# Introduction to Biological Data Analysis in Python

Stilianos Louca

Edition: October 9, 2022

# Contents

## Copyright notice



## Preface

This book introduces basic techniques of computational data analysis in biology and, more broadly, basics of scientific programming. The book covers much of the material taught by Stilianos Louca at the University of Oregon in corresponding courses. The book targets undergraduate students in biology with an interest in computational techniques, but could also be of interest to students in many other scientific disciplines. Examples and exercises are drawn from a wide spectrum across biology.

The computer programming language used throughout the book is `python` 3, which is particularly easy to learn and popular among data scientists. Emphasis is placed on fundamental principles of scientific programming and on a small number of particularly well-established packages (e.g., `numpy`, `scipy`, `pandas` and `matplotlib`), rather than covering a large number of discipline-specific packages. Conceptual understanding tends to outlive most packages.

Recommended prerequisites for this book are familiarity with basic biology and calculus (high-school level). Exercises are included throughout the book; the reader is highly encouraged to attempt these exercises. References to literature covering additional material are also provided for the interested reader.

# 1 Motivation

## 1.1 Why learn computer programming?

Biology, like most sciences and industrial sectors, is becoming increasingly data-driven and computational [1, 2]. High-throughput genome sequencing, satellite-based remote sensing of forest structure, gene expression analyses in cancer cells, epidemiological monitoring and forecasting, automated image recognition in wildlife monitoring networks, analyses of vocal communication in whale populations, individual-based animal tracking, genetic analyses of population structure and evolutionary reconstructions, to name a few examples, would be unthinkable without computers. These and many other biological research activities generate terabytes and even petabytes of data each day, thus making automated, computer-driven analyses a necessity [3–11]. Scientists deploying sophisticated computational analyses often communicate with computers through *computer programming*, i.e., writing precise computer-readable instructions (aka. "*code*"). For a biologist, there are several reasons to learn how to program computers:

- Modern programming languages typically offer much greater versatility and control over a desired computation, compared to graphical user interface-based ("point-and-click") software. For example, high-quality visualizations of scientific data are typically achieved via computer programming.
- Many modern data analyses require complex computational workflows specifically tailored to the task at hand. Some tasks can be so specialized or so novel that no readily available software exists for performing them, and hence scientists are often forced to develop their own software.
- Even if software exists, it may not perform the task exactly as desired, or it may require some modifications to be able to handle the data at hand, in which case it is useful to be able to modify existing software to one's needs. Fortunately the source code for a lot of scientific software is open, i.e., their code is viewable and editable, thus allowing for adjustments.
- A data analysis workflow may comprise dozens to hundreds of computational steps, each performed by a separate program and each applied multiple times to distinct pieces of data. Such multifaceted and/or repetitive workflows are usually best coordinated using a programming language.

It is not an understatement to say that the ability to program computers is becoming the new form of *alphabetism* in science and engineering.

## 1.2 Example outcomes

Through this book/course, students will learn the basics of scientific computer programming, in particular with regards to analyzing and visualizing biological data. The following are examples of the types of analyses that students will learn to perform via computer programming:

- Compute the frequency of each DNA nucleotide, and in fact of arbitrary nucleotide sequences, across the genome of an organism.
- Statistically compare the number of confirmed COVID19 cases and deaths to socioeconomic properties of countries.
- Compute the amount of time that a bird spent at sea and its total travel distance, based on GPS tracking data.
- Create various professionally looking visualizations of data.

- Determine the number of unique genera listed in a species catalog.
- Compute per-capita annual growth rates of the human global population.
- Detect peaks in electroencephalograms.
- Compute quality metrics of genome assemblies.
- Detect potential negative interactions between bacterial populations based on observed abundances over time.

### 1.3 Programming requires practice

Keep in mind that programming is a very hands-on skill. Similarly to learning a new sport or art, so learning how to write computer code and think like a data scientist requires a lot of practice. Do try to repeat all examples and complete all exercises of this book, and feel encouraged to "play around" to try out new things. Just like little cheetah cubs need to constantly play to develop their motor skills, you need to play around with python out of your own initiative to really develop your computational thinking skills.

## 2 Installing and running python

### 2.1 Installing `python`

Students working through this book are strongly advised to install a recent version of `python` 3 on their personal machines, as the material requires a lot of hands-on practice (all examples have been tested using version 3.9). Python is freely available for all major operating systems. Note that `python` 2 is incompatible with `python` 3 code, and should thus not be used for this course. The proper procedure for getting `python` installed on your machine depends strongly on the type of operating system. The material below only provides a brief overview of the typical installation procedures; situation-specific instructions and troubleshooting details are best sought online, e.g. via Google.

**Mac:** All modern Macs come with `python` 3 pre-installed and ready to go, although you may need to install some additional packages as we move through the course (for details on installing `python` packages see Section 2.2 below).

**Linux:** Most Linux users will already have `python` 3 installed; if not, they can easily do so using common package managers, for example in Ubuntu:

```
sudo apt-get install python3
```

or in Fedora:

```
sudo dnf install python3
```

**Windows:** Windows users will need to download and run the latest official `python` exe installer, available here. When prompted by the installer, you should check the checkboxes "Install launcher for all users" and "Add python to PATH". The default installation path is somewhere around `C:\Users\Username\AppData\Local\Programs\Python`, although this may differ somewhat between Windows versions.

## 2.2 Installing packages

Python packages are essentially optional software collections that enhance the functionality of python, often targeting specific disciplines. For example, the `scipy` provides various functions for advanced scientific computation, `scikit-learn` package provides machine learning capabilities, while the package `cartopy` provides functions for geographic visualizations. Many packages come pre-installed with standard `python` distributions, however thousands of additional packages are available for download (often for free).

The precise method to install a new package depends somewhat on the package, for example whether it is freely available or commercial, and on which online database it is available. The majority of `python` packages can be obtained through an automated package manager called `pip` ("Package Installer for Python"), which usually is automatically installed with `python`. To download and install a package using `pip` we use the terminal (on Mac and Linux) or command prompt (on Windows), typically as follows:

```
sudo pip3 install scipy
```

To update an existing package, modify the above command as follows:

```
sudo pip3 install --upgrade scipy
```

Packages used throughout this book include:

- `os`
- `sys`
- `numpy`
- `scipy`
- `matplotlib`
- `global-land-mask`
- `pandas`

## 2.3 JupyterHub

If you are following this book as part of a course at the University of Oregon, your instructor may have set up an online `JupyterHub` server that you can access to write and run `python` code without having to install `python` on your machine. `JupyterHub` allows one to create so called *Jupyter Notebooks*, which are documents that combine `python` code with normal text and visualizations, resulting in more human-readable program flows and *story lines* around a dataset. You can also install Jupyter Notebook on your own machine for free, thus enabling you to use its functionalities without an internet connection. The `JupyterHub` resource provided by your instructor should generally only be used if you cannot install `python` on your own machine, as

it requires an internet connection and responses may be slow at times (due to many users using it simultaneously).

## 2.4 Running `python` in interactive mode (console)

The simplest way to execute some basic `python` code is in interactive mode, aka. the console. On Macs, start the `Terminal` app located at `/Applications/Utilities/Terminal.app`, type the following command and hit *Enter*:

```
python3
```

On Linux, the location of the Terminal is system-dependent, but is usually prominently featured since it is a central part of the Linux user experience. On Windows, open the *Start* menu to find and start the `Command Prompt` program (typing `cmd` should quickly reveal it). This will open a new window, called a *command prompt*. Type `py` into this window and hit *Enter*. For more details on running `python` on Windows see e.g. this official webpage.

Either one of the above procedures should start up the `python` interpreter, which means that anything you type henceforth into your terminal (on Mac or Linux) or command prompt (on Windows) will be interpreted as `python` code and executed, one line at a time. For example, typing:

```
10+20+30
```

and hitting *Enter* will sum up the numbers 10, 20 and 30 and print out the result immediately underneath:

```
60
```

When used this way, `python` can be seen as a *very* fancy calculator, although its functionalities extend far beyond those of traditional handheld calculators. More accurately, `python` is a programming language, and the `python` interpreter's job is to translate what we type in that language into something a computer can understand and act upon. Similarly to natural human languages, `python` has rules, but there are some fundamental differences:

- `python`'s rules are relatively simple and consistent (i.e., with few exceptions). Most rules can be learned in a few weeks to a point where we can already accomplish many realistic tasks.
- `python`'s rules are rigid and with a very precise meaning, to the point where a single comma or period can dramatically affect what a given `python` code does (and whether it works at all). Natural languages are more vague and more flexible, which means that small mistakes can usually be glossed over and don't affect our ability to understand each other. A computer will follow our code to absolute precision, but if there's the slightest mistake it can easily generate a wrong result or even fail completely.

As you learn to write `python` code, you will encounter numerous error messages from the interpreter. Don't despair, as even experienced programmers encounter error messages all the time. Error messages may be hard to understand at the beginning, but they will become more understandable over time and will help you write correct `python` code.

## 2.5 Writing and running scripts

For anything more complex than just a few lines of code, it is recommended to write `python` code into self-contained scripts rather than into the console. Scripts are merely plain text files containing all the code to be executed, from top to bottom and typically one command per line. For example, a `python` script may comprise the following code, which generates a plot of the sine function and saves it as a PDF (don't worry if you don't understand this code yet):

```
import numpy
from matplotlib import pyplot

# evaluate the sine function on a regular X-grid
x = numpy.linspace(0,10,100)
y = numpy.sin(x)

# plot the sine function using pyplots
fig = pyplot.figure(figsize=(4, 3))
pyplot.plot(x, y, c = "#00507a", linewidth = 2, label="sin")

# add title & axis labels
pyplot.title("Sine function")
pyplot.xlabel("x")
pyplot.ylabel("sin(x)")

# save the plot to a PDF file
os.makedirs("figures", exist_ok=True)
pyplot.savefig("figures/script_example_sine_plot.pdf", bbox_inches='tight')
pyplot.close()
```

Because `python` scripts are plain text files, they can be created and edited by virtually any text editor, such as `TextEdit` or `BBEdit` on Mac, `Gedit` or `Vim` on Linux, and `Notepad` on Windows. You should avoid, however, editing `python` scripts in word editors such as `Word` or `OpenOffice`, as these tend to create non-plain text formats that are incompatible with the `python` interpreter. While `python` scripts are plain text, they should have the file extension `.py` instead of simply `.txt`, as this allows the operating system and text editors to recognize them as python scripts.

To run a `python` script, you can "call" the script in your terminal (Mac and Linux) or command prompt (Windows):

```
python3 path_to_my_script.py
```

The above command tells your operating system to interpret the entire contents of your script using `python`. Note that you will either need to provide the proper absolute path of your script (e.g., starting at `C:` in Windows) or a proper relative path. Depending on your `python` installation, you may also need to replace "`python3`" with some other version number, or simply with "`python`".

The same language rules apply when writing `python` code in interactive mode or in the form of scripts. There are, however, many benefits to writing scripts instead of typing commands line by line into the `python` console. For one, a script constitutes a precise and complete record of the computational steps taken that can easily be shared with colleagues, attached to reports or scientific publications for transparency, and be used again by yourself in the future, should the need arise. Second, scripts make it much easier to write complex code compared to the console. Many scientific analyses comprise hundreds or thousands of lines of code, and it would be a true nightmare if these had to be entered one by one into the console each time they are performed.

Third, scripts allow us to revise arbitrary parts of our code (for example, if a mistake is found) without having to re-enter each subsequent line into the console. It's like writing a novel on a computer versus a typewriter - a computer allows us to revise arbitrary parts of the text without needing to retype everything else again.

## 3 Conventions in this book

This book includes several `python` code examples, as well as examples of datasets, code outputs and terminal commands. To improve clarity, these are displayed using different styles. Valid `python` code is always displayed as follows:

```
import numpy
import matplotlib.pyplot as pyplot

x = numpy.linspace(0,10,100)
y = numpy.sin(x)
```

Invalid `python` code, for example showing code that would generate an error or yield the wrong result, is displayed using a different frame color:

```
number_of_genes = 3821
genome_size = number_of_genes * average_gene_size
average_gene_size = 1060
```

Text output generated by `python` code, i.e., printed to the screen, is displayed as follows:

```
Minimum mass: 91 kg
Maximum mass: 134 kg
Largest mass difference: 43 kg
```

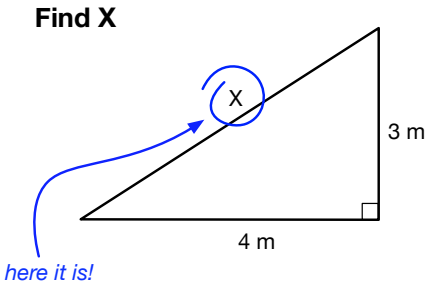
This format is also used for the contents of data files, for example:

```
# species   male_body_mass_kg   female_body_mass_kg
Mirza coquereli 0.304   0.326
Microcebus myoxinus 0.031   0.03
Microcebus rufus    0.043   0.042
Lemur catta 2.21    2.21
```

Shell commands written into the terminal (Mac and Linux) or command prompt (Windows) are displayed as follows:

```
python3 path_to_my_script.py
```

**Exercises:** The book includes numerous programming exercises, typically at the end of each section. Exercises are numbered and marked with a ★. Particularly challenging exercises are marked with ★★; these exercises should all be doable with the material covered thus far, but may require more creativity and troubleshooting. When an exercise asks you to compute and report a specific quantity or create a certain visualization, you are expected to *write* ***python*** *code* that will perform those tasks in a generic and automated way. For example, an exercise may give you a list of 10 numbers and ask you to write `python` code that determines the largest number in the list; in that case, your code should be written in such a way that it would find the largest number for any such provided list, not just report the number that happens to be largest in the specific example. In other words, don't be like the person in the joke:




Many exercises include helpful hints; these are meant as recommendations, but you are not required to follow them. Indeed, there are usually many alternative ways to solving a problem.

# 4 Introduction to python

## 4.1 Variables

A central concept in any modern programming language is that of *variables*, which are essentially representations of data stored in memory. For example, a variable may represent a single fixed number:

```
x = 3
```

or the numerical result of a mathematical calculation:

```
y = 5.1 + 10.4
```

or a piece of text:

```
z = "hello"
```

Variables are referenced based on their unique name; in the examples above the 3 variables had names `x`, `y` and `z`, however longer names (including numbers and underscores) are also allowed:

```
number_of_genes = 3821
species_name = "Mycobacterium lepraemurium"
treatment1 = "no antibiotics"
treatment2 = "with antibiotics"
```

Spaces and other non-alphanumeric characters (except underscores) are not allowed in variable names, since these characters have special meanings in `python`.

Variables are defined or modified (if they have already been defined) by assigning a value from the right to the left, and never from left to right. Note that the code

```
number_of_genes = 3821
```

is not asserting that `number_of_genes` is equal to 3821; rather, it *specifies* what the variable `number_of_genes` shall hereafter mean, i.e., it is essentially saying:

> *define the variable* ***`number_of_genes`*** *to hereafter represent the value 3821*

Once a variable has been defined, its value can be accessed in subsequent calculations. For example, in the following code we are first defining two new variables, `number_of_genes` and `average_gene_size`, and then assign the product of these two variables to a 3rd new variable

called `genome_size`:

```
number_of_genes = 3821
average_gene_size = 1060
genome_size = number_of_genes * average_gene_size
```

The variable `genome_size` now contains the value 4050260. We can hereafter use this value in our calculations by writing `genome_size` instead of the numerical value itself. You will have noticed that we use the "`*`" symbol to denote multiplication, instead of $\times$ or the dot; this is common in most programming languages. We will discuss mathematical operations in more detail later on.

To display the value of a variable in interactive mode, we can type its name into a new line and hit *Enter*. The value will then be displayed beneath. Note that in a script this approach does not work (nothing will be displayed). Alternatively, we can also explicitly tell `python` to print the variable's value to the screen, for example as follows:

```
print(genome_size)
```

```
4050260
```

In contrast to the 1st method, the 2nd method also works in scripts.

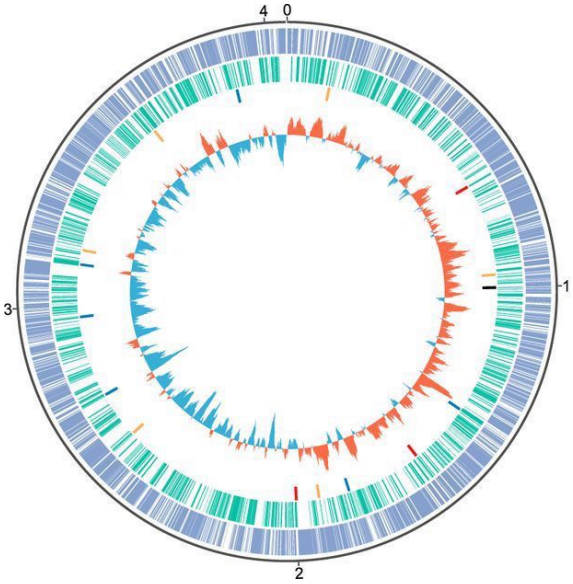



**Figure 4.1:** The genome of *Mycobacterium lepraemurium* strain Hawaii [12].

★ **Exercise 1:** Determine the final value of the variables `x`, `y`, `z` after the following code has executed. Try to first figure out the answer in your head without running the code, and then confirm the results by running the code.

```
x = 3
y = x * 5
x = y
z = x * 2
```

Remember that if you're running the code in a script, you will need to use `print` to actually see

the values of the variables `x,y,z`.

## 4.2 Basic syntax rules

It is at this point worth discussing some general syntax rules of the `python` language. First, the order in which we write commands matters! If we had instead written:

```
number_of_genes = 3821
genome_size = number_of_genes * average_gene_size
average_gene_size = 1060
```

we would have gotten an error, since by the time the variable `average_gene_size` is used we haven't actually defined it yet. This is because `python` reads and interprets code from top to bottom. Second, keep in mind that `python` is case-sensitive, which means letter capitalization matters. For example, the following code would not work, because we are accessing an undefined variable `Average_Gene_Size`:

```
number_of_genes = 3821
average_gene_size = 1060
genome_size = number_of_genes * Average_Gene_Size
```

Third, we can freely use whitespace (e.g., spaces and tabs) to make our code look "prettier", except at the beginning of new commands. For example, the following code is valid and equivalent to the earlier example:

```
number_of_genes     = 3821
average_gene_size   = 1060
genome_size         = number_of_genes   *   average_gene_size
```

whereas the following code is not valid:

```
    number_of_genes = 3821
  average_gene_size = 1060
        genome_size = number_of_genes * average_gene_size
```

We will see below that whitespace at the beginning of commands is interpreted by `python` in special ways, which is why we are not allowed to enter it for mere decorative purposes.

Fourth, we can split long python statements across multiple lines using the backslash "\" as a line separator, for example:

```
genome_size = number_of_genes\
              * average_gene_size
```

Note that in this case we are allowed to add decorative whitespace at the beginning of the 2nd line, because it is merely a continuation of the previous line. In some cases we are also allowed to split a line without appending a "\", as will become clear through examples in this book; if in doubt, it doesn't hurt to include the backslash.

## 4.3 Comment lines

Like most programming languages, `python` allows the insertion of *comments* into code, i.e., arbitrary segments of text that are not actually interpreted by `python`. Comments typically help us understand the context of some code, or make technical notes to ourselves. Anything

following a # in python is considered a comment up until the end of that line, unless the # is immediately preceded by a backslash (\). In the example below, comments are used to elaborate on the meaning of various variables:

```
number_of_genes   = 3821 # the number of genes in the bacterial genome
average_gene_size = 1060 # the average size of a gene in the bacterial genome

# compute the average genome size by multiplying the
# number of genes with the average gene size
genome_size = number_of_genes * average_gene_size
```

As you start writing increasingly complex code, you will come to appreciate the importance of properly documenting the logic of your code. This makes it easier for others as well as your future self to understand it. In this book, we will also be routinely using clarifying comments in our code examples.

★ **Exercise 2:** For each of the following pieces of python code, determine whether they are valid or invalid. For the valid ones, determine the final value of each variable. For the invalid ones, explain why the code is invalid.

*Hint: You may either run the codes to see if they generate an error, or you can just predict what will happen based on what we have learned.*

1. ☐

```
number_of_cells = 2000
cell_volume = 0.001
total_volume = number_of_cells * cell_volume
```

2. ☐

```
number of cells = 2000
cell_volume     = 0.001
total_volume    = number_of_cells * cell_volume
```

3. ☐

```
2000 = number_of_cells
cell_volume = 0.001
total_volume = number_of_cells * cell_volume
```

4. ☐

```
number_of_cells = 2000
cell_volume     = 0.001
total_volume    = number_of_cells * cell_volume
```

5. ☐

```
x = 10
z = x * y
```

6. ☐

```
x = 10
z = x * 5
```

7. ☐

```
x = 10
z = z * 5
```

8. ☐

```
x = z * 5
z = 10
```

9. ☐

```
x = 10
    z = x * 5
```

## 4.4 Mathematical operations

We have already encountered the symbol `*` for multiplying two numbers in `python`. This is an example of a mathematical operation, but as you have probably already guessed, other standard mathematical operations are also available, including addition, subtraction, division and raising powers:

```
x = 3
y = x + 4  # this assigns the value 3+4=7 to the variable y
z = y - 1  # this assigns the value 7-1=6 to the variable z
w = z / 4    # this assigns the value 6/4=1.5 to the variable w
q = w ** 3 # raise w to the power of 3 and assign the result to q
```

Note that `/` performs true devision, not integer division; for example `14/4` yields `3.5`, not `3`. To perform integer division, we can instead use `//`, as follows:

```
A = 14 // 4  # this assigns the value 3 to A
B = 6.5 // 4 # this assigns the value 1 to B
C = 0.5 // 4 # this assigns the value 0 to C
```

Another useful operation is modulo, i.e., computing the remainder of integer division, encoded by the `%` character:

```
r = 5 % 2 # this assigns the value 5 modulo 2 = 1 to the variable r
r = 5 % 5 # this assigns the value 5 modulo 5 = 0 to the variable r
```

Remember that assignments always happen from right to left: First the result of a mathematical operation (on the right side of the `=` sign) is computed, and then it is assigned to the variable on the left of the `=` sign. Hence, for example, the following code is invalid:

```
x + 4 = y
```

If we merely want to add/subtract a value from an existing variable, or simply want to increase/decrease it by some factor, we can also use the shorthands:

```
x += 4  # increment x by 4 and assign the result back to x
x -= 2  # subtract 2 from x and assign the result back to x
x *= 5  # multiply x by 5 and assign the result back to x
x /= 10 # divide x by 10 and assign the result back to x
```

We can also combine several mathematical operations, for example as follows:

```
x = 4 + 5 - 10
y = 10 ** 3
z = y*2 + x/3
```

Note that power raising is by default prioritized over multiplication and division, which are in turn prioritized over addition and subtraction, similarly to what we've learned in elementary school. Hence, in the last line above first `y` is multiplied by 2, then `x` is divided by 3, and finally the results of these two operations are added, and the result assigned to `z`. If needed, we can use round brackets to explicitly specify our desired priority of operations. For example, in the code below first `x` is added to `2y` and then the result is divided by 3:

```
z = (y*2 + x)/3
```

It is recommended to always use parentheses to explicitly specify the desired order of operations to avoid confusions. In some cases it can also be useful to split complex mathematical expressions into multiple stages using intermediate variables. For example, the following mathematical formula

$$f = x^{y+3 \cdot z^2} \tag{4.1}$$

can either be coded as follows:

```
f = x**(y + 3 * z**2)
```

or as follows:

```
exponent = y + 3 * z**2
f = x ** exponent
```

**Integers vs. decimals:** For future reference, it should be mentioned that technically speaking `python` distinguishes between numeric variables that can (but don't necessarily need to) include decimals (e.g., `30.5` or `0.04` or `0.0`), known as floating point numbers (or "floats"), and variables for representing integers (e.g., `30` or `0`). Floating point is a technical term in computer science referring to numbers with decimals stored in a specific way. In technical language, floats and integers are different "data types". For example, the following code defines `X` as a float and `N` as an integer, even though mathematically their values are equivalent:

```
X = 3.0
N = 3
```

In the vast majority of cases this technical distinction can be ignored, since integers are automatically converted to floats or vice versa when needed in most calculations. In some rare situations, however, we may need to represent a value explicitly as an integer, for example when asking `python` to repeat a computation $N$ number of times. If needed, we can convert any float to an integer as follows:

```
N = int(X)
```

Let's look at the difference:

```
print(X)
```

```
3.0
```

```
print(N)
```

```
3
```

★ **Exercise 3:** Match each of the following mathematical expressions:

1. $z + x \cdot y^2$
2. $z + (x \cdot y)^2$
3. $z + x^2 \cdot y$
4. $(z + x) \cdot y^2$
5. $(z + x)^2 \cdot y^2$

to one or more of the following `python` codes:

A. `z + (x * y) * (x * y)`

B. `z + (x * y**2)`

C. `(z + x) * (y**2)`

D. `((z + x)**2) * (y**2)`

E. `((z + x) * y)**2`

F. `z + x**2 * y**2`

G. `z + (x**2) * y`

Note that some mathematical expressions may match multiple codes.

★ **Exercise 4:** Write a `python` code that represents the following formula for converting temperatures from Fahrenheit to Celsius:

$$C = (F - 32) \cdot \frac{5}{9}. \tag{4.2}$$

Then evaluate the formula for $F = 50$ and print out the resulting value of $C$.

★ **Exercise 5:** Write a `python` code that represents the following mathematical expression:

$$\frac{x + y^{2z+x}}{100 \cdot (x + y^3)} + x \cdot z^4. \tag{4.3}$$

Then evaluate the expression for the values $x = 5$, $y = 2.1$ and $z = 4.6$.

★ **Exercise 6:** Consider a population of an annual plant, starting out with a single adult in year 1. Suppose that each adult plant generates on average 500 seeds, of which only 15 germinate and grow to adulthood in the following year (the remaining seeds die). Assuming no other losses or immigration, and using `python`, compute the expected number of adult plants in year 5.

## 4.5 Functions

A `python` function is a set of pre-defined instructions that are executed upon demand by writing out the name of the function. Functions typically take arguments that specify precisely what the function should do. Arguments are passed inside parentheses, after the function name. For example, the following code uses the function `print` to print a message to the screen:

```
print("Hello world")
```

producing the following printout:

```
Hello world
```

In the above example, we *call* the function `print` and pass to it the text "`Hello world`" as an argument. An argument serves as an "input" of information into the function. In the above example, the argument tells the `print` function what it is that we want to print. Many (but not all) functions take arguments, and in most cases the provided arguments are expected to be of a specific type (for example, a number, or a text, etc). If you are unsure what arguments a function expects, the best source of information is usually the function's official online documentation. A brief overview can also be obtained by calling the `help` function, with the name of the desired function as an argument, for example:

```
help(print)
```

The following code uses the function `abs` to compute the absolute value of some numerical variable `x`, and assigns the result to a new variable `y`:

```
y = abs(x)
```

Note that in contrast to the previous example, here we actually obtain a result from the function call (the absolute value of `x`), which we can assign to a variable (`y`). The result obtained from a function call is called the *returned value* of the function. Thus, arguments serve as "inputs" to a function while its returned value serves as an "output". Many functions such as `abs` return a value, while others such as `print` do not. Functions that don't return a value are used to make something *happen* (e.g., print or plot something), while functions that return a value are typically used to compute something (e.g., the result of a mathematical formula).

In both of the above examples we only passed a single argument to the function. However, in many situations more than one arguments may need to be passed. Multiple arguments must be separated by commas. For example, the following expression computes the minimum of 3 numbers and assigns the result to a new variable called `smallest_number`:

```
smallest_number = min(10, 5, 98)
```

Similarly:

```
largest_number = max(10, 5, 98)
```

The `print` function itself in fact also accepts multiple arguments, which can be used to print multiple pieces of text and variables on the same line. For example, the following function call:

```
print("The smallest number is:",smallest_number)
```

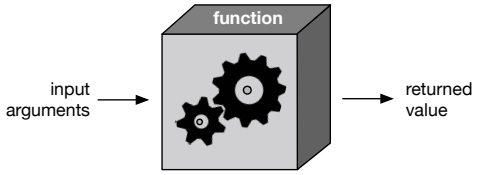
prints out the following to the screen:

```
The smallest number is: 5
```

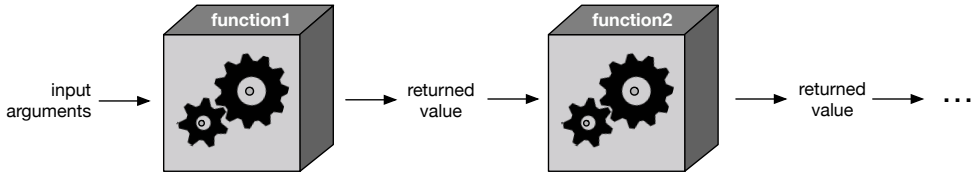
Some functions can also be used with no arguments at all. In that case, we still need to write the opening and closing parentheses right after the function's name. The following list summarizes the terminology introduced so far:

- Arguments are *passed* to the function, and the function may optionally *return* a value.
- Arguments serve as inputs to the function, while returned values serve as outputs.
- When a function is used in a statement (by writing its name followed by parentheses, possibly with arguments) we say that the function is *called*.
- Calling a function tells `python` to perform the function's internal computation.

