTCTCTGTGATTAAAT"

nucleotides_count = {"A": 0, "T": 0, "G": 0, "C": 0}
```

We want to loop through the keys (nucleotides) in the `nucleotides_count` and count the number of occurrences of each key in `e_coli_dna`. Just as a list is a collection of elements we can loop over, a dictionary is a collection of key-value pairs. This allows us to loop over the keys in a dictionary and print their values, like this:

```
for nucleotide in nucleotides_count:
    print(nucleotide, nucleotides_count[nucleotide])
```

```
A 0
T 0
G 0
C 0
```

In this `for` loop, the `nucleotide` variable gets assigned each key from the dictionary in order.



Looping over the dictionary keys (nucleotides) with a `while` loop

While it is the easiest to loop over the dictionary keys (nucleotides) by using a `for` loop, it can also be done using a `while` loop. The solution is to create a `nucleotides` list from the dictionary keys, and then index this `nucleotides` list to get the current nucleotide, which we use to index `nucleotides_count`:

```
nucleotides = list(nucleotides_count.keys())

index = 0
while index < len(nucleotides):
    nucleotide = nucleotides[index]
    print(nucleotide, nucleotides_count[nucleotide])

    index = index + 1
```

```
A
```

```

0
T 0
G 0
C 0

```

We have so far looped through `e_coli_dna` character by character to count the number of each nucleotide. There exists a much easier way to do this. Strings in Python have a function called `count(substring)` that returns the number of occurrences of `substring`. To count the number of A's in the DNA string we write:

```
print(e_coli_dna.count("A"))
```

13

which is what we obtained with our previous implementation. In fact our implementation for counting the nucleotides is exactly what the `count()` function does: search for and count the number of A's in `e_coli_dna`.

Below we use the `count()` function to count the nucleotides and store the result in a dictionary:

```

e_coli_dna = "AGCTTTTCATTCTGACAGCAACGGGCAATATGTCTCTGTGGATTAAAT"

nucleotides_count = {"A": 0, "T": 0, "G": 0, "C": 0}

for nucleotide in nucleotides_count:
    nucleotides_count[nucleotide] = e_coli_dna.count(nucleotide)

print(nucleotides_count)

```

{'A': 13, 'T': 17, 'G': 11, 'C': 9}

Here, we loop over the keys in `nucleotides_count`, count the number of occurrences of each key, and store the result as the corresponding value of the key in the dictionary `nucleotides_count`.

Similarly, to calculate the percentages we use:

```

nucleotides_percentage = {"A": 0, "T": 0, "G": 0, "C": 0}

for nucleotide in nucleotides_percentage:
    nucleotides_percentage[nucleotide] = 100 * nucleotides_count[nucleotide] / len(e_coli_dna)

```

In the code above, we create a new dictionary with the four nucleotides as keys, loop over each key, and divide the count `nucleotides_count[nucleotide]` by the total length of the DNA string. The result is stored in the dictionary. Finally, we print the result:

```
print(nucleotides_percentage)
```

{'A': 26.0, 'T': 34.0, 'G': 22.0, 'C': 18.0}

This code gives the same result as previously, but is less than half the length of the previous implementation. The code examples in this section is a good illustration of how there often are multiple many ways of solving a given problem in Python (or other programming languages). Some solutions are better than others, but they all get the job done. As a start you should make sure that your code works, and solves your problem. Only thereafter should you try to optimize your solution.

7.3 Calculating GC-content

GC-content is defined as the percentage of guanine and cytosine in a fragment of DNA. It is often used when analyzing DNA sequences, as it is correlated with biological features of genome organization. Examples are genes, with a higher GC-content found within genes than in the rest of the genome, DNA sequences that are repeated in the genome, and how likely regions of the genome are to be mutated. GC-content also provides useful information about a DNA fragment, such as its stability, melting temperature, and structure. These properties originate from the difference in bonding between the GC pairs and AT pairs.

A GC base pair consists of three hydrogen bonds while an AT pair only has two hydrogen bonds, as shown in Figure 7.5. Therefore, a GC base pair is more stable and subsequently a higher temperature is required to break the bonds. Thus, a DNA fragment with high GC-content is more stable. Bacteria that thrive at high temperatures tend to have a high GC-content, which likely is an adaptation to high temperatures to prevent the DNA double helix from breaking apart.

Finding the GC-content of a DNA strand is the same as adding the percentages of the nucleotides G and C together. When we have our `nucleotides_count` dictionary we can achieve this by:

```
GC_percentage = nucleotides_percentage["G"] + nucleotides_percentage["C"]
print("GC percentage:", GC_percentage)
```

```
| GC percentage: 40.0
```

The content of GC and AT must add up to 100%. This means that once we know the GC-content, we also know the AT content:

```
AT_percentage = 100 - GC_percentage
print("AT percentage:", AT_percentage)
```

```
| AT percentage: 60.0
```

We could also add the A and T percentages together:

```
AT_percentage = nucleotides_percentage["A"] + nucleotides_percentage["T"]
print("AT percentage:", AT_percentage)
```

```
| AT percentage: 60.0
```

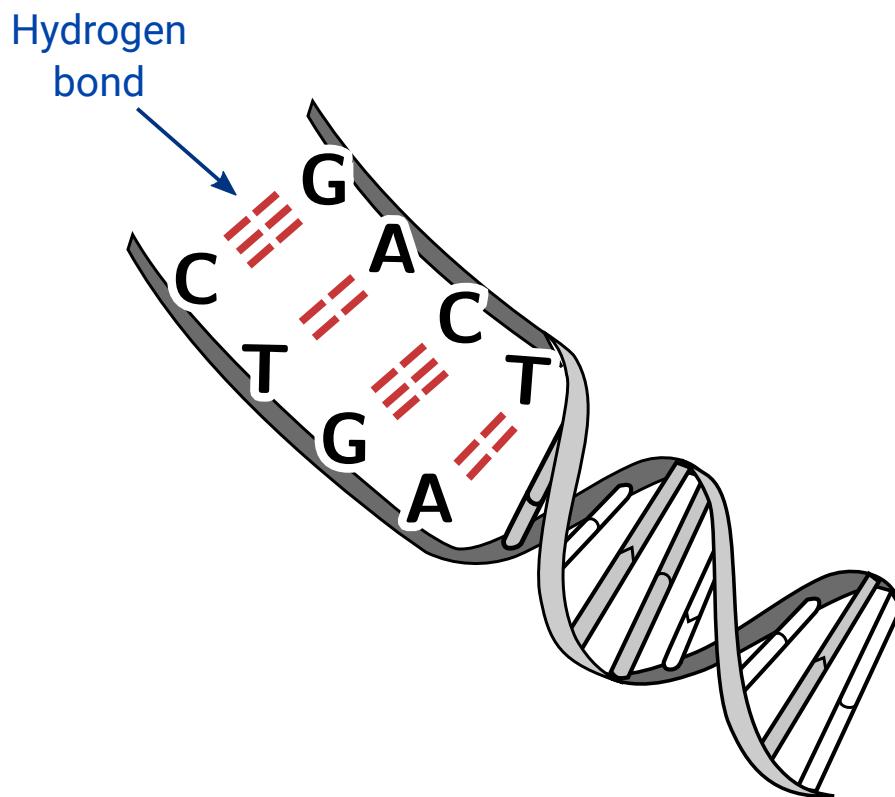


Figure 7.5: Nucleotide bonds between GC pairs and AT pairs. Dashed red lines correspond to hydrogen bonds. Adapted from [3].

The GC-content for our small fragment is higher than the AT-content. Since we only examine a short piece of *E. coli* DNA we are not able to make any definite conclusions why this would be the case.

7.4 Transcription of DNA to mRNA

As mentioned in the introduction, genes provide the instructions for making specific proteins. These proteins perform tasks in the cells and are involved in the expression of genetic traits. To understand inheritance, it is therefore important to know how the information stored in the DNA is used to make proteins. The process of making proteins consists of two major steps. The first step is *transcription*, where parts of a DNA strand are converted to an intermediate called RNA, and is the topic of this section. The second step is *translation*, where the intermediate RNA is converted to proteins, and is the topic of the next section.

RNA is similar to DNA, but there are a couple of important differences. RNA is a chain of nucleotides, but consists of a single-strand rather than the paired double-strand of DNA. Additionally, RNA uses the nucleotide uracil (U) instead of thymine (T). Different types of RNA exist, but the one we are interested in is called messenger RNA (mRNA). DNA is transcribed to mRNA inside the nucleus, while the resulting mRNA moves outside the nucleus, to the areas of the cell where proteins are produced. During transcription, often several copies of mRNA are produced for a single DNA sequence. Each mRNA sequence can be transcribed to a protein and this makes it possible to produce several units of a protein simultaneously.

Specific sequences along the DNA mark where the transcription of a gene begins and ends, and subsequently where the mRNA strand starts and ends. The transcription process that creates mRNA from DNA consists of several steps, involving many enzymes working together, but describing these in any detail is beyond the scope of this book.

How can we perform this transcription with Python? First, we need to find a DNA sequence to work with. This time, we will use a more or less random sequence, not specifically derived from any existing genome:

```
DNA = "CAATGGCACACATTCAAGTCTTCCAATAAATAGGAC"
```

As we described before, the important differences between DNA and RNA are that RNA is single stranded, and that it uses uracil (U) instead of thymine (T). We already represent the DNA as a single strand. What then remains for us to go from a DNA string to a mRNA string is substituting all occurrences of "T" with "U".

Strings have a `replace(substring_1, substring_2)` function that replaces all occurrences of `substring_1` with `substring_2` and returns a new string. We use that to replace all occurrences of T with U, and then print the result.

```
DNA = "CAATGGCACACATTCAAGTCTTCCAATAAATAGGAC"  
mRNA = DNA.replace("T", "U")  
  
print(mRNA)
```

```
CAAUGGCCAACAUUCAAGUCUUCCAAUAAAUGGAC
```

This is the corresponding mRNA sequence of the piece of DNA.

Transcribing a DNA sequence to its corresponding mRNA is something we want to do many times for different DNA sequences. Therefore, it is a good idea to put the code inside a function

that has the DNA sequence as input and returns the corresponding mRNA. To make the function easier to understand we also add a docstring that explains what the function does.

```
def transcribe(dna):
    """
    Translate a DNA string to its corresponding mRNA
    """
    rna = dna.replace("T", "U")
    return rna
```

We can now use our function as follows:

```
DNA = "CAATGGCACACATTCAAGTCTTCCAATAAATAGGAC"
mRNA = transcribe(DNA)

print(mRNA)
```

CAAUGGCAACAUUCAAGUCUUCCAAUAAAUGGAC

In prokaryotic cells, the transcription process is finished as soon as a mRNA strand is created. In eukaryotic cells, this transcription process results in what is called pre-mRNA, which needs additional processing, called *RNA splicing*. The pre-mRNA strand has long regions of so-called non-coding stretches of nucleotides in between regions which do code for protein. These non-coding regions are called *introns*. The coding regions are called *exons* and are combined in the final mRNA that codes for the protein, while the introns are spliced out. Introns and exons are illustrated in Figure 7.6. Introns regulate how the exons are spliced together. Depending on how the exons are spliced together, i.e. how they are combined, one gene can give rise to more than one protein.

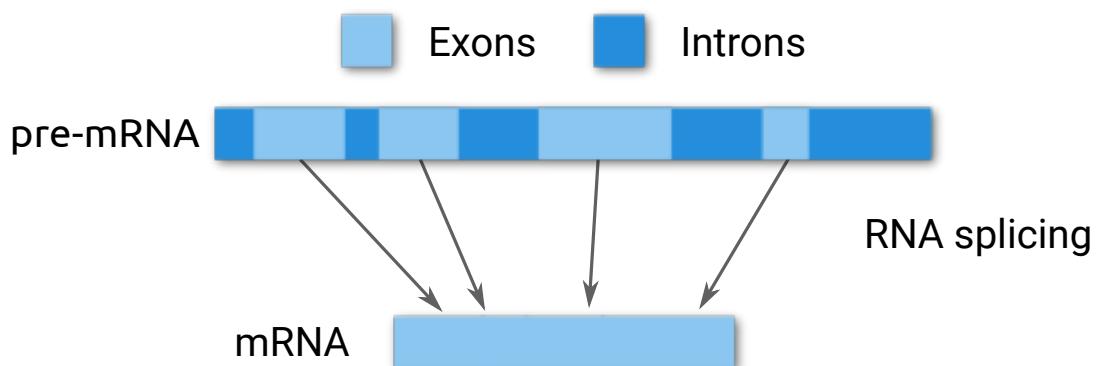


Figure 7.6: Exons and introns in a DNA strand.

Transcription for eukaryotic cells is more complicated due to introns and exons. We cannot just substitute T with U because we need to know which regions of the pre-mRNA are supposed to be assembled into the mRNA. This is very complicated since the behavior varies from one gene to the next and lies outside the scope of this book. We will therefore not transcribe a eukaryotic DNA strand.

7.5 Translating a mRNA strand to a protein

In the next part of the process, the mRNA is used to assemble a protein. This process called *translation*. As with transcription, translation involves many steps in how the mRNA is read and the protein is put together in the cell, again involving many enzymes. Just as with translation, we do not go into details here, also because these steps are too difficult to model. However, using the computer to find out which protein is created from the mRNA is much easier.

The building blocks of proteins are amino acids. Our cells use 20 different amino acids to create proteins. The question then comes up how the recipe for these proteins can be encoded in the DNA when it consists of only four different nucleotides? If each nucleotide corresponds to a single amino acid, the cell would only be able to make four different amino acids. Each amino acid must therefore be coded for by a combination of several nucleotides. If each amino acid were coded for by a combination of two nucleotides we have a total of 16 possibilities, four for the first nucleotide multiplied with four for the second. This is still insufficient. If we use three nucleotides, we get 64 possible combinations, which is more than enough. This means three nucleotides is the smallest possible unit that can encode for all possible amino acids.

It turns out that nature has chosen this solution with triplets (three consecutive nucleotides) as the unit to code for proteins. We call such a triplet a *codon*. The mRNA codon wheel in Figure 7.7 shows all codon combinations, and which amino acid they code for. The wheel shows the common three-letter abbreviations for the amino acids and the single-letter abbreviations on the outside. We will use these single-letter abbreviations in the Python model. As the codon wheel shows, there is no ambiguity in the sense that each codon only codes for a single amino acid. However, there is redundancy, the same amino acid can be produced by many codons. Note the presence of codons that are labelled "STOP", "STP", and *. We will return to these later. The process of translating a mRNA to protein then consists of taking all consecutive codons (triplets of consecutive nucleotides), and using these to assemble together the corresponding amino acids one by one.

7.5.1 Converting codons to amino acids

To set up our Python model of translation works, let us translate our piece of mRNA that we obtained before from translating the DNA sequence:

```
print(mRNA)
```

```
| CAAUGGCAACAUUCAAGUCUUCCAAUAAAUGGAC
```

Once we have our mRNA string, we want to read through it in sequences of three nucleotides, or codons, at the time. We thus need to create slices of the mRNA string starting at indices that grow in steps of three. This process is illustrated for the first three iterations in Figure 7.8.

Once we have extracted these codons, we will for each of them need to find the corresponding amino acid and use these to build the protein sequence.

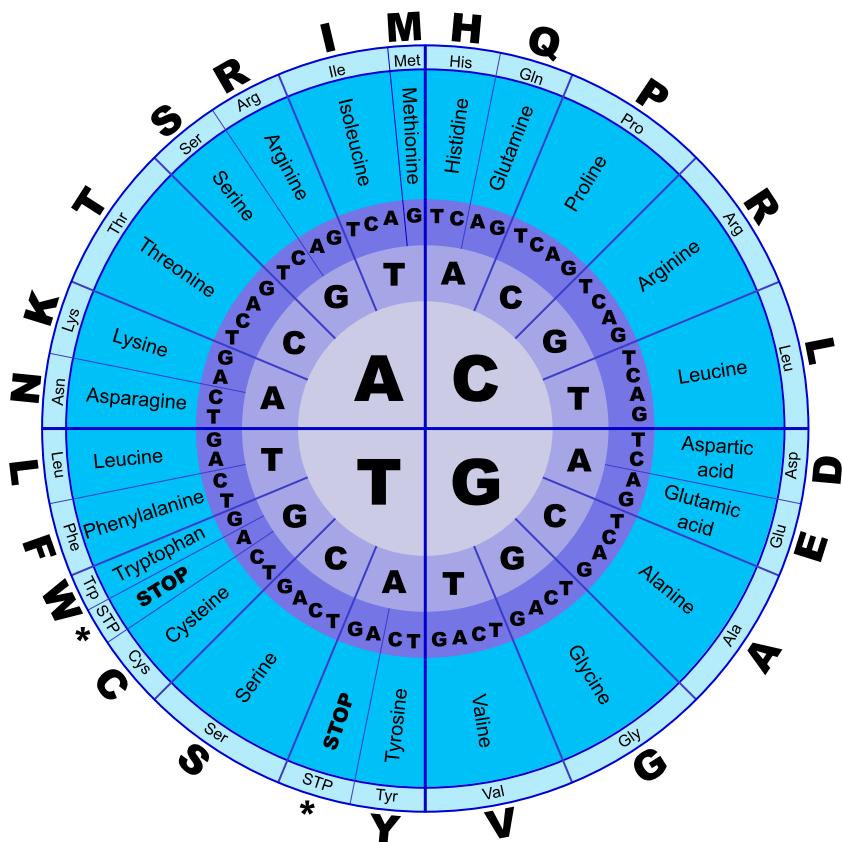


Figure 7.7: mRNA codon wheel. Start at the center and move outwards reading each letter of the codon to find the amino acid that codon encodes (denoted in the outermost circle). For example, the codon UCU codes for Serine. The wheel also shows the common three-letter abbreviations for amino acids (Ser for Serine) with the single-letter abbreviations in parenthesis (S). Adapted from [1].

Iteration	Slice	CAAUGGCAACAUUCAAGUCUUCAAUAAAUGGAC
1.	[0:3]	CAA
2.	[3:6]	UGG
3.	[6:9]	CAA

Figure 7.8: The first codons when iterating over the mRNA string with steps of three.

We will first show an implementation in Python using a for loop and the `range()` command. However, to make our model more similar to how the biological process of translation works in cells, we will switch to a while loop in the second implementation.

7.5.2 Using the range function in for loops

A function often used in cases where one needs to create a list of incrementally increasing numbers is the `range()` function, `range(start, stop, step)`. `range()` takes a `start` value, a `stop` value and the `step` size and generates a sequence of numbers from `start` up to but not including `stop` while incrementing the numbers with a step size of `step`.

Let's illustrate the `range()` function with an example:

```
for i in range(4, 16, 2):
    print(i)
```

```
4
6
8
10
12
14
```

In this example we generate a sequence starting from 4 up to, but not including 16, with step size 2.

If the `step` argument is omitted, the `step` defaults to 1. Calling the function with only `start` and `stop`, that is `range(start, stop)`, is thus the same as calling `range(start, stop, 1)`:

```
for i in range(4, 10):
    print(i)
```

```
4
5
6
7
8
9
```

The `range()` function is made in such a way that if it takes two arguments, then the first argument will become `start`, and the second argument `stop`: `range(start, stop)`. If it takes one argument then `start` defaults to 0, and the argument will become `stop`: `range(stop)`, which is the same as `range(0, stop)`:

```
for i in range(6):
    print(i)
```

```
0
1
2
3
4
5
```

This might seem confusing, but it makes `range` much more intuitive when using two arguments, as it otherwise would be hard to remember that we had to write it as `range(stop, start)`.

All arguments to the `range()` function must be integers, so `range(0.5, 10)` does not work.

We can use the `range` function to generate our subsequences of three nucleotides at the time with a for loop. We loop over the indices of the mRNA string from 0 to the length of the mRNA string, `len(mRNA)`, with steps of three, `range(0, len(mRNA), 3)`.

```
for index in range(0, len(mRNA), 3):
    codon = mRNA[index:index + 3]
    print(codon)
```

```
CAA
UGG
CAA
CAU
UUC
AAG
UCU
UCC
AAU
AAA
UAG
GAC
```

With `mRNA[index:index + 3]`, we start the slice at the current value of `index`, and end it three positions further, i.e. at `index + 3`.

The next step is to implement the process that converts each codon into the corresponding amino acid. Since each codon encodes for one specific amino acid, a reasonable way to link codons to amino acids is to use a dictionary with codons as keys, and amino acids as values. We use the single letter abbreviations for the amino acids. Note how some codons have an asterisk * as their corresponding amino acid. These are the so-called stop codons, and we will come back to those later.

Our dictionary of codons and corresponding amino acids then becomes:

```
codon_dict = {
    "UUU": "F", "UUC": "F", "UUA": "L", "UUG": "L",
    "UCU": "S", "UCC": "S", "UCA": "S", "UCG": "S",
    "UAU": "Y", "UAC": "Y", "UAA": "*", "UAG": "*",
    "UGU": "C", "UGC": "C", "UGA": "*", "UGG": "W",
    "CUU": "L", "CUC": "L", "CUA": "L", "CUG": "L",
    "CCU": "P", "CCC": "P", "CCA": "P", "CCG": "P",
    "CAU": "H", "CAC": "H", "CAA": "Q", "CAG": "Q",
    "CGU": "R", "CGC": "R", "CGA": "R", "CGG": "R",
    "AUU": "I", "AUC": "I", "AUA": "I", "AUG": "M",
    "ACU": "T", "ACC": "T", "ACA": "T", "ACG": "T",
    "AAU": "N", "AAC": "N", "AAA": "K", "AAG": "K",
    "AGU": "S", "AGC": "S", "AGA": "R", "AGG": "R",
    "GUU": "V", "GUC": "V", "GUA": "V", "GUG": "V",
    "GCU": "A", "GCC": "A", "GCA": "A", "GCG": "A",
    "GAU": "D", "GAC": "D", "GAA": "E", "GAG": "E",
    "GGU": "G", "GGC": "G", "GGA": "G", "GGG": "G"
}
```

If we retrieve an item from this dictionary with a codon as index, we get the corresponding amino acid. For example, the amino acid encoded by codon UUU is:

```
print(codon_dict["UUU"])
```

F

If we try to retrieve a value by using a key that does not exist, we will get an error message, in this case of type `KeyError`:

```
print(codon_dict["AGC"]) # exists in the dictionary
print(codon_dict["ABC"]) # does not exist in the dictionary
```

S

```
-----
KeyError                                                 Traceback (most recent call last)
Input In [291], in <cell line: 2>()
      1 print(codon_dict["AGC"]) # exists in the dictionary
----> 2 print(codon_dict["ABC"])

KeyError: 'ABC'
```

Inside our `for` loop we use each current codon as index to to retrieve the corresponding amino acid from `codon_dict`. To store our result for later use, we add each `amino_acid` to a string called `amino_acid_sequence`, that initially is an an empty string:

```
amino_acid_sequence = ""
for index in range(0, len(mRNA), 3):
    # Obtain the codon by slicing the mRNA
    codon = mRNA[index:index + 3]
    # Retrieve the corresponding amino acid
    # and add it to the protein
    amino_acid = codon_dict[codon]
    amino_acid_sequence += amino_acid

print(amino_acid_sequence)
```

QWQHFKSSNK*D

Note how we use the `+=` shorthand to add the amino acid to the `amino_acid_sequence` string. This program shows an amino acid sequence that can be created from this mRNA, but there is a problem we have not addressed. How do the enzymes responsible for translation know from where to start reading the codons?

7.5.3 Reading frames

Our ability to read mRNA is dependent on the correct grouping of codons; they have to be in the correct *reading frame*. A reading frame is one way of dividing mRNA into a set of non

overlapping codons. Consider the string “momhasthecar”. If we read from the first letter in groups of three (adding a space after each third character) we get: “mom has the car”. However, if we start at the second letter we end up with: “omh ast hec ar”. Just as this sentence becomes very different from the first one (and meaningless to us), a sequence of mRNA becomes vastly different if we start to read from the wrong position. Figure 7.9 illustrates three different reading frames for our mRNA.

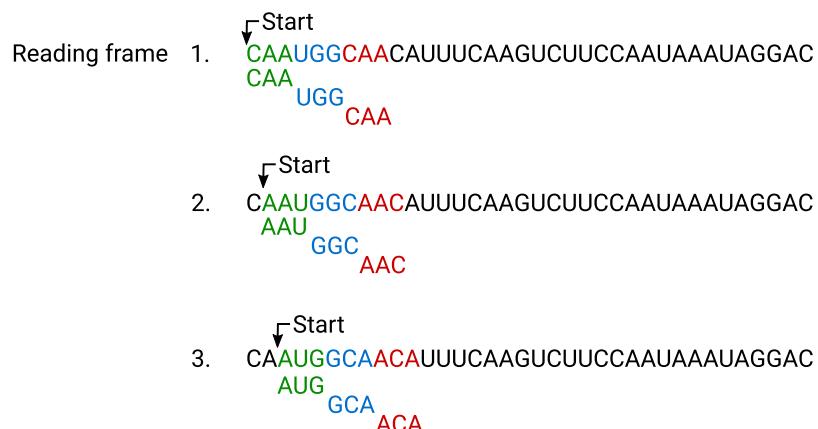


Figure 7.9: Different reading frames and the resulting codons.

Consider the mRNA from above:

```
mRNA = "CAAUGGCAAACAUUCAAGUCUUCCAAUAAAUGGAC"
```

If we start reading from the second nucleotide instead of the first, which you do by changing the starting index from 0 to 1, we get the following codons:

```
#           1 instead of 0
#
#           |
for index in range(1, len(mRNA), 3):
    codon = mRNA[index:index + 3]
    print(codon)
```

```
AAU
GGC
AAC
AUU
UCA
AGU
CUU
CCA
AUA
AAU
AGG
AC
```

We now see that we do not use the first nucleotide, but also that the last 'codon' is only two nucleotides: when we start from index 1, the last codon becomes AC. In the last iteration of the

loop, `index` becomes 34, and `index + 3` becomes 37, while the last index of the string `mRNA` is 35 (its length is 36 nucleotides). You may have expected an error when trying to obtain a slice beyond the end of the string, but in Python, such slicing operations will always return a string. If *both* indices don't fall within the range of the number of elements in the string, an empty string is returned. The same is true for slicing beyond the end of a list, which returns an empty list.

We should avoid trying to translate that last, too short codon. If we tried, we would get a `KeyError` since the codon `AC` is not a key in our `codon_dict` dictionary. We can avoid this problem by adding an `if` test to check the length of the codon, and only retrieve the corresponding amino acid when the codon is 3 nucleotides long:

```
if len(codon) == 3:  
    amino_acid = codon_dict[codon]  
    amino_acid_sequence += amino_acid
```

Here it is a good idea to add a comment that explains why we have added the `if` test, as it is not directly clear from the code:

```
# The last slice of the string may be shorter  
# if the length of the sequence to translate is not a multiple of 3 bases.  
# We thus need to check that the codon is 3 nucleotides.  
if len(codon) == 3:  
    amino_acid = codon_dict[codon]  
    amino_acid_sequence += amino_acid
```

This way, if we revisit this program some time from now, we will still understand why we wrote it this way. We thus translate the reading frame starting from the second nucleotide like this:

```
amino_acid_sequence = ""  
  
#           1 instead of 0  
#           |  
for index in range(1, len(mRNA), 3):  
    codon = mRNA[index:index + 3]  
    # The last slice of the string may be shorter  
    # if the length of the sequence to translate is not a multiple of 3 bases.  
    # We thus need to check that the codon is 3 nucleotides.  
    if len(codon) == 3:  
        amino_acid = codon_dict[codon]  
        amino_acid_sequence += amino_acid  
  
print(amino_acid_sequence)
```

NGNISSLPINR

Rather than manually changing the reading frame a third time, let us automate the process to investigate all three reading frames in one go. For this, we use a nested `for` loop. Remember that we saw nested `while` loops in Section 6.1.3. Just as with `while` loops, we can make nested `for` loops.

In the program below, we nest the loop that translates the mRNA starting from a particular position inside a loop that goes over the three starting positions we want to investigate. This

turns it into a nested `for` loop. For the outer loop we use `range(3)`, because we want to go over the starting positions 0, 1 and 2:

```
# Outer loop for starting positions
for starting_position in range(3):
    amino_acid_sequence = ""

    # Inner loop to translate the mRNA
    for index in range(starting_position, len(mRNA), 3):
        codon = mRNA[index:index + 3]
        # The last slice of the string may be shorter
        # if the length of the sequence to translate is not a multiple of 3 bases.
        # We thus need to check that the codon is 3 nucleotides.
        if len(codon) == 3:
            amino_acid = codon_dict[codon]
            amino_acid_sequence += amino_acid

    print(starting_position, amino_acid_sequence)
```

```
0 QWQHFKSSNK*D
1 NGNISSLPINR
2 MATFQVFQ*IG
```

These three amino acid sequences are very different from each other. If we would start from the fourth nucleotide we do not change the reading frame, we just skip a codon. The reason for this is that we do not change the triplets we read compared to when we start reading from the first nucleotide.

The program we have written so far allows us to translate the DNA sequence in all three reading frames. Our examples illustrate that if we start with a different reading frame we end up with a very different amino acid sequence. The question then is which one is the correct one reading frame when translating a mRNA? In cells, the solution to this problem is that certain codons function as translation start and stop signals.

7.5.4 Translation from start to stop

The codon `AUG` codes for the amino acid methionine and is the most common start codon. The presence of this codon tells the enzymes performing the translation process where to start translating the mRNA. If no such starting codon existed, there would be ambiguity in where to start reading the mRNA.

In addition to knowing where to start, we also need to know where to stop. There are three codons which do not code for any amino acid: `UAA`, `UGA` and `UAG`. These are called *termination codons*, and signal the end of a protein.

Going back to the codon wheel in Figure 7.7, we see that these are labelled "STOP", "STP", and the asterisk *, which is by convention. In our codon dictionary in Python, these are represented with the * character.

With this information, we can adjust our Python model to take start and stop codons into account. When we want to translate mRNA to proteins, we need to start to read the mRNA

from the first start codon AUG. Note that this also determines the correct reading frame. Our program should then translate the protein from that point until the first occurrence of any one of the three termination codons UAA, UGA or UAG.

It is possible to achieve our goal by modifying our `for` loop for translating the mRNA, but in this case it is easier to start from scratch and use a `while` loop. We will use this new approach to illustrate a useful role for `while` loops: when you don't know when the loop should end, a `while` loop often is a good choice. This means that for many loops we presented in earlier chapters of this book, we could have used `for` loops instead of the `while` loops we presented. We have taught `while` loops first because we believe these are more explicitly describing all the steps that happen, and thus better for learning about loops in Python, while it is less obvious how a `for` loop works.

Our `while` loop first needs to go over the mRNA sequence and retrieve the triplets of amino acids, i.e. the codons. We do this, as with our `for` loop, by using the loop index `index`, and slice from `index` to `index + 3`. For the next iteration of the loop, we then increase the loop variable by 3 for each execution of the loop:

```
index = 0
while index < len(mRNA):
    codon = mRNA[index:index + 3]
    print(codon)
    # increase index by three for slicing the next codon
    index += 3
```

```
CAA
UGG
CAA
CAU
UUC
AAG
UCU
UCC
AAU
AAA
UAG
GAC
```

Note how this is the `while` loop alternative of the `for` loop we wrote using `range(0, len(mRNA), 3)`.

But, instead of looping over all nucleotides, we want to start at the first start codon, in other words, the first occurrence of AUG in our string. We find the start codon with the `.index(substring)` function. The `.index(substring)` function is specific for strings, and returns the index of the first occurrence of `substring` in it:

```
starting_offset = mRNA.index("AUG")
print(starting_offset)
```

2

If we try to find a substring that does not exist in the string we get an error.

Our new program then becomes:

```
# find the first occurrence of AUG, the start codon
starting_offset = mRNA.index("AUG")

# start the loop at the start codon
index = starting_offset
while index < len(mRNA):
    codon = mRNA[index:index + 3]
    print(codon)
    # increase index by three for slicing the next codon
    index += 3
```

```
AUG
GCA
ACA
UUU
CAA
GUC
UUC
CAA
UAA
AUA
GGA
C
```

Note how the first codon we slice is in fact AUG, and that the last one is again shorter than three nucleotides.

Our `while` loop has the condition `index < len(mRNA)` to ensure it terminates once we are done looping over the entire string. But what we really want is to loop through the mRNA string until we encounter a termination codon. This means we will use a different loop variable and condition in our `while` loop. We will keep using the variable `amino_acid` inside the loop to store the amino acid corresponding to each codon. To check for a termination codon, we can now use the condition `amino_acid != "*"` for terminating the `while` loop. Remember that `!=` stand for "not equal".

Note that with that change to the while loop, we still need the `index` variable to be able to slice the codons. We also need to initialize our new loop variable `amino_acid` before the start of the loop, otherwise we would get a `NameError`. The simplest solution is to assign an empty string to the variable `amino_acid`.

Our new program, leaving out slicing and retrieving te amino acids for now, then becomes:

```
# find the first occurrence of AUG, the start codon
starting_offset = mRNA.index("AUG")

# start the loop at the start codon
index = starting_offset

# initialize the loop variable
amino_acid = ""
# let the loop run until the first stop codon
while amino_acid != "*":
    # Convert a codon to an amino acid
```

```
# and add it to our protein

# increase index by three for slicing the next codon
index += 3
```

The loop continues as long as `amino_acid` is not `"*"`. If the `amino_acid` is `"*"`, the next time the expression `amino_acid != "*"` is evaluated, it evaluates to `False`, and we exit the loop. This means that the stop codon (`*`) is added to the protein sequence *before* the loop finishes.

Now we can reuse some of the code we wrote for the `for` loop implementation:

- slice three elements of the mRNA string in each pass of the loop
- check that the slice is three nucleotides long
- convert the codon to an amino acid using `codon_dict` dictionary
- append the amino acid to a variable containing the protein sequence

As we are now translating into a real protein sequence, starting with a methionine and ending with a termination codon, we will change the name of the variable we store the amino acid sequence into `protein`.

Our program then becomes:

```
mRNA = "CAAUGGCAACAUUCAAGUCUUCCAAUAAAUGGAC"

# find the first occurrence of AUG, the start codon
starting_offset = mRNA.index("AUG")
index = starting_offset

protein = ""                                # New
amino_acid = ""

while amino_acid != "*":
    # Convert a codon to an amino acid
    # and add it to our protein
    codon = mRNA[index:index + 3]            # New
    # The last slice of the string may be shorter
    # if the length of the sequence to translate is not a multiple of 3 bases.
    # We thus need to check that the codon is 3 nucleotides.
    if len(codon) == 3:                      # New
        amino_acid = codon_dict[codon]        # New
        protein += amino_acid                # New
    # increase index by three for slicing the next codon
    index += 3
print(protein)
```

MATFQVFQ*

Congratulations, you have translated your first piece of RNA! A good exercise is to go through the flow of the above code manually, and keep track of what `index` is in each iteration and which nucleotides that `index` corresponds to.

In this example, we were lucky that there was a termination codon. What happens if our mRNA has no termination codons? We explore this in the next section.

7.5.5 Translation without a stop codon

The following mRNA string has no termination codon:

```
mRNA = "CAAUGGCAACAUUCAAGUCUUCAA"
```

If you try to run your translation code on this string, you get an endless loop.

What happens is that we continue through the entire mRNA string without finding a termination codon. Once `index` is greater than the length of the mRNA string, `codon = mRNA[index:index + 3]` gives us an empty string and `codon` never changes from this empty string. From this point forward, for every execution of the loop, `len(codon) == 3` always evaluates to False, `index` continues to increase with 3, but the condition `amino_acid != "*"` never evaluates to False, leading to an endless loop. Try printing out both `index` and `codon` to see what happens yourself.

We introduced this problem when we changed the condition of the `while` loop from `index < len(mRNA)` to `amino_acid != "*"`. What we really need is to test for *both* conditions. We should only continue the loop as long as

- `index` is less than the length of the mRNA string, `index < len(mRNA)`
- *and* we have not yet encountered a termination codon

In order to implement this solution we need to chain two boolean statements together, that both must be `True` to continue the loop. We show how to do this in the next section.

7.5.6 Logical operators for combining boolean expressions

The keywords `and` and `or` combine multiple boolean expressions in one single expression. If you combine two expressions with the `and` keyword, you get a new expression which is `True` only if both the original expressions are `True`:

```
my_number = 4

if my_number > 2 and my_number < 5:
    print("my_number is between 2 and 5!")
else:
    print("my_number is not between 2 and 5!")
```

```
| my_number is between 2 and 5!
```

If only of one of the conditions are met, the `else` block is executed instead:

```
my_number = 6

if my_number > 2 and my_number < 5:
    print("my_number is between 2 and 5!")
else:
    print("my_number is not between 2 and 5!")
```

| my_number is not between 2 and 5!

The `or`-keyword allows you to test if any of the two expressions are `True`. Only one expression needs to be `True` for the entire expression to be `True`:

```
my_number = 6

if my_number < 2 or my_number > 5:
    print("my_number is smaller than 2, or larger than 5!")
else:
    print("my_number is something else!")
```

| my_number is smaller than 2, or larger than 5!

In Figure 7.10 we show the possible outcomes when using the `and` and `or` operators.

A	B	A and B	A	B	A or B
True	True	True	True	True	True
True	False	False	True	False	True
False	True	False	False	True	True
False	False	False	False	False	False

Figure 7.10: The different possible outcomes when using the `and` and `or` operators. Left: Table representation of the conditional statement `A and B` (right column) for all combinations for `A` and `B`, where `A` and `B` are also conditional statements that evaluate to either `True` or `False`. Only when both `A` and `B` are `True` does `A and B` evaluate to `True`. Right: Table representation of the conditional statement `A or B`. Only when both `A` nor `B` evaluate to `False` does `A or B` evaluate to `False`.

Using `and`, we can make sure that the while loop in our translation program terminates at the end of the mRNA:

```
while amino_acid != "*" and index < len(mRNA):
    # Convert a codon to an amino acid
    # and add it to our protein
```

Here, both `amino_acid != "*"` and `index < len(mRNA)` must be `True`, in order to iterate over the `while` loop.

7.5.7 A complete translation function

As with transcribing a DNA sequence to its corresponding mRNA, translating mRNA to proteins is something we want to do many times. We will therefore also put our translation program inside a function, one that has the mRNA as input and returns the encoded protein.

The function combines the `codon_dict` dictionary and the `while` loop we developed for translating a mRNA to a protein. Note that it removes the need for the variable `starting_offset` by using `index = mRNA.index("AUG")` to initialize the `index` variable.

```

1  def translate(mRNA):
2      """
3          Translate a mRNA string to a protein.
4      """
5
6      codon_dict = {
7          "UUU": "F", "UUC": "F", "UUA": "L", "UUG": "L",
8          "UCU": "S", "UCC": "S", "UCA": "S", "UCG": "S",
9          "UAU": "Y", "UAC": "Y", "UAA": "*", "UAG": "*",
10         "UGU": "C", "UGC": "C", "UGA": "*", "UGG": "W",
11         "CUU": "L", "CUC": "L", "CUA": "L", "CUG": "L",
12         "CCU": "P", "CCC": "P", "CCA": "P", "CCG": "P",
13         "CAU": "H", "CAC": "H", "CAA": "Q", "CAG": "Q",
14         "CGU": "R", "CGC": "R", "CGA": "R", "CGG": "R",
15         "AUU": "I", "AUC": "I", "AUA": "I", "AUG": "M",
16         "ACU": "T", "ACC": "T", "ACA": "T", "ACG": "T",
17         "AAU": "N", "AAC": "N", "AAA": "K", "AAG": "K",
18         "AGU": "S", "AGC": "S", "AGA": "R", "AGG": "R",
19         "GUU": "V", "GUC": "V", "GUA": "V", "GUG": "V",
20         "GCU": "A", "GCC": "A", "GCA": "A", "GCG": "A",
21         "GAU": "D", "GAC": "D", "GAA": "E", "GAG": "E",
22         "GGU": "G", "GGC": "G", "GGA": "G", "GGG": "G"
23     }
24
25     index = mRNA.index("AUG")
26     protein = ""
27     amino_acid = ""
28     while amino_acid != "*" and index < len(mRNA):
29         # Convert a codon to an amino acid
30         # and add it to our protein
31         codon = mRNA[index:index + 3]
32         # The last slice of the string may be shorter
33         # if the length of the sequence to translate is not a multiple of 3 bases.
34         # We thus need to check that the codon is 3 nucleotides.
35         if len(codon) == 3:
36             amino_acid = codon_dict[codon]
37             protein += amino_acid
38             index += 3
39
40     return protein

```

Together with the `transcribe()` function we developed earlier, this `translate()` function enables us to

- transcribe a specific DNA sequence to its corresponding mRNA
- translate that sequence of mRNA to determine which protein it encodes for

For our example DNA sequence, we can thus do:

```

DNA = "CAATGGCAACATTCAAGTCTTCAATAATAGGAC"
mRNA = transcribe(DNA)
print(mRNA)

```

```
protein = translate(mRNA)
print(protein)
```

```
CAAUGGCAACAUUCAAGUCUCCAAUAAAUGGAC
MATFQVFQ*
```

We have now written two functions in Python that perform two biological functions. It is important to note, however, that both our functions do not model the process of how DNA is transcribed, or mRNA is translated. Instead, they convert the DNA *sequence*, represented as a string of characters, via the mRNA, to the final protein *sequence* the biological process would create. These are important steps in being able to discover what specific parts of our genes encode for which proteins, as it is the proteins that express the genetic traits stored in our DNA.

We will use our `translate()` function to examine the effects of mutations in Chapter 8.

7.6 More on dictionaries

There are some features of dictionaries we have not used in this chapter, that you should still know. In this final section we go through these.

7.6.1 Different types of keys in dictionaries

The keys in a dictionary can be strings or numbers:

```
my_dict = {"cars": 2, "beavers": 50}

my_dict["N"] = 1
my_dict[100] = 2

print(my_dict)
```

```
{'cars': 2, 'beavers': 50, 'N': 1, 100: 2}
```

Certain other types in Python can also be used as keys, but we have not encountered them yet. Lists and dictionaries are not allowed as keys. Lists and dictionaries are allowed as values, but this is beyond the scope of this book.

7.6.2 Removing a key-value pair from a dictionary

If we want to remove a key-value pair we can use:

```
my_dict = {"cars": 2, "beavers": 50}

del my_dict["cars"]

print(my_dict)
```

```
{'beavers': 50}
```

7.6.3 Assigning the same dictionary to two different variables

If two variables are assigned the same dictionary, the content of one is changed when we change the content of the other, similar to how lists behave. Let us create a new variable `my_dict_2` and set it equal to `my_dict`, then change one of the values in `my_dict_2`:

```
my_dict = {"cars": 2, "beavers": 50}
my_dict_2 = my_dict
my_dict_2["cars"] = 13
print(my_dict_2)
```

```
{'cars': 13, 'beavers': 50}
```

If we print `my_dict`, we see that it has also changed:

```
print(my_dict)
```

```
{'cars': 13, 'beavers': 50}
```

To avoid that `my_dict` is affected by changes in `my_dict_2` we need to create `my_dict_2` as a copy of `my_dict` with `.copy()`:

```
my_dict = {"cars": 2, "beavers": 50}
my_dict_2 = my_dict.copy()
my_dict_2["cars"] = 13
print(my_dict_2)
```

```
{'cars': 13, 'beavers': 50}
```

```
print(my_dict)
```

```
{'cars': 2, 'beavers': 50}
```

7.6.4 Check if a key is in a dictionary

Often, it is useful to check if a key is present in a dictionary. This is done by writing `if key in dictionary`. Here we first check if key is present - if yes, we print the value, if not, we print a message:

```
my_dict = {"cars": 2, "beavers": 50}

if "elephants" in my_dict:
    print(my_dict["elephants"])
```

```
else:  
    print("elephants are not present as a key in my_dict")
```

```
| elephants are not present as a key in my_dict
```

7.6.5 Accessing a key that does not exist

A common error when using dictionaries is to access the value of a key that does not exist in the dictionary:

```
my_dict = {"cars": 2, "beavers": 50}  
print(my_dict["elephants"])
```

```
-----  
KeyError                                     Traceback (most recent call last)  
Input In [314], in <cell line: 3>()  
      1 my_dict = {"cars": 2, "beavers": 50}  
----> 3 print(my_dict["elephants"])  
  
KeyError: 'elephants'
```

Here, we try to index our dictionary with `elephants`, which does not exist as a key in the dictionary. We therefore get a `KeyError`. A way to avoid `KeyError` when we try to access the value of a non-existing key is to use `dictionary.get(key)`. In this case `None` is returned if `key` does not exist as a key in the dictionary:

```
my_dict = {"cars": 2, "beavers": 50}  
print(my_dict.get("elephants"))
```

```
| None
```

7.7 Summary

In this chapter we have analyzed DNA sequences. You have learned how to count nucleotides in a DNA sequence and store the results in lists and dictionaries. In addition, you have learned how to translate mRNA to proteins.

7.7.1 Chapter glossary

Important terms introduced in this chapter are:

Nucleotides: Primary building blocks of DNA. There are four different types of nucleotides, or bases, in DNA: adenine (A), guanine (G), thymine (T), and cytosine (C).

DNA strands: Linear chains of nucleotides attached to each other.

Double helix: The structure formed by two strands of DNA, held together by hydrogen bonds.

In DNA the strands are wrapped around each other to form a twisted ladder-like structure.

Chromosomes: Made up of DNA tightly coiled many times around proteins that support its structure.

Genome: The complete genetic material of an organism.

Triplets/codons: Three subsequent nucleotides that code for a specific amino acid.

Gene: A region of DNA that contains all the triplets coding for a specific protein.

Base pair: Two bases from complementary strands of DNA bonded to each other. Adenine always pairs with thymine, cytosine always pairs with guanine.

RNA: A chain of nucleotides, but consisting of a single-strand rather than the paired double-strand of DNA. RNA has the nucleotide uracil (U) instead of thymine (T).

7.7.2 Strings

A string is a piece of text which consists of a set of characters. Strings in Python are enclosed with a " (double quotation mark) on each side, or a ' (single quotation mark) on each side. Both types are used, and you are free to choose, but make sure to never use one sign to start the string and another to end it, as Python will not understand this input. Everything inside the quotation marks is the string itself, so the marks are not considered part of the string. An example of a string is "Hello, world!".

Since a string is a collection of characters in an ordered sequence, it can be indexed and sliced similarly as lists. Note that only a subset of these are explained in the present chapter.

Syntax	Description	Result
S = ""	Initialize an empty string	""
S = "Hello, world!"	Initialize a string	"Hello, world!"
len(S)	number of characters in string S	13
S[1]	Index a string, get element 1	"e"
S[-1]	Get last character in a string	"!"
S[1:3]	Slice: copy data to substring	"el"
S.index("w")	Find index of first occurrence of "w"	7
S + "!"	Concatenate two strings	"Hello, world!!!"
S.count("l")	Count occurrences of substring "l"	3
S.replace("l", "k")	Replace occurrences of substring "l" with "k"	"Hekko, workd!"

Finally, strings have a function called `.strip()`, which removes all whitespace (tabs, newlines, and spaces) at the beginning and the end of the string:

```
my_string = "      spaces and newlines  \n\n\n"
print("With spaces and newlines:")
print(my_string)
```

```
print("Without spaces and newlines:")
print(my_string.strip())
```

With spaces and newlines:
spaces and newlines

Without spaces and newlines:
spaces and newlines

7.7.3 For loops

A `for` loop repeats a set of statements a specific number of times. It tells the computer that for each element in a list it should “do something”. Everything inside the indented code block that follows the colon after the `for` statement is run for each element in the list. One example:

```
1 numbers = [1, 2, 3]
2
3 for number in numbers:
4     print(number)
5
6 print("Finished printing numbers to screen!")
```

1
2
3
Finished printing numbers to screen!

Choosing between while and for loops. In many cases a `for` loop or `while` loop can be used to achieve the same goal. A rule of thumb to help us choose between the types is that when you don’t know when the loop should end, a `while` often is a good choice. If you know exactly how many times a loop should run, for example, when you need to go over each element in a collection, a `for` loop is often preferred.

7.7.4 Nested loops

We can also have loops within loops, called nested loops. These follow the same rules as normal loops, everything inside the loop must be indented. This also applies to the inner loop, which means everything inside the inner loop must be indented twice.

An example of a nested `while` loop is:

```
some_letters = ["A", "B"]
more_letters = ["P", "Q"]

index_outer = 0
while index_outer < len(some_letters):
```

```
outer = some_letters[index_outer]

index_inner = 0
while index_inner < len(more_letters):
    inner = more_letters[index_inner]
    print("outer:", outer, "inner:", inner)

    index_inner += 1

print("inner loop complete!")

index_outer += 1
print("outer loop complete!")
```

```
outer: A inner: P
outer: A inner: Q
inner loop complete!
outer: B inner: P
outer: B inner: Q
inner loop complete!
outer loop complete!
```

The same result with a nested `for` loop:

```
some_letters = ["A", "B"]
more_letters = ["P", "Q"]

for outer in some_letters:
    for inner in more_letters:
        print("outer:", outer, "inner:", inner)

    print("inner loop complete!")

print("outer loop complete!")
```

```
outer: A inner: P
outer: A inner: Q
inner loop complete!
outer: B inner: P
outer: B inner: Q
inner loop complete!
outer loop complete!
```

7.7.5 range

To loop over the indices of elements in a list, instead of the elements themselves we can use the `range(start, stop, step)` function. All arguments to the `range()` function must be integers.

Syntax	Description
<code>range(stop)</code>	From 0 up to, but not including, <code>stop</code> with step size 1
<code>range(start, stop)</code>	From <code>start</code> up to, but not including, <code>stop</code> with step size 1
<code>range(start, stop, step)</code>	From <code>start</code> up to, but not including, <code>stop</code> with step size <code>step</code>

7.7.6 Dictionary

A Dictionary is a collection of coupled elements, where each value in the dictionary is associated with a key, called a key-value pair. An example of a dictionary is:

```
D = {"A": 0, "C": 2, 100: 2}
```

Strings, floats, integers and several other Python data types not encountered yet can be used as keys. Lists and dictionaries are not allowed as keys. In the table below some important dictionary operations are shown, always using the dictionary `D = {"A":0, 100:2}`.

Syntax	Description	Result
<code>D = {}</code>	Initialize an empty dictionary D	{}
<code>D = {"A":0, 100:2}</code>	Initialize a dictionary D	{"A":0, 100:2}
<code>D["C"] = 10</code>	Set or create a key "C" with value 10	{"A":0, 100:2, "C":10}
<code>D["A"]</code>	Value associated with key "A"	0
<code>D.get("A")</code>	Value of "A" if "A" is in D, else None	0
<code>"A" in D</code>	Check if "A" is in D	True
<code>len(D)</code>	Number of key value pairs in D	3
<code>del D["A"]</code>	Remove "A" and its value from D	{100: 2, 'C': 10}
<code>D.keys()</code>	Get a view of all keys in D	dict_keys([100, 'C'])
<code>D.values()</code>	Get a view of all values in D	dict_values([2, 10])
<code>D.copy()</code>	Copy a dictionary D	{100: 2, 'C': 10}

7.7.7 Short-hand syntax for common operations

Code	Equivalent code
<code>n += 1</code>	<code>n = n + 1</code>
<code>n -= 1</code>	<code>n = n - 1</code>
<code>n *= 1</code>	<code>n = n*1</code>
<code>n /= 1</code>	<code>n = n/1</code>

7.7.8 If-elif-else tests

An `if-elif-else` test is a test that does something in one case, something else in another case, and something else for all other cases. It is possible to have several `elif` statements in one test:

```
codon = "UAG"
```

```
if codon == "UAA":  
    print("codon is a stop codon")  
elif codon == "UGA":  
    print("codon is a stop codon")  
elif codon == "UAG":  
    print("codon is a stop codon")  
else:  
    print("codon is not a stop codon")
```

| codon is a stop codon

7.7.9 Logical operators for combining boolean expressions

The keywords `and` and `or` combine multiple truth statements in the same `if` test. If you combine two expressions with the `and` keyword, you get a new expression which is `True` if both the original expressions are `True`:

```
my_number = 4  
  
if my_number > 2 and my_number < 5:  
    print("my_number is between 2 and 5!")  
else:  
    print("my_number is not between 2 and 5!")
```

| my_number is between 2 and 5!

The `or`-keyword allows you to test if any of the two expressions are `True`. Only one expressions needs to be `True` for the entire expression to be `True`:

```
my_number = 6  
  
if my_number < 2 or my_number > 5:  
    print("my_number is smaller than 2, or larger than 5!")  
else:  
    print("my_number is something else!")
```

| my_number is smaller than 2, or larger than 5!

Chapter 8

Mutations and DNA



Figure 8.1: A mutation is an alteration of the nucleotide sequence and can be due to mistakes made when copying DNA, or to incorrectly repaired damage to the DNA. Adapted from [3].

In Chapter 7, you learned about DNA and how it is related to an organism's characteristics. In nature there is an enormous diversity of organisms. This diversity is reflected in the genetic diversity found within, and between species. One important question one can ask is, how did this genetic diversity come to be? There are many aspects to this question. This chapter deals with one of them: direct changes that can happen to the nucleotide sequence itself.

During cell division, all DNA must be duplicated. As mentioned in the previous chapter, the human genome consists of around 3,200,000,000, or 3200×10^6 , base pairs. This means that during a single cell division several billion base pairs are copied. During our lifetime we experience somewhere around 10^{15} (10 quadrillion) cell divisions. The total number of nucleotides copied during our lifetime is thus staggering. The copying of DNA is a surprisingly robust process, almost always resulting in two faithful copies of the original DNA. However, despite this robustness, the process still sometimes goes wrong. This can result in *mutations*, which are permanent changes in the DNA sequence. These mutations can occur at large scales, where entire chromosomes are rearranged, or on small scales, where *point mutations* affect only a single pair of nucleotides.

The type of mutation discussed above is called a *spontaneous mutation* and arises either during DNA replication, DNA repair, or from recombination that goes wrong. Mutations also arise due to environmental causes, such as radiation or harmful chemicals that affect and change the DNA. Cells have repair mechanisms for dealing with such changes, but even these sometimes make mistakes, resulting in a mutation.

A mutation only affects the characteristics of the cell where it occurs. If the cell never reproduces the potential effect of that mutation is minimal. If the cell does reproduce, the mutation is spread to all future offspring of that cell. When such a mutation happens, it may sometimes have implications for the entire organism. One of the most drastic consequences may be the development of uncontrolled cell growth, ultimately leading to cancer. Luckily, most mutations do not lead to any significant changes in cell behaviour.

If a mutation occurs in cells that contribute to the next generation, for example the gametes, it can then be transferred to some or all offspring of that organism, and has the potential to affect the future of the entire species. Such mutations may have positive or negative consequences for the organism. Especially beneficial mutations may play a key role in the evolution of a species.

An example of mutations having an adverse effect are those causing *genetic disorders*, health problems caused by one or more abnormalities in the genome of an individual. Examining our DNA is key to detect and to understand how such disorders arise, and ultimately how they can be treated. In this chapter, we discuss point mutations, the class of mutations that affect only a single pair of nucleotides, and examine their effect on the proteins they encode. We also look at the genetic disorder sickle-cell anemia, and write a program to find the location of the mutation that causes this disease. Then we proceed to discuss *restriction cutting*, an experimental method where DNA is cut into smaller fragments. The fragments are examined by measuring the number of fragments and their lengths by a method called *gel electrophoresis*. The combination of these methods is used to find differences in the DNA from cells with and without the sickle-cell mutation. Finally, we create a program that simulates both restriction cutting and gel electrophoresis.



Learning outcomes

After this chapter you will know how:

- mutations affect DNA,
- to programmatically locate the mutation in the DNA/mRNA that causes sickle-cell anemia,
- restriction fragment analysis works, and
- to simulate restriction cutting.

The programming concepts we introduce in this chapter are:

- importing functions from other files, and
- reading from and writing to files with `open()`.

8.1 Point mutations

Point mutations only affect a single pair of nucleotides. Even though this might not sound much considering the large number of nucleotides in our DNA, the consequence of one such mutations can in certain cases be dramatic. Point mutations are divided into two categories, base pair *insertion and deletions* and base pair *substitutions*.

When a mutation is located in the DNA coding for a protein, all instances of the corresponding protein produced by the cell are mutated. It would be preferable to examine mutations in the DNA, but because of the challenges we encountered in the previous chapter with introns and exons, it is not so straightforward to go directly from DNA to protein. We therefore work with mRNA, that expresses a mutation located in the protein coding part of the DNA. Note that mutations can occur also in mRNA, but the mutations then only affects instances of proteins produced from that mRNA.

To study the effects of different types of mutations, we use the `transcribe()` and `translate()` functions from the previous chapter. However, instead of redefining these functions in this chapter, we show how we can store them in a file. This allows us to import the functions from that file when we need them again.

8.1.1 Importing functions from files

So far, we have used the `import` statement to enable the use of Python code for, among other things, plotting (via the `pylab` package), mathematical calculations (via the `math` package), importing data from `.csv` files (via the `pandas` package).

It is also possible to import code written by ourselves (or others) that has been stored in a file. To illustrate this, let us show how this is done for the `transcribe()` and `translate()` functions we developed in Section 7.4 and Section 7.5, respectively.

We put the Python code for the `transcribe()` and `translate()` function in a file named `bioinformatics.py` and import the function here by writing:

```
from bioinformatics import translate
```

This assumes that the file `bioinformatics.py` is in the same directory as the the file we are currently working on.

In general, a functions can be imported from a file using:

```
from my_filename import my_function
```

`my_filename` is the name of the file, and is given without the `.py` extension. A Python file consisting of functions and variables that can be imported into other programs is called a *module* or *package*.

Now that we have imported our `transcribe()` and `translate()` functions, we can use them to look at different types of mutations in DNA, and how they potentially affect the proteins these encode for.

8.1.2 Insertions and deletions

Insertions and deletions cause base pairs to be inserted or removed from DNA, as illustrated in Figure 8.2. Note that pairs are inserted or removed, so both strands of the DNA are affected at the same time.

Insertions and deletions in regions of the DNA that code for a protein can be particularly problematic. This has to do with the reading frames, see Section 7.5.3. To illustrate this phenomenon, consider how a deletion in a coding sequence can be compared to removing a letter from a sentence where each word has three letters. If you remove the seventh letter in the sentence “the mom has the car”, but keep dividing the text in words of three letters, you get “the mom ast hec ar”. Everything after the deletion becomes meaningless to us. The same is true for DNA with a deletion; depending on where the deletion or insertion occurs, the resulting protein may become very much altered and no longer functional.

Such mutations are called *frameshift mutations* because they cause what is called a *frameshift*: they offset the DNA sequence and shift the reading frame. A frameshift causes all the following triplets to be misinterpreted when they are read. Let us examine how a deletion affects the encoded protein.

Insertion: ATGTTCAACGGT^TCCTGTACACACTGCTAC
Deletion: ATGTTCAACGGTC^CCTGTACACACTGCTAC

Figure 8.2: An insertion (top) is a base pair that is inserted into the DNA. A deletion (bottom) happens when a base pair is removed from the DNA. For simplicity only a single DNA strand is shown, but the mutations affect both strands at the same time.

When translating the mRNA sequence from the previous chapter, we get:

```
print("original:", translate("CAAUGGCAACAUUCAAGUCUUCUCCAAUAAAUGGAC"))
```

```
original: MATFQVFQ*
```

If we introduce a deletion of C at the 7th position, we get:

```
#                                     C deleted
#
# print("mutation:", translate("CAAUGGAACAUUCAAGUCUUCUCCAAUAAAUGGAC"))
```