Functions are meant to encapsulate and modularize complex computations, such that we can focus on the net/final result of these computations without worrying about the internal details each time. In that sense, functions are generally best seen as useful *black boxes*, whose internal workings are (most of the time) invisible to us. We don't need to know precisely how the `print` function works internally (it's actually complicated) — we just need to know that whatever we provide as an argument, will get printed to the screen.




It is also often possible to nest multiple functions and mathematical expressions, i.e., directly using the return value of one function (or the results of a mathematical expression) as an argument to another function.




For example, the following code computes and prints the absolute value of $55 - 298$:

```
print(abs(55-298))
```

Equivalently to the above code, we could have of course also written:

```
X = 55-298
Y = abs(X)
print(Y)
```

however nesting functions can make code more concise and efficient. Note that nested calculations are always performed in outward order, similarly to mathematics, i.e., first the mathematical

expression $55 - 298$ is evaluated, then the result is passed as an argument to the function `abs` to compute its absolute value, and then the result returned by the `abs` function is passed as an argument to the function `print`. For example, the following alternative code would be meaningless:

```
abs(print(55-298))
```

since the `print` function does not return any value, and even if it did, there would be little use in computing that returned value's absolute value.

Always remember to close as many brackets as you open (as in mathematics and human natural language). Unbalanced brackets will always cause an error, since `python` cannot know for sure what you really meant to say. For example, the following code would cause an error, since we forgot to close one of the brackets.

```
abs(print(55-298)
```

As another example, consider the following code, which opens a text file and loads its entire contents into a new variable:

```
fin = open("documents/some_text_file.txt", "rt")
contents = fin.read()
fin.close()
```

In the first line, we use the function `open` to open the file, passing two arguments: The 1st argument specifies the path of the file to read and the 2nd argument specifies how to open the file (`rt` means read-text mode). The order in which we pass arguments in the above example is important, i.e., we couldn't just swap the first and 2nd argument. The variable `fin` now represents the opened file, and is called a file object. In the 2nd line we are calling something that looks like a function (`read`) associated with a variable (`fin`), to load the file's contents into a new variable called `contents`. Lastly, we call the method `close` to close the file, thus telling the operating system that we are no longer interested in working with that file.

Technically speaking, `read` is called a method rather than a function, and its action always refers to the object that it is attached to (the file object `fin` in this case). Methods can take arguments and return values just like functions, and in some programming languages they are in fact called *member functions*. In some sense, the object that a method is attached to can be interpreted as a special argument, written in front of the method instead of inside the parentheses due to its special status. Methods are a central feature of the modern "object-oriented programming" paradigm, which however we won't be discussing in depth in this introductory book.

Functions and methods are an integral part of `python`; nearly all tasks are performed by functions and methods, and in fact most functions and methods in turn use multiple functions/methods internally to accomplish their tasks. For a list of built-in `python` functions see this webpage. To find out more about a specific method, we generally need to look at the documentation of the type of object it is associated with (e.g., strings, file objects, etc). Throughout this course, we will learn about many functions and methods for accomplishing specific computations, however many thousands more exist that we won't cover. We will soon also learn how to define our own functions.

**Named function arguments:** One of `python`'s nice features is that we can (almost always) pass arguments to functions using designated names (aka. "*keywords*"), rather than based on their relative positions. This is particularly useful for functions that may take multiple arguments, especially when some arguments are optional. For example, the `open` function that we used

earlier to open a file accepts a mandatory argument called `file` (specifying the file path), as well as multiple optional arguments, such as `mode` (specifying how to open the file), `buffering` (which provides some control over the internal buffering) and `encoding` (for handling files with non-standard encodings). To pass any of these arguments, we can refer to them explicitly by their names, for example:

```
fin = open(file="documents/some_text_file.txt", mode="rt", encoding="latin_1")
```

Passing arguments by name has multiple advantages: First, it clarifies their meaning, thus making our code more readable. Second, it allows us to omit certain arguments that would otherwise need to precede other later arguments so as to have the latter at their proper position. For example, without naming the arguments as above, we would need to write:

```
fin = open("documents/some_text_file.txt", "rt", -1, "latin_1")
```

(notice the extra `-1`, representing the default value of the optional `buffering` argument). Third, it allows us to specify arguments in an arbitrary order rather than worrying about their relative positions, for example as follows:

```
fin = open(encoding="latin_1", file="documents/some_text_file.txt", mode="rt")
```

In some situations some function arguments (typically the first one) are strictly positional, meaning that they have no name and must always be specified at their appropriate position. These situations are usually clarified in a function's documentation or through an error message (if you accidentally provide a name).

★ **Exercise 7:** Consider the following masses (in kg) of African elephant calves born in captivity [13]: 114, 91, 98, 109, 91, 104, 125, 128, 93, 106, 134, 118. Use the `min`, `max` and `abs` functions to compute the mass of the smallest calf, the mass of the largest calf, and the mass difference between the smallest and largest calf.

## 4.6 Using functions provided by modules and packages

The vast majority of functions in `python` are provided through additional modules and packages, either included with our standard `python` installation or installed separately. The difference between modules and packages is not very important at this point, so for now you should just consider both to be a type of container of additional functionalities. To access the functionalities of an installed `python` module or package, we first need to "import" it into our `python` session. For example, the following line of code imports the module `math`:

```
import math
```

Importing the `math` module gives us access to a multitude of common mathematical functions, such as trigonometric functions (e.g., `sin`, `cos` etc), the exponential function (`exp`), logarithm (`log`) and the square root (`sqrt`). To call a function from a specific module we use the module and function names, separated by a dot. For example, to compute the exponential $e^{0.5}$ and assign the result to a new variable `E`, we can write:

```
E = math.exp(0.5)
```

To get access to more sophisticated mathematical functions, we can install and import the package `numpy`:

```
import numpy
```

In some cases we may only want to load the parts of a package that we actually need. For example, the following code imports from `numpy` only functionalities related to linear algebra and random number generation:

```
from numpy import linalg, random
```

Technically, the parts `linalg` and `random` are "modules" of the `numpy` package (so essentially a package can comprise multiple modules). To access a function inside an explicitly imported module, we can use the module and function names separated by dots, as before. For example, the following code will generate a random number, uniformly between 0 and 100, and assign it to the new variable `r`:

```
r = random.uniform(0, 100)
```

If we had not explicitly imported the `random` module, but instead simply imported the entire `numpy` package, we would need to write:

```
r = numpy.random.uniform(0, 100)
```

Generating random numbers, which we will discuss in detail later on, is useful for statistical hypothesis testing, estimating confidence intervals, certain optimization algorithms and simulating stochastic models [14–17].

Other commonly used modules and packages that we will encounter in the course include `os` and `sys` (for interacting with the operating system), `string` (for working with text), `datetime` (for working with dates) and `matplotlib` (for plotting).

★ **Exercise 8:** Consider a single bacterial population growing on an unlimited carbon source at a per-capita growth rate $r$. If $N_o$ denotes the initial cell count at time 0, then at any future time $t$ the cell count is given by the exponential function, $N(t) = N_o e^{r \cdot t}$. Assuming that $r = 0.2$ day$^{-1}$ and $N_o = 10^6$, use `python` to compute the cell counts after $t = 5$ days and after $t = 20$ days.

*Hint: You can use the* ***`exp`*** *function in the* ***`math`*** *module to compute exponentials.*

★ **Exercise 9:** Suppose that we have tracked a migratory falcon throughout a year, and have determined its summer nesting coordinates to be 127.483226°W 51.866239°N (West Canada),

and its winter location coordinates to be 103.487167°W 19.685403°N (Mexico). Use the following (Vincenty) formula to compute the great-circle distance (i.e., shortest-path distance) between these two locations:

$$D = R \cdot \arctan 2(A, B), \tag{4.4}$$

where we defined:

$$\begin{aligned} A &:= \sqrt{(\cos(y_2)\sin(\delta))^2 + (\cos(y_1)\sin(y_2) - \sin(y_1)\cos(y_2)\cos(\delta))^2}, \\ B &:= \sin(y_1)\sin(y_2) + \cos(y_1)\cos(y_2)\cos(\delta), \end{aligned} \tag{4.5}$$

and where $x_1, y_1$ are the longitude and latitude of the summer location, $x_2, y_2$ are the longitude and latitude of the winter location, $\delta = |x_1 - x_2|$, $R = 6371$ km is Earth's average radius, and $\arctan 2$ is the 2-argument arctangent.

*Hint: The required trigonometric functions* `sin`, `cos`, `atan2` *and the constant* `pi` *are all available in the* `math` *module. Keep in mind that the functions* `sin` *and* `cos` *expect angles to be given in radians, rather than degrees; to convert angles from degrees to radians, multiply them by* $\pi/180$.

## 4.7 Strings

### 4.7.1 Introduction

In addition to numbers, `python` variables can also be used to represent text; such variables are called strings. The word "string" is a programming term for a sequence of characters. In technical language, a string is another data type, just like floats and integers are special data types. Strings may be used to represent, for example, DNA sequences(e.g., ACGAATC...), gene or species names, a list of coordinates recorded by an animal GPS tracker, file paths or sample names, and so on, and have a myriad of applications. We have already encountered strings in Section 4.5, where we printed the string "`Hello world`" to the screen. Strings can be explicitly defined using single or double quotes, but may also be generated by some more involved computation. For example, the following code assigns a species name to a variable:

```
full_species_name = "Escherichia coli"
```

Here, `full_species_name` is the name of a string variable, and "`Escherichia coli`" is the value of that variable. Note that quotes are important when explicitly specifying a string's value! The following code would generate an error, since `python` would be pointlessly looking for a variable called `Escherichia`:

```
full_species_name = Escherichia coli
```

A string variable's name can (and usually does) differ from its value. To quickly inspect the value of a string variable, we can print it to the screen as follows:

```
print(full_species_name)
```

```
Escherichia coli
```

To determine the length of a string (i.e., the number of characters), we can use the function `len`, for example:

```
print("Length of full species name:",len(full_species_name))
```

```
Length of full species name: 16
```

In practice, we rarely explicitly specify the full value of a string as above. Instead, strings are often created as the result of some computation or loaded from a data file. For example, we can concatenate strings using the addition symbol, as follows:

```
genus_name = "Escherichia"
species_name = "coli"
full_species_name = genus_name + " " + species_name
```

By the same reasoning, we can also multiply a string with a positive integer $n$, which essentially repeats the string $n$ times:

```
codon = "CTA"
multiple_codons = codon * 10
print(multiple_codons)
```

```
CTACTACTACTACTACTACTACTACTA
```

Many more useful computations can be performed with strings, such as searching for a piece of text inside another text, extracting parts of a text, looking for complex patterns in text, replacing certain character sequences with others, splitting a comma-separated list of names at each comma, or joining a large list of text pieces. For an overview of common string operations see this official documentation.

**Accessing segments of a string:** A common need when working with strings is the ability to extract a specific segment from a string. For example, we may have a collection of gene nucleotide sequences, from which we wish to omit the start and end codons. Or we may have a list of species names that we wish to abbreviate. We can extract a segment of a string by specifying the position of the first character (counting from 0) and the position past the last character (also counting from 0) to extract, separated by a colon and enclosed in square brackets. The following code extracts the first 5 characters of `genus_name` as a new string and prints it to the screen:

```
genus_abbrev = genus_name[0:5]
print(genus_abbrev)
```

```
Esche
```

The square brackets `[]` are called the index operator, and the colon-separated pair of integers

`0:5` specifies the range of character positions to extract. Similarly, the following code extracts and prints out the third, fourth, fifth and sixth characters of `genus_name`:

```
segment = genus_name[2:6]
print(segment)
```

```
cher
```

To extract a single character, we simply provide its position without a colon, for example:

```
first_character = genus_name[0]
print(first_character)
```

```
E
```

We can also use negative positions, which are counted backwards from the end. For example, the following code prints out all but the last two characters of `genus_name`:

```
print(genus_name[0:-2])
```

```
Escherich
```

while the following code prints out the last character of `genus_name`:

```
print(genus_name[-1])
```

```
a
```

**Strings are not numbers:** It should be noted that while strings may in principle also encode numbers, they are not automatically interpreted in a numeric fashion. For example, a string `"1.5"` is not equivalent to a numeric `1.5`. Hence the following expression yields the string `"1.52.0"`:

```
"1.5" + "2.0"
```

while the following expression yields the number 3.5:

```
1.5 + 2.0
```

This is because strings are a fundamentally different type of data than numbers, and `python` always assumes that we mean to treat strings as text, regardless of their content. If we want to interpret a string numerically, we first need to convert it to a number, for example using the `float` or `int` functions. The function `float` converts strings to floating point numbers, while `int` converts strings to integers. `int` can thus only be applied to strings that represent a true integer (e.g., `"4"` but not `"4.5"`), and non-integer-representing strings will generate an error. The following code demonstrates their use:

```
print(float("1.5") + float("2.0"))
print(int("1") + int("4"))
```

```
3.5
5
```

★ **Exercise 10:** Predict the output of the following code without running it:

```
aminoacid_sequence = "ARNDARCLFPTVTSO"
print(aminoacid_sequence[3:6])
print(aminoacid_sequence[1:-1])
print(aminoacid_sequence[-2])
```

★ **Exercise 11:** Which of the following lines of code are in principle valid, i.e., will not cause an error? For the valid ones, can you predict the outcome? If you are unsure, try running them in `python`.

```
x = "10" + 5
x = 10 + 5 + "20"
x = float(5) + "10"
x = float("5") + 10
x = "1" + "2" + 3
x = "10" + "5"
x = 1 + 2 + 3
"x" = "10"
"10" + "5" = x
x = int("4.5") + 1
x = int("4") + 1
x = "3 + 4
```

### 4.7.2 Counting substring occurrences

Suppose that we have tracked the geographic location of an animal over time, recording the corresponding forest region name each day:

```
locations = "N valley, S valley, S valley, SE peak, E ridge, SE peak, N valley"
```

Suppose that we wanted to count how many days the animal was in "S valley". To achieve this, we can use the string method `count`, as follows:

```
Ndays = locations.count("S valley")
```

Observe that since the method `count` is by its very nature associated with the string variable `locations`, there is no need to specify the latter as an additional argument; we only need to specify the substring that we want to count. By the same logic, we can determine how often the animal moved directly from "S valley" to "SE peak":

```
Ntransitions = locations.count("S valley, SE peak")
```

Note that, as with variable names, string operations in `python` are generally case-sensitive. For example, the following code would not return the expected result:

```
Ntransitions = locations.count("s Valley, se Peak")
```

Using computer programming may seem like an overkill in these simple examples, however realistic datasets can be thousands of times larger, and we may need to perform this task for many more than just one forest region.

★ **Exercise 12:** In this exercise we will practice counting string occurrences. We will consider the following text file containing several animal species names:

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`animal_species_names_1000.txt`".

You can open this text file in any regular text editor, and you will notice that it lists one latin binomial name (e.g., `Leucascus albus`) per line. Download the above file, load its contents into `python` as a string (as demonstrated in Section 4.5) and count the number of times the genus *Sycon* is represented.

*Hint: Make sure not to accidentally count other genera that may begin with "Sycon".*

### 4.7.3 Replacing substrings

Another common task is to replace certain character sequences ("substrings") with others, or to simply remove certain substrings altogether. For example, suppose that we have a string variable containing a DNA sequence, such as the following:

```
DNA = "TATCGGCATCTACTAATCCGACTTTCCCTTCACGAGCTAAGGCAACGGCTAAATTTACTGCAACAGTAGCGA"
```

Suppose that we wanted to compute the complement of this sequence, thus replacing `A` with `T` (and vice versa), and replacing `G` with `C` (and vice versa). This can be achieved using the string method `replace`, as follows:

```
# swap A <--> T
complement = DNA.replace("A","a") # temporarily replace A with a
complement = complement.replace("T","A") # replace T with A
complement = complement.replace("a","T") # replace a with A

# swap G <--> C
complement = complement.replace("G","g") # temporarily replace G with g
complement = complement.replace("C","G") # replace C with G
complement = complement.replace("g","C") # replace g with C
```

The `complement` variable will now contain the complement DNA sequence:

```
ATAGCCGTAGATGATTAGGCTGAAAGGGAAGTGCTCGATTCCGTTGCCGATTTAAATGACGTTGTCATCGCT
```

Note that the string method `replace` always returns a modified copy of the original string, and does not actually modify the original; this is why we need to re-assign its return value to a variable each time. For example, in the following code the method `replace` temporarily creates a modified version of `DNA` (replacing `A` with `a`) and returns it, but since the returned value is not assigned to any variable, it is immediately lost.

```
DNA.replace("A","a")
```

Since `replace` returns the modified string, we can chain multiple `replace` calls to achieve the same outcome as above, but with fewer lines of code:

```
# swap A <--> T
complement = DNA.replace("A","a").replace("T","A").replace("a","T")

# swap G <--> C
complement = complement.replace("G","c").replace("C","G").replace("c","C")
```

Chained method calls such as the above are executed from left to right, i.e., first `A` is replaced with `a`, thus returning a new string, then `T` is replaced with `A` in this new string, again returning a new string, then `a` is replaced with `T` in this new string, again returning a new string, which is then assigned to the variable `complement`.

★ **Exercise 13:** Consider the example discussed in Section 4.7.3, where we computed the complement of a DNA sequence. Why would the following alternative shorter code not accomplish the same task?

```
complement = DNA.replace("A","T") # replace A with T
complement = complement.replace("T","A") # replace T with A
complement = complement.replace("G","C") # replace G with C
complement = complement.replace("C","G") # replace C with G
```

*Hint: Try running this code to see what's wrong with the result.*

### 4.7.4 String formatting (C-style)

A common use of strings is for printing specifically formatted information to the screen (i.e., messages to the user) or into a file. For example, a program that performs some computations for a dataset of thousands of bacterial genomes may print out progress reports and summaries of results for each genome processed. Suppose that we wanted to print a message to the screen that reports the species name for a genome, the length of the genome (in basepairs), and the frequency of adenines (A) in it, i.e., something like [18]:

```
Genome of Ecoli_K12 has length 4639221, adenine fraction 0.2452
```

Further, suppose that we have stored the species' name in a string variable `name`, its length in a numerical variable `L` and its adenine fraction in a numerical variable `A`:

```
name = "Ecoli_K12"
L    = 4639221
A    = 0.24527
```

How could we construct a string such as the above, displaying the information contained in the variables `name`, `L` and `A`? There are multiple alternative ways to achieve this. The "old" or "C-style" way, more familiar to users of C-style programming languages, uses special format

specifiers placed inside some text (called the format string), with values corresponding to the format specifiers provided as a list of arguments. Specifically in this example:

```
TX = "Genome of %s has length %d, adenine fraction %f"%(name,L,A)
print(TX)
```

Here, `"Genome of %s has length %d, adenine fraction %f"` is the format string. The format specifier `%s` tells `python` to insert another string (the species name) at the specified location, `%d` tells `python` to insert an integer (the genome length) and `%f` tells `python` to insert a fractional number (the adenine fraction). The three required values (species name, genome length, adenine fraction) are then provided as comma-separated arguments following a `%` operator at the end of the format string. The `%` operator tells `python` to insert the values of the provided variables (`name`, `L`, `A`) into the appropriate locations in the format string (i.e., at `%s`, `%d`, `%f`). The result of this operation (a new string) is then assigned to the variable `TX`. Note that the format specifiers (`%s`, `%d` and `%f` in this example) must generally match the type and order of variables provided at the end. For example, the following code would cause an error, because the first format specifier (`%d`) represents an integer whereas we are providing a string (`name`):

```
TX = "Genome of %d has length %s, adenine fraction %f"%(name,L,A)
```

Note that the `%f` format specifier may sometimes print out an excessive amount of decimal digits. To specify a desired number of decimal digits, we do so between the `%` and `f` symbols, e.g., as `%.3f` (3 digits) or as `%.10f` (10 digits). Further, to include a literal percentage sign (`%`), we need to write `%%` so that `python` does not confuse it with an intended format specifier. For example, to print the adenine fraction as a percentage with 1 decimal digit, we can write:

```
print("Adenine fraction %.1f%%"%(100*A))
```

```
Adenine fraction 24.5%
```

Many more format specifiers are available, for example for displaying numbers in scientific notation, all of which can be reviewed here. One special character that finds frequent use when formatting strings is the "newline" character, represented by a `\n`, which is used to define line breaks. For example, suppose we wanted to print to the screen the following 3-line message:

```
Genome of Ecoli_K12:
  Length: 4639221
  Adenine fraction: 0.245270
```

To achieve this, we would insert a newline character at the appropriate positions:

```
print("Genome of %s:\n  Length: %d\n  Adenine fraction: %f"%(name,L,A))
```

Apart from printing informative messages to the screen, string formatting is also often used to write specifically formatted contents to data files, such as tables in CSV (comma-separated values) or TSV (tab-separated values) format. We will learn how to write text to files later on.

★ **Exercise 14:** In this exercise we will practice C-style string formatting. Let us define the following `python` variables:

```
Nbeetles          = 105
```

```
average_weight    = 1.6487135
country_of_origin = "Kenya"
```

Using C-style string formatting and the function `print`, and using the above variables as string formatting arguments (i.e., without explicitly writing their values into the format string), print the following to the screen:

```
Across 105 examined beetles from Kenya, the average weight was 1.649 g.
```

Do pay attention to the number of displayed decimal digits.

★ **Exercise 15:** Consider the following `python` variables, representing the mass (gram) and basal metabolic rate (Watt) of a *Pinus massoniana* pine tree [19]:

```
mass = 596599.7
metabolic_rate = 48.65
```

Write a single `print` statement that uses the above variables and C-style string formatting to generate the following output:

```
Pinus massoniana:
  Mass: 5.97e+05 gram
  Basal metabolic rate: 48.6 Watt
```

*Hint: The notation* `%e+y`*, which represents* $x \times 10^y$*, is called scientific notation (aka. exponent notation), and can be achieved using the* `%g` *format specifier. See here for instructions.*

### 4.7.5 String formatting (new methods)

With the release of `python` 3, new alternative approaches to formatting strings were introduced. Consider, as in an earlier example, the following text:

```
Genome of Ecoli_K12 has length 4639221, adenine fraction 0.24527
```

which we would like to assign to a string variable called `TX`, based on the data stored in the variables `name`, `L` and `A`. To achieve this, we can use the string method `format`, which uses explicit references to values (called replacement fields) at the positions in which they should be inserted, as follows:

```
TX = "Genome of {0} has length {1}, adenine fraction {2}".format(name,L,A)
```

Here, the replacement fields `{0}`, `{1}` and `{2}` refer to the 3 arguments passed to the `format` method, i.e., `name`, `L` and `A`, respectively. Observe that every time we want to insert a variable's value into the text, we are explicitly referring to the variable's position in the argument list. Because we are explicitly referring to a variable's position, the order of the provided variables does not need to match the order in which their corresponding replacement fields appear in the format string, i.e., the following code is equivalent to the above:

```
TX = "Genome of {1} has length {2}, adenine fraction {0}".format(A,name,L)
```

As an alternative, we could provide the three variables as a *named* tuple and then refer to the variables by name rather than by position:

```
TX = "Genome of {na} has length {le}, adenine fraction {ad}"\
      .format(na=name,le=L,ad=A)
```

Hence, the replacement field `{na}` refers to the value associated with the name `na` in the argument list. Since references are name-based, we can also change the order in the argument list, for example the following code would produce the same output:

```
TX = "Genome of {na} has length {le}, adenine fraction {ad}"\
      .format(le=L,ad=A,na=name)
```

In the above example we chose to use the short names `na`, `le` and `ad`, but any other naming scheme could have been used instead, as long as the referenced variable names match, for example:

```
TX = "Genome of {some_name} has length {len1}, adenine fraction {aden_frac}"\
      .format(some_name=name,len1=L,aden_frac=A)
```

Note that instead of existing variables, we could have also used `python` expressions, such as mathematical formulas. For example, suppose that we are computing the fraction of adenines as the ratio between the number of adenines (`Naden`) divided by the genome size (`L`), i.e., `Naden/L`. In that case, we could have also written:

```
TX = "Genome of {na} has length {le}, adenine fraction {ad}"\
      .format(na=name,le=L,ad=Naden/L)
```

You probably noticed by now that in the above examples we have not been telling `python` how to format the various variables passed (e.g., as integers, fractionals etc), thus leaving this decision to `python` (which may not always be what we want). To control the precise formatting style, we can include additional format specifiers in the replacement fields. For example, to print out the adenine fraction with only 2 decimal digits, we would write:

```
print("Adenine fraction {ad:.2f}".format(ad=Naden/L))
```

```
Adenine fraction 0.25
```

Additional variants of the above string formatting methods exist (described here), although for nearly all purposes the above approaches are adequate. Whether one uses the old (C-style) or new methods to format strings is largely a matter of preference, and each may have its benefits depending on the situation.

★ **Exercise 16:** In this exercise we will practice string formatting using the `format` method. Let us define the following `python` variables:

```
Nbeetles        = 105
average_weight  = 1.6487135
country_of_origin = "Kenya"
```

Using the string method `format` and the function `print`, and using the above variables (i.e., without writing their values into the format string), print the following to the screen:

```
Across 105 examined beetles from Kenya, the average weight was 1.649 g.
```

Do pay attention to the number of displayed decimal digits.

★ **Exercise 17:** Consider the following `python` variables, representing the basal metabolic rate (Watt) and mass (gram) of a *Pinus massoniana* pine tree [19]:

```
mass = 596599.7
metabolic_rate = 48.65
```

Write a single `print` statement that uses the above variables and the `.format` string method to generate the following output:

```
Pinus massoniana:
  Mass: 5.97e+05 gram
  Basal metabolic rate: 48.6 Watt
```

*Hint: The notation* `xe+y`*, which represents* $x \times 10^y$*, is called scientific notation (aka. exponent notation), and can be achieved using the* `g` *format specifier. See here for instructions.*

★ **Exercise 18:** In this exercise we will consider the genome sequence of a strain of *Staphylococcus aureus*, a common human pathogen. The genome can be downloaded from the official NCBI database (accession number `GCF_000698045.1`) as well as from the following location:

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`Staphylococcus_aureus_A69_GCF_000698045.1.fasta`".

Note that while the file name has the extension `.fasta`, it is actually a plain text file, whose contents are formatted in a way suitable for storing DNA sequences. We can in fact open this file in any common text editor. The first few lines are shown below as example:

```
>NZ_JJOP01000001.1 Staphylococcus aureus strain A69 scaffold1_16.0_28093_28093
CGCCGCCAGCGTTCATCCTGAGCCAGGATCAAACTCTCCATAAAAATTATGATGTTTGATTAGCTCATAAATACTAAATA
ATGTTTGTAACTTATAGTTACGTTTTTTGGAATTAACGTTGACATATTGTCATTCAGTTTTCAATGTTCATTAATGTTCA
ATCTCTTTTATTCTACTTCATTTGTTTCTTGAAGTCAATAACTTTTTGAACGATTACTTTATTTATTCTATATTGTTTTT
ATAGTTATTTCAATGGTAAGTTTTACACTTTTGAAATCCTTCTTTAAAAACAACTGCGTCGTTTTTGACGACTTTATCAT
ATTATCAACTTTGGAATTTAAAGTCAATAACTTTTTTTAAACTTTTTGTGTTCACAACCCGCTTCTTTTTCAACGCGTTT
```

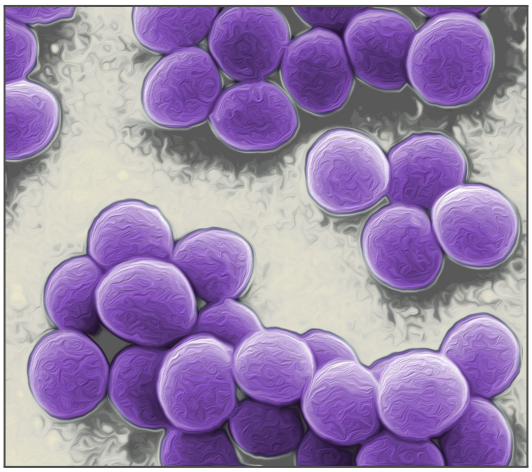
Observe that there are two types of lines in fasta files, those containing actual DNA sequences (e.g., `ATGTTGTCA..`), and those containing various metadata and starting with a `>` character. For more information on fasta text files see this wikipedia entry. Perform the following tasks:

1. After downloading the above fasta file, load its entire contents into a single string variable, as demonstrated in Section 4.5. Print out the total number of characters loaded (i.e., the length of the string) in a nicely formatted informative message, using the C-style approach discussed in Section 4.7.4 as well as the newer approach discussed in Section 4.7.5.