```
mutation: MEHFKSSNK*
```

In the mutated case, the sequence of amino acids is completely different, even though we only deleted one nucleotide. The resulting protein is also longer than the original, because a **STOP** codon is encountered later. As a result, the two proteins likely have very different properties.

A deletion or addition can have a disastrous effect on the resulting protein, because it changes the reading frame. Frameshift mutations are linked to several genetic diseases, for example Crohn's disease [9].

8.1.3 Substitutions

Another type of point mutation is substitution, where a base pair has been substituted with another pair of nucleotides, illustrated in Figure 8.3.

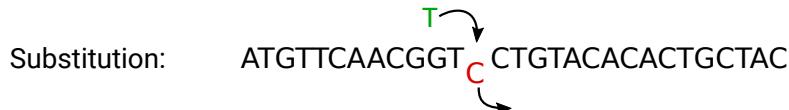


Figure 8.3: In a substitution, one base pair is exchanged for another base pair. For simplicity only a single DNA strand is shown, but the mutation affects both strands at the same time.

When a substitution occurs in DNA coding for a protein, it can be a *missense mutation*, a substitution that also changes one amino acid into another. In this example, a U is replaced with a C:

```
print("original:", translate("CAAUGGCAACAUUCAAGUCUUCUCCAAUAAAUGGAC"))
print("mutation:", translate("CAAUGGCAACAUCAAGUCUUCUCCAAUAAAUGGAC"))
#
#                                         C instead of U
```

```
original: MATFQVFQ*
mutation: MATSQVFQ*
```

This changes the amino acid **Phe** to the amino acid **Ser** in the resulting protein. The effect of such a change varies from protein to protein and amino acid to amino acid. If the new amino acid

has properties that are similar to the original, the mutation may have little effect. Otherwise, the mutation could dramatically alter the properties of the protein.

A *silent mutation* changes a codon into another codon that encodes for the same amino acid. This is the case if we substitute the following A with G:

```
print("original:", translate("CAAUGGCAACAUUCAAGUCUUCAAUAAAAGGAC"))
print("mutation:", translate("CAAUGGCAGACAUUCAAGUCUUCAAUAAAAGGAC"))
#
#                                     |
#                                     G instead of A
```

```
original: MATFQVFQ*
mutation: MATFQVFQ*
```

In this case, the resulting protein is unchanged.

A *nonsense mutation* alters a codon into a stop codon. In the following, we have substituted two nucleotides; UU for AA:

```
print("original:", translate("CAAUGGCAACAUUCAAGUCUUCAAUAAAAGGAC"))
print("mutation:", translate("CAAUGGCAACAUACAAGUCUUCAAUAAAAGGAC"))
#
#                                     ||
#                                     AA instead of UU
```

```
original: MATFQVFQ*
mutation: MAT*
```

Nonsense mutations lead to premature termination of the protein, as seen in the above example.

Some mutations can improve the protein or introduce novel capabilities, but most are detrimental to the function of the protein. One example of such a mutation is the substitution that leads to the disease sickle-cell anemia. This is the topic of the next section.

8.2 Searching for the sickle cell mutation

Hemoglobin is the protein responsible for binding oxygen in red blood cells. It consists of four smaller proteins, called subunits. Sickle-cell anemia occurs because of a single substitution in the DNA sequence that codes for one of these subunits. This substitution results in an amino acid change in one of the subunits of hemoglobin that has a dramatic effect on the way hemoglobin proteins behave in the cells. As a result, the red blood cells with the amino acid change become deformed. Normal red blood cells are disk-shaped, but the red blood cells with the abnormal hemoglobin assume a sickle-like shape, illustrated in Figure 8.4.

The sickle-shaped cells are much more fragile than the normal-shaped cells. They thus break down prematurely, which leads to a reduction in the total number of blood cells, a condition known as *anemia*. This causes shortness of breath, fatigue, and developmental delay in children. Another symptom of sickle cell anemia is blood clots that occur because sickle-shaped cells stick together in lumps that clog the veins.

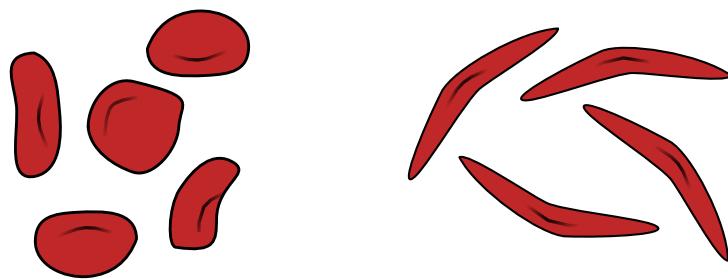


Figure 8.4: Comparison of a normal-shaped blood cell (left) to a sickle-shaped blood cell (right).

Sickle-cell anemia is a genetic disorder that occurs when a person inherits two mutated versions of the hemoglobin gene, one from each parent. If a person only has one copy of the mutated gene, he or she is said to have sickle-cell traits. Such a person produces both sickle-cell hemoglobin and normal hemoglobin, and only has limited symptoms of sickle-cell anemia. A person with two copies of the mutated gene has full-blown sickle-cell anemia.

Although there are many dangers associated with the sickle-cell mutation, there is one positive side-effect. Sickle-shaped blood cells protect against the tropical malaria parasite. The malaria parasite prefers to infect the sickle-shaped blood cells, which are then removed by the immune system. On the other hand, malaria makes the full blown sickle-cell anemia more severe, so the benefits of the protection are only relevant for people with sickle-cell traits. Having one copy of the mutation is therefore advantageous in areas with malaria, and the sickle-cell mutation is more prevalent in these areas.

Later in this chapter, we discuss how different molecular biology techniques are used to detect the sickle-cell mutation in hemoglobin. Before we can design experiments to detect the mutation with these techniques, we first need to find where the mutation occurs in hemoglobin DNA. This is the topic of this section.

8.2.1 Sickle-cell and normal hemoglobin

Below, we have listed the mRNA for the hemoglobin subunit where the sickle-cell mutation occurs. The following mRNA sequence is collected from the person without the symptoms of sickle cell disease and encodes for the normal hemoglobin subunit:

```
ACAUUUCGUUCUGACACAACUGUGUUCACUAGCAACCUCAAACAGACACCAUGGUGCAUCUGACUCCUGA  
GGAGAACUGUCGGUUAUCGCCUGUGGGCAAGGUGAACGGUGGAUGAUUGGGUGGAGGCCUGGGC  
AGGCUGCGUGGGGUUACCCUUGGACCCAGAGGUUCUUGAGGUCCCCUUGGGGAUCUGUCACUCCUGAUG  
CUGUUUAUGGGCAACCUAAGGUGAAGGCUCAUGGCAAGAAAGUGGUCCGGUCCUUUAGUGAUGGGCUGGC  
UCACCUUGGACAACCUAAGGGCACCUUUGGCCACACUGAGUGAGCUGCACUGGACAAGCUGCACGUGGAU  
CCUGAGAACUUCAGGCUCUGGGCAACCGUGCUGGUCCUGGGCCAUACUUUUGGCAAAGAAUUC  
CCCCACCAAGUGCAGGCUGCCUAUCAGAAAGUGGUCCUGGUCCUAUGCCCUGGCCACAAGUAUC  
CUAAGCUUCGUUCUUGCUGGUCCAAUUCUAUUAAGGUUCUUGGUCCUAAGGUCCACUACUAAACU  
GGGGAUAAAUGAAGGGCCUUGAGCAUCUGGUUCUGGUCCUAUAAAACAUUUAAAUCAUUGC
```

In a person with sickle-cell anemia, the same hemoglobin sequence was found to be the following:

```

ACAUUUCUUCUGACACAACUGGUUCACUAGCAACCUCAAACAGACACCAUGGUGCAUCUGACUCCUGU
GGAGAACUGCCGUUACUGCCCUGUGGGCAAGGGUACGUGGAUGAUGUGGGUGAGGCCUGGGC
AGGCUGCUGGUGGUUCACCUUCCUUGGACCCAGAGGUUCUUGAGGUCCUUUUGGGAUUCUGGUCCACUCCUGAUG
CUGUUUAUGGGCAACCCUAAUGGCUAAGGCUAUGGCAAGGAAAGUGGUCCGGUCCUUUAGUGAUGGGCCUGGC
UCACCUUGGGCAACCUUAGGGCACCUUUGGCCACACUGAGUGACUGGUCCACUGGUACAAGCUGCACGUGGAU
CCUGAGAACUUCAGGCUCUCCUGGGCAACGGCUGGUCCUGGUCCAUACUUUUGGCAAAGAAUUC
CCCCACCAAGUGCAGGCUGCCUAUCAGAAAGUGGUCCUGGUCCUAUGGCCUGGCCACAAGUAUCA
CUAAGCUUCGUUUUCUUGCUGGUCCAUUUAAAGGUCCUUUUGGUCCUAAGGUCCACUACUAAACU
GGGGAUAAAUGAAGGGCCUUGAGCAUCUGGUCCAUAAAACAUUUAAAUCAUUGC

```

Can you spot the difference?

Obviously this is hard to do for a human, but thankfully it is straightforward for a computer. Let us therefore write a program that finds the difference between the two sequences.

8.2.2 Reading FASTA files with the open-function

The hemoglobin sequences are so large, that we should not store them directly in the code. For this reason we have stored them in two files, one named `hemoglobin_normal.fasta` and one named `hemoglobin_sickle.fasta`.

Previously we used the `pandas` package to read in the content of `.csv` files. However, `pandas` is not suitable for the type of files like we use for the hemoglobin sequences. For this, we use a Python command for opening any file, named `open()`:

```
f = open("hemoglobin_normal.fasta", "r")
```

The first argument to `open()` is the name of the file we want to open given as a string. The second argument, the string `"r"` (for read), tells that we want to open the file for reading. The variable `f` is what is called a filehandle and can be used for reading from and writing to the file it refers to. The content of the file can be read in various ways, but a common method is to use the function `.readlines()`. This function returns a list where the lines in the file being read are the elements of the list.

```
lines = f.readlines()
print(lines)
```

```
[ '>NM_000518.4 Homo sapiens hemoglobin subunit beta (HBB), mRNA\n',
  'ACAUUUCUUCUGACACAACUGGUUCACUAGCAACCUCAAACAGACACCAUGGUGCAUCUGACUCCUGA\n',
  'GGAGAACUGCCGUUACUGCCCUGUGGGCAAGGGUACGUGGAUGAAGUUGGGUGAGGCCUGGGC\n',
  'AGGCUGCUGGUGGUUCACCUUUGGACCCAGAGGUUCUUGAGGUCCUUUUGGGAUUCUGGUCCACUCCUGAUG\n',
  'CUGUUUAUGGGCAACCCUAAUGGCUAAGGCUAUGGCAAGAAAGUGGUCCGGUCCUUUAGUGAUGGGCCUGGC\n',
  'UCACCUUGGACAACCUAAGGGCACCUUUGGCCACACUGAGUGACUGGUACAAGCUGCACGUGGAU\n',
  'CCUGAGAACUUCAGGCUCUCCUGGGCAACGGCUGGUCCUGGUCCAUACUUUUGGCAAAGAAUUC\n',
  'CCCCACCAAGUGCAGGCUGCCUAUCAGAAAGUGGUCCUGGUCCUAUGGCCUGGCCACAAGUAUCA\n',
  'CUAAGCUUCGUUUUCUUGCUGGUCCAUUUAAAGGUCCUUUUGGUCCUAAGGUCCACUACUAAACU\n',
  'GGGGAUAAAUGAAGGGCCUUGAGCAUCUGGUCCAUAAAACAUUUAAAUCAUUGC\n' ]
```

We should always close a file when we are finished working with it. This frees up the resources the computer used to open it, and helps prevent data loss. We close an opened file by using the `.close()` function:

```
f.close()
```

The hemoglobin sequence data is stored in the so-called FASTA format. FASTA is a popular file format used for storing DNA sequences. It is a text based format where the first line starts with a > (greater-than) symbol and contains different metadata ('data about data') for that specific sequence. This line is called a header. Our mRNA hemoglobin sequence files have the following header:

```
>NM_000518.4 Homo sapiens hemoglobin subunit beta (HBB), mRNA
```

This header tells us we have mRNA for a specific subunit of hemoglobin for humans. We do not need the header for our purpose, because we already know we are working with hemoglobin from humans. We are interested in the mRNA sequence itself.

Each line in the output above ends with the characters \n. This is the *newline* symbol and signifies a new line. When we print a string, the symbol \n is converted to a new line:

```
print("Hello\nWorld!")
```

```
Hello  
World!
```

Currently, the data read from the file is in a list consisting of several elements, one for each line. However, we would like to work with a single string containing the entire mRNA sequence. We turn the list of elements in `lines` into one string by starting with an empty string called `mRNA`, looping over each element of `lines`, and adding it to the `mRNA` string. We will use a `for` loop as we know how many lines there are to process. Each element in `lines` is a string and joining strings is done with + (the addition symbol). After the loop we print the result:

```
mRNA = ""  
for line in lines:  
    mRNA += line  
  
print(mRNA)
```

```
>NM_000518.4 Homo sapiens hemoglobin subunit beta (HBB), mRNA  
ACAUUUCUUCUGACACAACUGGUUCACUAGCAACCUAACAGACACCAGGGCAUGGUGCAUCUGACUCCUGA  
GGAGAAAGUCUGCCGUUACUGCCUGGGCAAGGGGAACQUGGAUGAAGUUGGGUGGAGGCCUGGGC  
AGGCUGCCUGGGGUUCACCCUUUGGACCCAGAGGUUCUUGAGGUCCCCUUGGGGAUCUGUCCACUCCUGAUG  
CUGUUUAAGGGCAACCCUUAAGGUGAAGGGCUAUGGCAAGAAAAGUGGUCCGGUCCUUUAGUGAAGGGCUGGC  
UCACCUUGGACAACCUAAGGGCACCUUUGGCCACACUGAGUGAGCUGCACUGUGACAAGCUGCACGUGGAU  
CCUGAGAACUUCAGGCUCUGGGCAACCGUCUGGUCCUGUGUGCCAUACUUUUGGCAAAGAAUUC  
CCCCACCAAGUGCAGGCUGCCUAUCAGAAAGUGGUCCUGGUCCUAUGCCCUGGCCACAAGUAUCA  
CUAAGCUCGCUUUCUUGCUGGUCCAAUUCUAUUAAGGUUCUUGUUCGUUAAGGUCAUCUAAACU  
GGGGAUUUUAUGAAGGGCCUUGAGCAUCUGGUUCUGGUUAUAAAAACAUUUUUUCAUUGC
```

We have now collected all lines into a single string, but it still contains the newline character, causing it to be printed over several lines. These newline characters also interfere with the next steps of our analysis. Strings have a function called `.strip()`, which removes all whitespace (tabs, newlines, and spaces) at the beginning and the end of the string:

```
my_string = "      spaces and newlines  \n\n\n"
print("With spaces and newlines:")
print(my_string)
print("Without spaces and newlines:")
print(my_string.strip())
```

```
With spaces and newlines:
spaces and newlines
```

```
Without spaces and newlines:
spaces and newlines
```

We use `strip()` to remove the newline symbols from each line in `lines`. We also want to ignore the first line with the header, and loop over all lines from the second line to the last with `lines[1:]`:

```
mRNA = ""
for line in lines[1:]:
    mRNA += line.strip()

print(mRNA)
```

```
ACAUUUGCUUCUGACACAAC ... (Truncated)... AAAACAUUUUUCAUUGC
```

The output of this code is a single string (without whitespaces) that contains the entire mRNA sequence. Reading a sequence from FASTA files is a task we are going to repeat several times, we should therefore put it in a function. We give the function a descriptive name, `load_fasta()`. It takes one argument, called `filename`, which is used by the `open()` function. Additionally we add a docstring that explains what this function does. The rest of the code is unchanged:

```
def load_fasta(filename):
    """
    Load a FASTA file and return the sequence as a string.
    """
    f = open(filename, "r")
    lines = f.readlines()
    f.close()

    mRNA = ""
    for line in lines[1:]:
        mRNA += line.strip()

    return mRNA
```

We can use `load_fasta()` to read any FASTA file and get the sequence stored in the file as a string, without the header. We have added `load_fasta()` to the file `bioinformatics.py`. We use `load_fasta()` to load both the normal hemoglobin mRNA and the sickle-cell hemoglobin mRNA from their respective files.

```
from bioinformatics import load_fasta
```

```
mRNA_original = load_fasta("hemoglobin_normal.fasta")
mRNA_mutation = load_fasta("hemoglobin_sickle.fasta")
```

Our next task is to find where these RNA fragments differ. To do this, we have to compare them, nucleotide by nucleotide.



FASTA files normally use T instead of U

Normally, FASTA files for mRNA sequences, the letter T is used, instead of the letter U. This is just a convention, since we can easily change them back and forth. In this book, to cause less confusion, we have files that use U instead of T, when we work with mRNA.

8.2.3 Comparing two mRNA strands

We have now the two mRNA sequences available, and we want to compare these at each position to see whether there are any differences between them, and potentially where the differences are. We thus need to loop through both mRNA strings simultaneously, nucleotide by nucleotide. This can be done using `for` loops (see the box below), but the simplest solution is to use a `while` loop. We loop over all indices, and access the corresponding nucleotide in each mRNA string, and if they differ we have found the mutated site.

```
index = 0
while index < len(mRNA_original):
    normal = mRNA_original[index]
    sickle = mRNA_mutation[index]

    if normal != sickle:
        print("Mutation has switched", normal, "to", sickle, "at", index)

    index += 1
```

```
Mutation has switched A to U at 69
```



Comparing two mRNA strands using a for loop and the zip-function

We know how to loop through all characters in a string, one at the time with a `for` loop. However, here we want to loop through both mRNA strings simultaneously, nucleotide by nucleotide. How can we loop over each position in *two* dna strings? It is possible loop over multiple lists at the same time using the `zip()` function. An example of the use of `zip()` is:

```
list_a = [1, 2, 3, 4]
list_b = [10, 20, 30, 40]

for a, b in zip(list_a, list_b):
    print(a, b)
```

```
1 10
2 20
3 30
4 40
```

Note how the `a` and `b` variables are assigned the values of the two lists, element for element, in turn. If the two lists are not of equal length, the loop stops when the end of the shortest list is reached:

```
list_a = [1, 2, 3, 4]
list_b = [10, 20, 30, 40, 50, 60]

for a, b in zip(list_a, list_b):
    print(a, b)
```

```
1 10
2 20
3 30
4 40
```

Since strings can be considered collections of elements, we can loop over our two hemoglobin mRNA strings using `zip()` in the same way. We can then test if each nucleotide in the two sequences are equal. If they differ, we print the nucleotides:

```
for normal, sickle in zip(mRNA_original, mRNA_mutation):
    if normal != sickle:
        print("Mutation has switched", normal, "to", sickle)
```

```
Mutation has switched A to U
```

Note that we are not able to tell at which position the change happened, unless we introduce a counter keeping track of the positions we are comparing.

There is a substitution from A to U in the mRNA for a hemoglobin subunit, and the corresponding substitution in the DNA is from an A to a T. For our molecular biology experiments, we need the sequence surrounding the mutation. To do this, we print the five characters before and after the index for the location when we find the mutation:

```
index = 0
while index < len(mRNA_original):
    normal = mRNA_original[index]
    sickle = mRNA_mutation[index]

    if normal != sickle:
```

```
print("Mutation has switched", normal, "to", sickle, "at", index)

print("Surrounding sequence, original:", mRNA_original[index - 5:index + 6])
print("Surrounding sequence, mutation:", mRNA_mutation[index - 5:index + 6])

index += 1
```

```
Mutation has switched A to U at 69
Surrounding sequence, original: UCCUGAGGAGA
Surrounding sequence, mutation: UCCUGUGGAGA
```

Notice that we put the `index += 1` statement at the end of the code block. This is to make sure we do not change the current index before we print it. Further, we slice from `index - 5` to `index + 6` because slicing goes from the start index to, but not including, the stop index.

8.3 Implementing restriction fragment analysis

We have found the mutation that leads to sickle-cell anemia, but detecting if someone has sickle-cell anemia is rarely done by directly examining their DNA. A more common method is *restriction fragment analysis*, which involves using a so-called restriction enzyme to cut the DNA at or near a specific sequence in the DNA. This is an experimental technique that has proved useful for many purposes. Restriction fragment analysis consists of three steps: *cloning*, *restriction cutting*, and *gel electrophoresis*.

8.3.1 Cloning

To perform the restriction fragment analysis we need many copies of the nucleotide sequence we want to analyze, which we can achieve with the help of cloning. There are many different techniques for cloning a specific *gene of interest*. One common molecular cloning technique uses bacteria and *plasmids*. Plasmids are small circular DNA sequences inside bacteria (see Figure 8.5), that exist separately from the chromosomes, and can replicate independently. The general idea behind this cloning technique is to extract plasmid DNA, insert the DNA of our gene of interest into the plasmid, and reinsert the modified plasmids into the bacteria. The bacteria then produce many copies of the plasmids in each cell, and by growing the bacteria to large numbers, significant amounts of plasmid DNA can be extracted from them. Such a modified plasmid is an example of *recombinant DNA*, which is DNA with nucleotide sequences from two different sources. We can use the above method to clone the hemoglobin gene from a person with sickle-cell anemia, as well as from a person with normal hemoglobin.

8.3.2 Restriction cutting

Once we have cloned our DNA sequence, we perform restriction cutting of the cloned plasmids. DNA can be cut open with *restriction enzymes*. Restriction enzymes find a specific DNA sequence called a *recognition sequence*, and cleave the DNA at or near that sequence, called the *restriction*

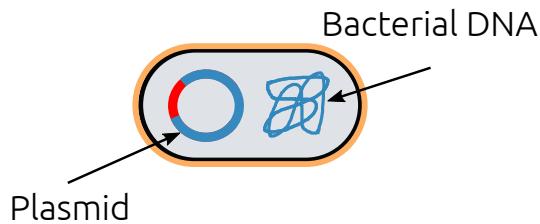


Figure 8.5: Illustration of a bacterium with chromosomal DNA and plasmid that contains the gene of interest (in red).

site. One example is the restriction enzyme EcoRI, which has the recognition sequence GAATTC and cuts between G and AATTC, as is illustrated in Figure 8.6



Figure 8.6: Double stranded DNA with the recognition sequence GAATTC is cut by the EcoRI restriction enzyme at the restriction site (green).

Note that the sequence of the EcoRI recognition sequence is *palindromic*, meaning that the sequence is the same on both DNA strands: when we consider the bottom strand in Figure 8.6, we need to read it in the reverse direction, from right to left. In fact, many recognition sequences are palindromic.

How can we use restriction cutting to detect DNA that has the sickle-cell causing mutation? Since restriction enzymes recognize a specific sequence, we choose a restriction enzyme whose recognition sequence is at the site of the mutation, such that it will cut the normal hemoglobin DNA at or near that site, but not the mutated DNA. We can thus use this enzyme to examine if there are any differences at (only) the restriction sites for different DNA sequences, a common use of the restriction cutting technique. Figure 8.7 shows restriction cutting of recombinant DNA with a mutation at one restriction site in the gene of interest. The green circle shows where the mutation is located and the restriction enzyme does not recognize this mutated cut site. This leads to different number of cuts and different length of the DNA fragments.

8.3.3 Gel electrophoresis

After restriction cutting, we are left with a soup of DNA fragments invisible to the naked eye. To be able to distinguish the DNA fragments, we use a technique called *gel electrophoresis*. Gel electrophoresis separates DNA fragments of different sizes, and visualises the fragments as distinct bands. An example of such bands is shown in Figure 8.8.

A single DNA molecule is not enough to form visible bands. This is the reason why we clone our gene of interest first: to obtain enough copies of the DNA fragments to get visible bands. The DNA fragments from the restriction cutting are added to a gelatinous material, (a 'gel'), which has a positive charge at one side, and a negative charge at the side where the DNA fragments

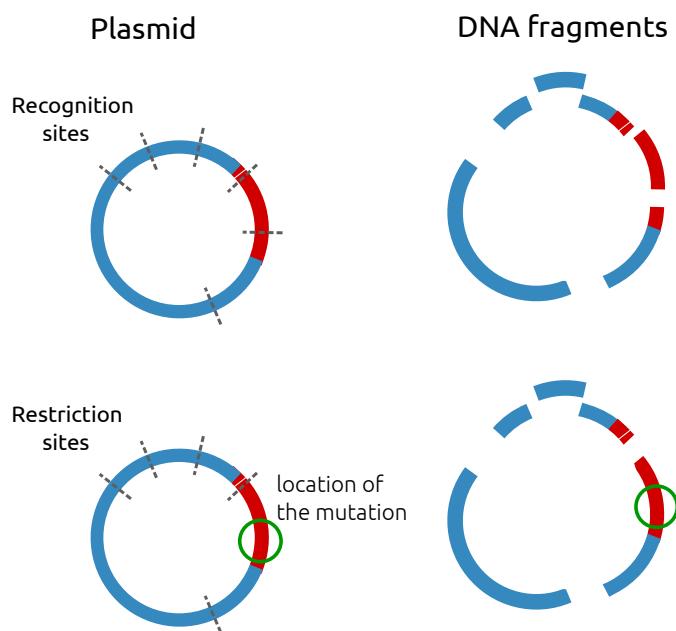


Figure 8.7: Restriction cutting of recombinant DNA (top) and recombinant DNA with a mutation at or on of the recognition sites (bottom). The green circles show where the mutation is located. Dashed lines indicate the recognition sequence. Because of the mutation, the recognition sequence is no longer there, and the restriction enzyme does not recognize this site.

are added. DNA is negatively charged and the fragments are pulled towards the positive charge through the gel. The DNA fragments move at different speeds through the gel depending on their size. Smaller fragments move faster than larger fragments, and after some time, the DNA fragments are separated into distinct bands depending on their length. A DNA-binding dye is added to reveal the bands.

The pattern of the bands is unique to the combination of analyzed DNA and the restriction enzyme used to cut it. By comparing these bands, we detect if there are differences in the number and length of the DNA fragments from restriction cutting of different DNA. However, we cannot observe the number of base pairs in each fragment directly. Instead, a DNA sequence with known length of the fragments is used for comparison. This DNA sequence is called a *marker*.

Now that we have described the different steps of restriction fragment analysis, we are ready to implement the process using Python programming. We will skip the cloning step and go right to restriction cutting.

8.3.4 Implementing restriction cutting of linear DNA

After cloning the gene of interest, we are left with circular DNA (the plasmids). An illustration of a plasmid with circular DNA is shown in Figure 8.9. Previously, we have only worked with linear DNA in Python, represented as strings. Circular DNA has no beginning or end, but it is

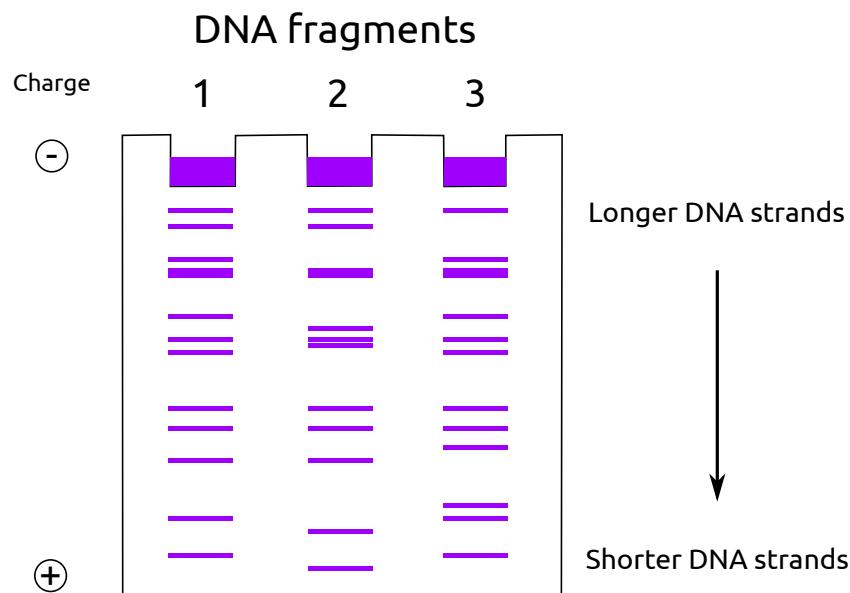


Figure 8.8: Gel electrophoresis is used to separate and visualise DNA fragments of different lengths. Three mixes of DNA fragments are added at the top, and an electric current is applied, causing the fragments to move towards the positive charge. Shorter fragments move faster than longer fragments. After visualising the separated DNA, the results indicate that the samples have some bands of identical size, but each has a different pattern and thus consists of a unique mix of DNA fragments.

still stored as a linear string in our program. We must take the circular behavior into account when we search for recognition sequences and cut the circular DNA.

However, most of our code will be similar regardless if we take circular DNA into account or not. For simplicity we therefore first implement a more simple solution, where our DNA is linear. We imagine having cut the plasmid first with a restriction enzyme that has one recognition sequence in the middle of the plasmid, outside the hemoglobin sequence. After this imaginary step, our DNA is linearised. This simplified solution will then help to illustrate the problems that arise due to plasmid DNA being circular.

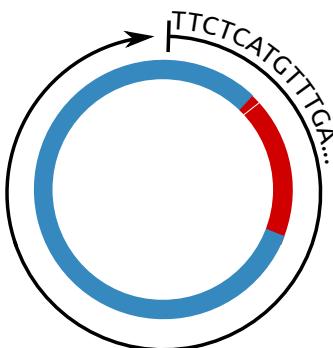


Figure 8.9: Representation of the circular plasmid.

Our goal is to create a Python function that takes a DNA string (in our case, a linearised recombinant plasmid) and the information defining the restriction enzyme, and returns a list with the DNA fragments obtained from the restriction cutting. In order to do this we go through the following steps:

1. Describe the restriction enzyme with code.
2. Find the cutting locations on the DNA sequence. This means to find indices where we need to slice the DNA string.
3. Use the cutting indices to slice the DNA string into fragments and return the list of fragments.

Restriction enzyme for detecting sickle-cell anemia. For discriminating between the normal and mutated forms of the hemoglobin sequence, we need a restriction enzyme that recognizes one of the corresponding sequences in the DNA, but not the other. In Section 8.2.3, we saw that the following sequences are surrounding the site where the mutation occurs, here represented as DNA sequences: TCCTGAGGAGA for the original, and TCCTGTGGAGA for the mutated site. It turns out that the sequence GAGG is recognized by a restriction enzyme named MnII. This fits perfectly for our purpose, because GAGG is part of the normal hemoglobin mRNA, but not the mutated hemoglobin mRNA, which instead has GTGG at the site of the mutation. We therefore use this enzyme when we perform the restriction cutting.

The restriction enzyme MnII recognizes the sequence GAGG, and cuts 10 nucleotides from the start of the restriction site. This restriction enzyme therefore only cuts the normal DNA near the sickle-cell mutation location, and leaves the mutated DNA intact at this point. However, the sequence GAGG occurs multiple times in the recombinant DNA consisting of the plasmid and hemoglobin sequences, and there will be multiple cuts in both mutated and non-mutated DNA, but *the number of cuts* will be different. With a different number of cuts, we also get fragments of different length, which appear as different bands on the gel after electrophoresis.

In order to describe a restriction enzyme in Python, we need to know two properties:

1. the recognition sequence, and
2. where the enzyme cuts the DNA relative to the recognition sequence, the restriction site.

The restriction enzyme MnII recognizes the sequence GAGG and cuts 10 nucleotides from the start of this sequence:

```
# Restriction enzyme MnII
recognition_sequence = "GAGG"
cut_site_offset = 10
```

Dummy DNA. Next, we define a dummy linear plasmid sequence to make it easier to debug and test our code:

```
dna = "----GAGG-----X-----GAGG-----X--"
```

We use “-” and “X” to denote nucleotides that are not part of a restriction site. Additionally, “X” is used to mark where the plasmid is cut (10 nucleotides from the start of GAGG). This dummy sequence is simple enough that we can perform the restriction cutting manually, illustrated in Figure 8.10. This gives us the three fragments ---GAGG-----, X-----GAGG----- and X--. We use this result to check that our code works as intended.

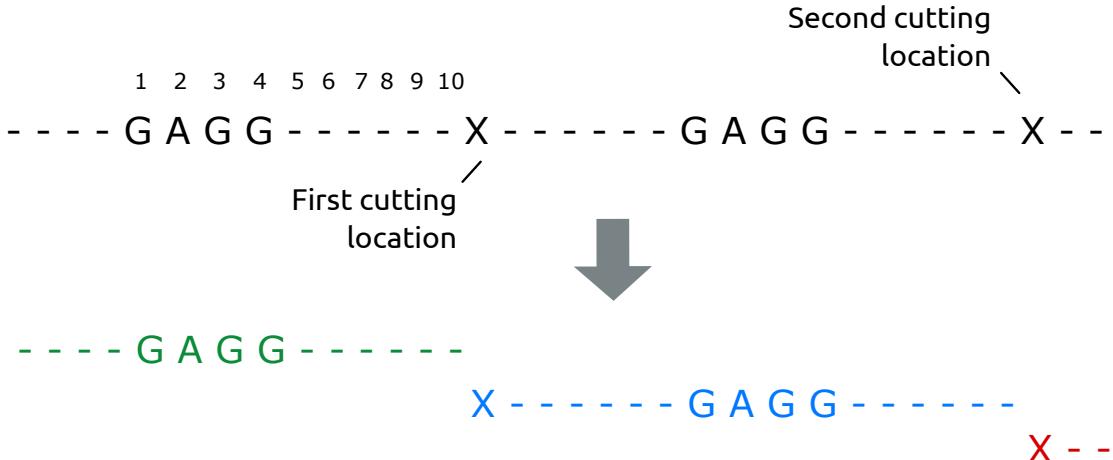


Figure 8.10: Manual restriction cutting of the linear dummy sequence. Top: Our linear dummy sequence, with recognition sites (orange) and cut locations ('X'). We start from the beginning and go through the sequence until we find the first GAGG sequence. We count 10 nucleotides from the start of the GAGG sequence to find where the restriction enzyme cuts the plasmid ('X'). We continue from the cut location until we find the next GAGG sequence, count 10 nucleotides from the start of this GAGG sequence to find the second cut. Bottom: the resulting fragments, with the first fragment in green, the second in blue and the third in red.

We want to create the function `restriction_cutting()`, which takes three arguments: the string `dna`, the string `recognition_sequence`, and the integer `cut_site_offset`. It should return a list of the cutting sites found for the specified index. We first implement the code piece by piece, and then put the code into a function once we have all the pieces. We use the dummy DNA, the MnlI restriction site, and the MnlI cut index as while building the code:

```
dna = "----GAGG-----X-----GAGG-----X--"
recognition_sequence = "GAGG"
cut_site_offset = 10
```

First we find the cutting locations, then we use those cutting locations to slice the `dna` string into fragments. We start with an empty list called `cutting_locations` to store the locations of the cuts. To find the cutting locations we loop through each index of the plasmid by using `range()` over the length of the `dna`, with `for index in range(len(dna))`. We choose a `for` loop here because we know exactly how many iterations we need. For each index, we slice a region that has the same size as the `recognition_sequence`:

```
# Find the cutting locations
cutting_locations = []
for index in range(len(dna)):
```

```
# Slice a region of the same size as the recognition_sequence
index_end = index + len(recognition_sequence)
region = dna[index:index_end]
```

We then compare the slice with the `recognition_sequence`. If they match, we have found a recognition site. We then know that the cutting location is located 10 nucleotides (`cut_site_offset`) from the current index. We print the DNA string from the cutting location to the end of the `dna` string, by writing `dna[cutting_location:]`. Finally, we add the new cutting location to the list `cutting_locations`:

```
# Find the cutting locations
cutting_locations = []
for index in range(len(dna)):
    index_end = index + len(recognition_sequence)
    region = dna[index:index_end]

    if region == recognition_sequence:                      # New
        cutting_location = index + cut_site_offset          # New

        print("found cutting location")                   # New
        print("cut index:", cutting_location)              # New
        print("remaining nucleotides:", dna[cutting_location:]) # New
        cutting_locations.append(cutting_location)

print("Cutting locations:", cutting_locations)
```

```
found cutting location
cut index: 14
remaining nucleotides: X-----GAGG-----X--
found cutting location
cut index: 31
remaining nucleotides: X--
Cutting locations: [14, 31]
```

We obtain two cut locations as expected, see Figure 8.10.

8.3.5 Finding DNA fragments

When we have the indices where the enzyme cuts the DNA, we can perform the cutting to get the DNA fragments. We need the number of DNA fragments, which is equal to the number of cutting locations. Additionally, we need a list to store the fragments:

```
nr_fragments = len(cutting_locations)
fragments = []
```

To find the DNA fragments we slice our `dna` string with two consecutive cutting locations, which gives the fragment between those cutting locations. We loop through the indices from 0 up to but not including `nr_fragments - 1`. We have to restrict the range to `nr_fragments - 1` since we inside the loop cut between `index` and `index + 1`. Where we to use `range(nr_fragments)`, `index + 1` would become higher than the number of elements in the list `cutting_locations`, and we get an `IndexError` error.

```

for index in range(nr_fragments - 1):
    fragment = dna[cutting_locations[index]:cutting_locations[index + 1]]
    fragments.append(fragment)

print("fragments:", fragments)

```

fragments: ['X-----GAGG-----']

Our dummy sequence only has 1 fragment, which means this code only performs 1 cut. This is because we wrote the program to give us the fragment *between* the cutting locations. This means we only obtain the middle one. The fragments we lack are the first one, and the last one. The same would happen if our sequence had more than two cut sites: our program would give us all of the fragments, except the first and last one. The first and last fragment are a thus special case, and we handle these outside our loop.

The first goes from the first base, `dna[0]`, to the first cutting location, `cutting_locations[0]`:

```

# Perform the first cutting
fragment = dna[0:cutting_locations[0]]
fragments.append(fragment)

```

The last fragment goes from the last cutting location, `cutting_locations[-1]` to the end of our sequence:

```

# Perform the last cutting
fragment = dna[cutting_locations[-1]:]
fragments.append(fragment)

```

In our function, we place the code for adding the first fragment before the loop, and the one for adding the last fragment after the loop. We now have all we need to define the complete `restriction_cutting` function:

```

1 def restriction_cutting(dna, recognition_sequence, cut_site_offset):
2     """Perform a restriction cutting of the given DNA strand."""
3     # Find the cutting locations
4     cutting_locations = []
5     for index in range(0, len(dna)):
6         # Slice a region of the same size as the recognition_sequence
7         index_end = index + len(recognition_sequence)
8         region = dna[index:index_end]
9
10        if region == recognition_sequence:
11            cutting_locations.append(index + cut_site_offset)
12
13    # Perform the cutting
14    nr_fragments = len(cutting_locations)
15    fragments = []
16
17    # Perform the first cutting
18    fragment = dna[0:cutting_locations[0]]
19    fragments.append(fragment)
20
21    # Perform the cutting for all fragments except the first and last one

```

```
22     for index in range(nr_fragments - 1):
23         fragment = dna[cutting_locations[index]:cutting_locations[index + 1]]
24         fragments.append(fragment)
25
26     # Perform the last cutting
27     fragment = dna[cutting_locations[-1]:]
28     fragments.append(fragment)
29
30 return fragments
```

We now test the function using our dummy linear sequence:

```
dna = "----GAGG-----X-----GAGG-----X--"
fragments = restriction_cutting(dna, recognition_sequence, cut_site_offset)
print(fragments)
```

```
['----GAGG-----', 'X-----GAGG-----', 'X--']
```

These results match the DNA fragments we found when we performed the restriction cutting by hand in Figure 8.12.

Our `restriction_cutting()` can now be used to perform restriction cutting of any linearised (plasmid) DNA sequence, as we will do in the following section.

8.3.6 Restriction fragment analysis of normal and sickle-cell hemoglobin

Let us use our `restriction_cutting()` function to perform a restriction fragment analysis of normal and sickle-cell hemoglobin, to see if we find any difference between the two.

To obtain the input for the function, we need to give our program access to the recombinant plasmid sequences. We have put the sequence for a recombinant plasmid with the gene for sickle-cell hemoglobin in the file `hemoglobin_sickle_cell_plasmid.fasta` and for normal hemoglobin in the file `hemoglobin_plasmid.fasta`.

These plasmids have been created by performing the above described cloning technique, using the normal and sickle-cell hemoglobin DNA sequences, the plasmid called pBR322, and the StyI restriction enzyme. It is important to note that we now work with DNA and not mRNA, since restriction cutting is performed on DNA strands. Also, since we cannot represent circular DNA in a file, the sequences are linear, with the plasmid cut in the middle.

We use `load_fasta()` to load `hemoglobin_sickle_cell_plasmid.fasta` and `hemoglobin_plasmid.fasta`. Then we perform the restriction cutting with the restriction enzyme MnlII on both of these sequences. This enzyme cuts where the sickle-cell mutation is located, and we therefore expect to see a different number of fragments for normal hemoglobin compared to sickle-cell hemoglobin.

```
from bioinformatics import load_fasta

# Restriction enzyme MnlI
recognition_sequence = "GAGC"
cut_site_offset= 10
```

```
# read fasta files
plasmid_normal = load_fasta("hemoglobin_plasmid.fasta")
plasmid_sickle_cell = load_fasta("hemoglobin_sickle_cell_plasmid.fasta")

# Perform restriction cutting
fragments_normal = restriction_cutting(plasmid_normal, recognition_sequence, cut_site_offset)
fragments_sickle_cell = restriction_cutting(plasmid_sickle_cell, recognition_sequence,
                                             cut_site_offset)

print("Number of DNA fragments for normal hemoglobin:", len(fragments_normal))
print("Number of DNA fragments for sickle-cell hemoglobin:", len(fragments_sickle_cell))
```

```
Number of DNA fragments for normal hemoglobin: 11
Number of DNA fragments for sickle-cell hemoglobin: 10
```

We do indeed get a different number of fragments after the restriction cutting, and are able to detect if a person has sickle-cell anemia or not.

In the laboratory we do not see the results after the restriction cutting, we only have a soup of non-visible DNA fragments. To be able to observe their lengths we need to perform a gel electrophoresis on these fragments.

We have created a function for you that creates plots similar to the results you get from performing a gel electrophoresis analysis in the laboratory. This function is named `plot_gel_electrophoresis()` and separates the different fragments depending on their length. The function takes a dictionary as argument, where the key is the name of the fragments, and the value is the list of DNA fragments you get from `restriction_cutting()`. This function is found in `gel_electrophoresis.py`, and is imported from that file.

We add `plot_gel_electrophoresis()` at the end of our program, which then becomes as follows:

```
1  from pylab import *
2  from bioinformatics import load_fasta
3  from gel_electrophoresis import plot_gel_electrophoresis
4
5  # Restriction enzyme MnI
6  recognition_sequence = "GAGG"
7  cut_site_offset= 10
8
9  # read fasta files
10 plasmid_normal = load_fasta("hemoglobin_plasmid.fasta")
11 plasmid_sickle_cell = load_fasta("hemoglobin_sickle_cell_plasmid.fasta")
12
13 # Perform restriction cutting
14 fragments_normal = restriction_cutting(plasmid_normal, recognition_sequence, cut_site_offset)
15 fragments_sickle_cell = restriction_cutting(plasmid_sickle_cell, recognition_sequence,
                                             cut_site_offset)
16
17 fragments = {"normal": fragments_normal,
18              "sickle-cell": fragments_sickle_cell}
19 plot_gel_electrophoresis(fragments)
20 show()
```

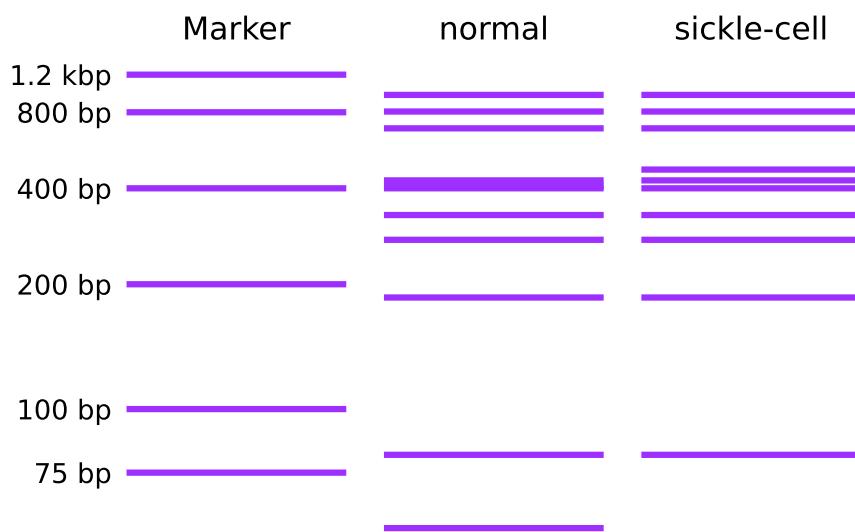


Figure 8.11: Gel electrophoresis of linearised recombinant plasmids. Result from running the gel electrophoresis program on the simulated restriction digestions of linearised plasmid containing normal (middle lane) and mutated (right lane) hemoglobin DNA sequences. Left lane: DNA size marker with sizes indicated. kbp: kilo base pairs (1000 base pairs).

This plot has a marker that contains bands made from DNA fragments of known lengths, given as a number of base pairs to the left. Such markers are used to estimate the size of the unknown fragments in the other samples. We see that we get a different number of bands for sickle-cell hemoglobin and normal hemoglobin. The fourth band from the top is slightly bigger in the fragments derived from the sickle cell DNA, and the smallest fragment seen in the normal hemoglobin is lacking among the sickle cell fragments. This indicates that one of the fragments from the sickle-cell hemoglobin has been split into two fragments in DNA from normal hemoglobin. You can confirm these observations by writing a short program to print the lengths of the fragments obtained from the restriction cutting.

If you were to perform this experiment in the laboratory, you would end up with a similar gel electrophoresis result.

8.3.7 Finding the cut locations on a circular DNA

The previous sections showed a solution to cutting linear DNA. The problem with this simplified implementation is that it does not reflect that plasmids are circular. In this section, we want to show how we can expand our program to take circular sequences in consideration. This section can be skipped if wanted.

When we developed the program to cut linear sequences, we used a dummy sequence to test and debug our code. Now that we want to implement cutting circular DNA, let us go back to this dummy sequence:

```
 dna = "----GAGG-----X-----GAGG-----X--"
```

From now on, we need to consider the situation that this dummy DNA sequence is in fact circular. We still have two cut sites, but when we circularise the dummy sequence, we only get two fragments instead of three! This is because the bases up to the first cut site become part of the second fragment.

If we again perform the restriction cutting manually, but now using the dummy sequence as a circular sequence, this gives us the two fragments X-----GAGG----- and X-----GAGG-----, which slightly differ in size, as illustrated in Figure 8.12. We use this result to check that our code works as intended.

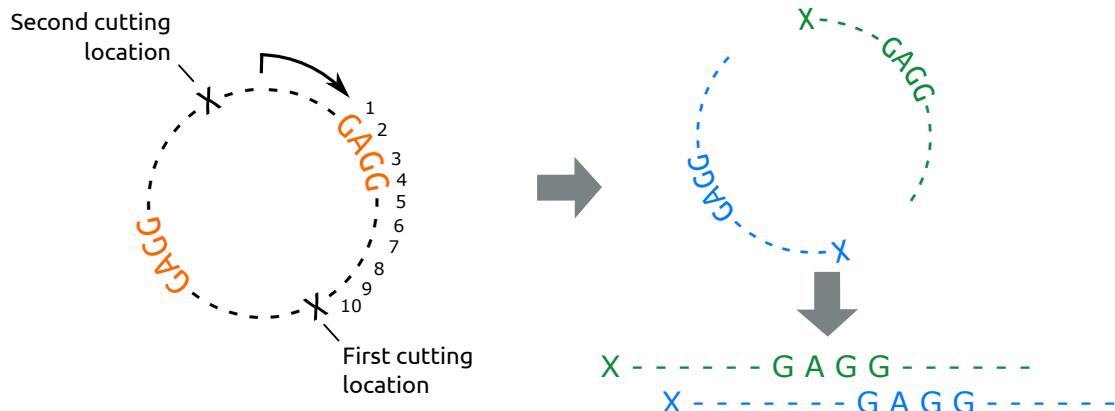


Figure 8.12: Manual restriction cutting of the circular dummy sequence. Left: Our circular dummy plasmid, with recognition sites (orange) and cut locations ('X'). We start from the arrow and go through the plasmid until we find the first GAGG sequence. We count 10 nucleotides from the start of the GAGG sequence to find where the restriction enzyme cuts the plasmid. We continue from the cut location until we find the next GAGG sequence, count 10 nucleotides from the start of this GAGG sequence to find the second cut. Top right: the plasmid is cut in two places. Bottom right: the resulting fragments, with the first fragment in green and the second in blue.

We now need to implement a function that handles circular DNA strings.

To better illustrate the problem of slicing a circular string, and to help us develop our new function, we use the following string of numbers, where each number corresponds to the index in the string:

```
circular_string = "0123456789"
```

Our goal is to write a function, which we name `circular()`, which correctly slices the string. To slice this correctly as a circular string, we need to consider three cases.

The first case is normal slicing, which is when the start index is lower than the stop index, and both are lower than the length of the string, see Figure 8.13 (left):

```
region = circular(circular_string, 3, 7)
print(region)

# We expect: "3456"
```

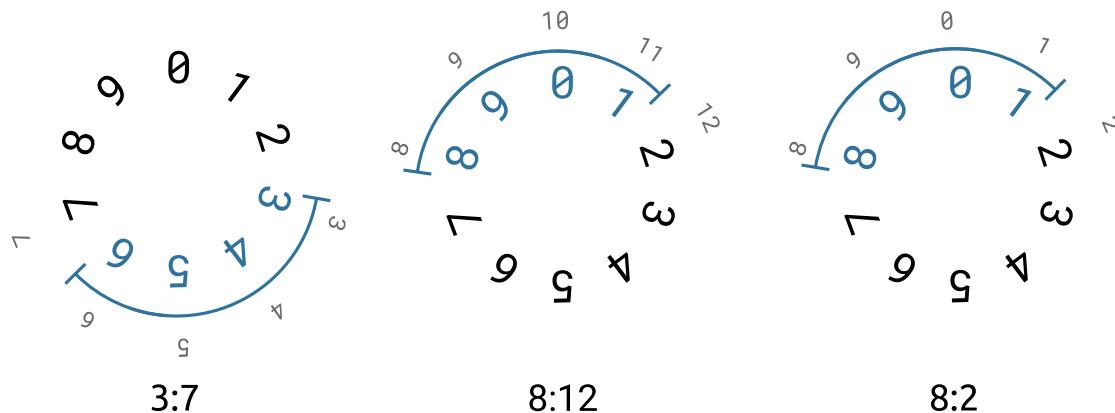


Figure 8.13: The three cases of slicing we have to take into account. The indexes of the fragment to obtain are written below for each case, and the fragment to slice with the indexes are indicated in blue. Left: the first case is normal slicing, which is when the start index (8) is lower than the stop index (7) and both are lower than the length of the string. Middle: the second case is when the stop index (12) is larger than the length of the string. Right: The final case is when the stop index (2) is smaller than the start index (8).

The second case is slicing when the stop index is larger than the length of the string, see Figure 8.13 (middle):

```
region = circular(circular_string, 8, 12)
print(region)

# We expect: "8901"
```

The final case is when the stop index is smaller than the start index, see Figure 8.13 (right):

```
region = circular(circular_string, 8, 2)
print(region)

# We expect: "8901"
```

We begin with writing a function that implements normal slicing:

```
def circular(dna, start_index, stop_index):
    """Slice a circular DNA string."""

    fragment = ""
    index = start_index
    for index in range(start_index, stop_index):
        fragment += dna[index]

    return fragment
```

Here, we start with an empty string called `fragment` to store our final slice. Then we loop through our DNA from the start index until we come to the stop index. Since we know exactly how many iterations we need, we can use a `for` loop with `range(start_index, stop_index)`: (excluding the stop index). For each pass of the loop we add the current nucleotide to `fragment`

and increase the `index` variable by one. When we encounter the stop index, we exit the loop and return the fragment.

Let us test this function on the first case:

```
region = circular(circular_string, 3, 7)
print(region)
```

```
| 3456
```

This works as expected, but the function is not yet able to handle the above cases. If we try the second case, where the stop index is beyond the length of the string, we get an error:

```
region = circular(circular_string, 8, 12)
print(region)
```

```
-----
IndexError                                                 Traceback (most recent call last)
Input In [361], in <cell line: 1>()
----> 1 region = circular(circular_string, 8, 12)
      2 print(region)

Input In [359], in circular(dna, start_index, stop_index)
      5 index = start_index
      6 for index in range(start_index, stop_index):
----> 7     fragment += dna[index]
      9 return fragment

IndexError: string index out of range
```

This is because the value of `index` is larger than the length of the string at the iteration where the program fails. This results in an `IndexError` when calling `dna[index]`. We need to do something about the `index` when it becomes higher than the length of the string. One solution is to make the index start over from zero. We achieve this by using the *modulo operator* `%`.

Modulo operator. The modulo operator `%` gives you the remainder of a number divided by another number:

```
print(5 % 3)
```

```
| 2
```

We pronounce this as “five modulo three”. We cannot divide five by three more than once, and if we do that, we are left with $5 - 3 = 2$. When we do “five modulo three”, we thus get two. The modulo operator has a feature we need; it makes numbers “wrap around” once they become larger than the number we divide by:

```
for i in range(7):
    print(i, "modulo 3 is:", i % 3)
```

```
0 modulo 3 is: 0
1 modulo 3 is: 1
2 modulo 3 is: 2
3 modulo 3 is: 0
4 modulo 3 is: 1
5 modulo 3 is: 2
6 modulo 3 is: 0
```

An analog clock works similarly to taking the modulo of 12. If the time is 09:00, and we wait 4 hours, the time becomes 13:00. However, for the same hours, the short hand on an analog clock first points to 9, and then to 1, as shown in Figure 8.14. This is because, once we pass 12 on the clock, we wrap around, and start counting from zero.

The modulo operator does the same:

```
print(9 % 12)
print(13 % 12)
```

```
9
1
```

This is the behavior we need to make our index “wrap around” once we get to the end of the string.

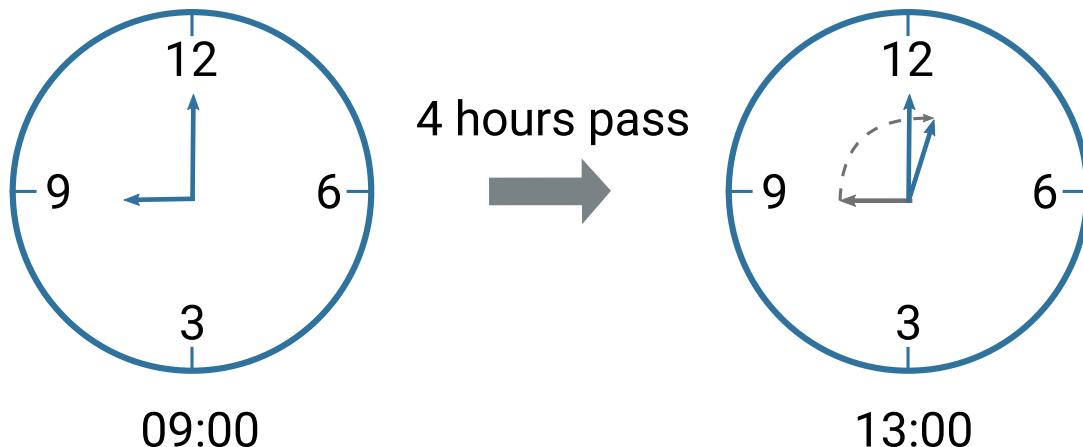


Figure 8.14: An analog clock uses modulo 12. If the time is 09:00, and we wait 4 hours, the time becomes 13:00. However, for the same hours, the short hand on an analog clock first points to 9, and then to 1. Once we pass 12 on the clock we start counting from zero.

Slicing with indices greater than the length of the string. We use the modulo operator with `index` and the length of the string `dna`, and store the result in the variable `circular_index`. We then use this variable to get the current element in the string:

```
def circular(dna, start_index, stop_index):
    """Slice a circular DNA string."""
```

```
fragment = ""
index = start_index
for index in range(start_index, stop_index):
    circular_index = index % len(dna)           # New
    fragment += dna[circular_index]             # New

return fragment
```

Let us test `circular()` on the case where we index beyond the end of the string:

```
region = circular(circular_string, 8, 12)
print(region)
```

8901

Perfect! This is exactly the result we wanted. Next, we need to implement the case where the start index is larger than the stop index.

Slicing from a higher index to a lower index. If we call the `circular()` function as it is defined now, with a stop index that is smaller than the start index, we get an endless loop. To avoid this, we use a trick of adding the length of the `dna` string to `stop_index`. The reason this works is illustrated in Figure 8.15.

```
def circular(dna, start_index, stop_index):
    """Slice a circular DNA string."""
    if stop_index <= start_index:                  # New
        stop_index += len(dna)                     # New

    fragment = ""
    index = start_index
    for index in range(start_index, stop_index):
        circular_index = index % len(dna)
        fragment += dna[circular_index]

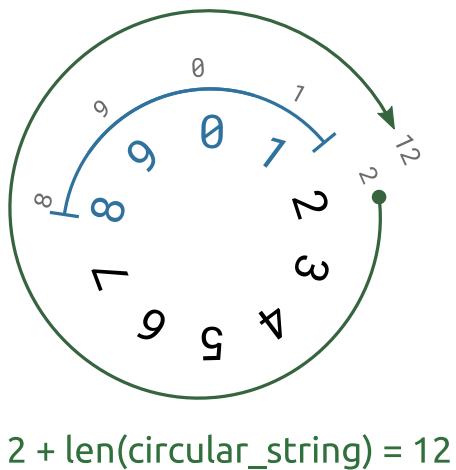
    return fragment
```

We are now able to slice with a higher from-index than the to-index:

```
region = circular(circular_string, 8, 2)
print(region)
```

8901

Our `circular()` function now works in all three cases. We go back to the restriction cutting and use the `circular()` function to slice our `dna` string instead of regular string slicing.



8:2 → 8:12

Figure 8.15: Adding the length of the dna string to the last index, moves the index one round along the circular string.

8.3.8 Implementing restriction cutting of circular DNA

We are now ready to test our new function with our dummy sequence. We expect two sequences, as we saw in Figure 8.12.

We will reuse the function `restriction_cutting`, but modify it to use our new `circular()` function. We should also give it a new name.

There are two important other changes we need to make, and these have to do with the first and last fragments. In our `restriction_cutting` function, we had to do the first and last cuts outside of the loop. This is because the DNA was already cut in the middle. In the case of circular DNA, the first cut is the one that starts at the first cut site, and the last one will include the fragment from the start of the linearised fragment to the first cut site. The beginning of our linearised fragment thus becomes part of the final fragment, see Figure 8.12. We thus do no longer need to do the first cut separately, but only the last one.

To cut the last fragment, we also need to adjust our function. The indexes of the last fragment are the from *last* cutting location to the *first* cutting location in our `cutting_locations` list, again see Figure 8.12:

```
fragment = circular(dna, cutting_locations[-1], cutting_locations[0])
```

Our final `restriction_cutting_circular` function than becomes:

```
def restriction_cutting_circular(dna, recognition_sequence, cut_site_offset):
    """Perform a restriction cutting of the given DNA strand."""
    # Find the cutting locations
    cutting_locations = []
    for index in range(0, len(dna)):
        region = circular(dna, index, index+len(recognition_sequence))
```

```

if region == recognition_sequence:
    cutting_locations.append(index + cut_site_offset)

# Perform the cutting
nr_fragments = len(cutting_locations)
fragments = []

for index in range(nr_fragments - 1):
    fragment = circular(dna, cutting_locations[index], cutting_locations[index + 1])
    fragments.append(fragment)

# Perform the last cutting
fragment = circular(dna, cutting_locations[-1], cutting_locations[0])
fragments.append(fragment)

return fragments

```

We now run it again on the dummy sequence:

```

dna = "-----GAGG-----X-----GAGG-----X--"
recognition_sequence = "GAGG"
cut_site_offset = 10
fragments = restriction_cutting_circular(dna, recognition_sequence, cut_site_offset)
print(fragments)

```

```
[ 'X-----GAGG-----', 'X-----GAGG-----' ]
```

These results match the DNA fragments we found when we performed the restriction cutting of the circular dummy plasmid by hand in Figure 8.12.

8.3.9 Redoing the restriction fragment analysis of normal and sickle-cell hemoglobin

Let us use our `restriction_cutting_circular()` function to redo the restriction fragment analysis of normal and sickle-cell hemoglobin.