   *Hint: You can use the function* `len` *to compute the length of a string.*

2. Use the string member function `count` introduced in Section 4.7.2 to determine how often the genome contains the nucleotide subsequences `AAAAAAAA` and `AACCTTGG`.

   *Hint: Don't worry about the fact that some parts of the fasta file are metadata and not actually DNA sequences; none of these will contain the subsequences that you are searching for.*



## 4.8 If statements

An important aspect of any programming language is the ability to perform some computations only if a certain condition is met, i.e., to "control the flow" of a program. For example, we may have a large dataset of camera trap images, and we wish to only attempt an automated species identification in those images that meet certain quality criteria (e.g., sufficient brightness or resolution). To only run a piece of code if a certain condition is met, we can use the `if` statement. For example, suppose that have a numeric variable, `brightness`, and a string variable, `image_name`, and suppose that we want to print the image's name to the screen only if the brightness exceeds a certain threshold, say 0.5. Then this can be achieved as follows:

```
if(brightness>0.5):
    print(image_name)
```

For example, the following code will print out the image's name, because `brightness` is greater than 0.5:

```
brightness = 0.7
image_name = "Yellowstone, camera A45, photo 453"
if(brightness>0.5):
    print(image_name)
```

```
Yellowstone, camera A45, photo 453
```

In contrast, the following code will not print out the image's name, because `brightness` is not greater than 0.5:

```
brightness = 0.3
image_name = "Yellowstone, camera A45, photo 453"
if(brightness>0.5):
    print(image_name)
```

Let's look at the `if` statement a bit closer:

```
if(brightness>0.5):
    print(image_name)
```

The expression inside the round brackets on the first line specifies the condition under which

the `print` statement is executed. Note that the inclusion of a colon (`:`) after the condition is mandatory. Also observe that we have written the conditional code further inwards compared to the condition line. This is called *indentation*, and it allows `python` to know which part of the code should only be executed if the preceding condition is satisfied. This becomes particularly useful if the conditional code spans multiple lines. Any line that is not indented is considered to be outside of the `if` statement. For example, in the following code the second `print` statement is always executed, regardless of the condition in the preceding `if` statement, while the two indented statements are only executed when `brightness` is greater than 0.5:

```
brightness = 0.3
image_name = "camera A45 - photo 453"
if(brightness>0.5):
    darkness = 1 - brightness
    print("%s has brightness %g, darkness %g"%(image_name,brightness,darkness))
print("This statement is always executed")
```

```
This statement is always executed
```

Indentation should either be done using a single "tab" character (typically the key above the caps-lock on your keyboard), or using a fixed number of space characters (typically 4 spaces); whichever convention you decide to use, you must be consistent throughout your entire code. All examples in this course use the tab character for indentation.


**Figure 4.2:** Photo of a puma, taken by an auto-activated wildlife camera trap in the Bosque del Apache National Wildlife Refuge. Credit J.N. Stuart, Creative Commons. Typical camera trap datasets can include tens of thousands of photos, the manual inspection of which can be very laborious; great efforts are thus being put into developing computational algorithms for automated analysis of such data [20–22].

**Alternatives with `else` and `elif`:** If we wanted to execute an alternative code whenever `brightness` is not greater than 0.5, we can use the `else` keyword (again followed by a colon), as follows:

```
brightness = 0.3
image_name = "camera A45 - photo 453"
if(brightness>0.5):
    darkness = 1 - brightness
    print("%s has brightness %g, darkness %g"%(image_name,brightness,darkness))
else:
    print("%s does not meet the brightness threshold"%(image_name))
```

```
print("This statement is always executed")
```

```
camera A45 - photo 453 does not meet the brightness threshold
This statement is always executed
```

Further, we can specify alternative conditions if previous conditions are not satisfied using the `elif` keyword (which stands for else-if or "otherwise if"), as in the following example:

```
brightness = 0.3
image_name = "camera A45 - photo 453"
if(brightness>0.5):
    print("%s brightness exceeds 0.5"%(image_name))
elif(brightness>0.2):
    print("%s brightness exceeds 0.2, but not 0.5"%(image_name))
elif(brightness>0.01):
    print("%s brightness exceeds 0.05, but not 0.2"%(image_name))
else:
    print("%s brightness does not exceed 0.01"%(image_name))
```

Note that an `elif` condition is only considered if all previous `if` and `elif` conditions turned out to be false, in other words `python` will only execute the conditional code associated with the first encountered true condition. Hence, the above code will produce the following output:

```
camera A45 - photo 453 brightness exceeds 0.2, but not 0.5
```

**Additional condition types:** In addition to the greater-than comparison (`>`), `python` also supports other common mathematical comparisons between numbers, including less-than (`<`), equal-to (`==`, note the double equal sign), greater-or-equal (`>=`), less-or-equal (`<=`) and not-equal (`!=`). For other types of variables, these comparisons may or may not make sense. For strings, these comparisons are conveniently interpreted in the context of lexicographic (aka. "alphabetical") order. For example, the following code determines which of two string variables precedes the other alphabetically:

```
name1 = "Escherichia coli"
name2 = "Staphylococcus aureus"
if(name1>name2):
    print("'%s' comes after '%s'"%(name1,name2))
else:
    print("'%s' comes before '%s'"%(name1,name2))
```

```
'Escherichia coli' comes before 'Staphylococcus aureus'
```

Other useful string-based conditions are whether a string starts or ends with some other string (`startswith` and `endswith`) or contains another string (`in`). For example, the following code:

```
name1 = "Escherichia coli"
if(name1.startswith("Escherichia")):
    print("'%s' is in genus Escherichia"%(name1))
else:
    print("'%s' is not in genus Escherichia"%(name1))

image_name = "camera A45 - photo 453"
if("camera A45" in image_name):
```

```
    print("Image '%s' was taken by camera A45"%(image_name))
```

produces the following output:

```
'Escherichia coli' is in genus Escherichia
Image 'camera A45 - photo 453' was taken by camera A45
```

**Nested if statements:** It is also possible to enclose ("nest") an `if` statement into another `if`, `else` or `elif` statement, using appropriate indentation. As an example, consider the case of camera trap images discussed earlier. Suppose that we want to perform a certain computation only when `brightness>0.5`, however the precise type of the computation depends on whether another parameter, `resolution`, is above or below 100. The general structure of our code would then look as follows:

```
if(brightness>0.5):
    if(resolution>100):
        # this code is only executed if brightness>0.5 and resolution>100
        ...
    else:
        # this code is only executed if brightness>0.5 and resolution<=100
        ...
```

Observe that the code conditioned by the inner (nested) `if` statement must be further indented compared to the code solely conditioned on the outer `if` statement. Essentially the inner `if` statement is considered just like any other `if` statement but only if the condition `brightness>0.5` is satisfied. As an example, consider the following two variables:

```
brightness = 0.7
resolution = 60
```

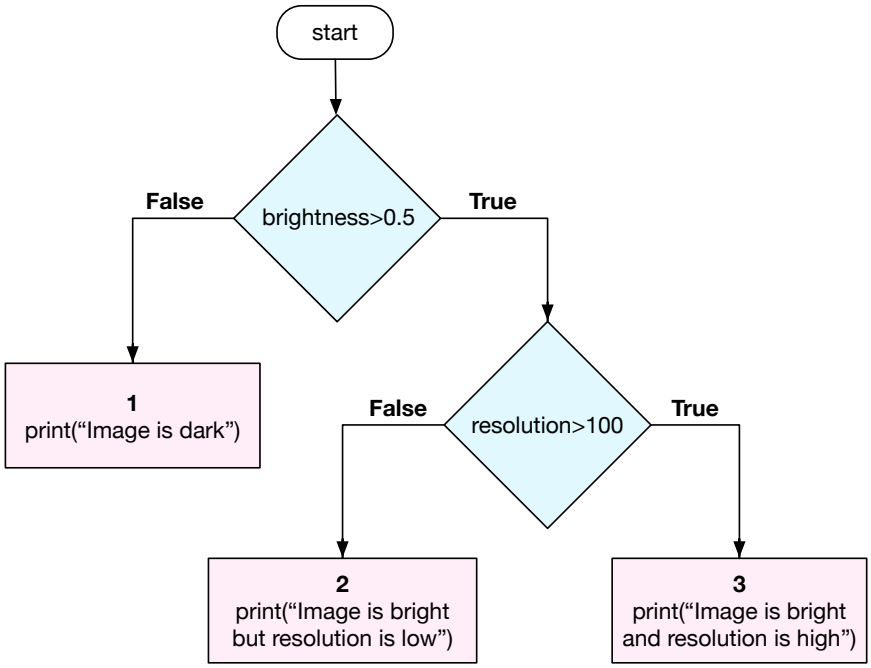
and the following nested `if` statements:

```
if(brightness>0.5):
    if(resolution>100):
        print("Image is bright and resolution is high")
    else:
        print("Image is bright but resolution is low")
else:
    print("Image is dark")
```

The above code generates the following output:

```
Image is bright but resolution is low
```

To understand what happens, it is useful to consult the corresponding flow diagram below:

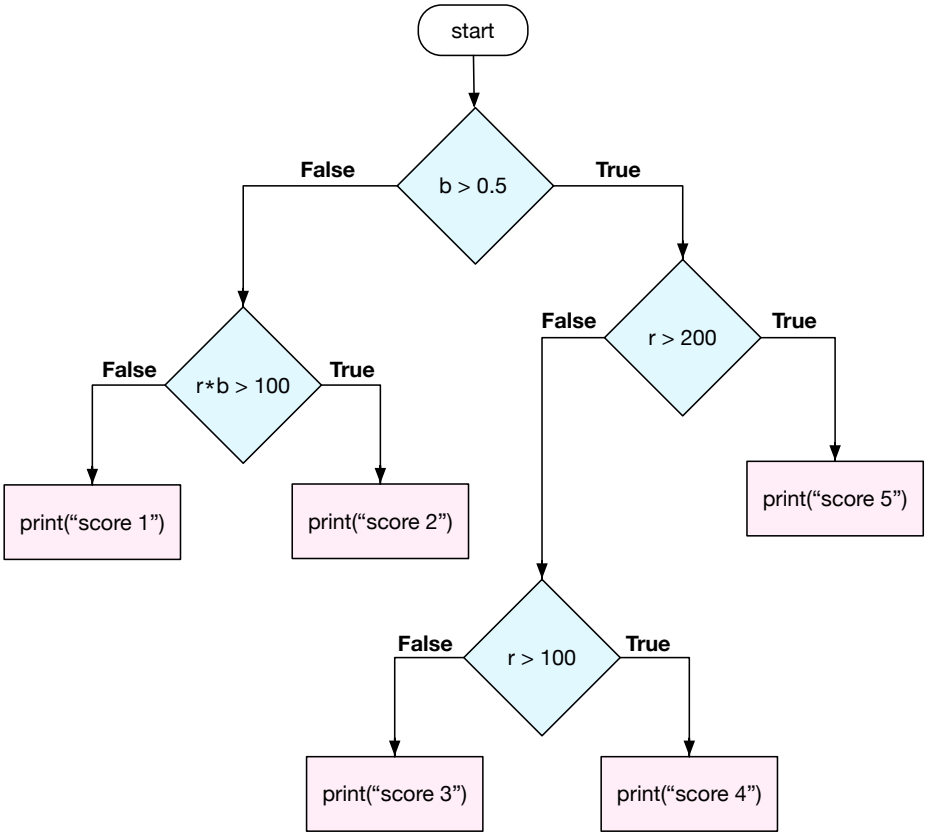



First, `python` checks if `brightness` is greater than 0.5, which is `True`. This leads `python` to the second (nested) `if` statement, to check whether `resolution` is above 100, which is `False`; this leads `python` to `print` statement 2, which is the only one executed.

★ **Exercise 19:** Predict the full output of the following code, without running it.

```
DNA = "ACTGCTTGAACGGTACACAACCC"
print("DNA: %s"%(DNA))
if(DNA.startswith("ACTCC")):
    print("First DNA condition is satisfied")
elif("ACTCC" in DNA):
    print("Second DNA condition is satisfied")
else:
    print("None of the two DNA conditions are satisfied")
print("Unconditional 1")

gene_size = 2100
print("gene size:", gene_size)
if(gene_size>2200):
    print("First gene-size condition is satisfied")
elif(gene_size<2100):
    print("Second gene-size condition is satisfied")
elif(gene_size<=2000):
    print("Third gene-size condition is satisfied")
else:
    print("None of the gene-size conditions are satisfied")
    if(DNA.startswith("ACT")):
        print("  However, the DNA condition is satisfied")
    else:
        print("  Neither is the DNA condition satisfied")
print("Unconditional 2")
```

★ **Exercise 20:** Consider the following flow diagram, which determines a score for a camera trap image based on its brightness (`b`) and resolution (`r`):




Write a code using nested `if` statements that corresponds to the above flow diagram. Execute your code using the following brightness and resolution values:

```
b = 0.6
r = 150
```

★ **Exercise 21:** Suppose that we have a numeric variable called `abundance`. Write a `python` code using `if` and `elif` statements that prints out the message "`is above 10`" if `abundance` is above 10, the message "`is between 5 and 10`" if `abundance` is above 5 and not above 10, and the message "`is between 0 and 5`" if `abundance` is above 0 and not above 5. Run your code with some example values of `abundance` to make sure it works as intended.

★ **Exercise 22:** Suppose that we have a string variable called `DNA` and representing a nucleotide sequence, i.e., consisting exclusively of the characters `A`, `C`, `G`, `T`. Write a `python` code using `if`, `else` and/or `elif` statements that determines whether the DNA sequence contains a segment of at least 5 consecutive `A` (e.g., `AAAAA`, or `AAAAAA` etc) but no more than 10 consecutive `A`. The code should print out the message "`yes`" if both conditions are satisfied, and "`no`" otherwise. Run your code with some example values of `DNA` to make sure it works as intended.

## 4.9 Booleans

We have seen in the previous section how to condition the execution of some code on a certain logical condition. As we will see later, it is possible and often necessary to formulate much more complicated logical conditions than in our examples so far. The representation and evaluation

of logical conditions in programming languages is formalized using boolean variables. A boolean is a type of variable that can only hold two possible values, `True` or `False` (you can also think of those as 1 and 0, respectively). In the following `if` statement:

```
if(brightness>0.5):
    print(image_name)
```

the expression `brightness>0.5` actually evaluates into a boolean that is either `True` or `False`, depending on the value of `brightness`. It is the value of that boolean that ultimately determines whether the conditioned `print` statement is executed or not. The following code is thus equivalent to the above:

```
mycondition = (brightness>0.5)
if(mycondition):
    print(image_name)
```

Representing the value of each logical condition as a boolean variable allows `python` to perform algebraic operations between them, similarly to those known from propositional logic in philosophy or boolean algebra in mathematics. For example, suppose that we have two variables that specify the quality of a camera trap photo, `brightness` and `resolution`, and we wanted to only perform a certain analysis if `brightness>0.5` and `resolution>100`. We thus need to combine two logical conditions into a composite condition that only evaluates to `True` if both `brightness>0.5` and `resolution>100` are `True`. This can be achieved using the `and` operator:

```
mycondition = (brightness>0.5) and (resolution>100)
if(mycondition):
    # perform some conditional computation
```

The `and` operator thus takes two boolean variables and yields a new boolean variable, whose value is only `True` if the two input variables were both `True`. Another common boolean operator is `or`, which evaluates to `True` if any one (or both) of the two input booleans are `True`. For example, of the following expressions the first 3 will evaluate to `True`, while the last one evaluates to `False`:

```
(10>4) or (5>0) # this evaluates to True
(10>4) or (5<0) # this evaluates to True
(10<4) or (5>0) # this evaluates to True
(10<4) or (5<0) # this evaluates to False
```

Hence, if we wanted to perform a certain conditional computation if either `brightness>0.5` or `resolution>100` (or both) are satisfied, we could write:

```
if((brightness>0.5) or (resolution>100)):
    # perform some conditional computation
```

A third important boolean operator is `not`, which is used to negate a certain condition (i.e., turn `True` to `False` and vice versa). Hence, the following two expressions are equivalent:

```
brightness>0.5
not (brightness<=0.5)
```

For example, the following code:

```
DNA = "AACGTCTACACGTA"
```

```
if(not DNA.startswith("AACGT")):
    print("DNA does not start with AACGT")
elif(not DNA.endswith("CGTA")):
    print("DNA does not end with CGTA")
else:
    print("DNA starts with AACGT and ends with CGTA")
```

generates the following output:

```
DNA starts with AACGT and ends with CGTA
```

Table 4.1 below summarizes the algebraic rules obeyed by the operators `and`, `or` and `not`.

**Table 4.1:** Summary of basic boolean algebra operations.

| expression (input) | evaluates to (output) |
|---|---|
| `True and True` | `True` |
| `True and False` | `False` |
| `False and True` | `False` |
| `False and False` | `False` |
| `True or True` | `True` |
| `True or False` | `True` |
| `False or True` | `True` |
| `False or False` | `False` |
| `not True` | `False` |
| `not False` | `True` |

Using parentheses, we can also combine multiple boolean operations to create even more complex conditions. Remember that boolean expressions inside parentheses are evaluated inside-out, as in regular mathematics. For example, the following boolean expression only evaluates to `True` if `brightness>0.5` and `resolution>100` and `location` does not start with "`Yellowstone`":

```
((brightness>0.5) and (resolution>100)) and (not location.startswith("Yellowstone"))
```

★ **Exercise 23:** Consider the following variables:

```
B = 0.7
R = 10
L = "Yellowstone, Mt. Howell"
```

Predict the output of the following `python` code, without actually running it:

```
print(((B>0.6) and (R<10)) and L.startswith("Yellowstone"))
print(((B>0.6) and (B<1)) and L.endswith("Yellowstone"))
print(((B>0.6) and (B<0.65)) and (not L.endswith("Mt. Table")))
print(((B>0.6) or (B<0.65)) and ((R>10) or (R>B)))
print(((B>0.6) or (B<0.65)) and ((R>10) or (not (R>B))))
print((((B>0.6) or (B<0.65)) and ((R>10) or (R>B))) and (L=="Yellowstone, Mt. Howell"))
```

★ **Exercise 24:** Suppose that we have a string variable (`x`) representing a single codon sequence, and that we want to perform a certain computation only if `x` codes for the amino acid

Valine, i.e., is equal to one of the following: "GTT", "GTC", "GTA", "GTG". Which of the following if statements properly encodes this logical condition?

1.
```
if(x == GTT, GTC, GTA, GTG):
    # do something
```

2.
```
if(x == "GTT", "GTC", "GTA", "GTG"):
    # do something
```

3.
```
if(x == "GTT" or "GTC" or "GTA" or "GTG"):
    # do something
```

4.
```
if(x=="GTT") or (x=="GTC") and (x=="GTA") or (x=="GTG"):
    # do something
```

5.
```
if((x="GTT") or (x="GTC") or (x="GTA") or (x="GTG")):
    # do something
```

6.
```
if((x=="GTT") and (x=="GTC") and (x=="GTA") and (x=="GTG")):
    # do something
```

7.
```
if((x=="GTT") or (x=="GTC") or (x=="GTA") or (x=="GTG")):
    # do something
```

★ **Exercise 25:** Suppose that we have two string variables, quality and extent, describing the quality and extent (completeness) of a fossil, respectively. Write a python code that uses a single if statement (without ifelse or else) and proper boolean algebra to print out the message "acceptable" whenever the following multifaceted condition is met: The fossil's quality must be either "moderate", "good" or "excellent", and the fossil's extent must contain at least one of the words "spine" or "skull" in a non-negated format (i.e., not preceded by "no "). Hence, the following examples are all acceptable:

```
quality = "moderate"
extent  = "spine,no skull,two front legs"
```

```
quality = "excellent"
extent  = "spine,skull,all legs,no toes"
```

```
quality = "good"
extent  = "no front legs,spine"
```

In contrast, the following examples are not acceptable (thus no message should be printed):

```
quality = "excellent"
extent  = "no spine,no skull,two front legs"
```

```
quality = "excellent"
extent  = "two front legs"
```

```
quality = "poor"
extent  = "three legs,skull,all toes"
```

You may assume that the strings `quality` and `extent` are entirely in lower case. Confirm your code's correctness using all of the above examples and 5 other examples of your choice.

**Context:** In this exercise we are simply printing a message whenever a condition is satisfied; in realistic situations, we would typically be using the condition to steer some complex analysis. For example, we may have a large database of fossils of varying qualities that we are analyzing, and we wish to automatically restrict some analyses to only fossils meeting our quality criteria.

★ **Exercise 26:** Consider the following `python` code that makes use of a hypothetical string variable called `species` and hypothetical numeric variable `age`:

```
if(species.startswith("Homo ")):
    if(species.startswith("Homo sapiens")):
        if(species=="Homo sapiens sapiens"):
            print("modern human")
        else:
            print("human")
    elif(species=="Homo erectus"):
        print("upright man")
    elif(species=="Homo habilis"):
        print("handy man")
    elif(age<2e6):
        print("other young homo")
    elif(age>1e7):
        print("other old homo")
else:
    if(species.startswith("Australopithecus") or species.startswith("Paranthropus")):
        print("Sister clade")
    else:
        print("Distant relative")
```

Rewrite the above code as a non-nested `if-elif-..-elif-else` statement, using only one `if` but as many `elif` as necessary.

*Hint: Use boolean algebra to construct appropriate composite conditions in your* ***`if`*** *or* ***`elif`*** *statements, as needed. Confirm the correctness of your code using various choices of* ***`species`*** *and* ***`age`****.*

## 4.10 Lists

### 4.10.1 Basic concepts

A central feature of any modern programming language is the ability to represent and process sequences of multiple numbers packaged into a single variable. In `python`, this can be accomplished using *lists*. Python lists represent sequences of one or more variables (called the list's elements or items), arranged in a specific order so as to facilitate access and processing. For example, the following is a list of 5 numbers:

```
[23, 3, 4, 10, 20.5]
```

We can assign lists to variables, just like we can assign individual numbers to variables:

```
mylist = [23, 3, 4, 10, 20.5]
```

Thus, `mylist` represents 5 distinct numbers, arranged in a specific order. To determine how many elements are in a list, we can use the function `len`:

```
print("There are %d elements in the list"%(len(mylist)))
```

```
There are 5 elements in the list
```

If we wish to print out the actual contents of a list, we can use the familiar `print` function:

```
print(mylist)
```

```
[23, 3, 4, 10, 20.5]
```

**Concatenating lists:** It is possible to concatenate two lists, thus creating a new, longer list. For example, the following code:

```
longer_list = mylist + [1000, 77, 66]
```

creates a new list of 8 numbers and assigns it to the variable `longer_list`. Let's look at its contents:

```
print(longer_list)
```

```
[23, 3, 4, 10, 20.5, 1000, 77, 66]
```

Note that `+` does not add the numbers in the two lists in the mathematical sense - the elements of the lists are merely combined into a longer list. If we wish to append the second list to the first list in place (i.e., modifying `mylist`), we can instead write:

```
mylist += [1000, 77, 66]
```

This will append the elements `1000, 77, 66` directly to `mylist`, so that `mylist` now contains in total 8 elements:

```
[23, 3, 4, 10, 20.5, 1000, 77, 66]
```

If we wish to only append a single element at a time, we can also use the list method `append`, for example:

```
mylist.append(55)
```

Hence, `mylist` now contains the following 9 elements:

```
[23, 3, 4, 10, 20.5, 1000, 77, 66, 55]
```

**List indexing:** We can access any one element in a list as an individual variable, by specifying its position (called index, counting from 0) in square brackets. For example, the following code prints the first two elements in the list:

```
print("First two elements in list: ",mylist[0],",",mylist[1])
```

```
First two elements in list:  23 , 3
```

while the following code prints the sum of the first 3 elements in the list:

```
print(mylist[0]+mylist[1]+mylist[2])
```

```
30
```

Note that we have already encountered a similar syntax for strings, where we used the index operator `[]` to extract segments of a string (Section 4.7.1). Similarly to strings, we can also index lists using negative indices, which are counted backwards from the end. For example, to access the last element in a list we can use the negative index -1:

```
print("Last element:",mylist[-1])
```

```
Last element: 55
```

Alternatively, we can first determine the length of the list using the function `len`, and then compute the proper index of the last element (remember that the last index is $N-1$ if there are $N$ elements in the list):

```
N = len(mylist)
print("Last element:",mylist[N-1])
```

```
Last element: 55
```

It is also possible to access a contiguous range of multiple elements, by specifying the range's start index (inclusive) and end index (exclusive) separated by a colon (`:`); such an operation is called "slicing". For example, the following code extracts the second, third and fourth elements in `mylist`, and returns them as a new (smaller) list:

```
smaller_list = mylist[1:4]
print(smaller_list)
```

```
[3, 4, 10]
```

We can also use the index operator to modify any of a list's elements. For example, the following code replaces the first element in the list with the value 10 and the last element with the value 100:

```
mylist[0]  = 10
mylist[-1] = 100
```

**Non-numeric lists:** Python lists are very versatile, as they can contain virtually any arbitrary datatype, such as strings, numbers, and even other lists. For example, the following defines a list of 4 strings:

```
species_names = ["E. coli", "S. aureus", "S. enterica", "V. cholearae"]
```

while the following defines a list whose three elements are themselves lists ("sublists") of numbers:

```
tree_heights = [[0.5, 0.8, 0.3, 1.1, 1.4], [0.1, 0.76, 2.3], [1.8, 2.3]]
```

Lists of lists are a convenient way to represent multi-layered datasets. In the above hypothetical example, `tree_heights` contains the heights of a specific tree species measured in multiple forest patches: 5 such trees were measured in the first patch, 3 such trees were measured in the second patch, and 2 such trees were measured in the third patch.

★ **Exercise 27:** How would you access the height of the first tree in the last forest patch, in the above example? Remember that each element of `tree_heights` is itself a list.

★ **Exercise 28:** Construct a list in `python` whose 3 elements are themselves lists (sublists), with the first sublist consisting of the numbers 1, 3, 5, the second sublist consisting of the numbers 10,30,50, and the third sublist consisting of the single number 100.

★ **Exercise 29:** Starting with the initial list:

```
mylist = [1]
```

and using only repeated list concatenations (as demonstrated in Section 4.10.1), construct a numeric list of length 4096, consisting entirely of 1's (i.e., `[1, 1, 1, ...]`). Try to keep your code below 30 lines.

*Hint: Notice that* $4096 = 2^{12}$.

### 4.10.2 More practical ways to create lists

We have already seen one way by which we can define lists, i.e., by explicitly writing out their elements as a comma-separated list enclosed by square brackets, for example:

```
mylist = [23, 3, 4, 10, 20.5]
```

It is also perfectly possible to use previously defined variables as elements, instead of explicit numerical values, for example:

```
H2S = 0.06
O2  = 0.10
CH4 = 1.4
NH4 = 0.009
chemical_concentrations = [H2S, O2, CH4, NH4]
```

Further, we can also create an empty list by not specifying any elements at all:

```
mylist = []
```

We can always add more elements to an existing list using the `append` method, for example:

```
mylist.append(4)
mylist.append(10)
print(mylist)
```

```
[4, 10]
```

Creating lists this way can be useful, for example, if we don't know a priori which or how many elements we need, and instead plan to gradually fill in the elements as part of some subsequent computations. On the other end of the spectrum, we can also easily create an arbitrarily long list with the same value at all positions, for example:

```
a_full_list = [0] * 10
```

The above code creates a new list comprising 10 zeros:

```
print(a_full_list)
```

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Creating a list of a specific length, such as above, may be useful for example if we know how many elements we want in a list but don't know right away what their values will be and instead will use some subsequent computations to compute those values.

Many other methods exist for creating lists, each suitable for different situations. A common method uses the `range` function, which can be used to define a list of consecutive integers between two desired extremes. For example, the following code constructs a list comprising the first 6 integers (0, 1, 2, .., 5):

```
X = list(range(6)) # this is equivalent to X = [0, 1, 2, 3, 4, 5]
```

If we instead want to start at some integer other than 0, we specify this as an additional first argument:

```
X = list(range(3,6)) # this is equivalent to X = [3, 4, 5]
```

Note that the function `range` itself does not return an actual list, but instead returns another object that represents a range of integers; `python` distinguishes between ranges and lists, because

ranges can be represented without the need to store all included members in memory (thus saving computational resources). It is only after we pass the range object returned by `range(3,6)` as an argument to `list` that we get an actual list.