```

# Perform restriction cutting
fragments_normal_circular = restriction_cutting_circular(plasmid_normal, recognition_sequence,
    cut_site_offset)
fragments_sickle_cell_circular = restriction_cutting_circular(plasmid_sickle_cell,
    recognition_sequence, cut_site_offset)

print("Number of DNA fragments for normal hemoglobin:", len(fragments_normal_circular))
print("Number of DNA fragments for sickle-cell hemoglobin:", len(fragments_sickle_cell_circular))

```

```
Number of DNA fragments for normal hemoglobin: 10
Number of DNA fragments for sickle-cell hemoglobin: 9
```

Compared to the number of fragments we obtained when we cut the linearised plasmid in Section 8.3.6, we do indeed get a different number of fragments after the restriction cutting of the circular plasmid using.

We can now also redo the gel electrophoresis:

```

1 fragments = {"normal": fragments_normal_circular,
2                 "sickle-cell": fragments_sickle_cell_circular}
3 plot_gel_electrophoresis(fragments)
4 show()

```

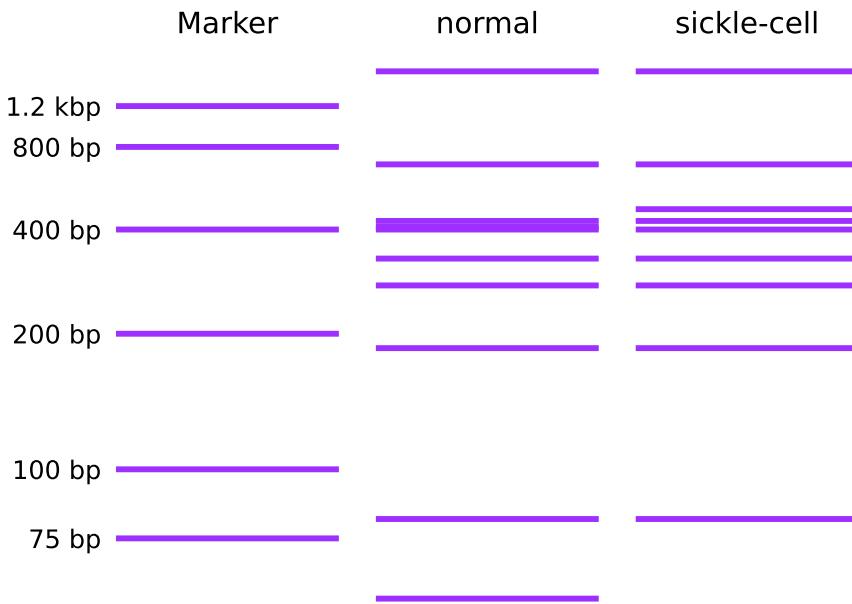


Figure 8.16: Gel electrophoresis of circular recombinant plasmids. Result from running the gel electrophoresis program on the simulated restriction digestions of (circular) plasmid containing normal (middle lane) and mutated (right lane) hemoglobin DNA sequences. Left lane: DNA size marker with sizes indicated. kbp: kilo base pairs (1000 base pairs).

We see a new band larger than the 1.2 kb marker that was split into two bands in the linear case, see Figure 8.11. Other than that, the result remain the same: we get a different number of bands for sickle-cell hemoglobin and normal hemoglobin.

What we have done in this section describes an experimental method you can use to detect DNA with the sickle-cell anemia causing mutation. We have simulated the experiment to predict what results we can expect without performing the experiment first. We confirmed that it should be possible to detect sickle cell anemia with this experimental technique.

Some experiments are extremely expensive and take an extraordinary long time to finish. In those cases, it can be very important to test if the experiment is feasible or not before you start.

In addition to the techniques you have learned in the last two chapters on bioinformatics, there exists a plethora of other useful tools and techniques for different types of DNA analysis, both experimental and computational. It can be useful to remember this when you later encounter problems not discussed in this book. If you go looking for it, you might find that there exists a tool or technique that solves exactly your problem.

8.4 Writing to files

We have read the DNA sequences we have digested using our Python program using the `open()` function. But what if we would want to write the resulting fragments to a file, for example to share them with a colleague?

We can also write to a file by opening it with the `open()` function. But instead of the argument "`r`" (for read), we use the argument "`w`" (for write) or "`a`" (for append):

```
f = open("our_file.txt", "w")
```

If the file already exists, and we want to keep the existing content, we must use the "`a`" argument. Opening the file with the "`w`" argument creates a new file if it does not yet exist, otherwise it removes all content from the existing file with the same name, effectively overwriting the file.

We write to the file with `f.write()`, which returns the number of characters written to the file:

```
f.write("The first line.\n")
```

| 16

```
f.write("The second line.")
```

| 16

`f.write()` does not automatically give us newlines from one function call to the next. To get a newline we therefore have to write `\n` at the end of our string.

We have to close the file once we are finished writing to it:

```
f.close()
```

If we take a look at the contents of `our_file.txt`, we find the following:

```
The first line.  
The second line.
```

If you try to read from a closed file, you get the following error:

```
f = open("our_file.txt", "w")  
f.close()  
  
lines = f.readlines()
```

```
-----  
ValueError                                Traceback (most recent call last)  
Input In [377], in <cell line: 4>()  
      1 f = open("our_file.txt", "w")
```

```
1     2 f.close()
2 ----> 4 lines = f.readlines()
3
4 ValueError: I/O operation on closed file.
```

This tells us we have `ValueError` because we try an `I/O operation on closed file`. I/O stands for Input/Output, which are things such as reading from or writing to a file.

For writing the fragments from digesting the normal hemoglobin as a circular plasmid to a file, we first open a new file with an appropriate name. Then we use a `for` loop to go over the list of fragments, and write each one to the file. We again have to add a newline to each fragment, otherwise all fragments be printed on the same line. We will keep things simple and not try to create the file in FASTA format, which was the format for the plasmid sequence. Our program to write the fragments to a file then becomes:

```
1 f = open("fragments_normal.txt", "w")
2 for fragment in fragments_normal:
3     f.write(fragment + "\n")
4 f.close()
```

Note that we can use `+ "\n"` for adding the newline, as `fragment` is of type string. Now we are ready to send the file to our colleague.

8.5 Summary

In this chapter we have studied point mutations and discussed one type of genetic disorder they might give rise to, namely sickle-cell anemia.

8.5.1 Point mutations

A point mutation changes a single pair of nucleotides in the DNA. There are two types of point mutations, insertions/deletions, and substitutions.

Insertions and deletions. Insertions and deletions cause base pairs to be either inserted or removed from the DNA and subsequently the mRNA if the DNA codes for a protein. When these mutations occur in DNA coding for a protein, they often cause frameshifts that offset the DNA sequence and shift the reading frame. If a frameshift occurs, it causes all the following triplets to be misinterpreted when they are read. Such a frameshift can have a disastrous effect on the resulting protein.

Substitutions. A substitution is when a base pair has been changed to another base pair. Substitutions in DNA coding for protein may or may not effect the resulting protein sequence. Some such substitutions are silent: a codon is changed to another codon that encodes for the same amino acid and the resulting protein is unchanged.

Substitutions might also change a codon into a codon that encodes for a different amino acid, called a missense mutation. These new proteins may be functionally different from the original protein.

Another case is if the substitution alters a codon coding for an amino acid into a stop codon, instead, called a nonsense mutation. This leads to premature termination of the protein and the resulting protein is vastly different. In most cases the protein will be nonfunctional. The reverse may also happen, such that a codon coding for a stop codon is changed into one coding for an amino acid, leading to an elongation of the protein sequence.

8.5.2 Restriction fragment analysis

The general idea behind restriction fragment analysis is to use restriction enzymes to cut DNA into smaller pieces. Since restriction enzymes cut at specific, predictable places in the DNA, we can examine if there are differences between two DNA sequences at the recognition sites for the restriction enzymes. Restriction fragment analysis consists of three steps:

Cloning. Plasmids are extracted from bacteria, and a DNA sequence of interest is inserted into the plasmid to create recombinant DNA. This recombinant DNA is reinserted into bacteria, which, when they reproduce, create copies of the plasmid, and thereby of the sequence of interest.

Restriction cutting. In this step restriction enzymes are used to cut DNA into small fragments. Restriction enzymes recognize all occurrences of a specific DNA sequence called recognition sequence, and cut the DNA at, or at a specific distance from these occurrences.

Gel electrophoresis. After the restriction cutting we get a soup of different DNA fragments that are invisible to the naked eye. To observe the differences we use gel electrophoresis to separate DNA fragments with different size and add a DNA-binding dye, resulting in distinct bands.

8.5.3 Importing functions from a file

We get access to the `translate` function from the `bioinformatics.py` by writing:

```
from bioinformatics import translate
```

Note that we leave out the extension `.py` of the `bioinformatics.py` file when importing. Importing other functions from other files is done similarly, by writing `from filename import function` or `from filename import *` to import all functions from the file.

8.5.4 Reading files

Python has built in functionality for loading any file with the `open()` command. To open the hemoglobin file with the name `hemoglobin_normal.fasta` we write:

```
f = open("hemoglobin_normal.fasta", "r")
```

`readlines()` returns a list where each line in our file is an element in the list.

```
lines = f.readlines()
```

`f` is a file object and we should always close this when we are finished with the file:

```
f.close()
```

8.5.5 Writing to files

We can write to a file by opening it with the argument "`w`" (for write) or "`a`" (append).

```
f = open("our_file.txt", "w")
```

We write to the file with `f.write()`, which returns the number of characters written to the file:

```
f.write("The first line.\n")
```

16

We close the file once we are finished writing to it:

```
f.close()
```

8.5.6 Traversing two or more lists simultaneously with zip

We traverse two or more lists simultaneously using the `zip()` function. `zip()` takes multiple lists as arguments and makes it so we iterate over all simultaneously. The loop stops when the end of the shortest list is reached:

```
list_a = [1, 2]
list_b = [10, 20, 30]

for a, b in zip(list_a, list_b):
    print(a, b)
```

1 10
2 20

In the example above we have unpacked the elements directly in the `for` loop line, but it can also be done inside the loop.

Chapter 9

Modeling epidemics

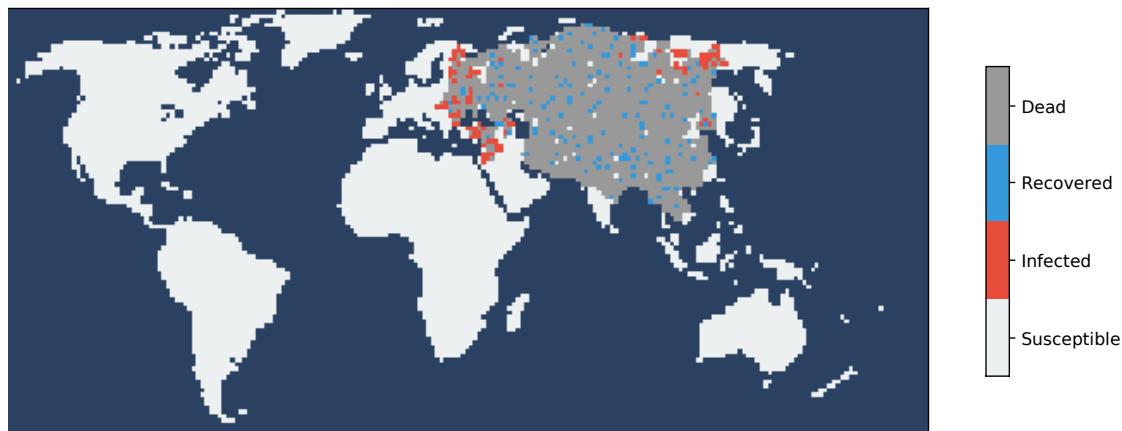


Figure 9.1: A simulation of a pandemic that is spreading through Asia and towards Europe. In this chapter, you will learn the tools to create similar simulations yourself.

An infectious disease that spreads to many people within a short period of time is called an *epidemic*. *The Black Death* is one of the most devastating epidemics in human history. Due to its global influence, the Black Death is called a *pandemic*. It originated in Asia in the 1330s, and spread throughout the continent before arriving in Europe, where up to 60% of the population died. The Black Death was caused by the *bubonic plague*, which is a bacterial infection that spreads through infected fleas. Without treatment, the bubonic plague resulted in death for up to 90% of the infected.

Knowledge of how diseases spread is vital to prevent pandemics such as the Black Death. In this chapter, we examine the spread of disease on a population level. We introduce the *SIR model* (Susceptible-Infectious-Recovered model) for the spread of infectious diseases, and use it to examine a real-world epidemic.

Most diseases spread from one person to another only if they are near each other. To take this into account, we expand the SIR model to include where people live. By taking the position of people into account, the SIR model becomes a *spatial model*. We use this model to examine strategies for how to stop a disease.



Learning outcomes

After working with this chapter, you will know:

- what the SIR model for infectious disease is,
- how to implement the SIR model as a population dynamics model,
- how to implement the SIR model as a spatial model, and
- the importance of herd immunity.

The programming concepts we introduce in this chapter are:

- arrays
- spatial models on a 2D grid
- tuples, and
- more on random numbers.

9.1 The SIR model of infectious disease

After The Black Death, the bubonic plague returned in epidemics with varying intervals up until the early 19th century. One of these occurrences was the Great Plague of London (1665-66), which killed almost a quarter of London's population in 18 months. During the Great plague of London, the plague spread to a small English village named Eyam. The villagers isolated themselves, and no one was allowed to enter or leave. The villagers managed to stop the spread of the plague to other nearby villages, but at the end of the epidemic only 83 of the original 289 villagers survived.

Historical records were kept of the deaths in the village. We have gathered the death toll for each week in the file `eyam.csv`. Let us load this file and plot the data:

```
from pylab import *
import pandas

data = pandas.read_csv("eyam.csv")
t_historical = list(data["t"])
death_toll_historical = list(data["D"])

plot(t_historical, death_toll_historical, "-o")
```

```

xlabel("Week")
ylabel("Death toll per week")
show()

```

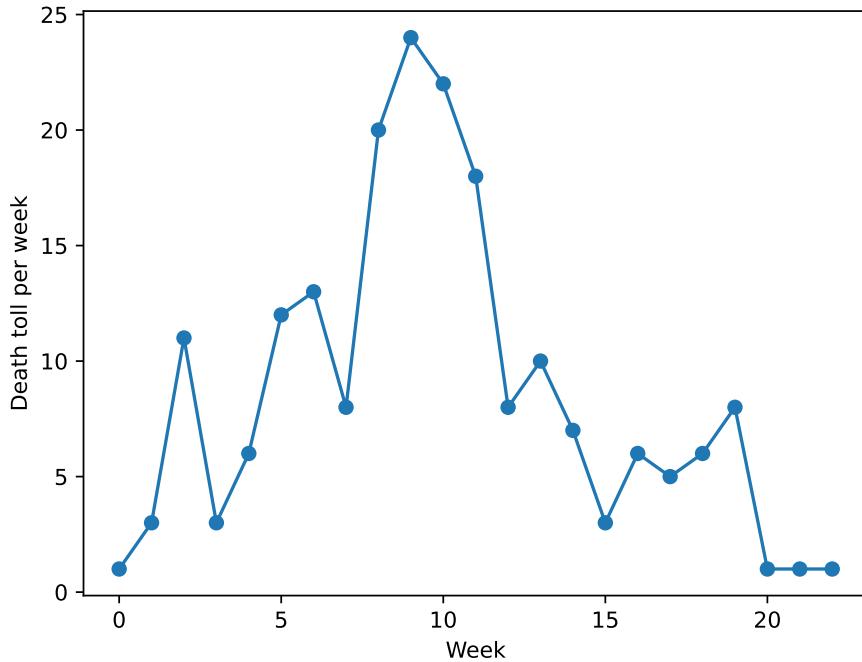


Figure 9.2: The recorded deaths from the bubonic plague in Eyam, 1666.

We want to model how this outbreak evolves by using the *SIR model*. This model works well for infectious diseases transmitted from human to human, where recovery makes you immune to the disease. It can therefore be used to model many different disease outbreaks. In the SIR model, people are placed in three groups:

- *S* - Susceptible. People in this group are currently healthy, but they are able to become infected.
- *I* - Infected. These are people that are currently infected, and can infect others.
- *R* - Recovered. These are people that neither can be infected or infect others. This can be for several reasons; they can have recovered from the disease, they can have died, or they can have been vaccinated and are immune.

We use the symbols *S*, *I*, and *R* to describe the number of individuals in each state, and the subscript *n* to denote the current week. We assume that individuals go only from susceptible, via infected, to recovered ($S \rightarrow I \rightarrow R$). This means that people cannot be reinfected in this model.

9.1.1 Deriving the SIR model equations

For simplicity, we assume that no births occur. The total number of people should therefore be constant (R_n also includes people that have died):

$$S_n + I_n + R_n = N. \quad (9.1)$$

As with the models of *E. coli* or plant growth, we can derive equations for how these three groups evolve over time.

Susceptible. Let us start with the equation for the number susceptible individuals in week n , S_n . Because every individual either stays susceptible or becomes infected, the number of susceptible individuals next week is:

$$S_n = S_{n-1} - \text{recently infected individuals}. \quad (9.2)$$

The disease has a chance to spread every time an infected individual interacts with a susceptible individual. With S_{n-1} susceptible individuals and I_{n-1} infected individuals, there are in total $S_{n-1}I_{n-1}$ unique combinations of interaction between susceptible and infected individuals. A factor a of these interactions leads to a susceptible becoming infected. This factor includes the ratio of interactions that actually happen, as well as the probability that the interaction spreads the disease. It does not have a direct interpretation, but a larger value for a means a more infectious disease. Thus, the number of newly infected individuals is:

$$\text{recently infected individuals} = aS_{n-1}I_{n-1}. \quad (9.3)$$

The complete equation for the number of susceptible individuals is then given by:

$$S_n = S_{n-1} - aS_{n-1}I_{n-1}. \quad (9.4)$$

Infected. Every infected individual either stays infected or recovers. The number of infected individuals is:

$$I_n = I_{n-1} + \text{recently infected individuals} - \text{recovered individuals}. \quad (9.5)$$

The ratio of infected individuals that recover every week is given by the *recovery rate* b . Thus, the total number of individuals that recover is given by:

$$\text{recovered individuals} = bI_{n-1}. \quad (9.6)$$

We already know the number of recently infected individuals, so the complete equation for the number of infected individuals is:

$$\begin{aligned} I_n &= I_{n-1} + \text{recently infected individuals} - \text{recovered individuals} \\ &= I_{n-1} + aS_{n-1}I_{n-1} - bI_{n-1}. \end{aligned} \quad (9.7)$$

Recovered. Recovered individuals have no chance of becoming infected again and stay recovered. The number of recovered individuals is:

$$\begin{aligned} R_n &= R_{n-1} + \text{recovered individuals} \\ &= R_{n-1} + bI_{n-1}. \end{aligned} \quad (9.8)$$

A figure with an explanation of the terms in Equation (9.4), (9.7), and (9.8) for the SIR model is shown in Figure 9.3. This system of equations is coupled, since the terms in one equation is dependent on the result from the other equations.

Both the factor a and the recovery rate b must be determined from the specific disease we would like to model. This process is called parameter estimation or parameter fitting and can be very complicated and technical. We therefore simply tell you what variables of a and b works for our case.

$$\begin{aligned} S_n &= S_{n-1} - aS_{n-1}I_{n-1} \\ I_n &= I_{n-1} + aS_{n-1}I_{n-1} - bI_{n-1} \\ R_n &= R_{n-1} + bI_{n-1} \end{aligned}$$

Figure 9.3: Explanation of terms in Equation (9.4), (9.7), and (9.8) for the SIR model.



Summary: The SIR model for an infectious disease

In the SIR model, individuals are placed in three categories:

- S - Susceptible. People in this group are currently healthy and can become infected.
- I - Infected. People in this group are currently infected and can infect others.
- R - Recovered. People in this group cannot become infected or infect others.

The SIR model works well for infectious diseases transmitted from human to human, where recovery makes you immune to the disease. Individuals go from susceptible, to infected, to recovered ($S \rightarrow I \rightarrow R$).

The factor a relates to how many susceptibles become infected. The ratio of infected individuals that recover (either become healthy or die) every week is called the *recovery rate*, b .

The equations for the number of individuals in each group is given by:

$$S_n = S_{n-1} - aS_{n-1}I_{n-1} \quad (9.9)$$

$$I_n = I_{n-1} + aS_{n-1}I_{n-1} - bI_{n-1} \quad (9.10)$$

$$R_n = R_{n-1} + bI_{n-1}. \quad (9.11)$$

To implement the SIR model in Python, we will be using *arrays* in Python, which we will introduce now.

9.2 Arrays: Like lists with support for mathematics

Arrays are important for scientific computing in Python. Arrays are similar to lists, but can only contain data of the same type (strings, floats, integers, etc., see Figure 9.4). However, arrays are much more suitable for mathematics because they support operations, like additions and subtractions, on all elements in an array at the same time. To use arrays we need the Numpy package. Fortunately, Numpy is a part of the `pylab` package and we get access to everything Numpy contains by writing

```
from pylab import *
```

We can make an array containing the numbers 1, 2, 3, and 4, and print it, by writing

```
a = array([1, 2, 3, 4])
print(a)
```

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

Arrays clearly look a lot like lists, so it is reasonable to compare them. Let us discuss the similarities and differences between lists and arrays, and the different mathematical operations we can perform on arrays.

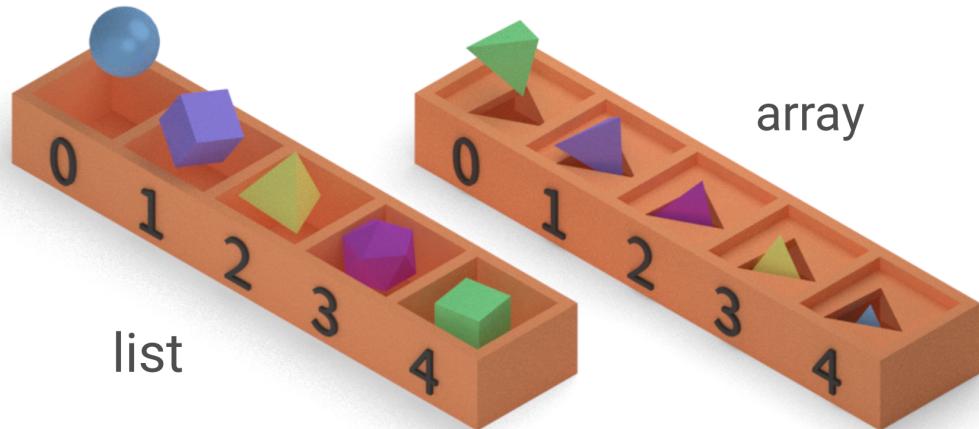


Figure 9.4: Lists can contain elements of many different types (illustrated as shapes above), while arrays can only contain elements of the same type, but each element can have different values (illustrated as colors above).

9.2.1 Similarities and differences between lists and arrays

Just like lists, elements of an array can be accessed using square brackets. Selecting the first element of an array, the element with index zero, is done by:

```
print(a[0])
```

| 1

We can use `len()` to get the length of an array:

```
print(len(a))
```

| 4

If we assign an array `a` to a new variable `b`, a change in `b` will modify `a`, just like for lists:

```
a = array([1, 2, 3, 4, 5, 6])
b = a
b[1] = 1

print(a)
```

```
[1 1 3 4 5 6]
```

Slicing also works similarly to lists. For example `a[2:-1]` picks out all elements except for the first two and last element:

```
a = array([1, 2, 3, 4, 5, 6])
print(a[2:-1])
```

```
[3 4 5]
```

There is, however, a very important difference between slicing arrays and lists. Slicing a list returns a copy of that part of the list. However, slicing an array returns a *view* to that part of the array. This means that if you modify the contents of the view, you also modify the original array (see Figure 9.5). To show this, we create an array `a` and create a view to the three first indices which we call `b`. We then change the value of one of the elements in `b` and see what happens to both `a` and `b`:

```
a = array([1, 2, 3, 4, 5, 6])
b = a[0:3]

print("a before change:", a)
print("b before change:", b)

b[1] = 1

print("a after change:", a)
print("b after change:", b)
```

```
a before change: [1 2 3 4 5 6]
b before change: [1 2 3]
a after change: [1 1 3 4 5 6]
b after change: [1 1 3]
```

As you can see `a` is affected by the change in `b`. This is because `b` is a view of `a`, so any change to `b` will also be reflected in the parts of `a` that `b` represent.

This behavior is important to remember whenever you want to pick out part of a larger array and modify it. If you do not wish to modify the original array, you can copy the part you are interested in like this:

```
a = array([1, 2, 3, 4, 5, 6])
b = a[0:3].copy()
```

Now any change to `b` will not affect `a`:

```
b[1] = 1

print("a after change:", a)
print("b after change:", b)
```

```
a after change: [1 2 3 4 5 6]
b after change: [1 1 3]
```

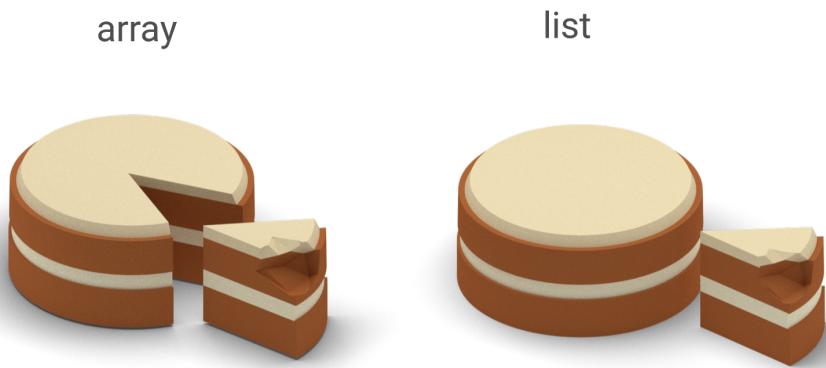


Figure 9.5: Modifying a slice of an array also changes the original, while modifying a slice of a list leaves the original untouched. With lists, you can have your cake and eat it too.

9.2.2 Mathematical operations on arrays

Say we have measured the bacterial population in two different petri dishes, and we want to know the total number of bacteria at each time step. If the data were stored in lists, we would have to add the elements one by one. For arrays, it is as easy as adding the two arrays together

```
E = array([10010, 9951, 10042, 25587, 76327, 212715, 619511, 1940838, 4240760])
E2 = array([9919, 10022, 10010, 9993, 19121, 28904, 49730, 75381, 136761])

E_total = E + E2
print(E_total)
```

```
[ 19929  19973  20052  35580  95448  241619  669241  2016219  4377521]
```

This is because arrays support mathematical operations, which is one of the reasons why arrays are so useful for mathematics. In the example above, element 0 of the first array is added to element 0 in the second array, element 1 of the first array is added to element 1 in the second array and so on.

We can add, subtract, multiply, and divide arrays of equal length. The operation is performed on each element of each array.

```
a = array([1, 2, 3])
b = array([1, 2, 3])
print(a + b)
```

```
[2 4 6]
```

```
print(a - b)
```

```
[0 0 0]
```

```
print(a*b)
```

```
[1 4 9]
```

```
print(a/b)
```

```
[1. 1. 1.]
```

If we have two lists and try to add them together, they are merged instead:

```
list1 = [1, 2, 3]
list2 = [1, 2, 3]
print(list1 + list2)
```

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

Lists neither supports subtraction, multiplication, or division. If you try, you get:

```
list1 = [1, 2, 3]
list2 = [1, 2, 3]
print(list1 - list2)
```

```
-----
TypeError                                     Traceback (most recent call last)
Input In [406], in <cell line: 3>()
      1 list1 = [1, 2, 3]
      2 list2 = [1, 2, 3]
----> 3 print(list1 - list2)

TypeError: unsupported operand type(s) for -: 'list' and 'list'
```

With arrays we can also add, subtract, multiply, and divide by a single number:

```
a = array([1, 2, 3])
print(a + 1)
```

```
[2 3 4]
```

```
print(a - 1)
```

```
[0 1 2]
```

```
print(a**2)
```

```
[2 4 6]
```

```
print(a/2)
```

```
[0.5 1. 1.5]
```

We can also calculate powers of two arrays or an array and a number:

```
print(a**b)
```

```
[ 1  4 27]
```

```
print(a**2)
```

```
[1 4 9]
```

If we try to perform the above mathematical operations with two arrays of different length we get a `ValueError`, that tells us the shape of our arrays are different:

```
a = array([1, 2, 3])
b = array([1, 2])
```

```
print(a + b)
```

```
-----
ValueError                                Traceback (most recent call last)
Input In [413], in <cell line: 4>()
      1 a = array([1, 2, 3])
      2 b = array([1, 2])
----> 4 print(a + b)

ValueError: operands could not be broadcast together with shapes (3,) (2,)
```

Most math functions from the `pylab` package evaluate the function on each element of the array:

```
from pylab import *
```

```
print(sqrt(a))
```

```
[1.        1.41421356 1.73205081]
```

In addition to making the code simpler, arrays speed up Python programs that deal with heavy calculations, which we make use of in this chapter.

9.2.3 Creating arrays

There are several ways to create an array. So far we have done this by converting a list to an array, using `array()`, but for our purpose we need to create arrays with a specific number of

elements. This can be done by using the `zeros()` function. This command, as the name implies, creates an array where all elements are zero. The argument is the number of elements in the array.

```
print(zeros(5))
```

```
[0. 0. 0. 0. 0.]
```

If you want an array where all the elements are equal, but different from zero, you can add a number to an array created using `zeros()`. The following code creates an array with five elements, all initialized to the value 9:

```
print(zeros(5) + 9)
```

```
[9. 9. 9. 9. 9.]
```

It is also useful to create arrays with the same step size between each consecutive element. This can be done using the function `arange(start, stop, step)`, which is similar to `range()`. `arange()` takes a start value, a stop value, and the step size and generates a sequence of numbers from `start` up to but not including `stop` with a step size of `step`. This means that to make an array containing the numbers 0, 1, 2, 3, 4, and 5, we write

```
a = arange(0, 6, 1)
print(a)
```

```
[0 1 2 3 4 5]
```

If we want to make an array of measurement times, where the measurements are made every 30 minutes, we can now do this as

```
t = arange(0, 270, 30)
print(t)
```

```
[ 0 30 60 90 120 150 180 210 240]
```

The difference between `arange()` and `range()` is that `arange()` can take floats as input arguments, while `range()` only can take integers.

```
a = arange(0.1, 6.5, 1.5)
print(a)
```

```
[0.1 1.6 3.1 4.6 6.1]
```

Finally, there are situations where you know the start and stop points of the array, but rather than giving a specific step, you want to set the number of elements in the array. This can be

done with the `linspace()` function. It also takes three arguments, but the last argument is the number of elements, `linspace(start, stop, number_of_elements)`. If we want 5 points spread out on the interval from 0 to 1, we write:

```
a = linspace(0, 1, 5)
print(a)
```

```
[0.  0.25 0.5  0.75 1. ]
```

9.3 Implementing the SIR model

In this section, we use the SIR model to simulate the death tolls in Eyam. To solve the equations for the SIR model, we use the same approach as for bacterial population growth (Chapter 3 and Chapter 4) and plant population growth (Chapter 5).

We first import the `pylab` package, set the number of weeks we want to simulate for, and the constants `a` and `b`. We do not go into how to find a and b parameters that makes the SIR model fit the data from Eyam, but it turns out that if we set $a = 0.0027$, and $b = 0.25$, we get a good fit. We simulate for 24 weeks:

```
from pylab import *

# Set parameters
N = 24      # number of weeks
a = 0.0027  # infection rate
b = 0.25    # recovery rate
```

Next, we define the arrays to store the number of individuals in each group over time:

```
# Create the arrays to store the number of individuals in each group
t = arange(N)
S = zeros(N)
I = zeros(N)
R = zeros(N)
```

Then, we set the initial conditions. In the first week there were 289 villagers, of which 282 were susceptible, and 7 were infected. This gives us the initial conditions:

```
S[0] = 282
I[0] = 7
R[0] = 0
```

We use a `for`-loop to calculate the number of individuals in each group for that week using Equations (9.9) - (9.11):

```
for n in range(1, N):
    S[n] = S[n-1] - a*S[n-1]*I[n-1]
    I[n] = I[n-1] + a*S[n-1]*I[n-1] - b*I[n-1]
    R[n] = R[n-1] + b*I[n-1]
```

Finally, we plot the results:

```
plot(t, S, "-o", label="Susceptible")
plot(t, I, "-o", label="Infected")
plot(t, R, "-o", label="Recovered")
xlabel("Week")
ylabel("Number of individuals")
legend()
show()
```

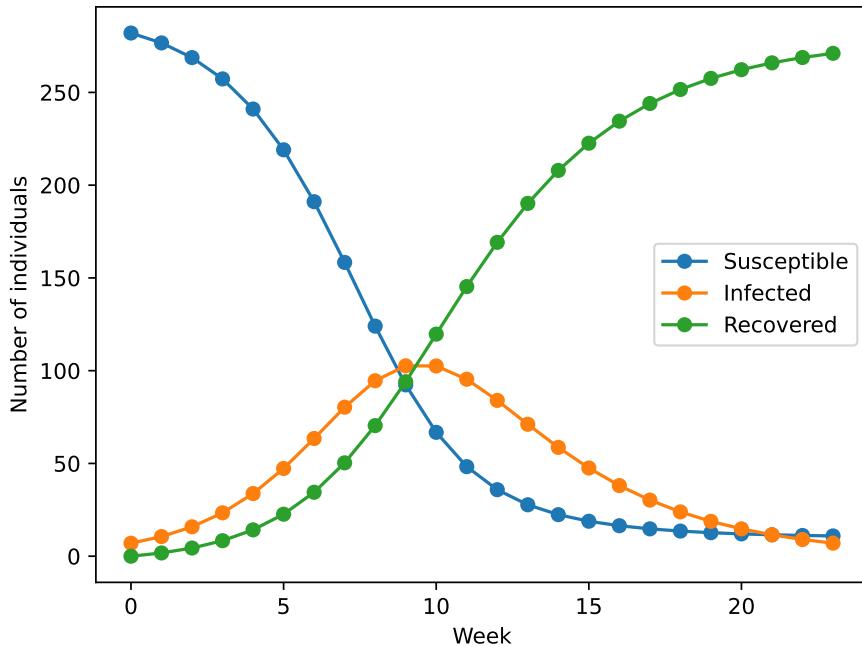


Figure 9.6: The number of individuals in each group for the SIR model

The behavior of this system can be separated into two distinct phases. At first, the disease spreads and the number of infected is rising. At some point, the number of infected peaks, and then starts to decrease. This is as expected; after a while there are almost no people left to infect, and the disease stops spreading. The peak occurs precisely when each of the infected individuals infect on average less than one susceptible person before they recover. This leads to fewer infected each week. It can be shown that this occurs when

$$S_n \frac{a}{b} < 1. \quad (9.12)$$

The case where $S_n a/b = 1$ is called the *epidemiological threshold*. If $S_n a/b > 1$, the number of infected will rise.

The SIR model highlights why vaccinations are important. In the SIR model, a vaccine moves an individual from the susceptible group to the recovered group. This makes sure the individual themselves are safe from the infection, but it has another very important effect. If enough people get the vaccine the system falls below the epidemiological threshold, since S_n is decreased. This

makes the disease die out, even if not everyone receives the vaccine. This phenomenon is called *herd immunity*.

Herd immunity against dangerous diseases is important. Some people are unable to develop immunity against certain diseases through vaccinations or cannot be vaccinated for medical reasons. Additionally, newborn infants are too young to receive many vaccines. Herd immunity keeps the people in the above groups safe, by preventing the disease from spreading in the population.

9.3.1 Calculating the death tolls from the SIR model

Let us finish this section by using the SIR model to simulate the death tolls in Eyam. We now have the number of susceptible, infected, and recovered individuals each week, but the recorded data is the number of deaths per week. The bubonic plague ended in death in as much as 90% of cases (keep in mind antibiotics were not discovered until 1928). To compare the SIR model to the recorded death tolls, we assume that 90% of the individuals that enter the recovered group dies.

The number of individuals that enters the recovered group each week is the difference between the number of recovered individuals the current week compared to the previous. To calculate this we use the `diff()` function. `diff()` takes an array as argument and returns an array with the difference between every consecutive pair of values:

```
d = array([4, 8, 10, 11])
print(diff(d)) # the result is [d[1]-d[0], d[2]-d[1], d[3]-d[2]]
```

[4 2 1]

Note that the result has one less element than the original array. The death toll is 90% of the people who enter the recovered group:

```
death_toll_simulated = 0.9*diff(R)
```

To get the weeks corresponding to each death toll we exclude the last week:

```
t_simulated = t[0:N-1]
```

We load the historical data and plot the simulated data together with the historical data:

```
data = pandas.read_csv("eyam.csv")
t_historical = list(data["t"])
death_toll_historical = list(data["D"])

plot(t_historical, death_toll_historical, "-o", label="Real death toll")
plot(t_simulated, death_toll_simulated, "-o", label="Simulated death toll")
xlabel("Week")
ylabel("Death toll per week")
legend()

show()
```

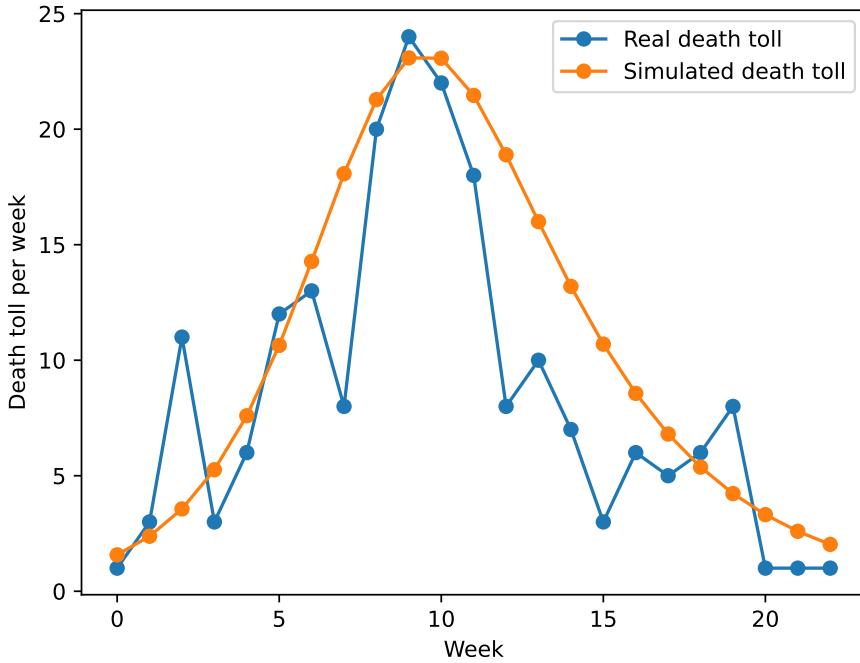


Figure 9.7: A comparison of the SIR model to the death tolls in Eyam village during the plague in 1666.

The recorded death toll appears more random than the simulated death tolls, but overall the SIR model is able to explain the evolution of the disease. If we count the total number of deaths during the 23 weeks, we notice that the simulated death toll is greater than the recorded death toll:

```
print("recorded death toll:", sum(death_toll_historical))
print("simulated death toll:", sum(death_toll_simulated))
```

```
| recorded death toll: 197
| simulated death toll: 243.95529102949172
```

In order to understand the discrepancy, we should consider the assumptions of the model. Among other things, we assumed that any infected individual is equally likely to infect any susceptible individual. This is unrealistic, as you normally have to be nearby, or even in physical contact with, an infected individual in order to catch the disease. In Eyam the villagers could avoid contact with the infected. This is why we use quarantines to stop the spread of infectious diseases today. Clearly, the spatial spread of the disease matters. In the next section, we introduce a more detailed approach which models every individual of the population, and takes their spatial location into account.

9.4 Disease on a grid: a spatial SIR model

In the rest of the chapter, we focus on the SIR model with space taken into account. We start by developing the basic tools we need to make such a model. Afterwards, we present an already finished Python module, called `epidemics`, for performing the simulations.

9.4.1 Working on a spatial grid

Contrary to what we did in the previous section, where only modeled the total population, here we model each individual by itself. We allow each individual to be either susceptible S , infected I , or recovered R . Working with these three groups as letters is problematic when we want to visualize the results, so we represent each group by an integer:

```
SUSCEPTIBLE = 1
INFECTED = 2
RECOVERED = 3
```

We model each individuals as fixed in one position. Each infected individual may only transmit the disease to their immediate neighbors, as that is the only people they are in contact with. Let us start with a small grid of people, with 8 rows and 6 columns. Each square in the grid is a person. We store these in a two-dimensional array. To create a two-dimensional array, we give the `zeros()` function a list with the number of rows and columns:

```
from pylab import *
rows = 8
columns = 6
grid = zeros([rows, columns])
print(grid)
```

```
[[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. 0. 0. 0.]]
```

We first set all individuals in the grid to susceptible. A two-dimensional array is indexed similar to how we indexed a Pandas `DataFrame`, access the elements array using square brackets. To get all elements we write:

```
grid[:, :] = SUSCEPTIBLE
print(grid)
```

```
[[1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1.]]
```

```
[1. 1. 1. 1. 1. 1.]  
[1. 1. 1. 1. 1. 1.]  
[1. 1. 1. 1. 1. 1.]  
[1. 1. 1. 1. 1. 1.]  
[1. 1. 1. 1. 1. 1.]
```

We access all elements in both dimension, similar to indexing all elements in one dimension with `[:]`. To access a single individual we use the row and column, here setting the individual in row 4 and column 3 to INFECTED (counting from zero):

```
grid[4, 3] = INFECTED  
print(grid)
```

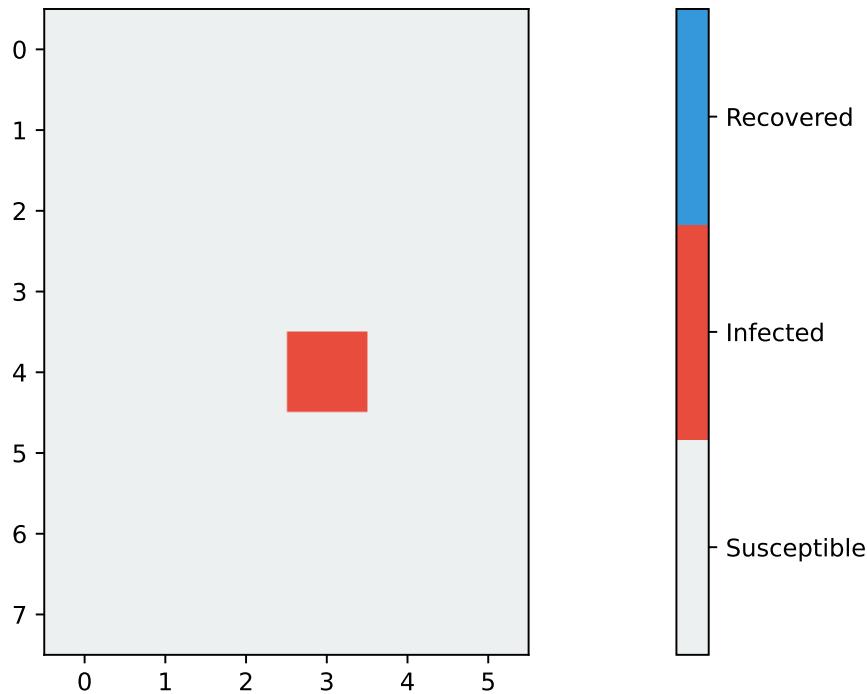
```
[[1. 1. 1. 1. 1. 1.]  
 [1. 1. 1. 1. 1. 1.]  
 [1. 1. 1. 1. 1. 1.]  
 [1. 1. 1. 1. 1. 1.]  
 [1. 1. 1. 2. 1. 1.]  
 [1. 1. 1. 1. 1. 1.]  
 [1. 1. 1. 1. 1. 1.]  
 [1. 1. 1. 1. 1. 1.]
```

The initial infected individual in a disease outbreak is often called *patient zero*.

We have added a plotting function, named `plot_SIR_grid()` to the `epidemics` package, which we use to better visualize the grid:

```
import epidemics  
  
epidemics.plot_SIR_grid(grid)  
show()
```

```
/Users/alexajo/anaconda3/envs/doconce_latest/lib/python3.10/site-packages/scipy/_init_.py:146:  
    UserWarning: A NumPy version >=1.16.5 and <1.23.0 is required for this version of SciPy  
    (detected version 1.23.1  
    warnings.warn(f"A NumPy version >={np_minversion} and <{np_maxversion}"
```



As you can see from the figure, there is one infected individual, while the rest are susceptible.

9.4.2 Rules of the spatial SIR model

We introduce the following rules for the spatial SIR model:

1. An infected individual can only infect the individuals that are closest to it on each side, and only if the neighbors are susceptible. This means that our patient zero can only infect individual [3, 3], [5, 3], [4, 2] or [4, 4] (see Figure 9.8). We use the position of each individual on the grid to distinguish them.
2. There is no guarantee that an infected individual infects its neighbors. An infected individual has a probability α of infecting a susceptible neighbor during a single time step.
3. For each time step, each infected individual can recover (die or become healthy, as in the last section) with probability β .

Here we have introduced the Greek letters α (alpha) and β (beta) for the probabilities. This is because they are related to the parameters a and b from the previous section, but they are not exactly the same.

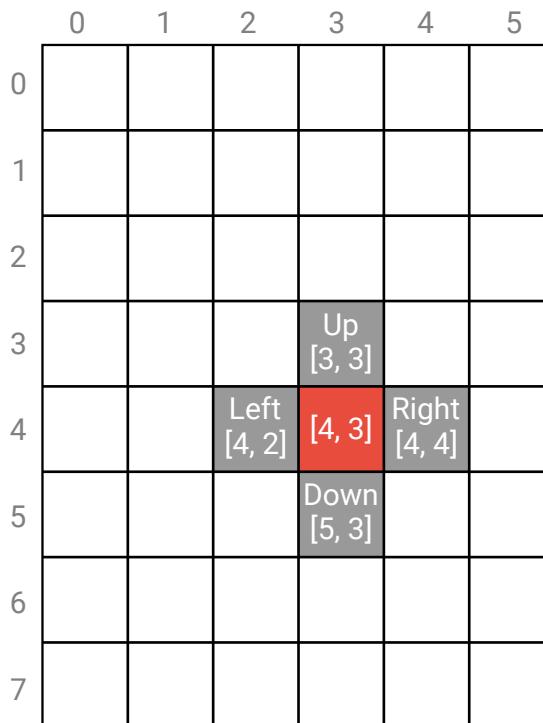


Figure 9.8: An infected individual can only infect the individuals that are closest to it on each side.

9.4.3 Using random numbers to represent probabilities

We have explored the concept of randomness in programs in Chapter 6 on Mendelian genetics. Here, we go a step further. If the probability that an infected person infects a susceptible neighbor is 20%, α is 0.2.

```
alpha = 0.2
```

In order to make the infected person have a 20% chance of infecting a susceptible neighbor, we use the `random()` function, which is a part of the `pylab` module. `random()` returns a number between 0 and 1, but it does not return the same number each time! The `random()` function is designed such that every number on the interval between 0 and 1 is equally likely to be returned, and there are no connection between one call of the function and the next. We can pick a random number, and test if it is lower or higher than α :

```
my_random_number = random()
print(my_random_number)

if my_random_number < alpha:
    print("Hello, world!")
else:
    print("Goodbye, world!")
```

```
0.11505456638977896
Hello, world!
```

Try to run this program a few times. Notice that `my_random_number` is changing each run, and that, on average, `Hello, world!` is printed 20% of the time, while `Goodbye, world!` is printed the other 80% of the time.

With this tool, we are ready to start working on the simulation. We go through each individual, one by one, and test if they are infected. We create a double `for` loop to go through all the rows and columns. For each individual, we check if it is infected. If yes, we print a message:

```
for i in range(rows):
    for j in range(columns):
        if grid[i, j] == INFECTED:
            print("Patient at", i, j, "is infected!")
```

```
Patient at 4 3 is infected!
```

We just found the individual that we set to infected in the initial grid!

Next, we need to consider if the infected individual will transfer the disease to its neighbors. To do this we require a Python concept called *tuples*.

9.4.4 Tuples: unchangeable lists

A tuple is a special type of list in Python. We create a tuple by using `()` (regular parentheses). This is different from lists, which are created with `[]` (square brackets):

```
a_tuple = (1, 2, 3)
print(a_tuple)
```

```
(1, 2, 3)
```

We index tuples similar to how we index lists:

```
print(a_tuple[1])
```

```
2
```

In contrast to regular lists, tuples cannot be changed once they have been created. If we try to change an element in a tuple you get the following error:

```
a_tuple[1] = 4
```

```
-----
TypeError                                Traceback (most recent call last)
Input In [441], in <cell line: 1>()
----> 1 a_tuple[1] = 4

TypeError: 'tuple' object does not support item assignment
```

This error tells us that we are not allowed to assign new values to the elements of a tuple. In addition, you cannot add or remove elements from a tuple.

9.4.5 Finding the neighbors

Each individual has neighbors in four directions: up, down, left, and right. We find the index for each of these directions by adding or subtracting 1 from the current value of *i* or *j*. We first illustrate the following for the infected individual, before we insert the code back into the above for loop, where *i* and *j* varies:

```
i = 4
j = 3

left = (i, j - 1)
right = (i, j + 1)
up = (i - 1, j)
down = (i + 1, j)

print("left:", left)
print("right:", right)
print("up:", up)
print("down:", down)
```

```
left: (4, 2)
right: (4, 4)
up: (3, 3)
down: (5, 3)
```

We store the indices as tuples, and not as lists. The reason for this is that we can index our two-dimensional grid with a tuple with two elements to get a specific individual

```
print(grid[(4, 3)])
```

```
2.0
```

With a list we instead get two rows, row 4 and row 3:

```
print(grid[[4, 3]])
```

```
[[1. 1. 1. 2. 1. 1.]
 [1. 1. 1. 1. 1. 1.]]
```

By putting the neighbor indices in a list, we can loop over them and test if they are susceptible:

```
neighbors = [left, right, up, down]
for neighbor in neighbors:
    if grid[neighbor] == SUSCEPTIBLE:
        print("Neighbor", neighbor, "is susceptible!")
```

```

Neighbor (4, 2) is susceptible!
Neighbor (4, 4) is susceptible!
Neighbor (3, 3) is susceptible!
Neighbor (5, 3) is susceptible!

```

For each susceptible neighbor, we want to infect them with probability α :

```

neighbors = [left, right, up, down]
for neighbor in neighbors:
    if grid[neighbor] == SUSCEPTIBLE and random() < alpha:
        print("Neighbor", neighbor, "is susceptible and should be infected!")

```

```

Neighbor (4, 4) is susceptible and should be infected!
Neighbor (5, 3) is susceptible and should be infected!

```

Finally, we want the infected individual to recover with probability $\beta = 0.15$, before moving on to the next individual:

```

beta = 0.15
if random() < beta:
    print("The current individual should recover!")

```

```

The current individual should recover!

```

9.4.6 Special behavior on the boundary

What happens if individual [4, 5], along the rightmost border, is infected? The program will try to access individual [4, 6], which should be to the right of individual [4, 5], but that is outside the grid array, and the program will crash.

We therefore need to treat the edges of the domain with special care, and decide upon a *boundary conditions*. We either have to assume that there are no individuals outside the grid, or we can add an extra set of recovered individuals along the edges. The latter option is called a *ghost boundary condition*, and is what we use.

This is implemented by not iterating over the individuals on the edges, which we do by modifying the loops over the rows and columns to start on 1 and subtract 1 from the stop index:

```

for i in range(1, rows - 1):
    for j in range(1, columns - 1):
        # check if infected, if so, check neighbors, and so on.

```

9.4.7 A complete spatial SIR model

Let us create a function called `time_step()` that takes the current grid and returns the grid in the next time step, summarizing all the steps we made above. We pass `time_step()` the `grid`, `alpha`, and `beta` as arguments. To get the number of rows and columns of a two-dimensional array, we use `.shape`. `grid.shape` is the number of rows and columns in the grid array as a

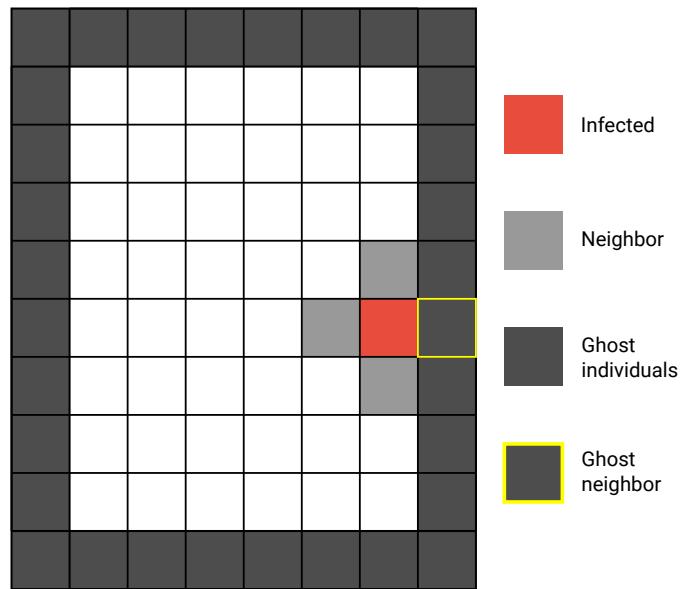


Figure 9.9: An illustration of ghost boundary condition.

tuple. Indexing `grid.shape` we get the rows and columns respectively, `rows = grid.shape[0]`, and `columns = grid.shape[1]`. In order to update the grid, we first copy the previous grid using the `copy()` function. By doing that we avoid comparing and updating the numbers at the same time inside the loops, which would give wrong result. Then we combine all the steps outlined above:

```
def time_step(previous_grid, alpha, beta):
    rows = previous_grid.shape[0]
    columns = previous_grid.shape[1]

    # copy the previous grid
    current_grid = previous_grid.copy()

    # loop over individuals
    for i in range(1, rows - 1):
        for j in range(1, columns - 1):
            left = (i, j - 1)
            right = (i, j + 1)
            up = (i - 1, j)
            down = (i + 1, j)

            if previous_grid[i, j] == INFECTED:          # check if individual is infected
                for neighbor in [left, right, up, down]: # check neighbors
                    if previous_grid[neighbor] == SUSCEPTIBLE and random() < alpha:
                        current_grid[neighbor] = INFECTED # infect the neighbor

            if random() < beta:                      # check if individual has recovered
                current_grid[i, j] = RECOVERED

    return current_grid
```

In the code below we use this function to perform 50 time steps, with $\alpha = 0.2$ and $\beta = 0.15$, and store the grid for every time step. Due to the ghost individuals we increase the size of the grid with one in each direction (two in total) from [8, 6] to [10, 8] to get a similar size as before.

```
from pylab import *
import epidemics

grid = zeros([10, 8])

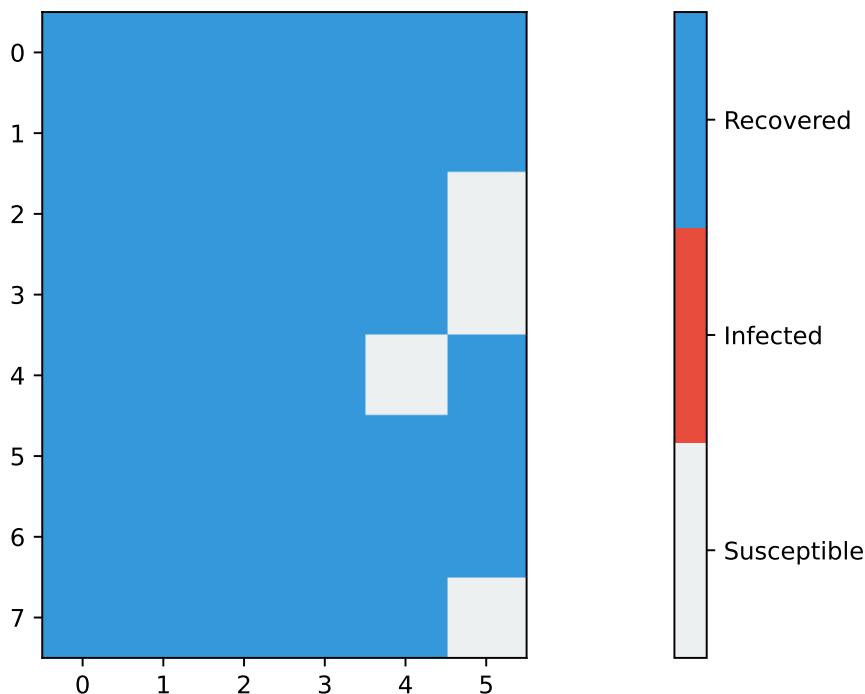
grid[:, :] = SUSCEPTIBLE
grid[4, 3] = INFECTED

grids = []
grids.append(grid)

for n in range(50):
    grid = time_step(grid, alpha=0.2, beta=0.15)
    grids.append(grid)
```

Let us plot the final grid without the boundary, as the boundary is not of interest and we ignore it in our calculations:

```
epidemics.plot_SIR_grid(grids[-1], without_boundary=True)
show()
```



We urge the reader to run the code a few times and inspect the outcome using the `plot_SIR_grid()` function. If we increase α , making the disease more contagious, we might see that the end result is everyone becoming infected.

The following lines create an animated video, which show up in a notebook.

```
import IPython
anim = epidemics.create_animation(grids)
IPython.display.HTML(anim.to_html5_video())
```

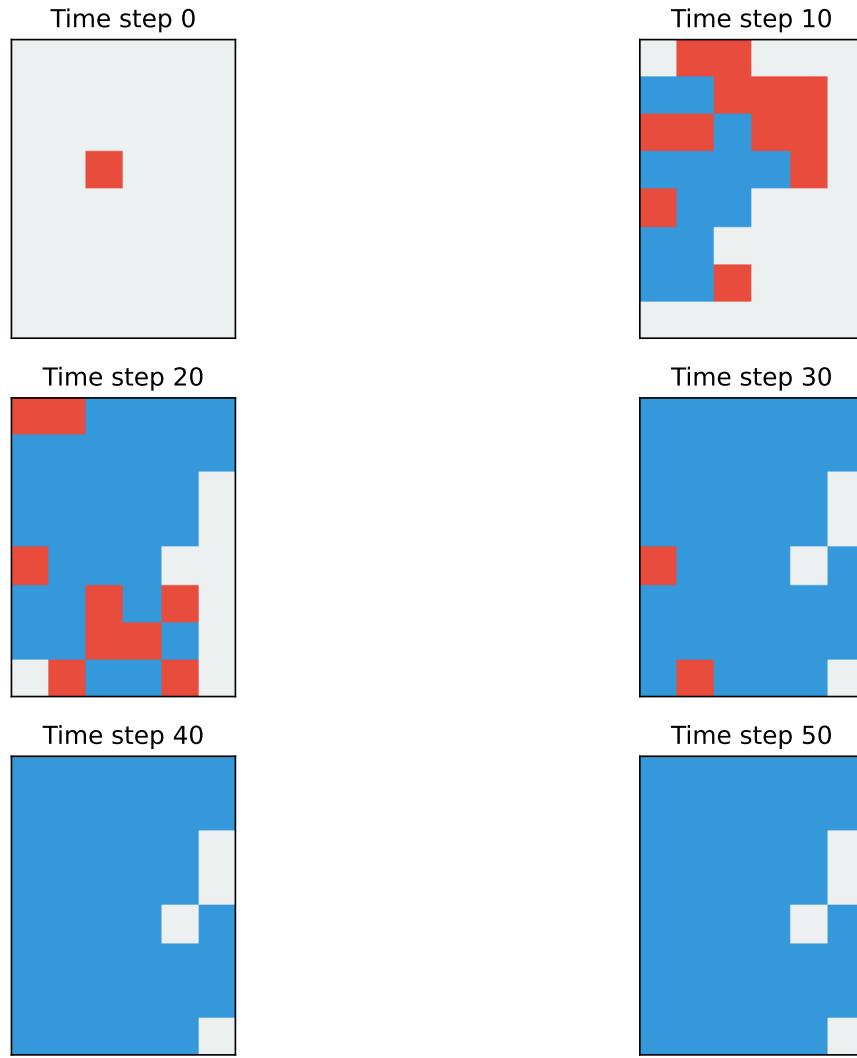
If you are not using a notebook, or if you want to save the simulation to a separate file, you can instead write:

```
filename = 'animation.mp4'
anim = epidemics.create_animation(grids)
epidemics.save_animation_to_file(anim, filename)
```

which creates a video file in your current working directory.

Additionally we have a function that shows a series of snapshots from the animation, which is used in the PDF:

```
epidemics.plot_snapshots(grids)
show()
```



We now have everything we need to perform larger simulations of how a disease spreads.

9.5 Exploring the spatial SIR model

Let us simulate the spread of a disease on a 50×50 grid over 200 time steps, with $\alpha = 0.2$ and $\beta = 0.15$. Our grid have to be 52×52 to accommodate the ghost individuals along the border.