A new list can also be obtained by performing some operation on each of the elements of another list. This can easily be achieved using the `map` function. For example, to take the square root of every element of a numeric list and store those square roots in a new list, we can apply the `sqrt` function (from the `math` module) as follows:

```
import math
numbers = [4, 9, 16, 100]
roots   = list(map(math.sqrt, numbers))
```

The above code tells `python` to apply `math.sqrt` to each element of `numbers`, and store the results of this repeated operation in a new list called `roots`. Hence, `roots` will be a numeric list with the following elements:

```
[2.0, 3.0, 4.0, 10.0]
```

Lastly, many `python` functions and methods return a list as a natural result of some computation. For example, the string member function `split`, which splits a string at a specific *delimiter*, returns a list with the various pieces. The following code creates a list whose elements are individual species names, by splitting a string consisting of semicolon-separated species names:

```
concatenated_species_names = "E. coli;S. aureus;S. enterica;V. cholearae"
separated_species_names    = concatenated_species_names.split(";")
print(separated_species_names)
```

```
['E. coli', 'S. aureus', 'S. enterica', 'V. cholearae']
```

The ability to split strings at specific characters of choice is very useful in data analysis, as many datasets are stored as plain text, with individual measurements separated by commas, tabs, semicolons or similar. Python even provides a special method for splitting strings at line breaks, called `splitlines`. This is useful, for example, for reading the contents of a text file and storing them as a list of individual lines (one string per line):

```
fin = open("documents/some_text_file.txt", "rt")
lines = fin.read().splitlines()
fin.close()
```

The new variable `lines` henceforth represents a list with as many elements as there were lines in the text file. If each line happens to consist of a number, we can convert the strings representing the individual lines into actual numeric variables (remember than in `python` `"1.5"` is not the same as `1.5`). This can be achieved by applying the `float` function to each element of `lines`, as follows:

```
numbers = list(map(float, lines))
```

Note that the above code will throw an error if any of the lines was non-numeric.

★ **Exercise 30:** In this exercise we consider the following text file, containing hourly air temperature measurements recorded at the Eugene airport on January 1st, 2022:

http://www.loucalab.com/archive/BioDataAnalysisPython
→ file "Eugene_airport_hourly_temperatures_NOAA_2022.01.01.txt".

Each line in this file contains a single hourly temperature measurement (in °C). Load the contents of this file as a list of strings representing individual lines, and then convert that list into a list of numbers, as discussed in Section 4.10.2. Print the resulting numeric list to the screen.

★ **Exercise 31:** Consider the following hypothetical time series of hourly measured expression levels for a single gene (one numeric value per hour), stored in the form of a single string:

```
series = "0.2,0.53,0.5,0,0.1,0.8,0.5,0.3,0.3,0.32,0.71,1.0,1.0,0.5,0.2,0.2,0.5"
```

Note that the first number refers to hour 1, the 2nd number refers to hour 2, and so on. Write a `python` code that computes and prints out the average gene expression level at hours 10 and 11 (i.e., the average of the two values recorded at time points 10 and 11). As usual, you are expected to use the provided `series` variable, rather than explicitly writing the two averaged expression level values into your code.

*Hint: You can split the string at each comma using the* ***`split`*** *method, then convert each piece into numbers using* ***`float`*** *and* ***`map`****, and then access the desired elements of the resulting numeric list as described in Section 4.10.1. Keep in mind that* ***`python`*** *list indices start counting at 0.*

### 4.10.3 Using lists as function arguments

Lists can be passed as function arguments pretty much like numeric or string variables, provided that this is meaningful. For example, if we have a list of numbers, we can pass that list as an argument to the `numpy` function `mean` as a single argument, to compute its arithmetic mean:

```
import numpy
X = [1.2, 3, 4.1, -4, 104, 501]
print("Mean = ",numpy.mean(X))
```

```
Mean =  101.55
```

Similarly, we can the compute the sum, standard deviation, minimum and maximum of `X`:

```
print("Sum = %.4g"%(numpy.sum(X)))
print("Standard deviation = %.4g"%(numpy.std(X)))
print("Minimum = %g"%(min(X)))
print("Maximum = %g"%(max(X)))
```

```
Sum = 609.3
Standard deviation = 182.6
Minimum = -4
Maximum = 501
```

Other functions accepting a list as argument may simply return a modified version of the list. For example, the function `sorted` takes a list as argument and returns a sorted copy of the input list:

```
Y = sorted(X)
print(Y)
```

```
[-4, 1.2, 3, 4.1, 104, 501]
```

We can also pass multiple lists as arguments to a function. For example, to compute the Pearson correlation ($r$) between two data vectors, as well as the associated two-sided statistical significance ($p$-value), we can use the `pearsonr` function from the `stats` module in `scipy` as follows:

```
from scipy import stats
r,p = stats.pearsonr([1, 2, 3, 4, 5], [10, 9, -30, 65, 4])
```

Observe that here we are passing two numeric lists as the two function arguments. The above code yields the following result:

```
r = 0.204063
p = 0.741994
```

★ **Exercise 32:** Following the examples in Section 4.10, compute the arithmetic mean for each of the following two data vectors:

```
X = [10, 32, 4, -9, 8.5, 12, 100]
Y = [0.4, 2, -0.1, 1, 24, 43, 98]
```

Then compute their Pearson correlation ($r$) and its two-sided statistical significance ($p$-value).

★ **Exercise 33:** Suppose that we have a long numeric list, such as the following:

```
measurements = [0.5, 1, -3, 4.4, 7.3, 2.9, 8.6, 2, 0, -34, 7, 7.7, 8.6, 11, 5, -5, 0.3]
```

Write a `python` code that prints out the 10 smallest elements in the list.

*Hint: You can use the* `sorted` *function to sort the list, and then print the first 10 elements.*

★ **Exercise 34:** Suppose that we have a list whose elements are themselves lists ("sublists"), each of which contains measured tree heights (in meters) at a specific surveyed location:

```
tree_heights = [[3.4, 5.1, 0.9, 4.8], [7.6, 6.8, 9.8], [0.8],  [0.2, 0.8]]
```

In the above example, 4 trees were measured at the first location, 3 trees were measured at the second location, one tree was measured at the third location, and 2 trees were measured at the fourth location. Write a `python` code that uses the `tree_heights` variable to compute the average tree height in the first location and the average tree height in the last location. Execute your code and print out the results for the above example.

*Hint: You can use the function* `mean` *in the package* `numpy` *to compute the average of a numeric list.*

★ **Exercise 35:** Consider the following DNA sequence, which also includes unknown nucleotides (`N`):

```
DNA = "GGCNAACGCTTGTGGANATCTAAAAGGTTAANNNAAATGACAAAANACATTGATNACNAACAGTGTTNTNTT"
```

Write a `python` code that computes the length of the longest segment of consecutive known (i.e., non-`N`) nucleotides.

*Hint: You can split the string at each `N` character using the `split` method, then compute the length of each resulting segment using `len` and `map`, and finally find the maximum segment length using the `max` function.*

★ **Exercise 36:** In this exercise we consider a text file containing hourly air temperature measurements recorded at the Eugene airport on January 1st, 2022:

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`Eugene_airport_hourly_temperatures_NOAA_2022.01.01.txt`".

Each line in this file contains a single hourly temperature measurement (in °C). Load the recorded temperatures as a numeric list, as discussed in Section 4.10.2 (also see Exercise 30), and compute their arithmetic mean, their standard deviation, their minimum and their maximum (as discussed in Section 4.10.3).

## 4.11 Sets

Another generic and basic data container in `python` is the set. Just like lists, sets are used to store multiple values, such as the names of bacterial species detected in an animal's gut. Unlike lists, however, the elements of a set are not ordered in any specific way, and each element can only appear once. Hence, a set only contains information on *which* elements are stored in it, but not the order in which they are arranged nor how often each element is represented. This may seem like a disadvantage, however there are situations in which this is useful. For example, we may want to keep track of which species were detected in a survey, regardless of the order in which they were detected, and avoiding duplicate listings of species that were detected multiple times.

To create an empty set (i.e., starting with no elements), we can use the function `set` without arguments:

```
myset = set()
```

We can then add elements to the set, one by one, using the method `add`, as follows:

```
myset.add("E. coli")
myset.add("S. aureus")
```

The set `myset` now consists of two elements, the strings `"E. coli"` and `"S. aureus"`. Adding an element more than once does not change the set, since each element can be represented at most once. For example, the following code yields the same set as above:

```
myset.add("E. coli")
myset.add("S. aureus")
myset.add("E. coli")
```

If we know beforehand which elements we want in a set, we can create the set more compactly by explicitly specifying all of its elements in curly brackets, as follows:

```
myset = {"E. coli", "S. aureus"}
```

We can also create a set using the elements in a list. For example, the following code yields the

same set as above:

```
myset = set(["E. coli", "S. aureus"])
```

This functionality is particularly useful if we want to reduce a list into a set of unique values, i.e., omitting any duplicates. For example, suppose that we have a list of species detected in a survey, as follows:

```
species = ["A. hirsuta", "L. cucumis", "A. acufera", "H elegans", "A. hirsuta"]
```

Using the `set` function, we can determine the unique detected species names, i.e., omitting duplicate elements:

```
unique_species = set(species)
```

To check whether a given value is part of a set, we can use the familiar `in` keyword (similarly to lists), for example:

```
if("L. cucumis" in unique_species):
    print("L. cucumis is included")
else:
    print("L. cucumis is not included")
if("L. impura" in unique_species):
    print("L. impura is included")
else:
    print("L. impura is not included")
```

```
L. cucumis is included
L. impura is not included
```

Similarly to lists, we can determine the number of elements in a set using `len`, for example:

```
print("Number of unique species:",len(unique_species))
```

```
Number of unique species: 4
```

We can also print out all elements of a set using our familiar `print` function (again, similarly to lists):

```
print(unique_species)
```

```
{'A. acufera', 'L. cucumis', 'A. hirsuta', 'H elegans'}
```

As mentioned earlier, the elements of a set are not arranged in any specific order, and hence it does not make sense to refer to them by their position (index) as in lists. For example, the following code will cause an error, since no specific element is at position 0:

```
print(unique_species[0])
```

If you really want to have elements arranged in a fixed order (regardless of what that order is), you can convert a set into a list, as follows:

```
ordered_unique_species = list(unique_species)
```

To obtain a list sorted in a standard way (e.g., ascending in the case of numbers, or alphabetical in the case of strings), you can instead use the `sorted` function:

```
ordered_unique_species = sorted(unique_species)
```

Switching between sets and lists is common, for example when we want to quickly remove any duplicates from a list but still want elements to have a fixed arrangement.

**Set operations:** Python sets support a wide range of set operations known from mathematics, such as computing set differences (i.e., finding elements in one set but not in another), computing set intersections (i.e., finding elements present in both sets), computing set unions (i.e., finding elements present in either set), or checking if one set is a subset of another set. For example, suppose that we recorded the bacterial species found in two animal guts:

```
gut1 = {"E. cloacae", "S. aureus", "E. coli", "S. humiferus", "V. cholerae"}
gut2 = {"S. aureus", "S. humiferus", "B. capreoli", "S. glaucescens"}
```

To determine which species were unique to the first gut, we can use set difference:

```
unique1 = gut1.difference(gut2)
print(unique1)
```

```
{'V. cholerae', 'E. coli', 'E. cloacae'}
```

To determine which species were found in both guts, we can use set intersection:

```
found_in_both = gut1.intersection(gut2)
print(found_in_both)
```

```
{'S. aureus', 'S. humiferus'}
```

To determine which species were found in either gut, we can use set union:

```
found_in_either = gut1.union(gut2)
print(found_in_either)
```

```
{'B. capreoli', 'S. humiferus', 'E. coli', 'S. aureus', 'V. cholerae', 'E. cloacae', '
    S. glaucescens'}
```

For a comprehensive overview of additional set operations see this official documentation.

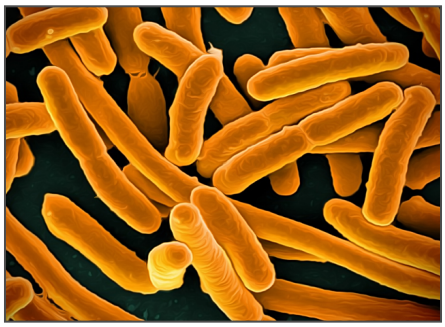
★ **Exercise 37:** Consider the following two text files, each of which lists the gene content of a different *Escherichia coli* strain:

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → files "`Escherichia_coli_genes_KOfam_GCF_008124005.1.tsv`"
> and "`Escherichia_coli_genes_KOfam_GCF_015831325.1.tsv`".

Note that each file consists of a single line of text, which specifies gene IDs as a comma-separated

list (e.g., `K07010,K01889,K07113,K24147,K07657,...`). Write a `python` code that analyzes the two files to compute the number of genes shared between the two genomes (i.e., their set intersection) as well as the total number of genes found in the two genomes (i.e., their set union).

*Hint: For each file, load its content into a single string, then split the string at the commas as described in Section 4.10.2, thus obtaining a list of strings (one string per gene), and then convert that list into a set as described in Section 4.11. Once you have the two gene lists in the forms of sets, you can use set operations as needed.*



## 4.12 For loops

One of the most useful aspects of computer programming is the ability to automatically repeat certain tasks a large number of times. The need to repeat a task many times is ubiquitous in science, for example when analyzing a large number of distinct genomes, or when examining the recorded behavior of a large number of individuals over time, or when dealing with EEG recordings of brain activity from multiple electrodes. In this section we will learn about one of the most common approaches to automating repetition, the `for` loop. The `for` loop is used whenever we wish to repeat a task a predictable number of times, for example analyzing an a priori known number of genomes.

In its most basic form, the `for` loop requires the definition of an iterator variable that keeps track of the current iteration or "cycle", a *range* or *sequence* through which the iterator variable will pass, and a body of code to execute with each iteration. For example, the following loop iterates over the first 4 integers (0, 1, .., 3) and prints out their squares to the screen:

```
for k in range(4):
    print(k**2)
```

Here, `k` is the iterator, `range(4)` is the range over which `k` iterates, and `print(k**2)` is the body of the loop. The first line, which specifies the iterator and range, is called the loop's header. The iterator's name can be arbitrary, following the same rules as any other `python` variable. The iterator can be accessed inside the loop's body, and allows the body to know which iteration is currently being executed (i.e., which of the 4 integers we wish to print next). Hence, the loop's body is actually executed 4 times, and each time the value of the iterator `k` is different. The above code thus produces the following output:

```
0
1
4
9
```

Note that the above loop is equivalent to the following (much less compact) code:

```
k = 0
```

```
print(k**2)

k = 1
print(k**2)

k = 2
print(k**2)

k = 3
print(k**2)
```

While in this simple example we could have gotten away with this alternative approach (essentially writing out the loop's body 4 times), in most realistic situations this is rather impractical. Indeed, a loop's body can — and usually does — consist of multiple lines of code, for example if complex computations need to be performed with each iteration.

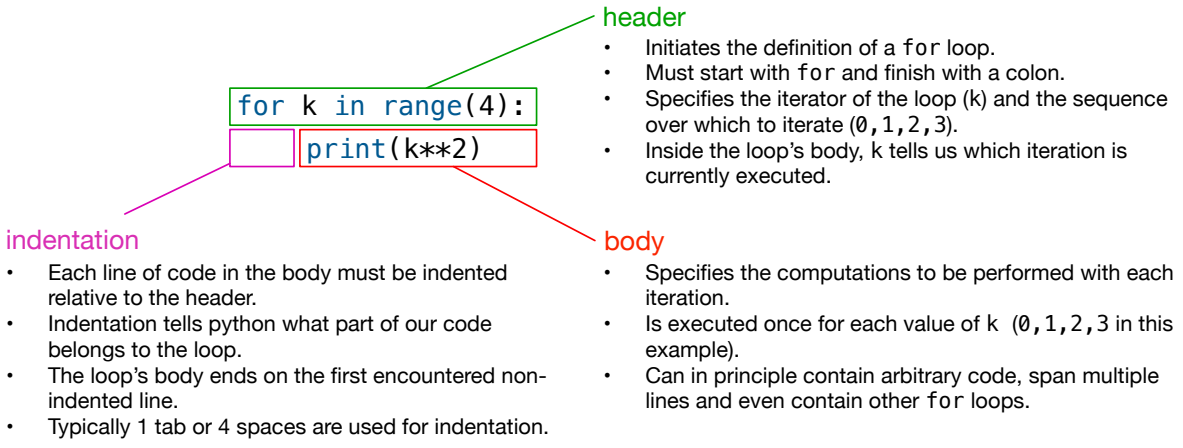
Observe that we have written the loop's body further inwards compared to the loop's header. This *indentation* is similar to what we have already encountered for `if` statements, and it allows `python` to know which part of the code belongs to the body of the loop. The loop's body extends down to (but not including) the first non-indented line after the header. For example, in the following code only the first `print` statement is part of the loop's body (and hence will be repeated 4 times), while the 2nd `print` statement is outside of the loop's body (and hence will only be executed once, after the loop is done):

```
for k in range(4):
    print(k**2)
print("This message is only printed once, after the whole loop is done")
```

Its output will thus be:

```
0
1
4
9
This message is only printed once, after the whole loop is done
```

The following schematic summarizes the structure of the above loop:




- Initiates the definition of a `for` loop.
- Must start with `for` and finish with a colon.
- Specifies the iterator of the loop (k) and the sequence over which to iterate (`0,1,2,3`).
- Inside the loop's body, k tells us which iteration is currently executed.

- Each line of code in the body must be indented relative to the header.
- Indentation tells python what part of our code belongs to the loop.
- The loop's body ends on the first encountered non-indented line.
- Typically 1 tab or 4 spaces are used for indentation.

- Specifies the computations to be performed with each iteration.
- Is executed once for each value of k (`0,1,2,3` in this example).
- Can in principle contain arbitrary code, span multiple lines and even contain other `for` loops.

A loop does not necessarily need to iterate over a contiguous range of integers, but can instead iterate over an arbitrary list/set of numbers, or an arbitrary list/set of strings, and so on. For example, the following code computes and prints out the frequency of each nucleotide (`A`, `C`, `G`,

T) in a given DNA sequence:

```
DNA = "CAAACCCCTCACATGCATTTAGGTGTTGTGCCTATGCGTGATGGAAAACTGCAAGGAAAAAATGTGTTTAAT"
for nucleotide in ["A","C","G","T"]:
    frequency = DNA.count(nucleotide)/len(DNA)
    print("Nucleotide %s has frequency %.3f"%(nucleotide,frequency))
```

In the above example, the loop's body consists of two lines of code, `nucleotide` is the loop's iterator, and the list `["A","C","G","T"]` specifies a sequence of strings over which `nucleotide` iterates. This code produces the following output:

```
Nucleotide A has frequency 0.319
Nucleotide C has frequency 0.181
Nucleotide G has frequency 0.222
Nucleotide T has frequency 0.278
```

**Iterating over a set:** As mentioned earlier, loops can also iterate over a set rather than a list, however note that in that case the order in which elements are iterated through is unpredictable. This is because the elements of a set are not ordered in any specific way. Depending on the situation, this may or may not be an issue. For example, the following code computes the frequency of each nucleotide, however the order in which the four nucleotides are examined is unpredictable:

```
DNA = "CAAACCCCTCACATGCATTTAGGTGTTGTGCCTATGCGTGATGGAAAACTGCAAGGAAAAAATGTGTTTAAT"
nucleotides = set(["A","C","G","T"])
for nucleotide in nucleotides:
    frequency = DNA.count(nucleotide)/len(DNA)
    print("Nucleotide %s has frequency %.3f"%(nucleotide,frequency))
```

```
Nucleotide C has frequency 0.181
Nucleotide G has frequency 0.222
Nucleotide T has frequency 0.278
Nucleotide A has frequency 0.319
```

**Accessing variables defined outside of the loop:** The body of a loop can also access and even modify variables outside of the loop, provided of course that these variables have been previously defined. Let's see how we can use this to write a loop that finds the maximum element in a numeric list. While `python` includes convenient functions for finding maxima and minima (which use loops internally), for now we will pretend that we don't know about those functions because it is really important to understand the logic of loops. Suppose that we have a numeric list containing tree heights, such as the following:

```
heights = [2, 4, 0.5, 3, 7.2, 8, 9.3, 10, 4, 2.1, 1.5, 11, 7.5, 14, 5]
```

Let's see how we can use a `for` loop to find the maximum listed height. How might a human do this manually? One approach would be to go through the heights one by one, from start to finish, at all times only remembering (tracking) the largest height encountered so far. Each time a new height is encountered, check if that height exceeds the largest height remembered so far, and if so, remember this new height and forget the previously remembered one. This strategy can actually easily be coded as a loop:

```
largest = heights[0] # start by looking at the first element
```

```
for height in heights:
    if(height>largest):
        # we found a larger element
        largest = height
print("Maximum height = %g"%(largest))
```

```
Maximum height = 14
```

Note that `largest` tracks the largest height encountered so far, however in order to access this variable in the loop body it first needs to be defined somewhere. In the above example, we chose to define it outside of the loop, setting it to the first element in the list (any other element would actually also work in this case). We can also use the familiar function `max` to avoid the `if` statement in the loop body and thus make it a bit more concise:

```
largest = heights[0]
for height in heights:
    largest = max(largest, height)
print("Maximum height = %g"%(largest))
```

```
Maximum height = 14
```

Each time the loop repeats, the tracking variable `largest` is updated (if needed) to the largest value found so far. Since tree heights are always positive, we could have also initialized `largest` to the value 0 prior to the loop, since surely this value is below the list's maximum element:

```
largest = 0
for height in heights:
    largest = max(largest, height)
print("Maximum height = %g"%(largest))
```

```
Maximum height = 14
```

Each time the above loop's body is executed, `largest` is set to the maximum between it's previous value and the most recently encountered height; hence, when the loop has finished running, `largest` will be equal to the maximum encountered tree height.

The last approach (i.e., initializing `largest` using a value that surely is below the list's maximum) can be useful when we don't yet have the elements whose maximum we wish to determine. For example, suppose that we want to compute the frequencies of all four nucleotides (A, C, G, T) in a DNA sequence and determine the maximum such frequency, using a single loop. Following an analogous approach as above, we can achieve this as follows:

```
max_frequency = 0
for nucleotide in ["A","C","G","T"]:
    frequency = DNA.count(nucleotide)/len(DNA)
    max_frequency = max(max_frequency, frequency)
print("Maximum frequency of any nucleotide: %.3f"%(max_frequency))
```

```
Maximum frequency of any nucleotide: 0.319
```

Each time the above loop's body is executed, we first compute the new nucleotide frequency, then `max_frequency` is set to the maximum between it's previous value and the most recently

computed nucleotide frequency; hence, when the loop has finished running, `max_frequency` will be equal to the highest encountered nucleotide frequency. The output of this code is shown below: If we wanted to also keep track of which nucleotide is the most frequent (rather than just its frequency), we could modify the code as follows:

```
max_frequency = 0
for nucleotide in ["A","C","G","T"]:
    frequency = DNA.count(nucleotide)/len(DNA)
    # check if this nucleotide is more frequent than any previous nucleotide
    if(frequency>max_frequency):
        max_frequency  = frequency
        top_nucleotide = nucleotide
print("Top nucleotide is %s, with frequency %.3f"%(top_nucleotide,max_frequency))
```

```
Top nucleotide is A, with frequency 0.319
```

**Concurrently iterating over multiple lists:** In some situations we may have multiple lists of data that we wish to iterate over in a *concurrent* manner. For example, we may have a list containing embryo masses and another list of the same length that contains corresponding embryo ages, and we wish to compute the average mass gained per time (mass/age) for each embryo. This can be achieved by iterating over all list indices, and then accessing the appropriate elements in the two lists inside the loop body:

```
masses = [0.1, 0.5, 0.3, 0.08, 0.65, 1.8]
ages   = [9,   52,  230, 89,   61,   131]
for k in range(len(masses)):
    mass_over_age = masses[k]/ages[k]
    print("Embryo mass/age = %f"%(mass_over_age))
```

In the above code, the iterator `k` cycles through the integers 0, 1, 2, and so on, and is used to track the embryo currently being investigated. The above code generates the output:

```
Embryo mass/age = 0.011111
Embryo mass/age = 0.009615
Embryo mass/age = 0.001304
Embryo mass/age = 0.000899
Embryo mass/age = 0.010656
Embryo mass/age = 0.013740
```

Python actually also provides convenient shorthands for performing such concurrent loops over multiple lists. One such shortcut is the `zip` function, which provides a separate iterator for each list. The following code is thus equivalent to the above:

```
for mass, age in zip(masses, ages):
    mass_over_age = mass/age
    print("Embryo mass/age = %f"%(mass_over_age))
```

Observe that both `mass` and `age` advance to their next respective value with each iteration, i.e., in the first iteration `mass` will be `0.1` and `age` will be `9`, in the second iteration `mass` will be `0.5` and `age` will be `52`, and so on. Another useful shortcut is provided by the `enumerate` function, which yields two iterators for looping over a list, one iterator acting as positional index and another iterator representing corresponding elements in the list. For example, in the following loop `m` specifies the integer position of the element currently represented by `mass`:

```
for m, mass in enumerate(masses):
    print("Mass of embryo #%d = %.2f g"%(m,mass))
```

```
Mass of embryo #0 = 0.10 g
Mass of embryo #1 = 0.50 g
Mass of embryo #2 = 0.30 g
Mass of embryo #3 = 0.08 g
Mass of embryo #4 = 0.65 g
Mass of embryo #5 = 1.80 g
```

Both `m` and `mass` advance concurrently through the loop, so in the first iteration `m` is `0` and `mass` is `0.1`, in the second iteration `m` is `1` and `mass` is `0.5`, and so on. The above code is thus equivalent to the following:

```
for m, mass in zip(range(len(masses)),masses):
    print("Mass of embryo #%d = %.2f g"%(m,mass))
```

Both `zip` and `enumerate` mostly serve as conveniences, allowing us to write more concise and readable code.

**Prematurely exiting a loop:** In some situations it may be necessary to exit a loop before it has iterated over the full provided list/set. For example, we may be using a loop to search for the first occurrence of a specific event in a time series, in which case we can skip the remainder of a loop as soon as the event is first encountered. To skip the remainder of a loop we can use the `break` statement. For example, the following code iterates over a list of species names and exits the loop as soon as "`E. coli`" is encountered:

```
names = ["S. aureus", "S. sonnei", "N. halophila", "E. coli", "R. tenuis", "F. filum"]
for name in names:
    print(name)
    if(name=="E. coli"):
        print("Exiting the loop early!")
        break
print("Done with the loop.")
```

```
S. aureus
S. sonnei
N. halophila
E. coli
Exiting the loop early!
Done with the loop.
```

Note that, as above, a `break` statement will usually be enclosed in some sort of condition, since an unconditioned `break` statement would mean that the loop never proceeds past its first iteration. For example, in the following code nothing gets printed to the screen because the loop exits right at the beginning of its first iteration:

```
for name in names:
    if(name=="S. aureus"):
        break
    print(name)
```

★ **Exercise 38:** Predict what the following code will print to the screen, without actually running it:

```
x = 1
n = 0
for k in [1, 2, 4, 6, 8]:
    x = k * x
    n += 1
    if(x>20):
        break
    print(x)
print(n)
```

★ **Exercise 39:** Suppose that we have two numeric lists of equal length, listing corresponding sample latitudes and longitudes (in degrees), respectively, such as the following:

```
latitudes  = [44.56,  37.67,  53.90,  61.82]
longitudes = [122.67, 113.90, 130.56, 110.07]
```

Write a `for` loop that uses the `zip` function to iterate concurrently over both lists, printing out the latitude-longitude pairs as well as their distance to the equator. The following shows the first two lines to be printed out, as an example:

```
Lat 44.56, lon 122.67, distance to equator 4954.8 km
Lat 37.67, lon 113.9, distance to equator 4188.7 km
```

*Hint: For any given latitude $y$ (in degrees), the distance to the equator can be calculated as $Ry\pi/180$, where $R = 6371$ km is Earth's average radius.*

★ **Exercise 40:** Suppose that we have a list of body masses, measured in pounds (lbs):

```
masses = [3.4, 5.2, 1.0, 9.1, 7.3, 0.4, 13.4]
```

Using a `for` loop, convert each element of `masses` from pounds to kilograms (kg), by multiplying it with the factor 0.4535. Your code should replace the original values in `masses` with their converted values. Print out the converted numeric list.

*Hint: You can reassign the element of a list at a specific index inside a loop's body.*

★ **Exercise 41:** Consider the following list of global human population sizes (in billions of people) during the years 2010–2020 (one element per year, source: Worldbank developmental indicators):

```
popsizes = [6.92, 7.00, 7.09, 7.18, 7.26, 7.35, 7.43, 7.52, 7.60, 7.68, 7.76]
```

Perform the following tasks:

1. Use the `range` function to define a list of integers, listing the years 2010–2020, as demonstrated in Section 4.10.2. Then, using a `for` loop, iterate over each year and print out the population size measured for that year, i.e., producing an output similar to the following:

```
Population size in 2010 was 6.92 billions
```

```
Population size in 2011 was 7.00 billions
Population size in 2012 was 7.09 billions
Population size in 2013 was 7.18 billions
Population size in 2014 was 7.26 billions
Population size in 2015 was 7.35 billions
Population size in 2016 was 7.43 billions
Population size in 2017 was 7.52 billions
Population size in 2018 was 7.60 billions
Population size in 2019 was 7.68 billions
Population size in 2020 was 7.76 billions
```

2. Using a `for` loop, compute and print out for each year (except for 2010) the per-capita annual population growth rate, defined as $r_k = (N_k - N_{k-1})/N_{k-1}$, where $N_k$ is the population size at year $k$. The following shows the first few lines to be printed out, as an example:

```
Per-capita annual population growth rate in 2011 was 0.011561
Per-capita annual population growth rate in 2012 was 0.012857
Per-capita annual population growth rate in 2013 was 0.012694
```

★ **Exercise 42:** Suppose that we have a list whose elements are themselves lists ("sublists"), each of which contains measured tree heights (in meters) at a specific surveyed location:

```
tree_heights = [[3.4, 5.1, 0.9, 4.8], [7.6, 6.8, 9.8], [0.8], [0.2, 0.8]]
```

In the above example, 4 trees were measured at the first location, 3 trees were measured at the second location, and so on. Write a `for` loop tat uses the `enumerate` function to iterate over the list `tree_heights`, computing and printing out the mean and maximum tree height in each location. The following shows the first two lines to be printed out, as an example:

```
Location 0: mean = 3.55 m, max = 5.1 m
Location 1: mean = 8.06667 m, max = 9.8 m
```

★ **Exercise 43:** Suppose that we have a numeric list specifying the number of eggs that a single gull laid each month, over the course of 2 years (one element per month):

```
monthly_eggs = [1,3,4,3,2,0,0,0,0,0,2,2,5,3,1,0,0,0,0,0,0,0,1,1]
```

Using a `for` loop and simple arithmetic operations (i.e., no list summation functions), compute and print out the cumulative number of eggs laid up until each month (i.e., total number of eggs until month 1, total number of eggs until month 2, and so on). The following shows the first few lines to be printed out, as an example:

```
Cumulative eggs until month 0: 1
Cumulative eggs until month 1: 4
Cumulative eggs until month 2: 8
```

*Hint: While iterating over the list* `monthly_eggs`*, use a numeric variable to keep track of the total number of eggs counted so far.*

★ **Exercise 44:** Suppose that we have been monitoring the nest visitations of a single bird during the breeding season, using an RFID tag [23–25]. Specifically, suppose that we have recorded each time (counted in minutes since the start of the recording) at which the bird passed through a detector at the nest entrance, either entering the nest or exiting the nest. For example:

```
times = [10, 10.8, 34.2, 38, 40.3, 50, 53.9, 65, 67, 80, 82.6, 97.2, 99.8, 100.2, 132]
```

Assume that immediately prior to the onset of the recordings the bird was outside of the nest, i.e., the first recorded time point corresponds to an entry of the bird. Perform the following tasks:

1. Write a code using a `for` loop that determines the total amount of time (in minutes) that the bird spent inside the nest.

   *Hint: One approach would be to use an auxiliary boolean variable that keeps track of the bird's location relative to the nest (inside vs outside), so as to only add up the time intervals between entries and their immediately following exits.*

2. Modify your code so that it instead computes the number of nest visits that lasted more than 1 minute, i.e., determine how often the bird was in the nest for more than 1 consecutive minute.

★ **Exercise 45:** The Fibonacci sequence is an increasing sequence of numbers that is widely encountered in mathematics and the natural sciences [26, 27]. It is defined using the following recursion formula:

$$F_n = F_{n-1} + F_{n-2}, \tag{4.6}$$

i.e., each member of the sequence is the sum of the previous two members, while the first two members are defined as $F_0 = 0$ and $F_1 = 1$. Hence, for example, the first few Fibonacci numbers are 0, 1, 1, 2, 3, 5, 8, 13, 21 and so on. Using a `for` loop in `python`, compute and print the 49-th and 50-th Fibonacci numbers (i.e. $F_{49}$ and $F_{50}$), as well as their ratio ($F_{50}/F_{49}$).

**Interpretation:** The ratio of successive Fibonacci numbers, $F_n/F_{n-1}$, converges to a fixed value as $n \to \infty$, known as the "Golden Ratio". The Golden Ratio appears in many natural phenomena and even theories of aesthetics, and has been the subject of great controversies and myths [28–30].

*Hint: One approach is to keep track of two variables in your loop body, the last Fibonacci number and the 2nd-last Fibonacci number, and update those as you cycle through the loop. Make sure to repeat the loop as many times as needed until you reach the numbers $F_{49}$ and $F_{50}$.*

★ **Exercise 46:** Consider the following text file, containing several animal species names:

http://www.loucalab.com/archive/BioDataAnalysisPython
→ file "`animal_species_names_1000.txt`".

You can open this text file in any regular text editor, and you will notice that it lists one latin binomial name (e.g., `Leucascus albus`) per line, except for some lines that begin with a `#` symbol (these are comment lines). Perform the following tasks using `python`:

1. Download the above file, load its contents as a list of strings (one string per line), and determine the total number of unique genomes listed.

   *Hint: You can use a* `for` *loop to iterate over each species, separating its generic and specific name parts, and using a set to keep track of unique genera encountered. Make sure to omit any lines beginning with* `#`*. See Section 4.10.2 on how to load the individual lines in the file as a list of strings.*

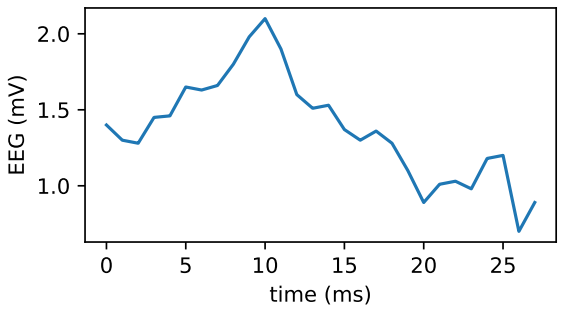
2. Determine the genus with the largest number of species in the list, and print out the genus name as well as its species count.

   *Hint: Use a loop that iterates over the set of unique genera, and determine for each of them how many times the genus is found in the text file, while keeping track of the maximum such number and the associated genus using auxiliary variables. The string method* `count` *may be of use. See the examples in Section 4.12.*

★ **Exercise 47:** Suppose that we measured the brain activity of a laboratory mouse over time using an electroencephalogram (EEG), and have stored the measured voltage (mV) at millisecond intervals in a list, such as the following:

```
EEG = [1.4, 1.3, 1.28, 1.45, 1.46, 1.65, 1.63, 1.66, 1.8, 1.98, 2.1, 1.9, 1.6, 1.51,\
       1.53, 1.37, 1.3, 1.36, 1.28, 1.1, 0.89, 1.01, 1.03, 0.98, 1.18, 1.2, 0.7, 0.89]
```

The above example EEG looks as follows, although note that in reality EEGs tend to be much longer than this one:




Write a `python` code that uses a `for` loop to iterate over the EEG to find the first occurrence of a peak. For this exercise, a point in the EEG is considered a peak if it is strictly higher than the two previous and the two subsequent points, and the previous point is strictly higher than the second previous point, and the next point is strictly higher than the second-next point. The code should print the time and EEG value of only the first encountered peak.

*Hint: The first two and last two data points can never be peaks, so it is reasonable to have your code not bother checking those.*

## 4.13 Reading and writing text files

The ability to read and write files is an indispensable tool for any data analyst, since datasets are almost always stored and transmitted in the form of files, from DNA sequences and species catalogues to demographic and epidemiological time series. Almost all computer programs written in biology involve some sort of file input/output. We have already learned about one specific way to load the contents of a text file into a string variable, using `open`, `read` and `close`:

```
fin = open("some_input_data.txt", "rt")
contents = fin.read()
fin.close()
```

Using a similar approach, we can also write a string into a text file. The following example code writes the value of a string variable into a text file named "`some_output_data.txt`" located inside a directory named "`py_output`":

```
mystring = "Some text"
fout = open("py_output/some_output_data.txt", "wt")
fout.write(mystring)
fout.close()
```

Here, `open` prepares the file for writing plain text (`wt` stands for "write text") and returns a file object that is then assigned to the variable `fout`. The method `write` writes the value of the hypothetical string variable `mystring` to the file (replacing all previous file contents), and the method `close` finalizes the file, telling the operating system that we are done working with it. After calling `close`, we can no longer use `fout` to add contents to the file unless we re-open the file. As another example, the following code will write 3 shark species names into a text file (one species per line):

```
fout = open("py_output/shark_species.txt", "wt")
fout.write("Carcharodon carcharias\nIsurus paucus\nCetorhinus maximus")
fout.close()
```

Note that we used the familiar `\n` symbol to denote line breaks between species names. The generated file will thus have the following content:

```
Carcharodon carcharias
Isurus paucus
Cetorhinus maximus
```

Keep in mind that each time we open a text file for writing using the above method, we are replacing all of its previous contents. To instead append text at the end of any existing text, use "`at`" (which stands for "append text") instead of "`wt`".

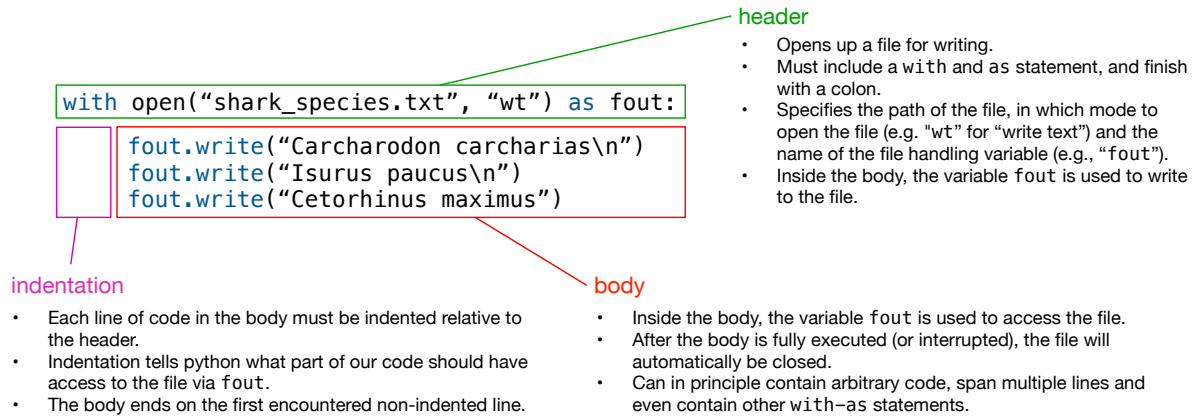
**A safer way to read/write files:** The above approach works fine in many situations, however it has one subtle flaw: If our program somehow crashes or is forcefully terminated (e.g., by the user) before the file is properly finalized using the `close` method, then the state of the text file is unpredictable and potentially corrupted. This is because `fout.write` does not necessarily immediately write everything to the file, but may instead place some or all of the text into a buffer (essentially a type of queue) to be written at some later time point. A safer and generally recommended way to read and write files is using a `with` statement, which allows us to specify a block of code to execute while the file is open in such a way that the file is properly closed if the program suddenly terminates. Hence, to write the 3 species names into a text file we could use:

```
with open("shark_species.txt", "wt") as fout:
    fout.write("Carcharodon carcharias\nIsurus paucus\nCetorhinus maximus")
```

Note that we use indentation to denote the code to be executed while the file is open. Also note that we no longer need to explicitly call the `close` method to finalize the file, as this is done automatically for us after the block inside the `with` statement is done. Thanks to indentation, we can include multiple lines of code, including multiple `write` calls, inside a `with` statement. For example, the following code produces the same outcome as the previous example:

```
with open("shark_species.txt", "wt") as fout:
    fout.write("Carcharodon carcharias\n")
    fout.write("Isurus paucus\n")
    fout.write("Cetorhinus maximus")
```

The following figure summarizes the structure of the above `with` statement:




We can also use a `with` statement to load the contents of a text file as a single string:

```
with open("some_text_file.txt", "rt") as fin:
    contents = fin.read()
```

To instead load the file's contents as a list of strings (one string per line), we can use:

```
with open("some_text_file.txt", "rt") as fin:
    contents = fin.read().splitlines()
```

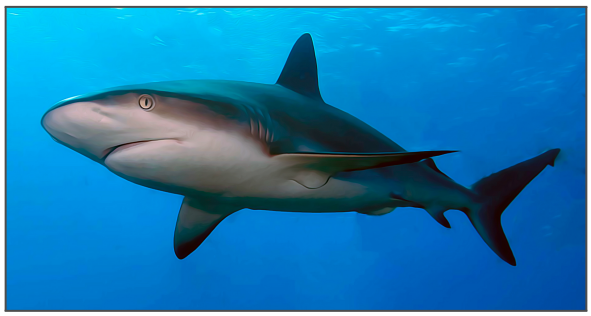
For the remainder of the book, we will be preferring the `with` method to read and write plain text files, however note that the previously discussed (less safe) method is still occasionally useful in practice. As we move through the book, we will also learn about specialized methods for reading more structured data from files, such as tabulated (e.g., CSV) datasets.

**Ensuring the existence of output directories:** When writing text to a file, we need to make sure that the directory containing the future file already exists, otherwise `python` will throw an error. For example, the following code will cause an error if the directory `documents/shark_data` does not exist beforehand:

```
with open("documents/shark_data/shark_species.txt", "wt") as fout:
    fout.write("Carcharodon carcharias\nIsurus paucus\nCetorhinus maximus")
```

Thus, we need to create the directory before executing the above code. Fortunately, we can use `python` to automatically generate the directory if it does not already exist, using the `makedirs` function in the `os` module as follows:

```
import os # load the os module if needed
os.makedirs("documents/shark_data", exist_ok=True)
```



**Working with gzip-compressed files:** Many of the data files encountered in biology today reach gigabytes or even terabytes in size, causing challenges in terms of storage space and network bandwidth. A common approach to ameliorating this issue is to compress data, for example temporarily for transfer or permanently for archival. Even lossless (i.e., fully reversible) file compression can lead to substantial reductions of file sizes, sometimes by a factor of 10 or more. One of the most well-established lossless compression formats for files, especially in the sciences, is gzip. Virtually any file type can be compressed using gzip, including fasta files (used for storing DNA sequences) and CSV/TSV files, and numerous free software exist for handling gzip files. By convention, the extension of gzipped files is amended with a `.gz`, for example `fasta.gz` or `csv.gz`, to clarify that a file has been compressed while keeping the original file type easily recognizable. Python seamlessly supports directly reading and writing gzipped files via the `gzip` package. As an example, consider the following gzipped plain text file, listing over one million animal species names (one species per line):

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`animal_species_names.txt.gz`".

Without compression, this file would take about 30 MB, while the compressed version only takes 8.4 MB. To load the contents of this text file as a list of strings (one per line), we can use the function `open` in the `gzip` package, much like we would use `python`'s standard `open` function:

```
import gzip
with gzip.open("data/animal_species_names.txt.gz", "rt") as fin:
    lines = fin.read().splitlines()
```

Let's check how many lines of text we loaded:

```
print("Loaded {0} lines".format(len(lines)))
```

```
Loaded 1454912 lines
```

Note that if we had used `python`'s standard `open` function instead of `gzip.open`, we would be reading the compressed version of the data without decompressing it, which would essentially appear to be gibberish.

★ **Exercise 48:** Construct a `python` code that uses the `with` statement to write the following precise text (representing surveyed shark weights) to a text file:

```
Carcharodon carcharias:
  1830, 2039, 1620

Isurus paucus:
  51, 69, 72.5, 43.8
```

★ **Exercise 49:** Consider the following text file, containing several animal species names:

http://www.loucalab.com/archive/BioDataAnalysisPython
→ file "`animal_species_names_1000.txt`".

If you open this file in any regular text editor you will notice that it lists one latin binomial name (e.g., `Leucascus albus`) per line, except for some lines which begin with a `#` symbol (these are comment lines). Write a `python` code that loads the species names from the above text file, determines the set of all genera encountered, counts the number of species listed for each genus, and writes the genus names and the associated number of species to a new text file called `animal_genus_sizes.txt` (one genus per line in alphabetical order, avoiding duplicates). A excerpt of the desired output format is shown below:

```
Achramorpha: 8
Alcinoe: 2
Amphiute: 2
Aphroceras: 19
Arthuria: 10
```

*Hint: Use a* `set` *to determine the unique genera found, then convert that set into an alphabetically sorted list using the* `sorted` *function, and then use a* `for` *loop to write out the genus names and their number of species. To count the number of species encountered for a given genus you may use the string method* `count`*. Note that you can use* `for` *loops inside a* `with-open` *statement.*

★★ **Exercise 50:** Consider the genome assembly of *Escherichia coli* strain ZZb4, stored in the following fasta file [31]:

http://www.loucalab.com/archive/BioDataAnalysisPython
→ file "`Escherichia_coli_ZZb4_GCF_013403045.1.fasta`".

Recall that a fasta file is merely a plain text file with a particular structure, suitable for storing multiple DNA sequences. Specifically, each line in the above fasta file is either a DNA sequence (consisting entirely of the characters `A, C, G, T`) or a title/metadata line (starting with a `>`) that provides information about the subsequent DNA sequence. Each DNA sequence listed represents a small contiguous part of the *E. coli* genome, called *contig*. Genome assemblies consisting of fewer and longer contigs are considered to be less fragmented and thus of higher quality. The goal of this exercise is to compute an important summary statistic representing this notion of fragmentation, called N50. Given a genome assembly such as the above, consisting of $n$ contigs, N50 is defined as the length of the longest contig such that all shorter contigs make up less than 50% of the total assembly. To compute the N50, we can thus sort all contigs in order of decreasing length (thus obtaining a list of lengths $L_1 \geq L_2 \geq .. \geq L_n$), find the first contig in the sorted list up until which 50% or more of the cumulative sum of the assembly's length has been covered (i.e., find the smallest $m \in \mathbb{N}$ such that $\sum_{k=1}^{m} L_k \geq 0.5 \cdot \sum_{k}^{n} L_k$), and set N50 to the length of that contig (i.e., N50=$L_m$). Use the above approach to compute the N50 of the

provided *E. coli* genome assembly.

*Hint: Note that the N50 only depends on the lengths of the contigs, not their actual nucleotide contents. You could loop through the fasta file line by line, building a list of the contig lengths encountered, then determine the total assembly size using the* `numpy` *function* `sum`*, sort the list using the function* `sorted`*, and use a loop to determine the first element of that list up until which the sum of lengths is at least 50% of the assembly size. To sort a list in decreasing order, use the optional argument* `reverse=True`*.*

## 4.14 Batch processing multiple files

A common situation in data analysis is the need to automatically process a large number of similar files. For example, we may have a collection of genome sequences for multiple species, each stored in a separate fasta file, and we need to check how prevalent a specific gene sequence is across those genomes. In `python`, we can easily find all fasta files in a directory, and use a loop to process them one by one. Similarly, we may have a collection of TSV files listing GPS coordinates of multiple tracked animals (one file per animal). Again, we can use `python` to find and process all such TSV files in a directory.

Let's consider the following example collection of genomes for 10 *Bacillus subtilis* strains, downloaded from NCBI and stored as individual fasta files in a single directory, available (as a ZIP archive) at:

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`Bacillus_subtilis_genomes.zip`".

Upon downloading the above ZIP archive and expanding it, we obtain a directory called `Bacillus_subtilis_genomes` that contains multiple fasta files (one per genome). Suppose that we wish to determine the prevalence of the precise DNA sequence `CTTACATGCAGGAAGGGATGCTTTTCCATTCCCT` in the genomes (for context: this sequence is part of a mycobactin peptide synthetase *mbtE* gene allele). We begin by finding all fasta files using the `glob` function in the `glob` module. This function takes as single argument a string specifying the file path pattern to match, and returns a list of strings representing the paths of all files found:

```
import glob
fasta_paths = glob.glob("data/Bacillus_subtilis_genomes/*.fasta")
```

Here, the "`data/`" part specifies that the directory `Bacillus_subtilis_genomes` is inside another directory named `data`. The "`*`" character is called a wildcard, and it represents ("captures") any arbitrary text in the file paths. By using this wildcard we don't need to worry about the precise file names in our dataset — all we need to know is the directory where the fasta files are located and their extension (`.fasta`). Hence, the above pattern tells `python` to search for all files ending with `.fasta` and located inside `data/Bacillus_subtilis_genomes`. For more details and examples using `glob` see its official documentation. The above code will yield a list of strings representing the file paths of all fasta files found. Let's inspect the first few found elements in this list to confirm our expectation:

```
print(fasta_paths[0:3])
```

```
['data/Bacillus_subtilis_genomes/GCF_000934165.1.fasta', 'data/
    Bacillus_subtilis_genomes/GCF_000830695.1.fasta', 'data/Bacillus_subtilis_genomes/
    GCF_000830645.1.fasta']
```

Next, we create a `for` loop that iterates over all fasta file paths, loads each file's content as a

string, and checks if the focal DNA sequence is found in that string. Note that each fasta file may consist of a mix of forward and reverse DNA strands, which means that we must also look for the focal sequence's reverse-complement version.

```
seq             = "CTTACATGCAGGAAGGGATGCTTTTCCATTCCCT"
seq_rev_compl   = "AGGGAATGGAAAAGCATCCCTTCCTGCATGTAAG"
Nfound          = 0 # number of genomes containing the sequence
for fasta_path in fasta_paths:
    with open(fasta_path, "rt") as fin:
        fasta_content = fin.read()
    Nfound += ((seq in fasta_content) or (seq_rev_compl in fasta_content))

print("Sequence was found in %d out of %d genomes"%(Nfound,len(fasta_paths)))
```

```
Sequence was found in 7 out of 10 genomes
```

While in the above example the number of genomes was small enough and the computational task simple enough to alternatively do it "by hand", in many realistic situations we are are dealing with hundreds or even thousands of files and much more complex analyses. Hence, the ability to automatically process large batches of files using programming is a very useful skill.

★ **Exercise 51:** Consider the example analysis of multiple genomic fasta files in Section 4.14. Modify the code presented therein to count and print out the number of genomes in which the focal DNA sequence is found, separately for the forward and reverse-complement versions.

## 4.15 Numpy arrays

In Section 4.10 we learned about lists as a means to represent multiple numeric (or other) variables in a specific order (i.e., along a single dimension). One might think of a list as being analogous to a single column in a spreadsheet. In many situations, however, we need to work with "tabular" data, i.e., a 2-dimensional arrangement analogous to a spreadsheet containing multiple rows and columns. In typical time series datasets, for example, each row represents a different time point, and each column represents a different physical/biological variable measured over time. In a cross-sectional study of an animal population, each row may represent an individual animal and each column a different trait measured. In an age-structured linear population model, each entry of such a table represents the fluxes from one age group into another. In some situations even higher-dimensional data collections are encountered, for example a 3-dimensional structure would be analogous to a stack of multiple spreadsheets. In `python` this type of data can be represented using `numpy` arrays. We have already encountered the `numpy` package as a collection of various mathematical functions; however, `numpy` is actually best known for its powerful arrays.

Before we move into the specifics of `numpy` arrays, it's worth clarifying for a moment why we would need them in the first place. Indeed, one could argue that any table can in principle be represented as a list of sublists, just like a spreadsheet can be seen as a collection of columns. In practice, however, `numpy` arrays provide a much more efficient means to handling such data (in terms of memory and computing time), especially for large numeric datasets. Even in cases where we just want to store a sequence of numbers in a 1-dimensional arrangement (i.e., similar to a numeric list), `numpy` arrays tend to be a more efficient solution. Further, as we will see below, `numpy` arrays allow us to formulate mathematical operations in a more concise and elegant way than lists would. In fact, the `numpy` package includes many sophisticated mathematical routines (for example, from linear algebra) specifically designed for `numpy` arrays. Knowing how to use both lists and `numpy` arrays depending on the circumstance is as important as knowing how to

use both nails and screws: some jobs are much better done with one or the other.

In this section we will learn how to construct and modify `numpy` arrays, how to load the contents of text files as `numpy` arrays, and how to perform basic mathematical calculations with them. Since this is an introductory course, we will focus on 1- and 2-dimensional arrays. Throughout, we assume that we have already imported the `numpy` package, as follows:

```
import numpy
```

### 4.15.1 Creating arrays

A `numpy` array (hereafter simply "array") is a variable that represents a 2- or higher-dimensional arrangement of values, typically (but not necessarily) numbers. A 2-dimensional numeric array, for example, could be used to represent the following numeric matrix:

$$\mathbb{A} = \begin{pmatrix} 0.1 & 0.2 & 0.3 & 0.4 \\ 1 & 2 & 3 & 4 \\ 10 & 20 & 30 & 40 \end{pmatrix}. \tag{4.7}$$

To define an array representing the above matrix, we can proceed as follows:

```
A = numpy.array([[0.1, 0.2, 0.3, 0.4], [1, 2, 3, 4], [10, 20, 30, 40]])
```

Observe that we provided a list of sublists as a single argument to the `numpy.array` function. Each of the sublists represents a specific row in the matrix, and hence we are providing a total of 3 sublists (since there are 3 rows), each of length 4 (since there are 4 columns). To improve readability, we could have also spread the sublists across multiple lines and included some extra spaces:

```
A = numpy.array([[0.1, 0.2, 0.3, 0.4],
                 [1,   2,   3,   4],
                 [10,  20,  30,  40]])
```

There are many other ways to create `numpy` arrays, each suitable for different situations. For example, we can easily create a numeric array of any arbitrary size and consisting entirely of zeros using the function `numpy.zeros`, as follows:

```
B = numpy.zeros([3,4], dtype=float)
```

Hence, `B` will represent a $3 \times 4$ numeric matrix (3 rows $\times$ 4 columns) consisting entirely of zeros. Arrays consisting entirely of zeros are often used when we have a subsequent computation that gradually fills in values at specific positions of the array.

We can also easily create a 1-dimensional array (i.e., a single row) containing an evenly spaced sequence of numbers using the functions `numpy.linspace` and `numpy.arange`. For example, to

create a 1-dimensional array containing 9 evenly spaced numbers between 0 (inclusive) and 1 (inclusive) arranged in a single row, we can write:

```
C = numpy.linspace(0,1,9)
```

The array `C` will thus contain the following numbers:

```
[0.    0.125 0.25  0.375 0.5   0.625 0.75  0.875 1.   ]
```

Alternatively, to create a 1-dimensional array containing a sequence numbers between 0 (inclusive) and 1 (exclusive) with a step of 0.1, we can write:

```
D = numpy.arange(0,1,0.1)
```

which would yield the following array:

```
[0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9]
```

Hence, `numpy.linspace` allows us to create evenly spaced sequences of a specific *length*, while `numpy.arange` allows us to create evenly spaced sequences with a specific *step*. Both functionalities may be used, for example, to specify a spatial or temporal grid for biogeochemical or ecological models, or for plotting mathematical functions (as we will see later). Whether we think of a 1D `numpy` array as a row or column is usually a matter of personal preference, however in some situations the distinction becomes important. For example, by default `numpy` will treat 1D arrays as rows when converting a list of 1D arrays into a 2D array.

★ **Exercise 52:** Use the function `numpy.linspace` to define a 1-dimensional array consisting of the numbers $0, 0.5, 1, 1.5, 2, 2.5, ..., 1000$.

### 4.15.2 Getting information about an array

To see the contents of a `numpy` array we can use the familiar `print` function, for example:

```
print(C)
```

```
[0.    0.125 0.25  0.375 0.5   0.625 0.75  0.875 1.   ]
```

If we want to modify the number formatting when printing a `numpy` array, we can use the `with numpy.printoptions` statement with appropriate optional arguments. For example, to print the array with only two decimal digits:

```
with numpy.printoptions(precision=2):
    print(C)
```

```
[0.   0.12 0.25 0.38 0.5  0.62 0.75 0.88 1.  ]
```

For a full list of `numpy` print options see here.

If we want to know the shape of the array, for example the number of rows and columns in the case of a 2-dimensional array, we can use the attribute `shape` (an attribute is like a variable, but tied to a specific object). The `shape` attribute of an array is a tuple with as many entries as there are dimensions (axes) in the array. For example, consider the following array:

```
A = numpy.array([[0.1, 0.2, 0.3, 0.4],
                 [1,   2,   3,   4],
                 [10,  20,  30,  40]])
```

To print its shape to the screen, we can write:

```
print(A.shape)
```

```
(3, 4)
```

Hence, `A.shape[0]` is the number of rows and `A.shape[1]` is the number of columns in `A`.

★ **Exercise 53:** Predict the output of the following code, without running it:

```
A = numpy.array([[3, 4, 8], [2, 9, 0]])
print(numpy.zeros(A.shape))
```

★ **Exercise 54:** Define a 2-dimensional `numpy` array representing the following $4 \times 2$ matrix:

$$\mathbb{M} = \begin{pmatrix} 57 & 8 \\ 6 & -2 \\ -5 & 78 \\ 10 & 15 \end{pmatrix}. \tag{4.8}$$

Print the array to the screen to confirm the correctness of your code.