```
time_steps = 200
alpha = 0.2
beta = 0.15

grid = zeros([52, 52])
```

We initialize the disease by infecting a group of individuals in the center of the grid, by slicing the grid in both dimensions at the same time. we slice from index 22 to 28 in the rows and from index 22 to 28 in the columns.

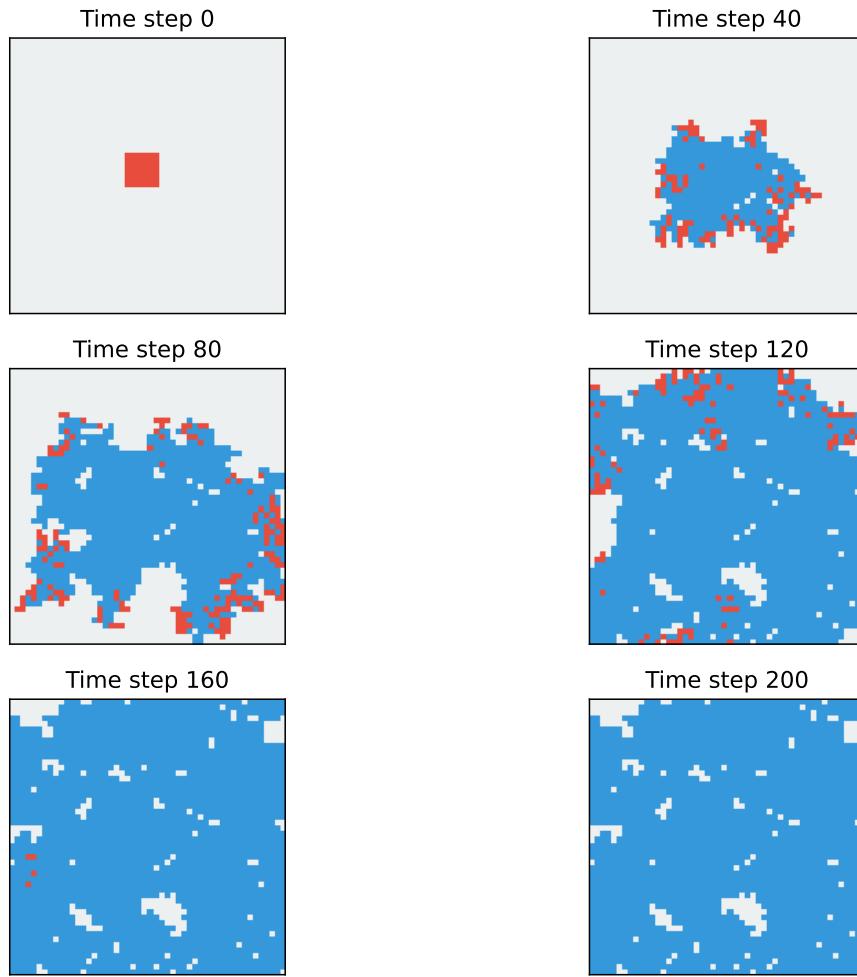
```
grid[:, :] = SUSCEPTIBLE
grid[22:28, 22:28] = INFECTED
```

We then run the simulation.

```
grids = []
grids.append(grid)
for n in range(time_steps):
    grid = time_step(grid, alpha, beta)
    grids.append(grid)
```

Finally, we plot snapshots from the simulation:

```
anim = epidemics.plot_snapshots(grids)
```



The disease quickly spreads to the edges of the grid, but eventually dies out completely. Just like in the population model, not everyone gets infected. This is expected, but the spatial distribution of the uninfected individuals is interesting. The individuals who never get infected are left in small susceptible “islands” inside large areas of recovered individuals. These can never be reached by the disease and stay susceptible throughout the simulation. These islands are often larger than a single individual.

We can explain these groups of uninfected individuals by considering an example. A susceptible individual, “Adam”, may be next to an infected individual, “Brian”. However, Adam is lucky, and he avoids infection for long enough time so that Brian recovers. Adam has another neighbor, “Charlotte”. Charlotte is now also less likely to get infected, because Adam avoided getting infected by Brian. This is why these groups of unaffected individuals tend to clump together.

Animations help us understand how the disease spreads, but it is also useful to plot the total populations of each group of individuals, as we did in the first section in this chapter. We create a set of arrays to store the population count for each time step:

```
S = zeros(time_steps)
I = zeros(time_steps)
R = zeros(time_steps)
```

For each time step, we loop over all individuals in the grid and count the number of individuals in each group (remember to avoid counting the boundary):

```
for n in range(time_steps):
    grid = grids[n]

    rows = grid.shape[0]
    columns = grid.shape[1]

    for i in range(1, rows - 1):
        for j in range(1, columns - 1):
            if grid[i, j] == SUSCEPTIBLE:
                S[n] += 1
            if grid[i, j] == INFECTED:
                I[n] += 1
            if grid[i, j] == RECOVERED:
                R[n] += 1
```

Finally, we plot the results:

```
plot(S, label="Susceptible")
plot(I, label="Infected")
plot(R, label="Recovered")
xlabel("Time step")
ylabel("Population count")
show()
```

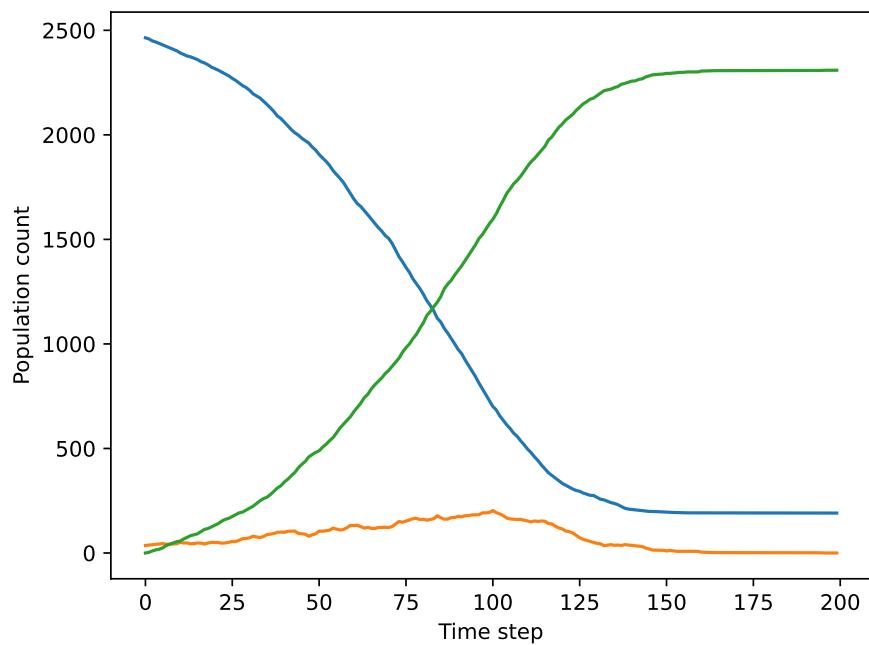


Figure 9.10: Population counts for the spatial SIR simulation.

If we compare Figure 9.10 to Figure 9.6, we see that there are similarities. In both cases the number of infected initially rises, until it reaches a top. This is the point where each infected individual on average infects less than one susceptible. However in our simulation the number of infected stays small for the entire simulation. This is because once an individual has been infected, it quickly recovers.

Try playing around with different parameters to see how the simulation changes. If we decrease β we increase the time an individual spends as infected, and the peak in the infected population is increased. An interesting side effect is that this leads to much fewer pockets of susceptible individuals, which is because the infected stay infected longer, and therefore have an increased chance of infecting all neighbors.

9.6 The effect of vaccinations

We can use the spatial SIR model to better understand how to fight infectious diseases. To do this, we introduce a new group in our model, V - vaccinated. A vaccinated individual behaves exactly like a recovered individual, they cannot get the disease, and therefore cannot spread it. However, since we want to examine the importance of vaccines, it is useful to keep them as a separate group. In the following example, we run the same simulation as above, but we vaccinate 10% of the individuals at the beginning.

First, we set up our grid, and create a variable to represent vaccinated individuals:

```
VACCINATED = 4

time_steps = 200
alpha = 0.2
beta = 0.15

grid = zeros([52, 52])
grid[:, :] = SUSCEPTIBLE
grid[22:28, 22:28] = INFECTED
```

Next, we find the number of individuals to vaccinate:

```
individuals_count = 52*52 # total number of individuals
vaccination_count = int(0.1*individuals_count) # vaccinate 10 percent
```

Then we loop through all these vaccinations and pick a random individual in our grid. We pick this individual with the `randint()` function. This function takes two arguments, `randint(low, high)`, and returns a random integer from `low` and up to, but not including `high`. We use this to pick a random positions in our grid, excluding the boundary. If the individual is susceptible, we set it to vaccinated instead.

```
for k in range(vaccination_count):
    i = randint(1, rows-1)
    j = randint(1, columns-1)
```

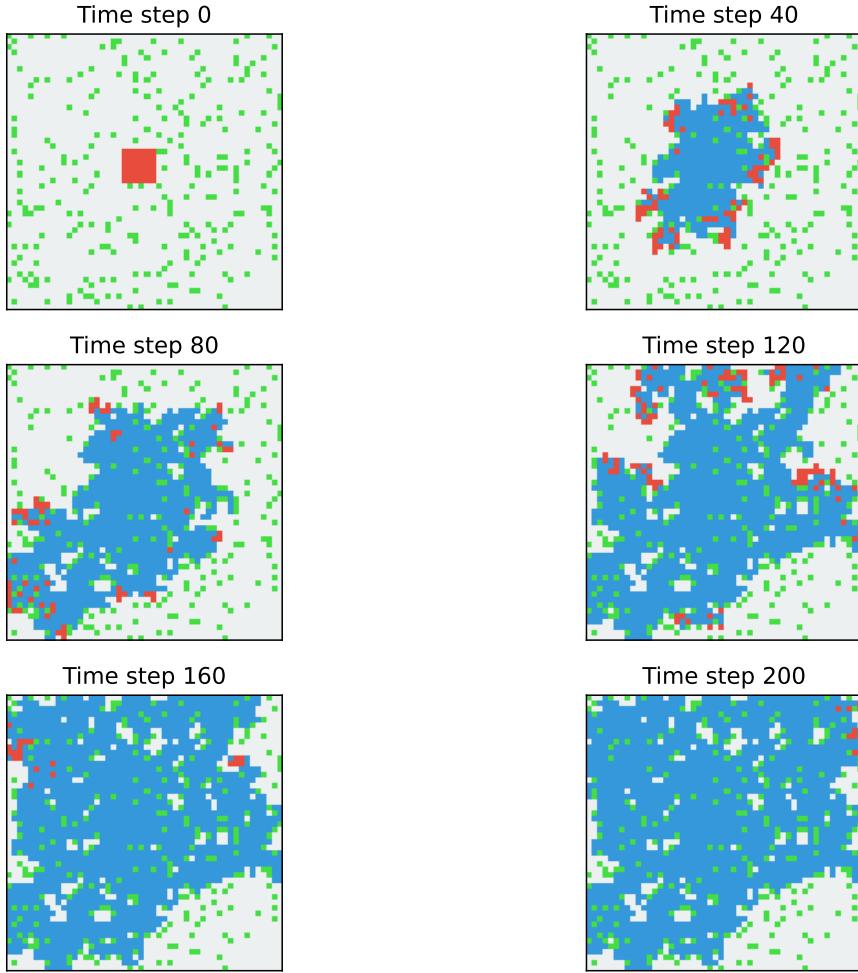
```
if grid[i, j] == SUSCEPTIBLE:  
    grid[i, j] = VACCINATED
```

This is not a perfect implementation, because we might vaccinate the same person twice, but it is close to 10% vaccination. We run the simulation as before:

```
grids = []  
grids.append(grid)  
  
for n in range(time_steps):  
    grid = time_step(grid, alpha, beta)  
    grids.append(grid)
```

Finally, we plot snapshots from the simulation (green is used to represent vaccinated individuals):

```
epidemics.plot_snapshots(grids, vaccinated=True)  
show()
```



In this particular case we saved the bottom right corner of our grid by vaccinating just 10% of the population. This shows the effect of herd immunity; vaccinated individuals protect susceptible individuals. It is important to remember that this changes every time you run the simulation, due to the random numbers in our code. In the next section, you are challenged to stop a pandemic from spreading across the world using vaccines.

9.7 Pandemics: Disease on a world map

In this section we want to examine how pandemics spread across a world map. In order to do this we introduce the `pandemics` module. It works almost like the `epidemics` module, but the disease now spreads on a grid resembling a world map. We also introduce two new groups:

- *D* - Dead. This is used to mark infected individuals that die. From a theoretical point of view, there is no reason to split these from the recovered individuals, as they have the

same properties in our simulation. However, it makes the simulations more interesting. In the simulations, we represent these by a gray color.

- U - Unavailable. This is used to mark uninhabited areas such as oceans. Again, this group has the same properties as the R group, we include it to make the simulations cool. This allows us to relate how the disease spreads to how real epidemics have spread across the planet, such as the Black Death and the Spanish Flu. In the simulations we represent these by a dark blue color.

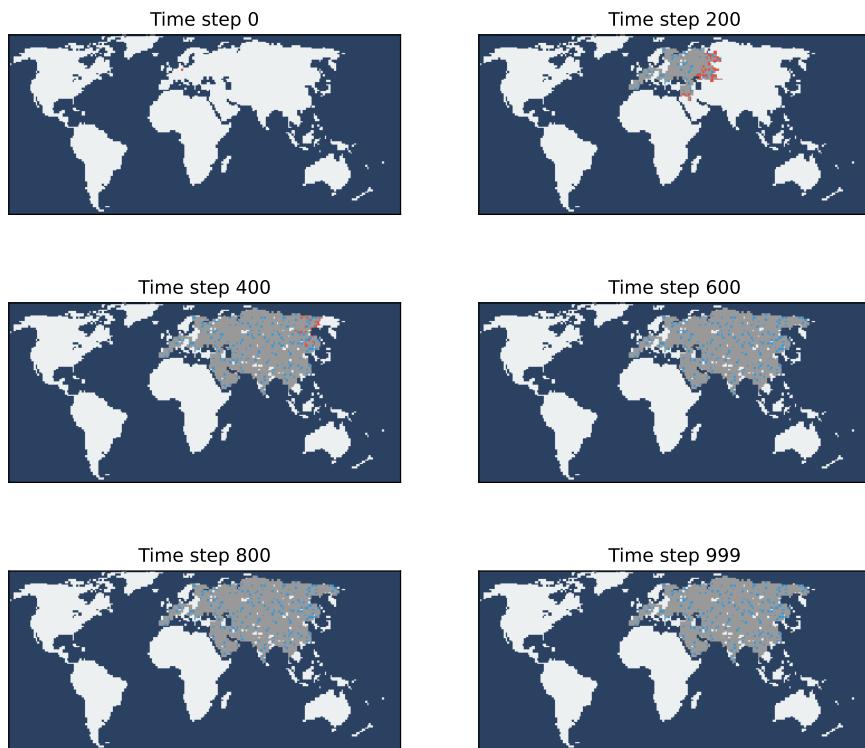
When we introduce a world map as our grid, instead of modeling each position as an individual, we assume that each position in the map is a large group of individuals living in the same area. We call each position on the map a site. However, everyone in this area are in the same state. In other words, if a site is SUSCEPTIBLE, all people on this site are susceptible, and equivalently if the position is INFECTED, all people on this site are infected.

Let us begin by simply running the simulation with the pandemics module. The following code runs our pandemic simulator and plots the results:

```
from pylab import *
import pandemics

grids = pandemics.simulate()

epidemics.plot_snapshots(grids, pandemic=True)
show()
```



The disease starts in Central Europe before it spreads across most of the continent and into Asia. For each time step, any infected site has a 1% chance of recovery and a 9% chance of dying. In the remaining 90% of the cases, the infection remains in the given site. It is quite clear that this pandemic produces devastating results, and leaves a long trail of death wherever it spreads.

Let us examine some different scenarios for how to stop the pandemic from spreading too far.

9.7.1 A naive vaccination strategy

The above simulation runs until there are no more infected sites on the grid. We are left with many dead sites and a few that recovered on their own. One of the most useful tools we have to stop a pandemic is to develop a vaccine against the disease, and then distribute this vaccine around the world.

We want to reduce the number of sites that get infected during the disease spread, and examine what is the best strategy for distributing the vaccine. To do this with the `pandemics` module, we implement a function called `distribute_vaccines` that can be passed to the `pandemics.simulate` function as the first argument. In this scenario it takes 50 time steps for the vaccine to be developed and distributed, and the `distribute_vaccines()` function is called by the simulation after 50 time steps. This means that the disease spreads freely for 50 time steps, and then we have to try to stop it by distributing vaccines to different areas of the world.

The `distribute_vaccines` function receives the current grid as input and must return a two-dimensional array of sites to vaccinate. This is done by setting all values in the map to 0 except a number of sites that are set to 1 to indicate that these are to be vaccinated.

In this first trial, we randomly distribute vaccines to different areas of the world. We pick 2000 random sites to vaccinate and hope that this is enough to stop the disease from spreading further. The strategy is implemented in the following code, and the result shown below:

```
def distribute_vaccines(current_map):
    # Create the vaccination map with the same shape as the world map
    rows = current_map.shape[0]
    columns = current_map.shape[1]

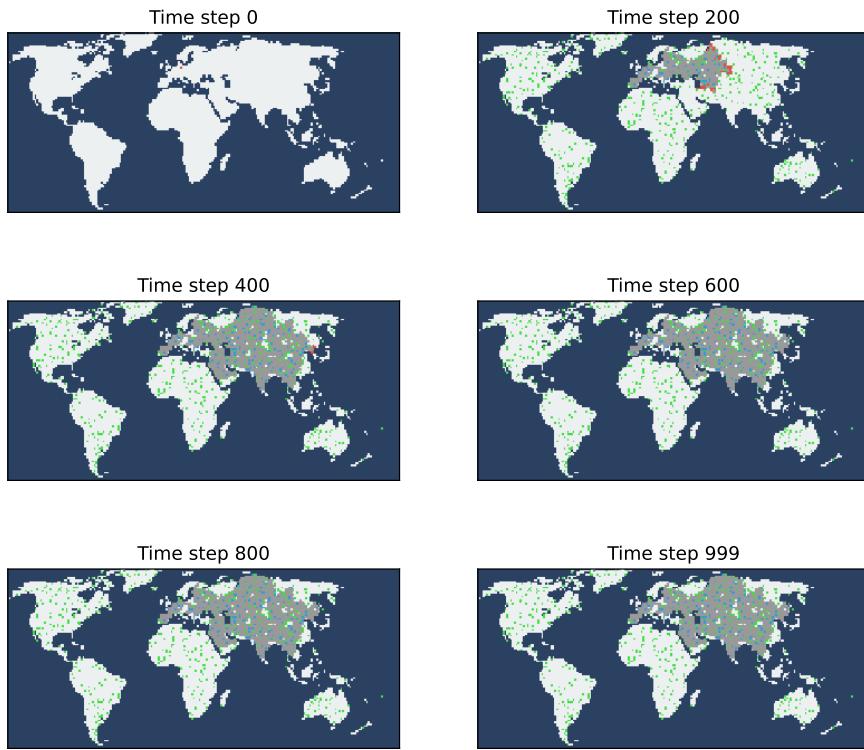
    vaccination_map = zeros([rows, columns])

    # Randomly chose sites to vaccinate
    for k in range(2000):
        i = randint(0, rows)
        j = randint(0, columns)

        vaccination_map[i, j] = 1

    return vaccination_map

grids = pandemics.simulate(distribute_vaccines)
epidemics.plot_snapshots(grids, pandemic=True)
show()
```



This strategy will in most cases stop the disease from spreading too far, but it requires many vaccines. This result is typical when a large portion of the population can be vaccinated. Because we chose the site randomly from the whole map, some of the sites we try to vaccinate are in the middle of the ocean. This means that we do not actually distribute 2000 vaccines. We ignore this detail.

9.7.2 A limited number of vaccines

In most cases when a pandemic occurs there just a short time to find and create a vaccine. The number of vaccines that can be distributed is therefore most often limited. We assume that we only have enough vaccines to vaccinate 300 sites. We have made the `simulate` function in such a way that it can simulate many different scenarios. The scenario is selected by passing the argument `level` to the function. The scenario which simulates a limited vaccine count is selected by setting `level=1`. We therefore have to adjust the number of vaccines to 300:

```
def distribute_vaccines(current_map):
    # Create the vaccination map with the same shape as the world map
    rows = current_map.shape[0]
    columns = current_map.shape[1]

    vaccination_map = zeros([rows, columns])

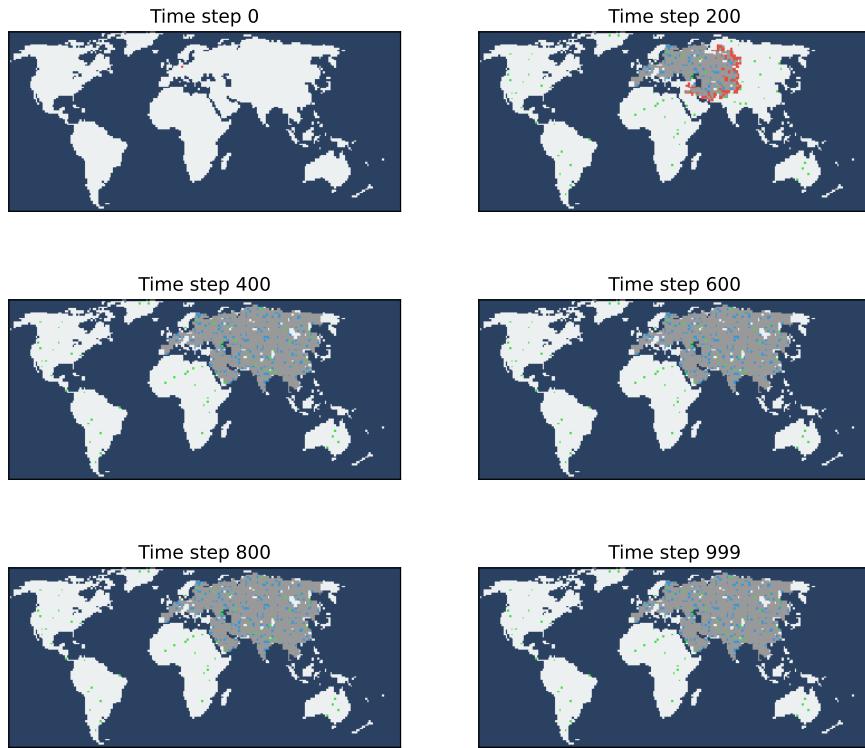
    # Randomly chose sites to vaccinate
    for k in range(300):
        i = randint(1, rows - 1)
```

```
j = randint(1, columns - 1)

vaccination_map[i, j] = 1

return vaccination_map

grids = pandemics.simulate(distribute_vaccines, level=1)
epidemics.plot_snapshots(grids, pandemic=True)
show()
```



We see that our strategy of randomly distributing vaccines is unable to stop the pandemic, if we have a limited number of vaccines. The disease spreads to almost all of Europe and Asia. Try to run the code a couple of times to verify that this happens almost every time.

We have to find a better strategy. Before reading on, try to come up with a strategy to stop the disease. You can change the `distribute_vaccines` function above and see if your strategy works. Just remember that when calling `simulate` with `level=1`, you cannot vaccinate more than 300 sites.

A simple strategy we can apply in this case is to draw a line of vaccinated sites, to prevent the disease from spreading further. We know that the epidemic starts to spread from Europe and if we draw a line between Europe and Asia, we might be able to stop the disease from spreading. Because our map only has 100 sites from top to bottom, this is a strategy that works when we only have developed 300 vaccines. Since there also are many sites that are unavailable because they are ocean and not land, we likely spend less than 100 vaccines.

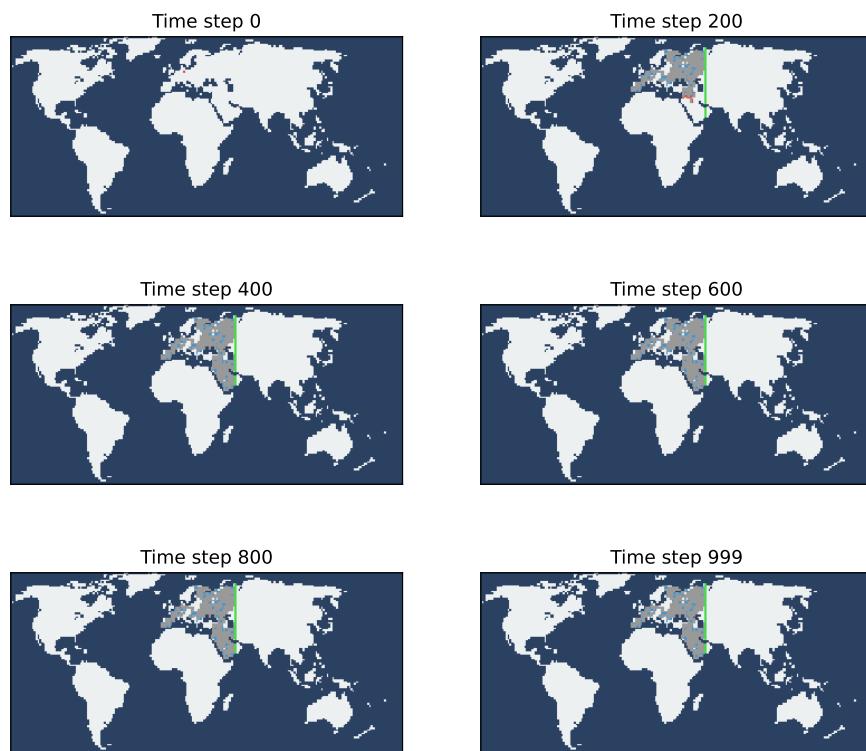
The following code implements the strategy of drawing a line of vaccinated sites down column 123:

```
def distribute_vaccines(current_map):
    rows = current_map.shape[0]
    columns = current_map.shape[1]

    vaccination_map = zeros([rows, columns])
    vaccination_map[:, 123] = 1

    return vaccination_map

grids = pandemics.simulate(distribute_vaccines, level=1)
epidemics.plot_snapshots(grids, pandemic=True)
show()
```



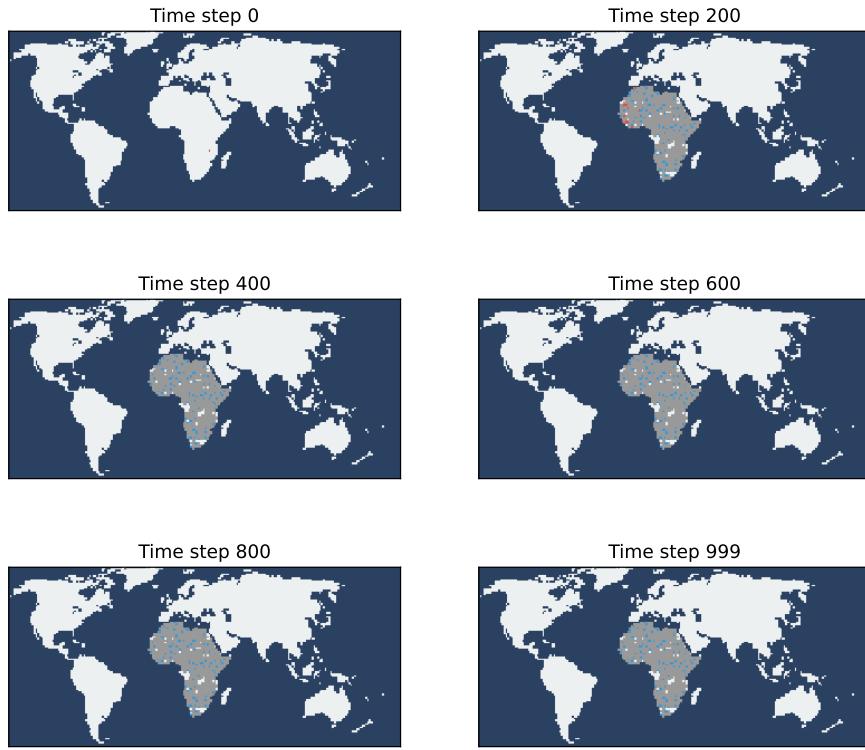
Column 123 is just about where the Ural Mountains are located, which is a typical reference used to divide Europe and Asia. This line of vaccinated sites prevents the disease from spreading further and protects everyone to the right of the line.

9.7.3 Random patient zero

One large assumption in the above strategy was that the disease always started in the middle of Europe. In a real pandemic, patient zero may appear anywhere in the world. By setting `level=2`

the initial location of patient zero becomes random. Here you see how the simulation chooses a random starting point and spreads without any vaccination strategy:

```
grids = pandemics.simulate(level=2)
epidemics.plot_snapshots(grids, pandemic=True)
show()
```



In reality we always vaccinate around where the disease is located. Try to come up with way to vaccinate in our simulation in this case. Remember that you may use the information contained in the `current_map` parameter that is received by the `distribute_vaccines` function. This is a matrix representing the map of the world. Each value in the matrix is one of the following

- 0 - `pandemics.UNAVAILABLE`
- 1 - `pandemics.SUSCEPTIBLE`
- 2 - `pandemics.INFECTED`
- 3 - `pandemics.RECOVERED`
- 4 - `pandemics.VACCINATED`
- 5 - `pandemics.DEAD`

Try to implement a version of `distribute_vaccines` that uses this information to vaccinate strategic sites.

One strategy is to draw the vaccine line close to the infection, rather than at a fixed position. To do this, we inspect the `current_map` argument we receive in the `distribute_vaccines` function. The goal is to find the rightmost and leftmost column that are infected so that we vaccinate just outside these two columns. We therefore need to find the highest and lowest column number that is infected.

We loop over all the sites in the system to find the indices of columns with infected sites. We use `min()` and `max()` to find the lowest and highest column number and store these in `infected_minimum` and `infected_maximum`, respectively. Finally, we set the column to the left of the `infected_minimum` and to the right of `infected_maximum` to vaccinated. To do this, we subtract 1 from `infected_minimum` to find the one to the left and add 1 to `infected_maximum` to find the one to the right.

The result can be seen by running the code below:

```
def distribute_vaccines(current_map):
    vaccination_map = zeros(current_map.shape)

    rows = current_map.shape[0]
    columns = current_map.shape[1]

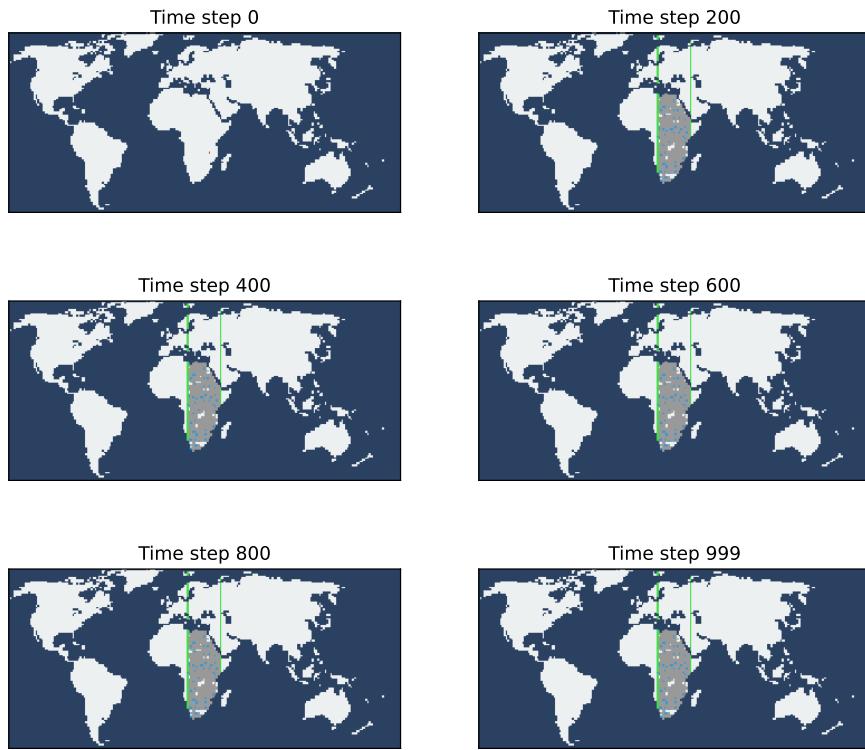
    # Find all columns that are infected
    infected_columns = []
    for i in range(1, rows - 1):
        for j in range(1, columns - 1):
            if current_map[i, j] == pandemics.INFECTED:
                infected_columns.append(j)

    # Locate leftmost and rightmost infected columns
    infected_minimum = min(infected_columns)
    infected_maximum = max(infected_columns)

    # Vaccinate to the left and right of the above columns
    vaccination_map[:, infected_minimum - 1] = 1
    vaccination_map[:, infected_maximum + 1] = 1

    return vaccination_map

grids = pandemics.simulate(distribute_vaccines, level=2)
epidemics.plot_snapshots(grids, pandemic=True)
show()
```



In this simulation, we see that the disease spreads for some time, but is severely inhibited by our vaccination strategy. While it may continue to spread along the columns, that is in the north-south direction, it is no longer able to spread along the rows.

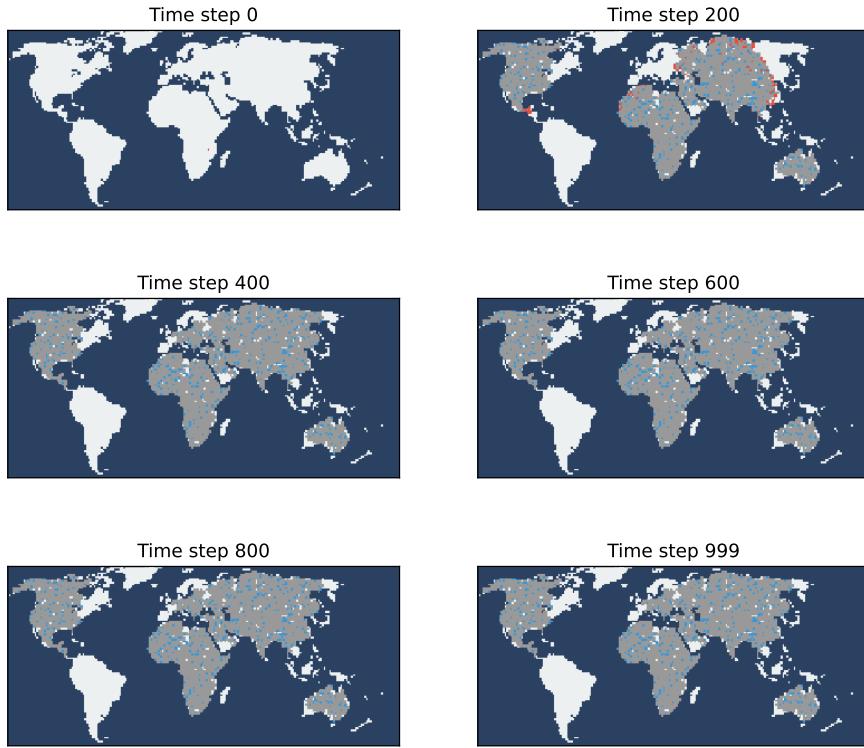
Try to write a version of this strategy that also keeps the disease from spreading further north and south. This is done by vaccinating two rows, one above the disease and one below. When you do this, you might have to make sure that you limit the number of vaccines to 300 that are available.

9.7.4 Globalization

A huge problem in our modern world is how easily a disease spreads from one point of the world to a completely different region through travel. Both people and cargo are moved vast distances with planes, cars and boats, which enables a disease to travel very rapidly. It took The Black Death seven years to spread through Europe, while today, a disease can travel across the entire world in just a few weeks. In the previous section we vaccinated lines near the outer points of the infection, this is no longer sufficient.

To simulate globalization, we add a small chance for an infected individual to spread the disease to a random site anywhere on the map. For simplicity globalization only happens once during our simulation, namely after time step 30. After this, we assume that all transportation has been banned to limit the spread of the disease. Below is a simulation of the pandemic spreading with globalization and no vaccination:

```
grids = pandemics.simulate(level=3)
epidemics.plot_snapshots(grids, pandemic=True)
show()
```



The disease is now able spread to another continent. What vaccination strategy works best in this case? Try to implement `distribute_vaccines()` in a way that stops the disease from spreading. Remember that you can call `numpy.random.seed()` before the simulation to try your solution on the same problem multiple times. Remove the seed when you want randomly generated problems again. Your solution should be able to solve the problem in all cases.

One strategy is to vaccinate all neighbors of the infected sites. We loop through all the sites in the system, and if a site is infected, we vaccinate all its neighbors.

```
def distribute_vaccines(current_map):
    rows = current_map.shape[0]
    columns = current_map.shape[1]

    vaccination_map = zeros([rows, columns])

    for i in range(1, rows - 1):
        for j in range(1, columns - 1):
            if current_map[i, j] == pandemics.INFECTED:
                left = (i, j - 1)
                right = (i, j + 1)
                up = (i - 1, j)
                down = (i + 1, j)
```

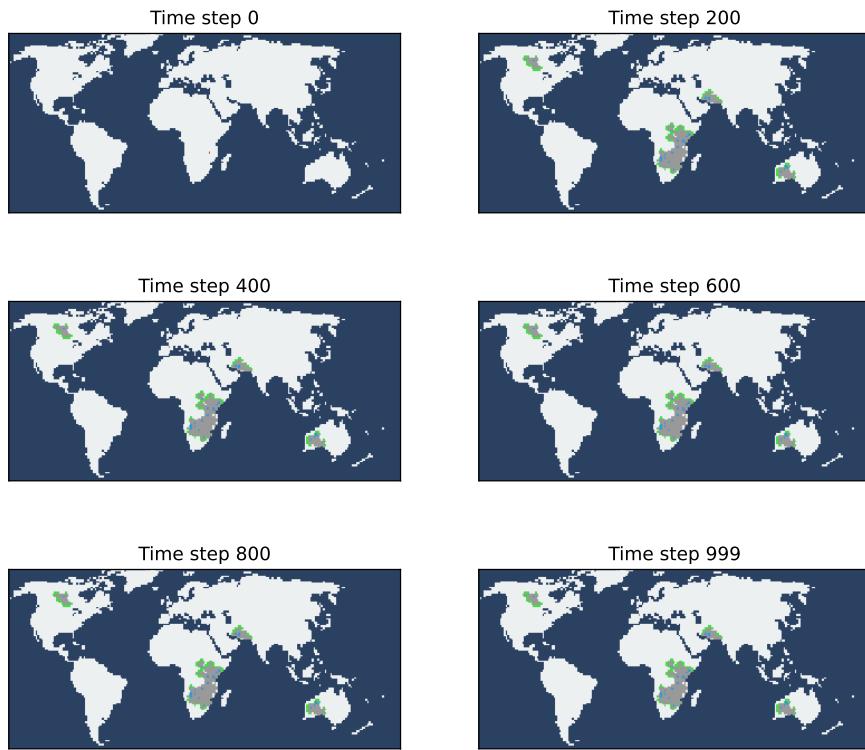
```

        for neighbor in [left, right, up, down]:
            if current_map[neighbor] == pandemics.SUSCEPTIBLE:
                vaccination_map[neighbor] = 1

    return vaccination_map

grids = pandemics.simulate(distribute_vaccines, level=3)
epidemics.plot_snapshots(grids, pandemic=True)
show()

```



By applying the vaccine to neighboring sites of the infected ones, we are able to encapsulate the pandemic even in the case of globalization. This is the strategy that was used to eradicate smallpox, an infectious disease with around 35% mortality rate. Smallpox is estimated to have been the cause of 200 to 500 million deaths in the 20th century. Each outbreak was stopped by isolating the infected and vaccination everyone that lived close by. This process is called *ring vaccination*, and is exactly what we have implemented to stop the disease in our simulation.

Real life experiments of the spread of epidemics are often impossible or expensive, and also unethical. In these cases, mathematical models are useful for analyzing the dynamics of infectious diseases. Models give us a way to use the current information about the state and progress of an outbreak, to predict the future. In addition, models can be used to evaluate different strategies to control an epidemic. For example, in this chapter we have used the spatial SIR model to test different strategies for stopping the spread of a disease by vaccination. Mathematical

modeling is a fundamental element in planning control strategies against outbreak of infectious diseases.

9.8 Summary

In this chapter, we have studied the SIR model for disease outbreaks. We first studied a population dynamics version of this model without taking space into account, but were still able to model the death tolls in Eyam village reasonably well. Later, we expanded the model to also include the spatial aspects.

We introduced the following biology terms:

Epidemiological threshold: The threshold where each infected individual infects exactly one susceptible individual. For a disease to grow, each individual must infect at least one susceptible individuals. If not, the disease will die out.

Herd immunity: To keep a population safe, we do not need to vaccinate everyone, but we must vaccinate enough that the disease is well below the epidemiological threshold. Some people have disorders that makes it dangerous for them to get vaccines.

In terms of programming, we introduced the concept of arrays, tuples, modeling on a two-dimensional grid, and expanded on random numbers.

9.8.1 Arrays

Arrays are similar to lists, but can only contain data of the same type (strings, floats, integers, ...). However, arrays are much more suitable for mathematics because they support operations, like additions and subtractions, on all elements in an array at the same time. We get access to arrays with `from pylab import *`.

There are several different ways we can create an array:

Syntax	Description
<code>array([5, 6, 7, 8])</code>	Convert a list to an array
<code>zeros(N)</code>	With N zeros
<code>arange(stop)</code>	From 0 up to, but not including, stop with step size 1
<code>arange(start, stop)</code>	From start up to, but not including, stop with step size 1
<code>arange(start, stop, step)</code>	From start up to, but not including, stop with step size step

Some array operations when we have two arrays of equal length, `a = array([1, 2, 3])` and `b = array([1, 2, 3])`:

Syntax	Description	Result
<code>len(a)</code>	Number of elements in array <code>a</code>	3
<code>a[1]</code>	Index the array, get element at index one	2
<code>a[1:3]</code>	Slice: get a view of the data	array([2, 3])
<code>a.copy()</code>	Creates a copy of an array	array([1, 2, 3])
<code>a + b</code>	Element-wise addition	array([2, 4, 6])
<code>a + 2</code>	Add 2 to each element of <code>a</code>	array([3, 4, 5])
<code>a - b</code>	Element-wise subtraction	array([0, 0, 0])
<code>a - 2</code>	Subtract 2 from each element of <code>a</code>	array([-1, 0, 1])
<code>a*b</code>	Element-wise multiplication	array([1, 4, 9])
<code>a*2</code>	Multiply each element of <code>a</code> with 2	array([2, 4, 6])
<code>a/b</code>	Element-wise division	array([1, 1, 1])
<code>a/2</code>	Divide each element of <code>a</code> with 2	array([0.5, 1., 1.5])
<code>a**b</code>	Element-wise power	array([1, 4, 27])
<code>a**2</code>	Each element of <code>a</code> to the power of 2	array([1, 4, 9])
<code>sqrt(a)</code>	The square root of each element in <code>a</code>	array([1., 1.41421356, 1.73205081])

9.8.2 linspace

The `linspace()` function creates an array given the start and stop points, and the *number of elements* in the array.

It takes three arguments: `linspace(start, stop, number_of_elements)`. If we want 5 points spread out on the interval from 0 to 1, we write:

```
a = linspace(0, 1, 5)
print(a)
```

[0. 0.25 0.5 0.75 1.]

9.8.3 Tuples

A tuple is a special type of list in Python. We create a tuple by using regular parentheses () instead of brackets [].

```
a_tuple = (1, 2, 3)
```

We index tuples similarly to lists:

```
print(a_tuple[1])
```

2

However, we cannot change the content of the tuple.

9.8.4 SIR model

In the SIR model, individuals are placed in three categories:

- S - Susceptible. People in this group are currently healthy and can become infected.
- I - Infected. People in this group are currently infected and can infect others.
- R - Recovered. People in this group cannot become infected or infect others.

The SIR model works well for infectious diseases transmitted from human to human, where recovery makes you immune to the disease. Individuals go from susceptible, to infected, to recovered ($S \rightarrow I \rightarrow R$).

The factor a relates to how many susceptibles become infected. The ratio of infected individuals that recover (either become healthy or die) every week is called the *recovery rate*, b .

Equation describing the model. The equations for the number of individuals in each group is given by:

$$S_n = S_{n-1} - aS_{n-1}I_{n-1} \quad (9.13)$$

$$I_n = I_{n-1} + aS_{n-1}I_{n-1} - bI_{n-1} \quad (9.14)$$

$$R_n = R_{n-1} + bI_{n-1}. \quad (9.15)$$

Implementation of the model. The core loop of this model is:

```
for n in range(1, N):
    S[n] = S[n-1] - a*S[n-1]*I[n-1]
    I[n] = I[n-1] + a*S[n-1]*I[n-1] - b*I[n-1]
    R[n] = R[n-1] + b*I[n-1]
```

9.8.5 Spatial SIR model

We model individuals as fixed in one position. Each infected position may only transmit the disease to their immediate neighbors, as that is the only individuals they are in contact with. Each square in the grid is any number of individuals with a fixed position. The rules for the model are:

1. An infected individual can only infect the individuals that are closest to it on each side, and only if the neighbors are susceptible.
2. There is no guarantee that an infected individual infects its neighbors. An infected individual has a probability α of infecting a susceptible neighbor during a single time step.
3. For each time step, each infected individual can recover (die or become healthy, as in the last section) with probability β .

When dealing with the edges of the grid of individuals, we use the technique of *Ghost individuals*; we add extra individuals along the edges of the grid, that we do not include in the simulation.

Bibliography

- [1] Codons sun; shows which base sequence encodes which amino acid. https://commons.wikimedia.org/wiki/File:Aminoacids_table.svg.
- [2] NASA/Apollo 17 crew; taken by either Harrison Schmitt or Ron Evans. Earth as seen from apollo 17. License: Public domain, https://www.nasa.gov/multimedia/imagegallery/image_feature_329.html.
- [3] Graphic decomposition of a chromosome (found in the cell nucleus), to the bases pair of the dna. https://commons.wikimedia.org/wiki/File:Chromosome_en.svg.
- [4] Escherichia coli: Scanning electron micrograph of escherichia coli, grown in culture and adhered to a cover slip. https://commons.wikimedia.org/wiki/File:EscherichiaColi_NIAID.jpg.
- [5] Evan-Amos. Orange. License: CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=36735411>.
- [6] 3d atomic rendering of insulin-like growth factor. https://commons.wikimedia.org/wiki/File:1IGL_Insulin-Like_Growth_Factor_Ii_01.png.
- [7] Leah Edelstein Keshet. *Mathematical Models in Biology*. Society for Industrial and Applied Mathematics, 2005.
- [8] Madprime. Escherichia coli culture. License: CC-BY-SA-3.0, https://commons.wikimedia.org/wiki/File:Ecoli_colonies.png.
- [9] Yasunori Ogura, Denise K. Bonen, Naohiro Inohara, Dan L. Nicolae, Felicia F. Chen, Richard Ramos, Heidi Britton, Thomas Moran, Reda Karaliuskas, Richard H. Duerr, Jean-Paul Achkar, Steven R. Brant, Theodore M. Bayless, Barbara S. Kirschner, Stephen B. Hanauer, Gabriel Nuñez, and Judy H. Cho. A frameshift mutation in NOD2 associated with susceptibility to crohn's disease. *Nature*, 411(6837):603–606, 2001.
- [10] Pisum sativum pods. https://commons.wikimedia.org/wiki/File:Doperwt_rijserwt_peulen_Pisum_sativum.jpg.
- [11] L. A. Urry, M. L. Cain, S. A. Wasserman, P. V. Minorsky, and R. Orr. *Biology: a Global Approach*. Pearson, twelfth edition, 2020.

Appendix A

Additional information

A.1 Packages

There are multiple ways to import packages in Python. In this book, we use

```
from pylab import *
```

as our preferred method because this simplifies the introduction to Python in general.

However, this way of importing packages is risky, because we can run into name clashes. This is the case if for instance two functions in different packages have the same name. One example is the sine function, which exists both in `pylab` and `math`. If we first import `pylab`, which has a sine function that works with arrays and then import `math`, which has a sine function that only works with floats, we will receive an error if we try to get the sine value of an array of numbers:

```
from pylab import *
from math import *
numbers = array([1.0, 2.0, 3.0])
print(sin(numbers))
```

```
-----
TypeError                                 Traceback (most recent call last)
Input In [489], in <cell line: 4>()
      2 from math import *
      3 numbers = array([1.0, 2.0, 3.0])
----> 4 print(sin(numbers))

TypeError: only size-1 arrays can be converted to Python scalars
```

However, if we import `math` first and `pylab` after, the code works:

```
from math import *
from pylab import *
numbers = array([1.0, 2.0, 3.0])
```

```
print(sin(numbers))
```

```
[0.84147098 0.90929743 0.14112001]
```

This prints the sine values of the numbers.

The above issue is the reason why it is not recommended to use the *-based import, especially in larger projects. We did however show you the *-based import because it simplifies the code greatly for a newcomer in Python. As you move on to bigger projects, we recommend using one of the following ways of importing Python packages:

Importing the package explicitly. This is the safest way of importing packages and greatly reduces the chance of name clashes:

```
import math
```

The contents of the package is then accessed by writing the name of the package and a dot, ., before the name of the function we wish to access. Let us test this by importing the math package and then take the square root of the number 4 by using the `math.sqrt` function.

```
import math
print(math.sqrt(4))
```

```
2.0
```

Importing the package explicitly with an alias. This is the same as the above, but allows you to give the package a shorter alias of your preference, making it quicker to type:

```
import math as mt
print(mt.sqrt(4))
```

```
2.0
```

Importing only what you need. We can also directly import the `sqrt` function by writing `from math import sqrt`. We then have access to `sqrt` in our program without having to write `math.sqrt()`:

```
from math import sqrt
print(sqrt(4))
```

```
2.0
```

We can also import multiple functions this way by listing them comma-separated:

```
from math import sqrt, sin, cos
print(cos(4))
```

```
-0.6536436208636119
```

This is better than the `*`-based import, but you still risk importing two functions with the same name from two packages if you do not pay attention to what you import.

Importing everything. Finally, we may import everything from a package using the already mentioned `*`-based import:

```
from pylab import *
```

And this, as we said before, comes with the risk of importing many of the same names from two different packages, with the accompanying unwanted behavior.

A.2 Terminal basics

In this appendix we show you a few terminal commands. First we create a new folder to store the programs for chapter 1 using the command `mkdir` (make directory):

```
unix ~$ mkdir chapter1
```

This creates a folder named `chapter1`. Then we enter that folder with the `cd` (change directory command)

```
unix:~$ cd chapter1
```

and we see that we are in `chapter1` as `chapter1` has been added before the dollar sign, `$`, in our terminal:

```
unix:~/chapter1$
```

We can then write our program and save it as before. If we want to check if we have managed to create a file you can use the `ls` command. This command lists all files that are in the current working directory:

```
unix:~/chapter1$ ls
cell_theory.py
```

Now we open our text editor by

```
unix:~/chapter1$ atom
```

The `&` (ampersand) at the end tells the terminal to keep working, if we did not add `&` the terminal would wait until we were finished with the editor before we could enter new commands. We can now enter our program for writing the tree laws of cell theory into the editor,

A.3 Lists with mixed contents

If we have a list with both numbers and strings, and we want to save it, `save()` converts the entire list to an array of strings, and if we load what we saved each element is now string, even the numbers. An example that shows you this behavior:

```
mixed_list = ["invalid", 2, 3, 4]
save("mixed_list.npy", mixed_list)
mixed_result = load("mixed_list.npy")
print(mixed_result)
```

```
["invalid" "2" "3" "4"]
```

```
| ["invalid" "2" "3" "4"]
```

As we can see, each element of `mixed_result` is now a string, which means that the data changed in the process of saving it to file and loading it back in. You should be aware of this if you ever try to store lists with mixed types.

Index

- E. coli*, 65
- Escherichia coli*, 65
- Addition, 16
- Algorithm, 79
- Alleles, 171
 - dominant, 171
 - recessive, 171
- Amino acids, 205, 224
- and, 235
- Anemia, 251
- Annual plants, 136
- arrays, 287
 - creation, 288, 293
 - indexing, 288
 - `len()`, 288
 - `linspace()`, 294, 326
 - mathematical operations, 290
 - slicing, 289
 - view, 289
 - `arange()`, 293
 - `zeros()`, 293
- Asexual reproduction, 66
- assignment, 14
- AT/GC rule, 207
- Binary fission, 66
- Blending hypothesis, 170
- Boolean values, 189
- Boundary condition, 304
- Bubonic plague, 282
- Character, 169
- Chromosomes, 206
- Circular strings, 269
- Cloning, 258
- Codon, 224
- Combining boolean expressions, 235
- Commenting, 23
- Condition, 87, 188, 189
- Death model
 - advanced, 124
 - simple, 118
- Death phase, 75
- Death rate, 117
- Debugging, 24
- Deoxyribonucleic acid, 207
- Dictionaries, 216
 - `copy()`, 239
 - `del`, 238
 - `for` loops, 218
 - `get()`, 240
 - `in`, 239
 - key-value pair, 216
 - `keys()`, 217
 - `values()`, 217
- Difference equations, 101, 161
 - closed-form solution, 102
 - coupled, 161
 - first-order, 101, 161
 - iterative solution, 102
 - one-year model, 143
 - second-order, 102, 161
 - system, 161
 - two-year model, 161
- Division, 16
- DNA
 - base pairs, 207
 - bases, 207
 - double helix, 207
 - exons, 223
 - introns, 223
 - nucleotides, 207

sequence, 207
strands, 207
triplets, 207
Docstring, 21
Dot notation, 35
Double loop, 174, 230
Doubling time, 74, 76, 78
dynamic model, 79
elif, 214
else, 191
Epidemic, 282
Epidemiological threshold, 295
Errors
 IndentationError, 88, 198
 IndexError, 34
 NameError, 89
 SyntaxError, 88, 199
 TypeError, 24
Eukaryote, 66
Exponential decay, 118, 145
Exponential growth, 74
Exponential growth model, 79, 84
 assumptions, 80
 implementation, 90
 limitations, 98
Exponential growth phase, 74
False, 189
fileformats
 .csv, 58
 .eps, 49
 .pdf, 49
 .png, 49
 .svg, 49
 FASTA, 254
Files
 append, 277
 close(), 253
 closed, 277
 open(), 253
 readlines(), 253
 write(), 277
float(), 20
Floats, 18
for loops, 210
Frameshift mutations, 249
Frameshifts, 249
Functions, 14, 178
 arguments, 21
 def, 179
 default values, 196
 docstring, 199
 global variables, 197
 local variables, 197
 parameters, 179
 return, 179
 usage, 21
GC-content, 220
Gel electrophoresis, 259
 marker, 260
Gene, 171, 205, 207
Gene of interest, 258
Generation time, 67, 74
Genetic disorder, 247
Genome, 206
Genotype, 172
Germination, 136
Germination rate, 147
Ghost boundary condition, 304
Globalization, 322
Grid, 299
Growth rate, 76
 multiplicative growth rate, 79
 Relative growth rate, 82
Heart rate, 18
Herd immunity, 296, 314
Heterozygous, 172
Homozygous, 172
if, 188
If-elif-else-tests, 214
If-else-tests, 191
If-tests, 188
import, 25
 from file, 248
Indentation, 87
Indented code block, 87
Initial condition, 83
insulin, 30
int(), 20
Integers, 18

- Jupyter Notebook, 11
 cell, 12
 creating notebook, 11
 starting, 11
- Lag phase, 72
- Lag phase model, 126
- `len()`, 35
- Linear functions, 51
- linear growth, 42
- `list()`, 32
 `copy()`, 34
 indices, 32
 negative indices, 34
 slicing, 38
 sublist, 38
- Logarithmic scale, 70
- Logarithms, 70, 77
- Logistic equation, 110
- Logistic growth, 106
- Logistic growth model, 110
- Logistic model, 110
 carrying capacity, 107
- Long-term stationary phase, 75
- Machine code, 9
- Model organism, 65
- Model parameters, 128
- Module, 248
- Modulo operator, 271
- mRNA, 222
- Multiplication, 16
- Mutations, 247
- Nested loop, 174, 230
- `numpy`, 287
- obesity, 31
- one-based indexing, 33
- One-year model, 136
 difference equation, 143
- Operators
 assignment, 87, 189
 comparison, 189
 Modulo, 271
- Optical density, 67
- `or`, 235
- Order of operations, 22
- Packages, 25
- Palindromic sequence, 259
- `pandas`, 58, 176
 `DataFrame()`, 176
 `read_csv()`, 58
 `iloc[]`, 176
- Pandemic, 282
- Parameters, 128
- Patient zero, 299
- Peas, 136
- Phase diagram, 146
- Phenotype, 172
- Plasmids, 258
- `plot()`, 36
 color, 47
 `legend()`, 45
 line style, 47
 markers, 47
 multiple curves, 43
 save, 49
 `savefig()`, 49
 semilogarithmic, 70
 `show()`, 37
 `subplot()`, 159
 `title()`, 37
 `xlabel()`, 37
 `ylabel()`, 37
 `yscale('log')`, 70
- Point mutations, 247
 deletions, 249
 insertions, 249
 missense, 250
 nonsense, 251
 silent, 251
 substitutions, 250
- Population dynamics, 65
- Power, 16
- pre-mRNA, 223
- `print()`, 14
- Prokaryote, 66
- Proteins, 205
- Punnett square, 171, 172
- Pylab, 25
 `choice()`, 187
- `randint()`, 313
- Random number generator, 195

`random()`, 302
`range()`, 138, 226
Ratio, 170
Reading frames, 228
Recognition sequence, 259
Recombinant DNA, 258
Recovery rate, 285
Recurrence relations, 101
Reserved keywords, 15
Restriction enzymes, 259
Restriction sites, 259
Ring vaccination, 324
RNA, 222
RNA splicing, 223
`round()`, 19
Row-by-column, 176

secretion, 31
secretion rate, 42
`seed()`, 195
Semilogarithmic, 70
Sickle-cell anemia, 251
Simulate, 65
Simulation, 79
SIR model, 282, 284
 Infected, 284, 285
 Recovered, 284, 286
 Susceptible, 284, 285
Spatial model, 283, 300
Spatial SIR model, 300
Spontaneous mutation, 247
Stationary phase, 75
`str()`, 20
Strings, 14, 241
 `count()`, 219
 `join()`, 20
 multi-line, 199
 newline, 254
 `replace()`, 222
 `strip()`, 255
Sublists, 38
Subtraction, 16
Survival rate, 147

Termination codons, 231
The Black Death, 282
Time step, 81

Traits, 167, 169
Transcription, 222
Translation, 224
`True`, 189
True breeding, 169
Tuples, 302
Two-year model, 152
 difference equation, 161

Vaccination, 296, 312
Variables, 14
 names, 15, 139

`while`, 85
`while loops`, 85
 never ending, 87
Whitespace characters, 87
Whole genome sequencing, 205
Winter survival rate, 139

zero-based indexing, 32
`zip()`, 256