### 4.15.3 Array indexing

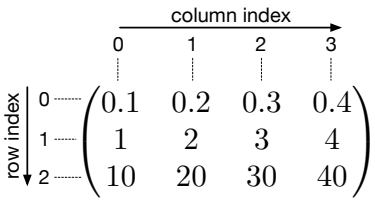
Accessing a specific element in a `numpy` array works similarly to lists, i.e., based on integer positions (indices) of elements. One difference is that for multi-dimensional arrays we need to provide multiple indices, i.e., one for each dimension (aka. axis). Consider, for example, the following 2-dimensional array:

```
A = numpy.array([[0.1, 0.2, 0.3, 0.4],
                 [1, 2, 3, 4],
                 [10, 20, 30, 40]])
```

representing the following matrix:

$$\mathbb{A} = \begin{pmatrix} 0.1 & 0.2 & 0.3 & 0.4 \\ 1 & 2 & 3 & 4 \\ 10 & 20 & 30 & 40 \end{pmatrix}. \tag{4.9}$$

As with list indexing, row and column indices in arrays start counting at 0, as summarized in the schematic below:



To access the third element in the first row of $\mathbb{A}$, we would thus write `A[0,2]`, where the `0` refers to the first row and the `2` refers to the third column. Similarly, to access the element in the last row and 2nd-last column, we can write `A[-1,-2]`. We can also access a range of multiple consecutive rows or columns, by specifying the beginning (inclusive) and end (exclusive) of the desired index range. For example, the following code prints out all elements in the first row and in the second, third or fourth column:

```
print(A[0,1:4])
```

```
[0.2 0.3 0.4]
```

Note that `1:4` stands for the indices `1, 2, 3` (i.e., excluding 4). Open-ended ranges are also allowed, for example `:4` means all indices from 0 to 3, while `2:` means all indices from 2 until the end. Thus, the following code prints all elements in the first row with column indices 2 or greater:

```
print(A[0,2:])
```

```
[0.3 0.4]
```

We can also extract an arbitrary (not necessarily contiguous) subset of rows, by specifying their indices as a list of integers. For example, the following code prints out all elements in the first row and in the second or fourth column:

```
print(A[0,[1,3]])
```

```
[0.2 0.4]
```

Note that element will be extracted in the order corresponding to the provided indices. For example, the following code extracts the same part of `A` as above, but in a different order:

```
print(A[0,[3,1]])
```

```
[0.4 0.2]
```

We can also use indexing to extract a *slice* from a multidimensional array, such as a specific row or column from a 2-dimensional array. For example, to extract the second column of `A`, thus obtaining a 1-dimensional array, we can write:

```
second_column = A[:,1]
```

Observe that we use the colon `:` without any particular indices for the first axis, which tells `python` to get elements from all rows, while the index `1` provided for the 2nd axis tells `python` to only get elements from the second column. Similarly, we can extract any arbitrary row:

```
first_row = A[0,:]
```

Both `second_column` and `first_row` will be 1-dimensional `numpy` arrays:

```
print(second_column)
```

```
print(first_row)
```

```
[ 0.2  2.  20. ]
[0.1 0.2 0.3 0.4]
```

To extract multiple rows at once, we can provide the appropriate range of indices either using a colon or using a list of indices. For example, the following code extracts the first 2 rows of `A`, thus yielding a new 2-dimensional array with 2 rows and 4 columns:

```
first_two_rows = A[0:2,:]
print(first_two_rows)
```

```
[[0.1 0.2 0.3 0.4]
 [1.  2.  3.  4. ]]
```

As before, open-ended index ranges are also allowed. For example, the following code extracts all rows except the first one:

```
all_rows_but_first = A[1:,:]
print(all_rows_but_first)
```

```
[[ 1.  2.  3.  4.]
 [10. 20. 30. 40.]]
```

Indexing of `numpy` arrays is a deep topic, comprising many more methods and nuances than covered here. To learn more see `numpy`'s in-depth official user guide.

**Modifying parts of an array:** So far we have seen how we can extract the values of an array using a variety of indexing methods. However, we can also use those indexing methods to modify the elements of an array in place. This if useful, for example, if we want to apply some type of computation to only specific columns of an array, or wish to substitute missing data points. By far the simplest situation is when we want to replace a subset of the array's elements with a single value. For example, to replace all elements in the first and second column of `A` with the value 5, we can write:

```
A[:,0:2] = 5
```

Let's see what `A` looks like now:

```
print(A)
```

```
[[ 5.   5.   0.3  0.4]
 [ 5.   5.   3.   4. ]
 [ 5.   5.  30.  40. ]]
```

Similarly, to replace all elements in the last row, but restricted to the first two columns, with the value 6, we can write:

```
A[-1,0:2] = 6
print(A)
```

```
[[ 5.   5.   0.3  0.4]
 [ 5.   5.   3.   4. ]
 [ 6.   6.  30.  40. ]]
```

The situation is more complex if we wish to replace multiple elements with multiple corresponding values. In that case, the right hand side of the assignment must match the indexed subset on the left hand side in terms of shape (or must be appropriately broadcastable, but we will not discuss that option here). For example, to replace all elements in the first row of `A` with an equal number of corresponding values, we must provide as many values as there are columns in `A`, as illustrated below:

```
A[0,:] = [11, 22, 33, 44]
print(A)
```

```
[[11. 22. 33. 44.]
 [ 5.  5.  3.  4.]
 [ 6.  6. 30. 40.]]
```

If we instead want to replace the values in multiple rows, then we should provide an appropriately sized list of sublists, with each sublist representing the values to be assigned to a different row, for example:

```
A[[0,1,2],:] = [[11, 22, 33, 44], [110, 220, 330, 440], [1100, 2200, 3300, 4400]]
print(A)
```

```
[[  11.   22.   33.   44.]
 [ 110.  220.  330.  440.]
 [1100. 2200. 3300. 4400.]]
```

The requirement that the shapes of both sides of the assignment must be compatible makes intuitive sense: Unless we provide a reasonably matched number of replacement values, `python` cannot know which value it should assign to which slot of the array.

If we want to transpose a 2-dimensional array, i.e., swap its rows and columns, we can use the array method `transpose`, which returns the transposed version of the array:

```
A = A.transpose()
print(A)
```

```
[[  11.  110. 1100.]
 [  22.  220. 2200.]
 [  33.  330. 3300.]
 [  44.  440. 4400.]]
```

Transposition is useful for some algebraic calculations. Some statistical functions also expect data arrays in a specific orientation (e.g., rows corresponding to samples and columns corresponding to variables), in which case transposition may be needed to satisfy that expectation.

**Examples:** The following two simple examples demonstrate the use of the `shape` method and array indexing. The first example shows how we can compute the sum of all elements in the first column of a 2-dimensional array, using a loop that iterates over all rows and adds up the appropriate elements:

```
S = 0 # initialize our summation variable
for r in range(A.shape[0]):
    S += A[r,0] # add the element in row r and column 0
print("Sum of first column:",S)
```

```
Sum of first column: 110.0
```

Similarly, to add all elements in the first row, we could use a loop that iterates over all columns and adds up the corresponding elements in the first row, as follows:

```
S = 0 # initialize our summation variable
for k in range(A.shape[1]):
    S += A[0,k] # add the element in row 0 and column k
print("Sum of first row:",S)
```

```
Sum of first row: 1221.0
```

As we will learn soon, there are more concise and efficient ways than loops to achieving the above tasks, using special routines provided by `numpy`.

★ **Exercise 55:** Consider a 2-dimensional `numpy` array, such as the following:

```
A = numpy.array([[1,    2,    3,    4,    5],
                 [10,   20,   30,   40,   50],
                 [100,  200,  300,  400,  500],
                 [1000, 2000, 3000, 4000, 5000]])
```

Use `numpy` array indexing/slicing, to extract and print out the elements restricted to the first 2 rows and to the second or fourth column (retaining the original relative arrangement of those elements).

★ **Exercise 56:** Predict what the following code will print to the screen, without running it:

```
A = numpy.array([[1,    2,    3,    4,    5],
                 [10,   20,   30,   40,   50],
                 [100,  200,  300,  400,  500],
                 [1000, 2000, 3000, 4000, 5000]])
print(A[-1,2:4],"\n")
print(A[:,2:],"\n")
print(A[1:,:3],"\n")
print(A[[0,3],[1,2]],"\n")
print(A[[2,1,0,1,0],:],"\n")

A[0:2,:] = 0
print(A,"\n")
```

### 4.15.4 Loading arrays from a file

It is also possible (and common) to load an entire array from a text file containing tabulated numeric data. Consider, for example, the following dataset of monthly phytoplankton productivity

measurements by the Hawaii Ocean Time Series program [32, 33], stored as a comma-separated-value (CSV) file:

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "HOTS_monthly_pp.csv".

The first few lines of this CSV file look as follows:

```
#
# time,chlA,phaeopigments,light12
1989.5,0.332748,0.263504,3.27627
1989.58,0.346524,0.264326,3.65575
1989.67,0.308802,0.372161,5.02991
1989.75,0.197382,0.28989,1.85003
```

To load a the contents of such a CSV file as an array, we can use the `numpy` function `genfromtxt` as follows:

```
data = numpy.genfromtxt("data/HOTS_monthly_pp.csv", delimiter=",")
```

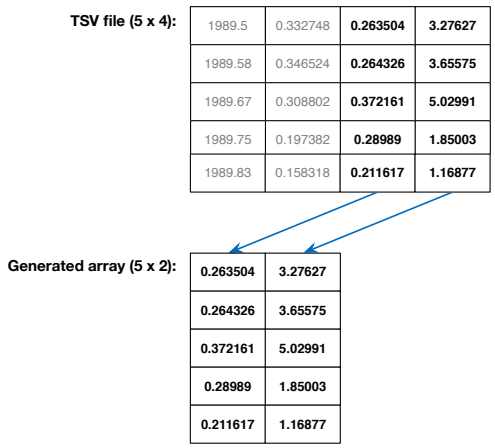
Henceforth, `data` will be a 2-dimensional numeric array, representing the tabulated contents of the file. Note that by default any comment lines in the file (i.e., starting with `#`) are ignored. To load a tab-separated-values (TSV) file, we need to adjust the `delimiter` argument:

```
data = numpy.genfromtxt("data/HOTS_monthly_pp.tsv", delimiter="\t")
```

It is also possible to only load a specific subset of columns, by passing a list of desired column indices as a `usecols` argument. For example, to only load the third and fourth column of a TSV table, we could write:

```
data = numpy.genfromtxt("data/HOTS_monthly_pp.tsv",\
                        delimiter="\t", usecols=[2,3])
```

In this case the column indices of the constructed `numpy` array will not match the original column indices in the input file. In the example above, the first two columns of the generated array correspond to the third and fourth columns, respectively, of the input file. The following figure illustrates this effect:




Loading only a subset of columns may be useful, for example, if the input file has a large number of columns that we don't actually need. Another useful feature of `genfromtxt` is that it can automatically load gzip-compressed files, provided that their file extension is `.gz`. For additional

options see the documentation of `genfromtxt`.

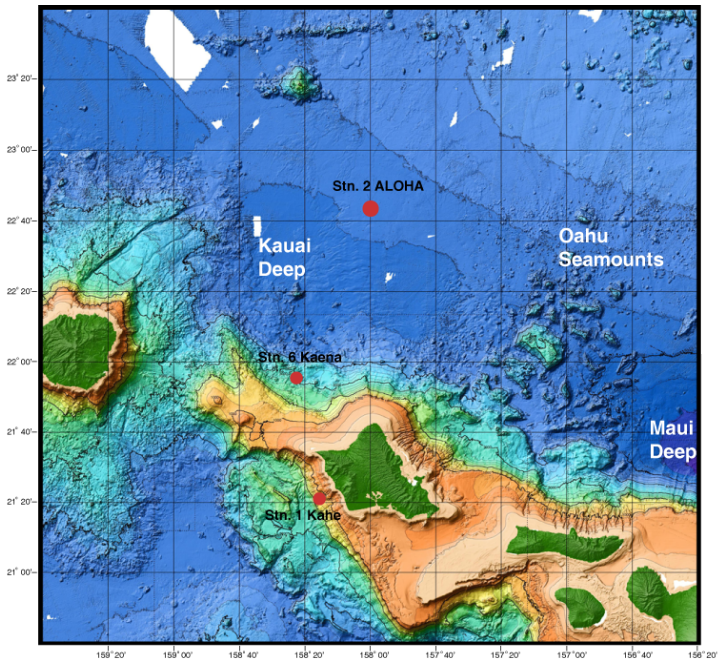



**Figure 4.3:** Location of station ALOHA, the main sampling site of the Hawaii Ocean Time Series program [32]. Image courtesy HOTS.

**Row names and columns with different data types:** Note that `numpy` arrays cannot directly store row or column names, i.e., rows and columns are entirely identified based on their integer positions. If we want to associate specific names with rows and/or columns, we need to either store those in a separate variable (such as a list of strings), or store them as part of the array (e.g., as a separate column or row, respectively), or use what's called a "structured array" (which we will not be discussing here). Let's look at the first option. Consider, for example, the following TSV file, listing average adult body masses for various primate species, separated by sex [34]:

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`primate_body_masses.tsv`".

The first column lists species names, the second column lists male body masses, and the third column lists female body masses. Let's look at the first few rows of this file:

```
Mirza coquereli 0.304   0.326
Microcebus myoxinus 0.031   0.03
Microcebus rufus    0.043   0.042
Lemur catta 2.21    2.21
```

We cannot simply load this file as a single numeric array, because the first column contains text rather than numbers. Instead, we can load the first column of the table separately from the other two (numeric) columns:

```
# load the first column as a string array
species_names = numpy.genfromtxt("data/primate_body_masses.tsv", dtype=str,
                                 delimiter="\t", usecols=[0])

# load the other 2 columns as a numeric array
```

```
body_masses = numpy.genfromtxt("data/primate_body_masses.tsv", dtype=float,
                               delimiter="\t", usecols=[1,2])
```

Observe that we explicitly specified the data type for the two arrays using the argument `dtype`, to help `numpy` decide how to interpret the file's contents in each column. The data type `str` tells `numpy` to load data as strings, which is appropriate for species names. The data type `float` tells `numpy` to load data as numbers, which is appropriate for the body masses. The array `species_names` will be 1-dimensional since we only loaded a single column, and can henceforth be interpreted as a list of row names for the 2-dimensional array `body_masses`. The relationship between the two arrays is illustrated in the figure below:

| **species_names** |
|---|
| Mirza coquereli |
| Microcebus myoxinus |
| Microcebus rufus |
| Lemur catta |
| ⋮ |

| **body_masses** | |
|---|---|
| 0.304 | 0.326 |
| 0.031 | 0.03 |
| 0.043 | 0.042 |
| 2.21 | 2.21 |
| ⋮ | ⋮ |

Later on, we will learn about another, more versatile, data structure for storing tabulated data, called `pandas` dataframes (Section 5.5). Dataframes natively support row and column names as well as mixed data types, however they are somewhat more complex than `numpy` arrays.

**Loading tables with a header:** Many CSV or TSV tables include a header row, i.e., a non-comment line containing column names immediately preceding the data rows. An example TSV file with a header row (henceforth called a HTSV file, denoted by the file extension `htsv`) is shown below:

```
# Original data downloaded on June 6, 2021, from:
#   https://hahana.soest.hawaii.edu/hot/hot-dogs/ppextraction.html
#
time    chlA    phaeopigments   light12
1989.5  0.332748    0.263504    3.27627
1989.58 0.346524    0.264326    3.65575
```

If we were to load such an HTSV table using the previously discussed methods, we would end up with an extra row at the top of the array containing `NaN` values (`NaN` stands for "not a number").

This is because by default `genfromtxt` tries to interpret all values as numbers, but fails to do so when confronted with text that does not represent a number. While `genfromtxt` does not provide an option to automatically ignore header rows, we can easily omit the first row from the loaded array in retrospect using basic indexing, for example as follows:

```
# load numeric table
data = numpy.genfromtxt("data/HOTS_monthly_pp.tsv", delimiter="\t")

# omit the first row from the loaded array, which represents the header row
data = data[1:,:]
```

Or condensed into a single line of code:

```
data = numpy.genfromtxt("data/HOTS_monthly_pp.tsv", delimiter="\t")[1:,:]
```

★ **Exercise 57:** Consider the following gzipped TSV file listing the coordinates of all major coastlines in the world (latitudes in the first column, longitudes in the second column):

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`coastline_coordinates_50m.tsv.gz`".

Load this file as a `numpy` array, and print out the number of rows and columns loaded.

★ **Exercise 58:** Consider the following dataset of annual animal population size estimates in the North-American tundra [35], stored in TSV file format (one row per year, the first column lists years, each of the remaining columns lists densities of a different species):

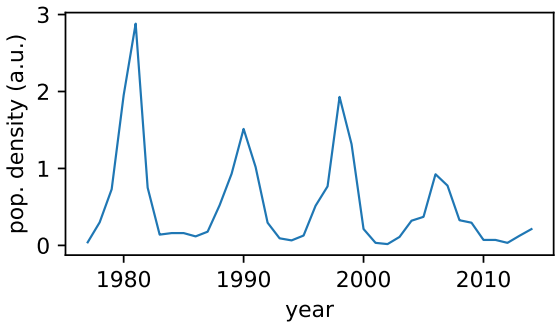
> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`Hudson_Bay_fur_bearing_animals.tsv`".

Load the first 3 columns of this file into a numeric `numpy` array using the function `numpy.genfromtxt`, and print its shape to the screen using the `print` function.

★ **Exercise 59:** Consider the following dataset of annual snowshoe hare population density estimates in Kluane National Park [36], stored in HTSV file format (one row per year, years in the first column, densities in the second column):

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`Krebs2011_hare.htsv`".

For reference, the following figure shows the hare population density estimates over time:




Load the data into a numeric `numpy` array using the function `numpy.genfromtxt`, and compute the average, the highest and the lowest population density across all years using a `for` loop that

iterates over all rows. Also print out the years in which the population density was more than double that of the previous year.

*Hint: Note that this file contains a header row, which should be discarded from the loaded array. If you can't figure out how to compute all of the requested quantities, at least try the easier ones.*

★ **Exercise 60:** Sexual dimorphism in size and morphology is ubiquitous in the animal world and has long been of great interest to biologists [37–39]. In this exercise we consider a dataset of average adult body masses for multiple primate species, separated by sex [34], stored in the form of a TSV file (one row per primate species, species names in the first column, male and female body masses in the second and third column, respectively):

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`primate_body_masses.tsv`".

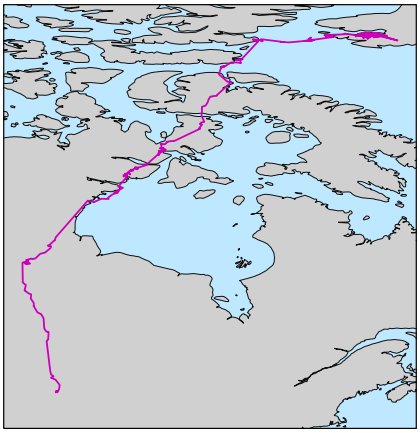
Load the first column of this file as a 1-dimensional string array, and the other two columns as a 2-dimensional numeric array, as demonstrated in Section 4.15.4. Use a `for` loop to iterate over each species and print out the names of the species for which the ratio of male to female body mass is greater than 2 or less than 0.5 (do also print out the male/female mass ratios of those species).

**Figure 4.4:** Female and male northern white-cheeked gibbons (*Nomascus leucogenys*), showing the strong sexual dimorphism that exists in some animal species. Photo credit Jinterwas, CC-BY-2.0.

★ **Exercise 61:** In this exercise we consider the migratory route of a peregrine falcon [40], recorded as a sequence of coordinates in the following TSV file:

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`falcon_migration_route_Burnham2012.tsv`".

Note that this TSV (tab-separated-values) file contains two numeric columns, listing latitudes and longitudes in degrees, respectively, recorded at successive time points (one row per recorded time point). The map below shows the route for reference.



The goal of this exercise is to compute the total distance traveled by the falcon. Perform the following tasks:

1. Download and load the above TSV file into `python` as a 2-dimensional `numpy` array, and print the total number of recorded points (number of rows), the coordinates of the first point (first row) and the coordinates of the last point (last row) to the screen.
2. Compute the total distance traveled, by adding all the great-circle distances (arc lengths) between successively recorded coordinates. To compute the great circle distance $D$ between two points with coordinates $(x_1, y_1)$ and $(x_2, y_2)$ (where $x$ are longitudes and $y$ are latitudes, in degrees), use the following (Vincenty) formula:

$$D = R \cdot \arctan 2(A, B), \tag{4.10}$$

where we defined:

$$\begin{aligned} A &:= \sqrt{(\cos(y_2)\sin(\delta))^2 + (\cos(y_1)\sin(y_2) - \sin(y_1)\cos(y_2)\cos(\delta))^2}, \\ B &:= \sin(y_1)\sin(y_2) + \cos(y_1)\cos(y_2)\cos(\delta), \end{aligned} \tag{4.11}$$

and where $\delta = |x_1 - x_2|$, $R = 6371$ km is Earth's average radius and $\arctan 2$ is the 2-argument arctangent.

*Hint: The required trigonometric functions* `sin`, `cos`, `arctan2` *and the constant* `pi` *are all available in the* `numpy` *package. Keep in mind, however, that the functions* `sin` *and* `cos` *expect angles to be given in radians, rather than degrees; to convert angles from degrees to radians, multiply them by* $\pi/180$*. You can use a loop to iterate over all but the last rows in the array, to add up the individual arc lengths.*

3. Compute the great-circle distance between the first and last recorded coordinates, i.e., the hypothetical distance if the falcon had taken the shortest possible path. How does this distance compare to the actual distance traveled, computed earlier?
4. Use the function `is_ocean` in the `globe` module of the `global-land-mask` package to determine the fraction of time points during which the falcon was above the ocean.

*Hint: For any point with a given latitude and longitude (*`lat` *and* `lon`*, in degrees), the* `python` *expression* `globe.is_ocean(lat,lon)` *evaluates to* `True` *if the point is above the ocean. For more detailed instructions and examples see here.*

### 4.15.5 Math with arrays

One of the benefits of `numpy` arrays is that they greatly facilitate mathematical calculations for large datasets. This is because we can write mathematical expressions involving arrays as if

they were simple numbers, such that `python` evaluates these expressions for each element in the arrays. For example, suppose we have two `numpy` arrays, `infections` and `deaths`, which list the monthly number of reported infections and deaths, respectively, for a transmissible infectious disease:

```
infections = numpy.array([220, 100, 40, 28, 32, 25, 24, 32, 80])
deaths     = numpy.array([11, 10, 5, 7, 4, 10, 6, 16, 20])
```

Suppose that we want to estimate the conditional probability of death given a reported infection, a quantity commonly known as case-fatality-ratio (CFR) [41, 42], separately for each month. One rough estimation method would be to divide the monthly number of deaths by the monthly number of reported infections, i.e., compute the ratios 11/220, 10/100 and so on. We could achieve this using a loop that iterates over all months, but `numpy` arrays actually allow us to achieve this in a much simpler way:

```
monthly_CFR = deaths/infections
```

The above statement will create a new `numpy` array of the same shape as `deaths` (and `infections`), whose elements are the pairwise ratios of the elements of `deaths` over the corresponding elements in `infections`, and assign that new array to the variable `monthly_CFR`. Thus, `monthly_CFR` will be a 1-dimensional array (a single row) containing the following values:

```
[0.05  0.1   0.125 0.25  0.125 0.4   0.25  0.5   0.25 ]
```

Other mathematical operations such as multiplication, addition, subtraction and power raising work similarly. These operations can also accept arguments of different shapes, for example a 1-dimensional array and a single number; in that case, the single number is temporarily treated as a repeated sequence of the same shape as the other array. For example, to divide every element of the `monthly_CFR` array by 100 (so converting CFRs into percentages), we could write:

```
monthly_CFR_percent = monthly_CFR/100
```

The new array `monthly_CFR_percent` would thus contain the following values:

```
[0.0005  0.001   0.00125 0.0025  0.00125 0.004   0.0025  0.005   0.0025 ]
```

Standard mathematical functions provided by `numpy`, such as the exponential and logarithmic functions (`exp` and `log`), trigonometric functions (`sin`, `cos` etc) or rounding functions (`round`, `floor` and `ceil`), can also be applied to an entire array; in that case, the returned variable will be an array of the same shape as the input array, whose elements are the values returned by the mathematical function applied to each element of the input array. For example, it is common in ecological analyses to log-transform population size measurements prior to a statistical analysis, i.e., replacing them with their logarithms. If population sizes are stored in a `numpy` array called `sizes`, then their logarithms can easily be computed as follows:

```
logsizes = numpy.log(sizes)
```

Arrays can also serve as input into functions that compute summary statistics of number sets, such as the arithmetic mean (`mean`), standard deviation (`std`), median (`median`), sum (`sum`), maximum (`max`) or minimum (`min`). These functions generally do not return an array of the same shape as the input array, but instead by default return a single number that summarizes/represents the entire input array. For example, to compute the average CFR and its standard

deviation over the entire measurement period, we can use the `numpy` functions `mean` and `std`:

```
average_CFR = numpy.mean(monthly_CFR)
std_CFR = numpy.std(monthly_CFR)
```

To compute the maximum and minimum CFR ever encountered, we can write:

```
min_CFR = numpy.min(monthly_CFR)
max_CFR = numpy.max(monthly_CFR)
```

If we instead want to know *which* month exhibited the lowest and highest CFR, we would use the `numpy` functions `argmin` and `argmax`, as follows:

```
min_CFR_month = numpy.argmin(monthly_CFR)
max_CFR_month = numpy.argmax(monthly_CFR)
```

Note the difference between `max_CFR` and `max_CFR_month`:

```
print("max_CFR:",max_CFR)
print("max_CFR_month:",max_CFR_month)
```

```
max_CFR: 0.5
max_CFR_month: 7
```

Hence, the highest CFR was equal to 0.5, and was observed at index 7, which corresponds to the 8-th month in our time series (since `python` indices start counting at 0).

For multidimensional arrays, it is possible to compute summary statistics *across* a specific dimension (aka. axis), rather than across all elements of the array. For example, suppose that we have a 2-dimensional numeric array containing population sizes for multiple species at multiple time points (one row per time point, one column per species), such as the following:

```
popsizes = numpy.array([[0.5,  10,  45],\
                        [0.8,  12,  43],\
                        [0.4,  13,  49],\
                        [0.65, 15,  56],
                        [0.3,  16,  39]])
```

To compute the average population size separately for each species (i.e., averaged over time), we can use `numpy.mean` with the optional `axis` argument:

```
mean_popsizes = numpy.mean(popsizes, axis=0)
print(mean_popsizes)
```

```
[ 0.53 13.2  46.4 ]
```

Hence, `mean_popsizes` is a 1-dimensional array with as many elements as there were columns in `popsizes`, each listing the average across all elements of a specific column of `popsizes`. The `axis` argument thus specifies the axis over which we want to average (in this case, the first axis); that axis is thus absent from the returned array. A similar approach can be used with other `numpy` functions that compute summary statistics, including `sum`, `std`, `median`, `min` and `max`.

★ **Exercise 62:** Suppose that we have have obtained daily average temperatures at a field site over the course of several days, and have stored those temperatures in °F in a numeric list, for example as follows:

```
temperatures_F = [55, 51, 60, 63, 69, 72, 71, 54, 60, 64]
```

After converting the list of temperatures to a `numpy` array, convert its entries to °C using the formula $(°F - 32)/1.8$ and using array operations as described above. Round the resulting temperatures to the closest integer and print them to the screen. Then, compute and print out the average temperature (in °C) across all days.

★ **Exercise 63:** Consider the following HTSV file, listing the proportions of 3 bacterial species in a wastewater treatment plant, measured at weekly intervals (one row per week, one column per species) [43]:

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`WWTP_AOB_abundances_Ofiteru2010.htsv`".

Perform the following tasks:

1. Load the above TSV table into python as a `numpy` array, and compute separately for each species its mean proportion and associated standard deviation, over time. Also compute for each species the *coefficient of variation* (CV, a common measure of relative variability), i.e., the ratio of the standard deviation over the mean, and determine which species exhibits the greatest CV. Try to use the techniques discussed in Section 4.15.5, and avoid using loops.

   *Hint: Note that this TSV file includes a header row, which should be removed from the loaded* `numpy` *array as discussed in Section 4.15.4.*

2. Compute the Pearson correlation coefficient ($r$) between the proportions of the first and second bacterial species across time, and similarly between the first and third species, and between the second and third species.

   *Hint: You can use the* `pearsonr` *function from the* `stats` *module in the* `scipy` *package, by passing the appropriate columns of the loaded array as arguments, similar to the example in Section 4.10.3.*

★ **Exercise 64:** In this exercise we will compute the mean flight velocities of Galapagos albatrosses during their foraging trips to Peru. We will consider the following GPS tracking dataset, recorded in 2008 [44]:

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`multiple_albatross_foraging_routes_Dodge2013.zip`".

Note that the above ZIP archive contains multiple HTSV files (one file per bird, with file names representing bird IDs). In each file, each row stores one record, and flight speeds are stored in the fourth column (in meters per second). Write a `for` loop that iterates over all HTSV files, and computes and prints out the average speed of each bird, with birds listed alphabetically. Do include each bird's ID and corresponding average speed in the printouts. For example, for the first 2 birds the output should look similar to the following:

```
albatross_1094 average speed = 0.441 m/s
albatross_1103 average speed = 0.485 m/s
```

*Hint: You can use the* `glob` *package to find the paths of all HTSV files, as described in Section*

*4.14. To extract the bird ID from each file path, you can use the functions* `basename` *and* `splitext` *in the module* `path` *of the package* `os` *(i.e.,* `os.path.basename` *and* `os.path.splitext`*). You can load the fourth column of each file as a 1-dimensional array as described in Section 4.15.4, and you can compute the mean of that array as described in Section 4.15.5.*

★ **Exercise 65:** In this exercise we consider a TSV table obtained from a survey of the microbial communities living inside the foliage of bromeliads in Brazil [45]:

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`bromeliad_OTU_overlaps_Louca2016_nonames.tsv`".

The table lists pairwise microbiome overlaps between individual bromeliads, with every row corresponding to a distinct bromeliad and every column corresponding to a distinct compared bromeliad. For example, the value in the first row and third column represents the microbiome overlap (fraction of shared microbial species) between the first and third bromeliad. Load the table as a `numpy` array, and compute the average microbiome overlap between all bromeliad pairs. Also compute for each bromeliad its average microbiome overlap with all other bromeliads, and determine which bromeliad has the lowest such value (i.e., is the least similar to the other bromeliads). Do not use any loops.

*Hint: The* `numpy` *functions* `mean` *and* `argmin` *may be of use.*

**Figure 4.5:** *Aechmea nudicaulis* bromeliads, shown here in a restinga sand dune forest in Brazil. The foliage of these bromeliads forms a deep central cavity (tank) that accumulates rainwater and dead organic material, such as leaves from nearby trees. The decomposition of this material sustains a highly productive and diverse food web inside the tank [45].

### 4.15.6 Array sorting

In some cases we need to sort the contents of an array in a specific order, for example sorting the rows of a 2-dimensional array according to the elements in a specific column. There are two main functions used to sort arrays: `sort` and `argsort`. The first function takes as input an array and returns a sorted version of the input array (in increasing order), similarly to the `sorted` function for lists. Consider, for example, the following 1-dimensional array:

```
A = numpy.array([4, 2, 3, 6, 10, 11, 3, 0])
```

The following code computes and prints out a sorted version of `A`:

```
A_sorted = numpy.sort(A)
print(A_sorted)
```

```
[ 0  2  3  3  4  6 10 11]
```

Note that `sort` also generally works similarly for non-numeric arrays, including arrays containing strings (in which case sorting is done alphabetically). For multidimensional arrays the behavior of `sort` is somewhat more complicated, so here we just focus on the more common 1-dimensional case. The function `argsort` performs a slightly different task: Instead of returning a sorted version of the array, it returns the indices of the input array's elements in sorted order. In other words, the returned indices are such that they *would* turn the input array into a sorted array. Let's look at what this means for our example:

```
ordered_indices = numpy.argsort(A)
print(ordered_indices)
```

```
[7 1 2 6 0 3 4 5]
```

Observe that `ordered_indices` does not list the actual elements of `A`, but instead lists integers covering the range 0,..,7. The first integer is the index of the smallest element in `A`, the second integer is the index of the second-smallest element in `A`, and so on. Theoretically, if we were to extract elements from `A` using the indices listed in `ordered_indices` (and in that order), we should get a sorted version of `A`. Indeed:

```
print(A[ordered_indices])
```

```
[ 0  2  3  3  4  6 10 11]
```

Why is this useful? Often our datasets are multidimensional, and perhaps even spread out across multiple arrays. Obtaining the indices corresponding to a re-ordering allows us to apply the re-ordering to multiple arrays, or to multiple slices of an array. For example, consider the following 2-dimensional $4 \times 5$ array:

```
B = numpy.array([[7,  2,   3,  6, 10],\
                 [9,  1,   4,  6, 2],\
                 [3,  7,   10, 1, 1],\
                 [2,  100, 0,  5, 81]])
```

Such an array may, for example, represent 5 morphological traits measured across 4 different species. Suppose that we want to sort the rows of `A` according to the values in the first column (note that this is different from sorting the elements in each column). We can use `argsort` to first determine the appropriate row order based on the elements in the first column:

```
row_order = numpy.argsort(B[:,0])
print(row_order)
```

```
[3 2 0 1]
```

and then apply that order to the rows of `B`:

```
B_sorted = B[row_order,:]
print(B_sorted)
```

```
[[  2 100   0   5  81]
 [  3   7  10   1   1]
 [  7   2   3   6  10]
 [  9   1   4   6   2]]
```

Observe that in each row of `B_sorted` neither the elements nor their order have changed compared to `B`; it is just the order of the rows that has changed, in such a way that the elements in the first column are increasing.

**Reverse sorting:** Unfortunately neither `numpy.sort` nor `numpy.argsort` provide an easy way to directly sort in decreasing order. This is, however, not a big issue, because we can easily reverse the order of any array using a special indexing specification. For example, to obtain a reversed copy of `A`, we can write:

```
A_reversed = A[::-1]
print(A_reversed)
```

```
[ 0  3 11 10  6  3  2  4]
```

Hence, to sort `A` in decreasing order we can combine `sort` and `[::-1]`, as follows:

```
A_decreasing = numpy.sort(A)[::-1]
print(A_decreasing)
```

```
[11 10  6  4  3  3  2  0]
```

A similar reasoning can be applied to `argsort`. For example, to sort the rows of `B` according to the elements in the first column (in decreasing order), we can write:

```
row_order = numpy.argsort(B[:,0])[::-1]
B_sorted = B[row_order,:]
print(B_sorted)
```

```
[[  9   1   4   6   2]
 [  7   2   3   6  10]
 [  3   7  10   1   1]
 [  2 100   0   5  81]]
```

As before, the order of elements in each row of `B_sorted` is the same as in `B`, however the order of the rows has changed in such a way that the elements in the first column of `B_sorted` are decreasing.

★ **Exercise 66:** Predict what the following code will print to the screen, without running it:

```
C = numpy.array([[7,  2,   3,  6, 10],\
                 [9,  1,   4,  6, 2],\
                 [3,  7,   10, 1, 1],\
                 [2,  100, 0,  5, 81]])
X = numpy.argsort(C[1,:])[::-1]
C_sorted = C[:,X]
print(C_sorted[-1,0:4])
```

*Hint: You can use pencil and paper to sketch out the outcome of each step.*

★ **Exercise 67:** Consider the following dataset of average adult body masses for multiple primate species, separated by sex, stored in the form of a TSV file (one row per primate species, species names in the first column, male and female body masses in the second and third column, respectively) [34]:

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`primate_body_masses.tsv`".

Load the first column of this file as a 1-dimensional string array, and the other two columns as a 2-dimensional numeric array, as demonstrated in Section 4.15.4. Print out the names of the species with the 10 smallest male/female body mass ratios.

*Hint: You can first compute the male/female body mass ratios, then use* ***`numpy.argsort`*** *to sort the array listing species names according to those ratios, and then print out the first 10 elements in the sorted species names array.*

## 4.16 Basic plotting

Arguably one of the most useful tools in data analysis is visualization (aka. "plotting"). In `python`, the most commonly used package for plotting data is `matplotlib`, especially its module `pyplot`. While `matplotlib` provides many powerful functions for plotting with a lot of control over details, `pyplot` provides a more simplified interface that is sufficient for most practical purposes. In the following, we will learn how to use `pyplot` to perform various common data visualizations. In all examples that follow in this section, we assume that we have loaded the `pyplot` module as follows:

```
from matplotlib import pyplot
```

### 4.16.1 Scatterplots

One of the most common type of data visualization is the scatterplot, i.e., showing pairs of numbers as individual points on a plane; hence, the pairs are interpreted as cartesian coordinates. Note that 3-dimensional scatterplots, i.e., showing coordinate triplets as points in 3D space, are

also possible, but we won't be discussing those here. As an example, consider the following dataset of mammal body masses (gram) and corresponding basal metabolic rates (Watt) across multiple species [19], stored in the following HTSV file (one row per mammal, masses in the first column, metabolic rates in the second column):

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "mass_metabolic_rate_Hatton2019_mammals.htsv".

We begin by loading the data into an array:

```
data = numpy.genfromtxt("data/mass_metabolic_rate_Hatton2019_mammals.htsv",
                        delimiter="\t")[1:,:]
```

Let's print out the first few rows to check that everything worked:

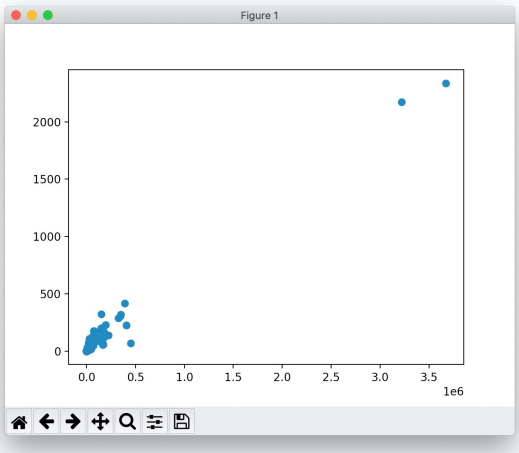
```
with numpy.printoptions(precision=3, suppress=True):
    print(data[0:6,:])
```

```
[[   34.6       0.361]
 [   42.3       0.322]
 [   27.        0.275]
 [38446.1      61.77 ]
 [37900.       50.103]
 [   42.        0.258]]
```

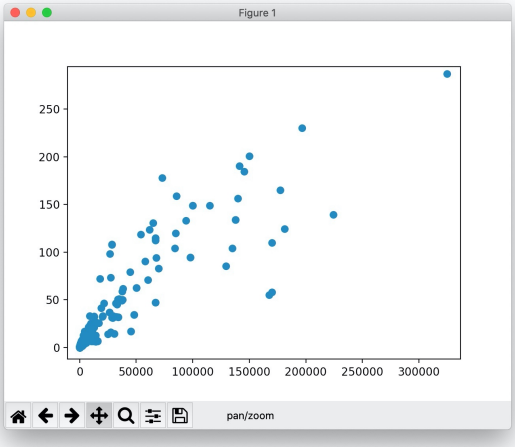
To create a scatterplot showing body masses on the horizontal axis and corresponding metabolic rates on the vertical axis, we can use the `pyplot` function `scatter` with two arguments (the body masses and the metabolic rates):

```
# initialize a new empty figure
pyplot.figure()

# add the scatterpoints to the figure
# by passing a 1-dim array for the x values and a 1-dim array for the y values
pyplot.scatter(data[:,0], data[:,1])

# show the figure to the user
pyplot.show()
```

When executed in `python` interactive mode, the above code will open a new window showing and allowing interaction with the scatterplot, similar to the following:




We can use the buttons at the bottom to move around and zoom into the scatterplot. For example, we can use the magnifying glass symbol to focus on the lower-left *cloud* of points:

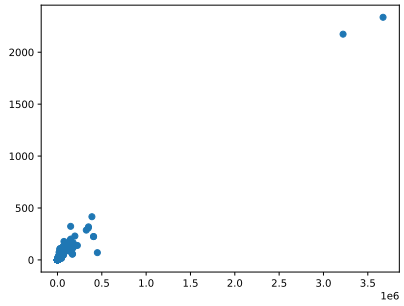



When the above code is executed as part of a script, the execution of the script pauses right after the `pyplot.show()` statement, until we close the newly opened plot window. To save the plot as an image file, we can use the button that looks like a floppy disk.

Instead of opening the plot in a new interactive window, we can also automatically write it into a file at a location of our choosing. For example, the following code generates a scatterplot and saves it as a PDF file:

```
# initialize a new figure
pyplot.figure()

# add the scatterpoints to the figure, as before
pyplot.scatter(x=data[:,0], y=data[:,1])

# save the plot as a PDF file
os.makedirs("py_output", exist_ok=True)
pyplot.savefig("py_output/mammal_mass_vs_metabolism_scatter1.pdf", bbox_inches='tight')
```

The generated PDF will look similar to the following:




Notice that the plot is saved with some default width and height that may not be the most suitable for our purposes; if we wanted a specific width and height, we would need to pass those as an argument to the `figure` function, for example as follows:

```
pyplot.figure(figsize=[4, 3]) # set width to 4 inches and height to 3 inches
```

The above scatterplot is clearly missing something important: Axis labels. To add those, we use the `pyplot` functions `xlabel` and `ylabel`. Similarly, to add a title, we use the `pyplot` function `title`. In addition, because the data in this example span multiple orders of magnitude, making it hard to see any trends, it seems worth plotting both axes on a logarithmic scale. This can be achieved using the `pyplot` functions `xscale` and `yscale`. Last, the scatterpoints in our previous plot are a bit too large, thus causing too much overlap between them; to address this we can use the optional `s` argument, to specify the points' surface area. In fact, we can also make the points slightly transparent so that we can see points hidden behind others using the optional `alpha` argument, which controls opacity. Last, it is generally advised to call `pyplot.close()` when done working with a specific plot, since otherwise the plot is still kept open in `python`'s memory (even if not shown):

```
pyplot.close()
```

Note that the above statement will only close the latest figure. If you forgot to close previous figures, or you just want to be sure that all previous figures are, use:

```
pyplot.close("all")
```

So let's give this another try, incorporating all of the above recommendations.

```
# specify the width & height of the plot, in inches
pyplot.figure(figsize=(3, 3))

# add the scatterpoints to the figure
pyplot.scatter(data[:,0], data[:,1], s=5, alpha=0.5)

# add title & axis labels
pyplot.title("Mammal body mass vs metabolic rate")
pyplot.xlabel("body mass (g)")
pyplot.ylabel("metabolic rate (W)")

# make both axes logarithmic
pyplot.xscale('log')
pyplot.yscale('log')

# ensure that the output directory exists prior to calling savefig
os.makedirs("py_output", exist_ok=True)
```

```
# save the plot as a PDF file
pyplot.savefig("py_output/mammal_mass_vs_metabolism_scatter2.pdf", bbox_inches='tight')

# close everything related to this plot
pyplot.close()
```

The above code will generate the following, much better looking, scatterplot:

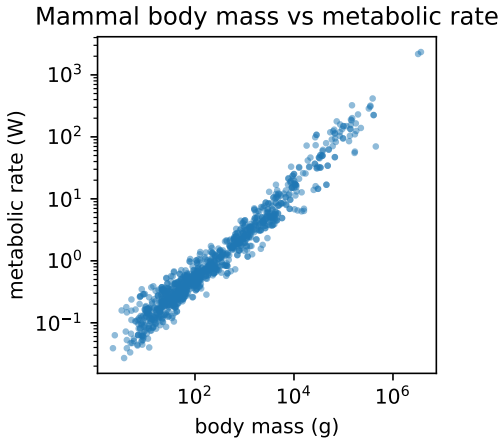



**Figure 4.6:** Mammal body masses and basal metabolic rates, both on a logarithmic axis (one point per mammal).

Voila! We can now see a clear linear trend on a log-log axis, which suggests that mammal metabolic rates and body masses roughly follow a power-law relationship. This apparent relationship is one of the cornerstones of the *metabolic theory of ecology* [46].

We could further customize the above scatterplot, for example by coloring each mammal family with a different color, or varying the point sizes according to some 3rd variable such as body temperature. For a list of additional options see the official documentation of the `scatter` function.

★ **Exercise 68:** Consider the following HTSV table listing pairwise average nucleotide divergences (AND) and gene-content differences (GCD) between multiple *Streptococcus mutans* genomes (one row per genome pair, ANDs in the third column, GCD in the fourth column):

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`Strep_mutans_AND_vs_eggNOG_differences.htsv`".

Load the third and fourth columns of this TSV file into a `numpy` array, and create a scatterplot showing ANDs on the horizontal axis and GCDs on the vertical axis. Use a linear scale for both axes.

**Context:** The AND is a commonly used measure for delineating bacterial species. It is defined as the fraction (in percent) of nucleotide difference between two genomes, across the entirety of homologous regions (i.e., essentially across all shared genes). The GCD between two genomes is computed as $1 - A/B$, where $A$ is the number of shared genes and $B$ the average number of genes in the two genomes. Hence, while the GCD measures how different two genomes are in terms of shared genes, the AND measures how much the sequences of the shared genes differ between the two genomes. As you will see through this exercise, a strong positive correlation exists between the AND and GCD, across *Streptococcus mutans* genomes.

★ **Exercise 69:** Consider the following TSV table, listing body masses (gram) and corresponding growth rates (1/year) for various mammal species (one row per species, masses in the fourth column, growth rates in the fifth column) [19]:

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "mass_vs_growth_rate_Hatton2019_mammals.htsv".

Perform the following tasks:

1. Load the fourth and fifth columns of this TSV table into `python` as a `numpy` array, and create a scatterplot showing body masses on the horizontal axis and growth rates on the vertical axis, both on a logarithmic scale. Make sure to properly label and title your axes.
2. Create a scatterplot showing body masses and productivities (computed as the product of body mass times growth rate).

   *Hint: To compute the productivities, you could extract the two columns from the* ***numpy*** *array as described in Section 4.15.3, and multiply them as described in Section 4.15.5 to create a third 1-dimensional array.*
3. Repeat the previous task, but restricting the range of body masses shown to $10^5$–$10^7$ g and the range of productivities shown to $10^8$–$10^{13}$ g/yr.

   *Hint: Look into the functions* ***pyplot.xlim*** *and* ***pyplot.ylim*** *for restricting axis ranges.*

### 4.16.2 Curve plots

Another common class of visualizations are curve plots, i.e., showing continuous curves on a 2-dimensional plane. In practice, a plotted curve is defined by many individual points (coordinates), which are connected using linear segments, thus generating the impression of a continuous curve. Curve plots are often used in time series analysis, where the value of a continuously changing numerical variable is shown over time. Curve plots are also useful for displaying trajectories in 2-dimensional space, such as the migratory route of a bird. As an example, let's consider an electroencephalogram (EEG) measured at a specific location of a subject's brain over time, following a short pain stimulus as part of a neurological experiment [47]. Suppose that we have stored the EEG in an HTSV file, with each row representing a different time point, the first column containing times (milliseconds) and the second column containing EEG values (mV):

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "EEG_IRF_Kanda1996.tsv".

Let's begin by loading the data into an array:

```
data = numpy.genfromtxt("data/EEG_IRF_Kanda1996.htsv", delimiter="\t")[1:,:]
```

Let's look at the shape of this array and the first few rows, to get a feeling for this dataset:

```
print("Data shape:",data.shape)
```

```
Data shape: (55, 2)
```

```
print(data[0:5,:])
```

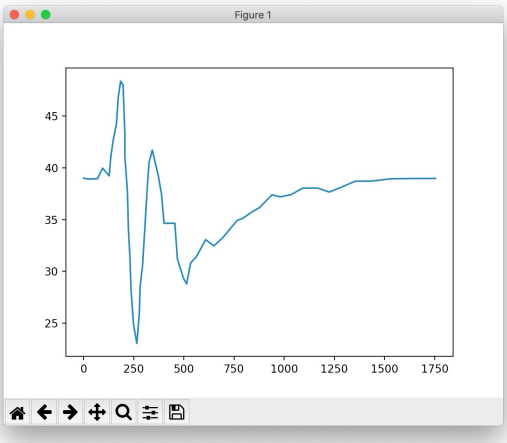
```
[[  0.        38.994354]
 [ 28.689049  38.91    ]
 [ 70.06748   38.9543  ]
```

```
 [ 94.780396   39.9631   ]
 [127.8451     39.23644 ]]
```

To plot the EEG as a curve over time, we can use the `pyplot` function `plot` with two arguments (the times and the corresponding EEG values):

```
# initialize a new empty figure
pyplot.figure()

# add the curve to the figure
# by passing a 1-dim array for the x values and a 1-dim array for the y values
pyplot.plot(data[:,0], data[:,1])

# show the figure to the user
pyplot.show()
```

As with scatterplots, the above code will open a new window showing and allowing interaction with the plot, similar to the following:




As with scatterplots, we can also automatically save this plot as a PDF file, using the `pyplot` function `savefig`:

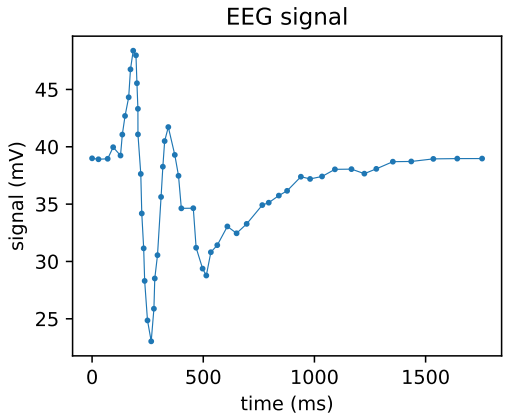
```
pyplot.savefig("EEG_curveplot1.pdf", bbox_inches='tight')
```

As with scatterplots, we can add axis labels and a title using the `pyplot` functions `xlabel`, `ylabel` and `title`. In addition, we can control the curve width using the optional argument `linewidth`, and we can display the original data points with the optional arguments `marker` (which sets the point style) and `markersize` (which sets the point size). So let's give this another try:

```
# specify the width & height of the plot, in inches
pyplot.figure(figsize=(4, 3))

# add the curve to the figure
# while choosing the curve width and also displaying the original data points
pyplot.plot(data[:,0], data[:,1], linewidth=0.6, marker='o', markersize=2)

# add title & axis labels
pyplot.title("EEG signal")
pyplot.xlabel("time (ms)")
pyplot.ylabel("signal (mV)")
```

```
# ensure that the output directory exists prior to calling savefig
os.makedirs("py_output", exist_ok=True)

# save the plot as a PDF file
pyplot.savefig("py_output/EEG_curveplot2.pdf", bbox_inches='tight')

# close everything related to this plot
pyplot.close()
```

The above code will generate the following improved plot:




For many additional options, such as setting the curve and point colors, see the official documentation.

**Plotting multiple curves:** It is possible to show multiple curves on top of each other (i.e., within the same figure), by calling `pyplot.plot` multiple times prior to closing the figure or saving it to a file. For example, consider the following dataset of snowshoe hare and Canadian lynx population sizes, measured at yearly intervals in Kluane National Park [36] and stored in HTSV file format (one row per year, years in the first column, densities in the second column):

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → files "`Krebs2011_hare.htsv`"
> and "`Krebs2011_lynx.htsv`".

Let's load both time series into arrays:

```
hare = numpy.genfromtxt("data/Krebs2011_hare.htsv", delimiter="\t")[1:,:]
lynx = numpy.genfromtxt("data/Krebs2011_lynx.htsv", delimiter="\t")[1:,:]
```

Let's normalize each curve so that they are measured relative to their maximum value, thus facilitating comparison between them:

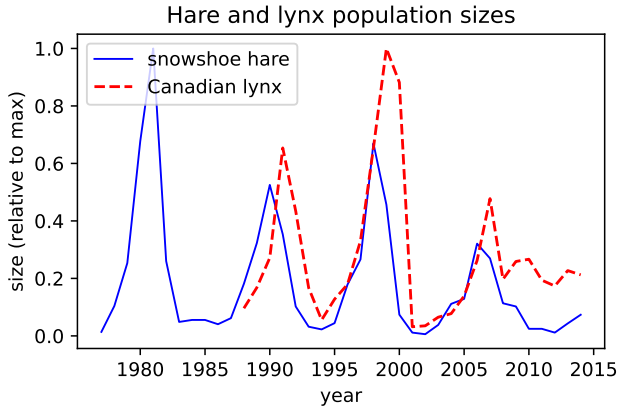
```
hare[:,1] /= numpy.max(hare[:,1])
lynx[:,1] /= numpy.max(lynx[:,1])
```

Let's plot both time series in the same figure, using a different color and line style for each. To help distinguish the two curves, we also assign to each of them a unique label (a brief description) using the argument `label`; the assigned labels are then used to identify the curves in a legend, which we add using the `pyplot` function `legend`.

```
# instantiate a new figure with specific width & height
```

```
pyplot.figure(figsize=(5, 3))

# plot the two curves in the same figure
pyplot.plot(hare[:,0], hare[:,1],
            linewidth=1, color="blue", linestyle="-",
            label="snowshoe hare")
pyplot.plot(lynx[:,0], lynx[:,1],
            linewidth=1.5, color="red", linestyle="--",
            label="Canadian lynx")

# add title & axis labels
pyplot.title("Hare and lynx population sizes")
pyplot.xlabel("year")
pyplot.ylabel("size (relative to max)")

# add the legend
pyplot.legend()

# save & close the figure
pyplot.savefig("py_output/hare_lynx_abundance_curves.pdf", bbox_inches='tight')
pyplot.close()
```

We thus obtain the following figure, showing both population sizes over time:




Observe that both the hare and lynx populations fluctuate at roughly 10-year periods, a phenomenon that has been heavily studied in the past [36, 48–50].

### 4.16.3 Plotting mathematical functions

Curve plots are not just useful for visualizing data, but can also be used to visualize mathematical functions of biological significance, or regression curves that we want to compare to data. As an example, consider the Monod equation (aka. "Michaelis-Menten" equation), which is often used to describe nutrient uptake rates in bacteria and phytoplankton [51–54]:

$$R = \frac{V \cdot C}{K + C}. \tag{4.12}$$

Here, $C$ is the concentration of the limiting nutrient (e.g., the carbon source), $R$ is the cell-specific nutrient uptake rate (amount absorbed per time per cell), and $V, K$ are fixed parameters. Note that in some contexts $R$ represents the growth rate rather than the nutrient uptake rate, although these two rates tend to be proportional to each other. A common way to examine the Monod equation and its implications for cell growth is to plot the rate $R$ as a function of the concentration $C$. Consider, for example, the case of an acetate-utilizing methanogen, with

parameters $K = 0.442$ mM and $V = 5.5 \times 10^{-13}$ mol $\cdot$ cell$^{-1}$ $\cdot$ day$^{-1}$ [55, 56]. Let's investigate how the rate of acetate uptake ($R$) changes as the concentration of acetate ($C$) varies from 0 up to 4 mM. We start by creating a 1-dimensional `numpy` array that consists of 100 evenly spaced $C$-values between 0 and 4:

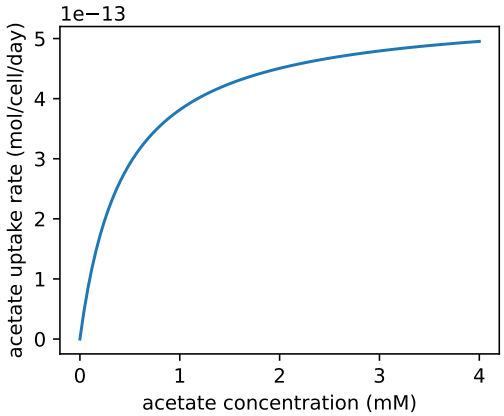
```
import numpy
C = numpy.linspace(0,4,100)
```

Next, we compute the corresponding $R$-values based on Eq. (4.12), as a 1-dimensional array:

```
V = 5.5e-13
K = 0.442
R = V*C/(K+C)
```

We can now plot the paired $C$-$R$ values as a continuous curve, with $C$ values shown on the horizontal axis and $R$ values on the vertical axis:

```
pyplot.figure(figsize=(4, 3))
pyplot.plot(C, R)
pyplot.xlabel("acetate concentration (mM)")
pyplot.ylabel("acetate uptake rate (mol/cell/day)")
pyplot.savefig("py_output/Monod_kinetics_curve1.pdf", bbox_inches='tight')
pyplot.close()
```

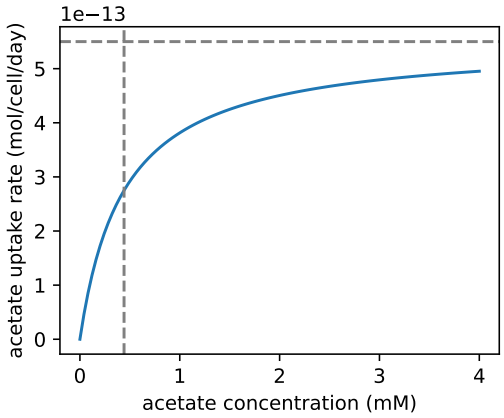



Strictly speaking we have only calculated $R$ for a discrete set of $C$ values, and hence the plotted curve is essentially a series of connected linear line segments; nevertheless, because the length of each line segment is so small (due to the small intervals between successive $C$ values), the curve looks smooth to the naked eye. Visualizing the Monod equation as above facilitates our understanding of its biological implications. For example, we see that for small substrate concentrations the rate $R$ increases rapidly as $C$ increases, however towards higher concentrations ($>$3 mM) the rate eventually saturates at some maximum value. Increasing $C$ much further only has a marginal effect on the rate $R$. It can easily be shown mathematically that the maximum rate possible (i.e., as $C$ tends to infinity) is $V$. Similarly, it can be shown that the concentration at which $R$ attains half of its maximum value (i.e., where $R = V/2$) is given by the parameter $K$ (which is why $K$ is often called "half-saturation constant"). We can easily visualize these two facts, by adding a horizontal line at $R = V$ (using the `pyplot` function `axhline`) and a vertical line at $C = K$ (using the `pyplot` function `axvline`):

```
pyplot.figure(figsize=(4, 3))
pyplot.plot(C, R)
```

```
pyplot.axvline(K, linestyle="--", color="gray")
pyplot.axhline(V, linestyle="--", color="gray")
pyplot.xlabel("acetate concentration (mM)")
pyplot.ylabel("acetate uptake rate (mol/cell/day)")
pyplot.savefig("py_output/Monod_kinetics_curve2.pdf", bbox_inches='tight')
pyplot.close()
```

Note that we used the optional arguments `linestyle` and `color` to make the two added lines dashed and gray-colored. We thus obtain the following figure:




★ **Exercise 70:** Consider the following time series dataset listing the normalized abundances of three bacterial populations in a wastewater treatment plant, measured at weekly intervals and stored in an HTSV file (one row per week, one column per species) [43]:

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`WWTP_AOB_abundances_Ofiteru2010.htsv`".

Perform the following tasks:

1. Load the above TSV file into `python` as a `numpy` array, and plot the abundances of the three bacterial species as curves over time. Show all three curves in the same plot using different colors and different line styles to facilitate their comparison, and include a legend specifying which curve represents which species. The horizontal axis should show time measured in weeks. *Hint: Use the* `color` *and* `linestyle` *arguments of the* `plot` *function to set the color and line style of each curve. For a list of available color names see here. Some available* `linestyle` *options are* `"-"`, `"--"`, `"-."` *and* `":"`. *To create an array listing the week numbers you can use the* `numpy` *function* `arange`.

2. Compute the weekly abundance increments for each species, i.e., the amount by which each species' abundance increased or decreased compared to the previous week, and create a scatterplot comparing the weekly abundance increments of the first two species. Hence, each point in the scatterplot will correspond to a different week, with the horizontal axis representing the increments of the first species and the vertical axis representing the increments of the second species.

   *Hint: You can use the* `numpy` *function* `diff` *to efficiently compute the weekly increments, or you can use a loop that iterates over all weeks but the first one.*

★ **Exercise 71:** The following equation describes the predicted population size of a single bacterial batch culture, growing on a limited carbon source, as a function of time [57]:

$$N(t) = \frac{N_o \cdot \left(1 + \frac{\gamma R_o}{N_o}\right)}{1 + \frac{\gamma R_o}{N_o} \cdot e^{-(N_o + \gamma R_o) \cdot \alpha t}}. \tag{4.13}$$

Here, $N(t)$ is the population size at time $t$, $N_o$ is the initial population size (i.e., at time $t = 0$), $R_o$ is the initial quantity of the carbon source, $\alpha$ is a fixed parameter that measures how fast a single cell can absorb the carbon source, and $\gamma$ is a fixed parameter that specifies how many cells are produced per mol carbon consumed. Suppose that $N_o = 10^4$ cells, $R_o = 0.1$ mol, $\alpha = 5 \times 10^{-7}$ cell$^{-1} \cdot$ hr$^{-1}$ and $\gamma = 10^8$ cells $\cdot$ mol$^{-1}$. Plot the predicted population size as a continuous curve over time for the first 4 hours. Hence, your horizontal axis should represent time (in hours) and your vertical axis should represent numbers of cells. Also show in the same plot a horizontal dashed line at the value $N_o + \gamma R_o$. Based on the plot, what is the relationship between the predicted population size and the horizontal line?

★ **Exercise 72:** Consider the following time series data of annual population size estimates in the North American tundra [35], stored as a TSV file (one row per year):

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`Hudson_Bay_fur_bearing_animals.tsv`".

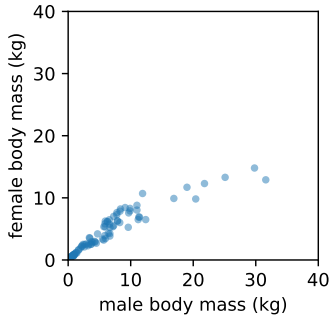
Note that the first column lists years, while the remaining columns list population size estimates for various animal species, including North American beavers (second column), American martens (third column) and so on (in arbitrary units). Load the first two columns of this file as a `numpy` array, and plot the beaver population sizes as a curve over time. Then, find all peaks in the beaver time series, and plot these as small red circles on top of the first plot at their corresponding time and value. In other words, the circles should highlight the peaks in the time series plot. For the purpose of this exercise, we consider a point in the time series to be a peak if it is strictly higher than the five previous and the five subsequent points.

*Hint: Use a* `for` *loop to find all peaks and store their times and values in temporary lists. Show the peaks in the same figure as the beaver time series using the* `pyplot.scatter` *function.*

### 4.16.4 Reference lines and linear regression

One special type of a curve plot is a single straight line. Straight lines are commonly used in scientific plots to indicate a reference value, to display the predictions of a linear regression model, or to facilitate the comparison of two variables across samples. We can easily plot a straight line segment using the familiar `pyplot.plot` function, by providing the coordinates (X

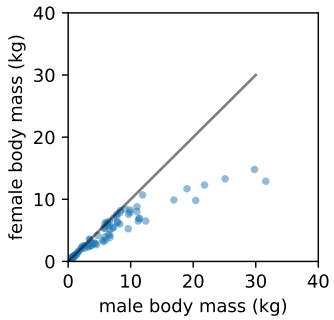
and Y values) of its two endpoints. As an example, consider the adult body masses of female and male primates [34], shown as a scatterplot below (one point per primate species):




The following code would add a semi-transparent diagonal line segment to the previous scatterplot, passing through the points $(x = 0, y = 0)$ and $(x = 30, y = 30)$:

```
pyplot.plot([0,30],[0,30], color="black", alpha=0.5)
```

The diagonal line segment serves as a useful reference that allows us to more easily see the sexual body mass dimorphism in primates, particularly for larger species. If there was no dimorphism all points would be lying on the diagonal, which however is not the case.




One weakness of using `pyplot.plot` to display lines is that the generated line segment does not automatically extend beyond the two specified endpoints; hence, if we want the line to span the entire figure dimensions, we would need to know the axis limits beforehand so as to choose appropriate endpoints. An alternative approach is to plot an "infinitely long" straight line, i.e., one that automatically extends to the full width and height of the figure, using `pyplot.axline`. This function allows specifying a straight line either in terms of two points, or in terms of a single point and a slope, or in terms of its intercept (the position at which it crosses the Y-axis) and a slope, all of which can be useful in some situations (Fig. 4.7).

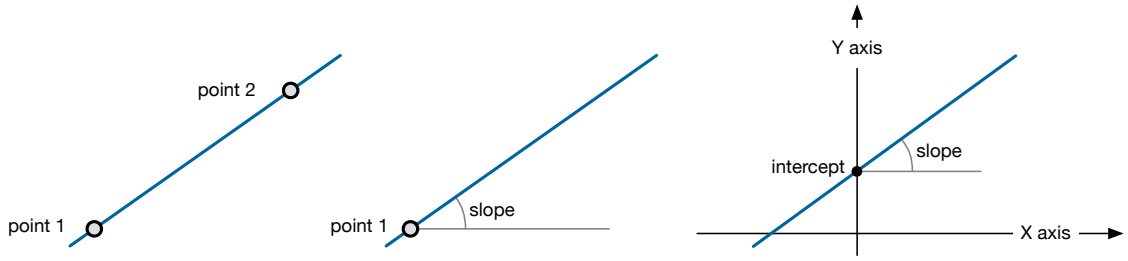
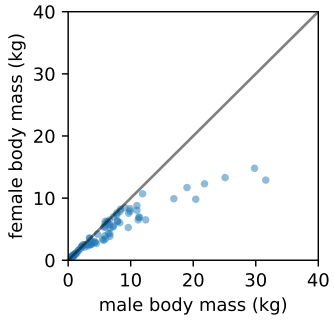



**Figure 4.7:** Mathematically, we can define a straight line in a variety of alternative ways, for example through the coordinates of two points, or through the coordinates of one point and a slope, or through an intercept and a slope. The function `axline` in the module `pyplot` can accommodate all of these three approaches.

For example, to add an infinite diagonal line to the previous scatterplot we could specify one point at the axis origin and a slope of 1:

```
pyplot.axline(xy1=[0,0], slope=1, color="black", alpha=0.5)
```



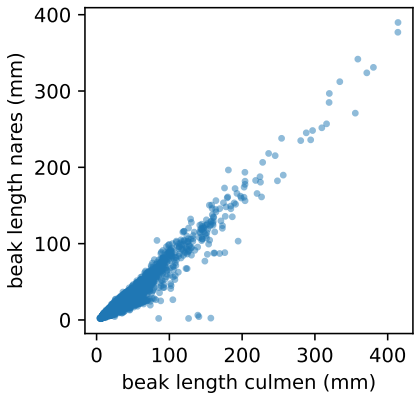

**Linear regression:** In many cases showing a diagonal reference line is not meaningful, particularly when the two axes represent very different variables (e.g., body mass versus metabolic rate). Nevertheless, when the two variables exhibit a positive or negative relationship, it may still be useful to display a straight line "through the data cloud" that summarizes this relationship. In general, such a line does not need to have an intercept of 0 and/or a slope of 1, and instead must be chosen such that it reasonably represents the data. Choosing an appropriate intercept and slope that represents a particular scatter dataset is known as linear regression [58], or linear curve fitting. Linear regression is one of the most common statistical/visual tools in biology [59]. Briefly, in linear regression the intercept and slope are chosen ("fitted") in such a manner that the sum of squared vertical distances between the data and the line (aka. "residuals") is minimized. Here, we shall not discuss the underlying statistical theory, and instead only briefly illustrate its use in data visualization. As an example, consider the following dataset of avian traits [60]:

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`AVONET_bird_traits.htsv`".

This HTSV file lists a variety of morphological traits for thousands of bird species (one row per species), including beak lengths measured using two different conventions: from the tip to the base of the skull along the culmen (column 9), and from the tip to the anterior edge of the nares (column 10). Let's plot these two traits across all species as a scatterplot (one point per species):

```
from matplotlib import pyplot
import numpy
traits = numpy.genfromtxt("data/AVONET_bird_traits.htsv",
```

```
                          delimiter="\t", usecols=[9,10])[1:,:]
pyplot.figure(figsize=(3, 3))
pyplot.scatter(traits[:,0], traits[:,1], s=6, alpha=0.5)
pyplot.xlabel("beak length culmen (mm)")
pyplot.ylabel("beak length nares (mm)")
pyplot.savefig("py_output/bird_beak_lengths.pdf", bbox_inches='tight')
pyplot.close()
```




We see that there is a strong and roughly linear relationship between the two beak lengths. To run a linear regression on these data we can use the function `linregress` in the module `stats` of the package `scipy`, whose two main arguments are the X values (beak length along culmen) and corresponding Y values (beak length to nares):

```
import scipy.stats
LR = scipy.stats.linregress(x=traits[:,0], y=traits[:,1])
```

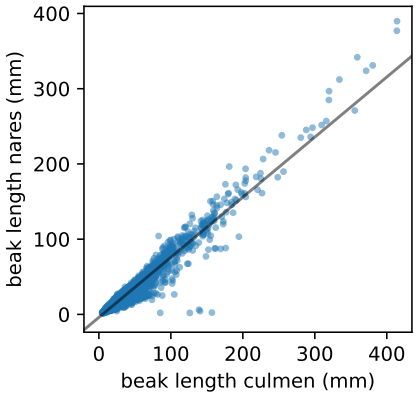
Once the above code is executed, `LR` will be an object containing various information about the linear regression just performed. We can extract the fitted intercept and slope from the `LR` object as `LR.intercept` and `LR.slope`, respectively, and print those to the screen:

```
print("Linear regression intercept = %.3g, slope=%.3g"%(LR.intercept, LR.slope))
```

```
Linear regression intercept = -3.99, slope=0.799
```

We can also use `pyplot.axline` to plot the line corresponding to the fitted intercept and slope on top of the data:

```
pyplot.figure(figsize=(3, 3))
pyplot.scatter(traits[:,0], traits[:,1], s=6, alpha=0.5)
pyplot.axline(xy1=[0,LR.intercept], slope=LR.slope, color="black", alpha=0.5)
pyplot.xlabel("beak length culmen (mm)")
pyplot.ylabel("beak length nares (mm)")
pyplot.savefig("py_output/bird_beak_lengths_lin_reg.pdf", bbox_inches='tight')
pyplot.close()
```




The close alignment of the data to the fitted line confirms that the two beak lengths following an approximately linear relationship, although some outliers exist. In practice, such visual inspections are generally accompanied by a quantitative measure of how closely the fitted line reflects the data ("goodness of fit") as well as confidence intervals on any fitted parameters of interest (e.g., the slope).

★ **Exercise 73:** Consider the following dataset of prokaryotic genome properties obtained from the NCBI RefSeq database [31] (one row per genome):

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`RefSeq_prokaryotic_genome_properties.htsv`".

This HTSV file lists genome sizes (in basepairs) in the second column and numbers of detected protein-coding genes in the third column. Create a scatterplot showing genome sizes (in kbp, i.e., thousands of basepairs) on the horizontal axis and protein counts on the vertical axis (one point per genome). Also include in the plot a linear regression line for comparison. What is the fitted slope (proteins per kbp)?

★ **Exercise 74:** A "power law" is a hypothesized or observed functional relationship between two variables $X$ and $Y$ of the form $Y = a \cdot X^b$, where $a$ and $b$ are constant parameters. Power laws have a long tradition in biology, where they are used to describe scaling relationships between various traits across species [19, 46, 61, 62]. In the following, we consider the following dataset of mammal body masses (gram) and corresponding basal metabolic rates (Watt) across multiple species [19], stored as an HTSV file (one row per mammal, masses in the first column, metabolic rates in the second column):

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`mass_metabolic_rate_Hatton2019_mammals.htsv`".

Plot the body masses ($X$-axis) and metabolic rates ($Y$-axis) as a scatterplot on a log-log axis, similarly to Figure 4.6. Include in the plot a straight reference line obtained via linear regression of the log-transformed data. Based on the fitted intercept and slope, what is the approximate power law describing the relationship between body mass and metabolic rate, i.e., what are the parameters $a$ and $b$?

*Hint: Performing linear regression on log-transformed data (i.e., on* $\ln(X)$ *and* $\ln(Y)$*) is equivalent to fitting a power law to the original data, with the fitted intercept and slope corresponding to* $\ln(a)$ *and* $b$*, respectively. You will need to show the curve* $Y = a \cdot X^b$ *as a reference, which will look like a straight line on a log-log axis. You cannot use* `axline` *for that (but feel free to try it for yourself); instead, you can use the approach described in Section 4.16.3.*

### 4.16.5 Bar plots

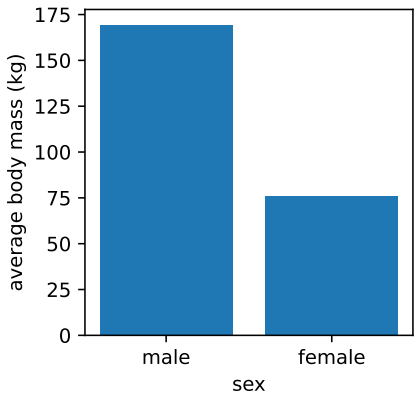
Another common type of visualization is the bar plot (aka. bar chart), which shows quantitative values associated with discrete categories as parallel bars. For example, bar plots may be used to compare the average body mass of different mammal species, or to compare a quantitative outcome between different experimental treatments. As an example, consider the average adult body mass (kg) of male and female gorillas [34], stored in the following list:

```
body_masses = [169.3, 75.7]
```

To generate a simple bar plot with these data, we can use the `pyplot` function `bar`. The two main pieces of information needed for a simple bar plot are a list of category names/labels (e.g., "male" and "female"), and a corresponding list of numerical values (`body_masses`), which are provided as arguments to `pyplot.bar`. Most other steps, such as initializing a figure, adding axis labels, saving the figure as a PDF and closing it afterwards, are the same as for scatterplots and curve plots.

```
# specify the width & height of the plot, in inches
pyplot.figure(figsize=(3, 3))

# add the bars to the figure
pyplot.bar(["male", "female"], body_masses)

# add labels
pyplot.xlabel("sex")
pyplot.ylabel("average body mass (kg)")

# save the plot as a PDF file
pyplot.savefig("py_output/barplot_gorilla_body_masses1.pdf", bbox_inches='tight')

# close everything related to this plot
pyplot.close()
```

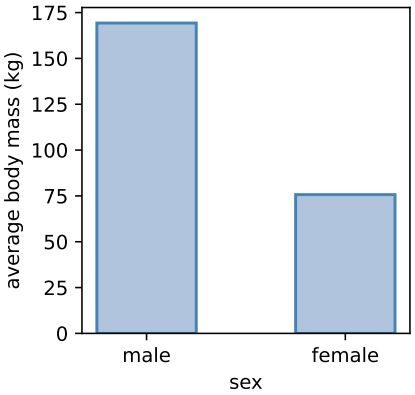
The above code will generate the following bar plot:




Let's try to improve this figure. We can control the fill color for each bar with the `color` argument (for a list of options see here), specify a darker color for the edges of each bar using the `edgecolor` argument, set the bar widths using the `width` argument, and control the edge widths using the `linewidth` argument:

```
pyplot.figure(figsize=(3, 3))
pyplot.bar(["male", "female"],
           height    = body_masses,
```

```
            color     = "lightsteelblue",
            edgecolor = "steelblue",
            width     = 0.5,
            linewidth = 1.5)
pyplot.xlabel("sex")
pyplot.ylabel("average body mass (kg)")
pyplot.savefig("py_output/barplot_gorilla_body_masses2.pdf", bbox_inches='tight')
pyplot.close()
```




Let's consider a slightly more complicated example. Suppose that we want to compare the body masses of males and females in three different primates [34]: gorillas, chimpanzees (*Pan troglodytes*) and Bornean orangutans (*Pongo pygmaeus*). Let's define their body masses in this order, but separated by sex into two lists:

```
male_masses   = [169.3, 42.7, 78.3]
female_masses = [75.7, 33.7, 35.8]
```

A simple bar plot showing these 6 values can be created similarly to the previous example, although we do need to rotate the bar labels to make them fit:

```
pyplot.figure(figsize=(5, 3))

pyplot.bar(["male gorilla", "male chimpanzee", "male orangutan",
            "female gorilla", "female chimpanzee", "female orangutan"],
            height    = male_masses + female_masses,
            color     = "lightsteelblue", edgecolor="steelblue",
            width     = 0.75,
            linewidth = 1.5)

pyplot.xlabel("sex & species")
pyplot.ylabel("average body mass (kg)")
pyplot.xticks(rotation=45, ha="right") # adjust orientation and alignment of bar labels

pyplot.savefig("py_output/barplot_primate_body_masses1.pdf", bbox_inches='tight')
pyplot.close()
```

Note that we used the plus (+) sign to concatenate the two mass lists, `male_masses + female_masses`, into a single list of bar heights. The above code produces the following figure:

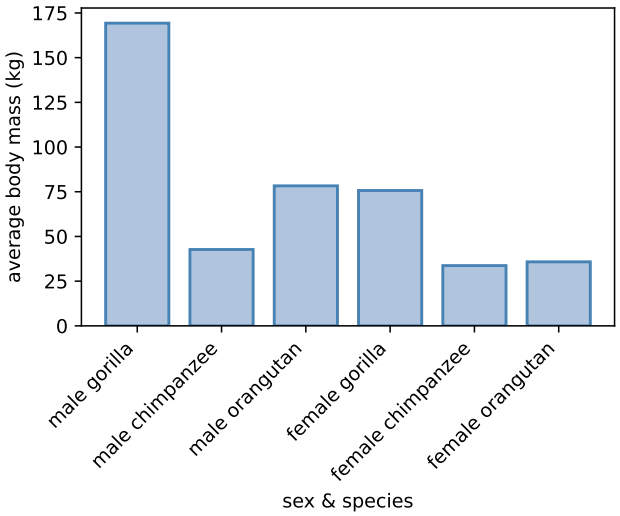



**Figure 4.8:** Bar plot of average adult body masses of various primate species.

However, this figure is rather unstructured and hard to read, so let's make some improvements. First, it would be better if bars were clustered together by sex. To place bars at non-evenly spaced positions, we need to explicitly provide their x-coordinates as a list (one x-coordinate per bar). By default, their x-coordinates are 0, 1, 2, .., 5, so to cluster bars by sex we can simply double the space between the 3rd and 4th bar, i.e., set the bar coordinates to 0, 1, 2, 4, 5, 6. To specify these coordinates, we need to provide them as the first argument instead of the labels, and then provide the labels as another argument called `tick_label`. Second, it would be nice if bars were colored by species. To achieve this, we need to explicitly specify the fill color of each bar, rather than specifying a single fill color for all bars. This can be achieved by passing a list of colors to `color` (one color per bar). So let's give this another try:

```
pyplot.figure(figsize=(5, 3))

# specify the fill color for each species
species_colors = ["lightsteelblue", "deepskyblue", "paleturquoise"]

pyplot.bar( [0, 1, 2, 4, 5, 6],
            height     = male_masses + female_masses,
            tick_label = ["male gorilla", "male chimpanzee", "male orangutan",
                          "female gorilla", "female chimpanzee", "female orangutan"],
            color      = species_colors + species_colors,
            edgecolor  = "dimgray",
            width      = 0.75,
            linewidth  = 1.5)

pyplot.xlabel("sex & species")
pyplot.ylabel("average body mass (kg)")
pyplot.xticks(rotation=45, ha="right") # adjust orientation and alignment of bar labels

pyplot.savefig("py_output/barplot_primate_body_masses2.pdf", bbox_inches='tight')
pyplot.close()
```

The above code yields the improved bar plot shown in Fig. 4.9 below.

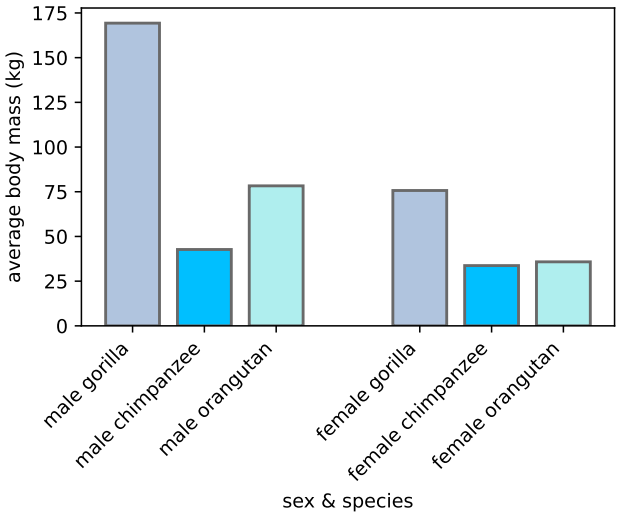



**Figure 4.9:** Improved bar plot of average adult body masses of various primate species, clustered by sex and colored by species.

We can now more easily see that while female chimpanzees and orangutans have very similar weights, male orangutans tend to be much larger than male chimpanzees. For additional customization options, such as adding errorbars, see this official documentation.

★ **Exercise 75:** Consider the following dataset of average adult body masses for multiple macaque species, separated by sex [34]:

```
species = ["M. mulatta", "M. fuscata", "M. radiata", "M. assamensis", "M. sinica"]
male_masses   = [11, 11, 6.67, 11.3, 5.68]
female_masses = [8.8, 8.03, 3.85, 6.9, 3.2]
```

Note that the order of elements in the two lists `male_masses` and `female_masses` matches the order of species names in `species`. Perform the following tasks:

1. Create a bar plot showing the above data, with bars colored by species and clustered by sex, similar to the one in Fig. 4.9, Section 4.16.5.
2. Create another bar plot with the same data, with bars colored by sex and clustered by species.

   *Hint: You will need to show bars in pairs (male & female of the same species), with extra distance between each species. The order in which you specify the bar heights via the argument* `height` *and bar labels via the argument* `tick_label` *will also change.*

★ **Exercise 76:** Consider the following dataset of average adult body masses for multiple primate species, separated by sex [34], stored in the form of a TSV file (one row per primate species, species names in the first column, male and female body masses in the second and third column, respectively):

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`primate_body_masses.tsv`".

Load the first column of this file as a 1-dimensional string array, and the other two columns as a 2-dimensional numeric array, as demonstrated in Section 4.15.4. Create a bar plot showing female body masses for the species with the 30 largest females, and sorted in order of decreasing size. Ensure that the species names are clearly legible underneath each bar (i.e., are not overlapping).

### 4.16.6 Histograms

In the following we will learn how to perform another important type of data visualization: Histograms. Histograms provide a simple and intuitive means to visualize the distribution of a repeatedly measured numeric variable, such as the body masses of individuals in a population, the number of bees found in each of several hives, or the expression levels of a specific gene measured at multiple time points. Histograms not only yield information about the "typical" values of a variable (e.g., its mean), but also its spread around the mean and biases towards larger or smaller values. Histograms are also useful for comparing the distribution of a variable between different experimental treatments or cohorts.

As a simple example, consider the following univariate dataset of hundreds of egg lengths (in mm) of the great black-headed gull (*Larus ichthyaetus*), measured across nests in Lake Chany, Russia [63]:

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`gull_egg_lengths_Yurlov2022.txt`".

Let's load this dataset as a 1-dimensional numeric array:

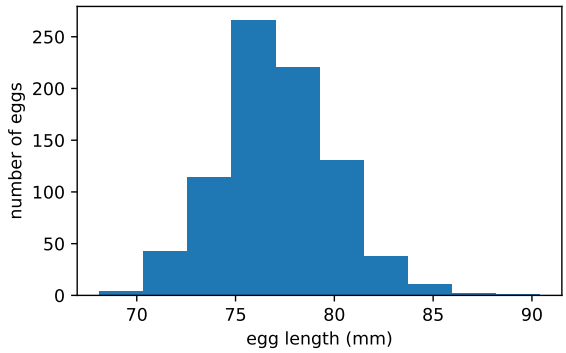
```
import numpy
egg_lengths = numpy.genfromtxt("data/gull_egg_lengths_Yurlov2022.txt")
print("Loaded %d egg lengths.\nMean length = %.2f mm"\
      %(len(egg_lengths),numpy.mean(egg_lengths)))
```

```
Loaded 831 egg lengths.
Mean length = 77.08 mm
```

To generate a simple histogram of these egg lengths, we can use the `pyplot` function `hist`. The only mandatory argument to this function is the list of values whose distribution we wish to visualize (`egg_lengths`). Most other steps, such as initializing a figure, adding axis labels, saving the figure as a PDF and closing it afterwards, are the same as for scatterplots, curveplots and barplots.

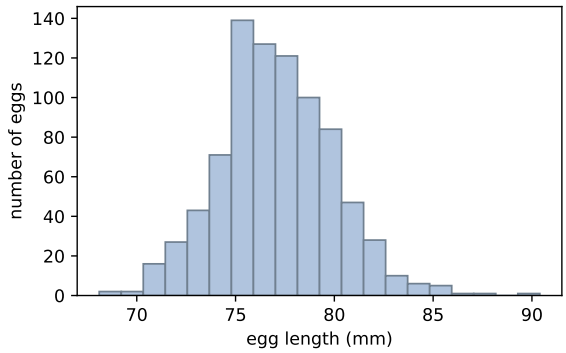
```
# specify the width & height of the plot, in inches
pyplot.figure(figsize=(5, 3))

# add the histogram to the figure
pyplot.hist(egg_lengths)

# add axis labels
pyplot.xlabel("egg length (mm)")
pyplot.ylabel("number of eggs")

# save the plot as a PDF file
pyplot.savefig("py_output/histogram_egg_lengths1.pdf", bbox_inches='tight')

# close everything related to this plot
pyplot.close()
```

The above code will generate the following histogram:




Each vertical bar represents a separate interval ("bin") of egg lengths, and the bar's height corresponds to the number of eggs found in that length interval. Since the bins have equal width, the overall shape of the histogram approximately reflects the probability distribution of egg lengths. For example, we can see that the mode (peak) of this probability distribution is around 75–77 mm. The number of bins used for a histogram is one of the most common knobs to tweak; more bins provide a higher resolution of the distribution, but at the cost of less accurate (more noisy) bin heights. The number of bins can be specified using the argument `bins`. Further, as with bar plots, we can set the bar fill color using the argument `color` and the bar edge color using `edgecolor`. So let's give it another try:

```
pyplot.figure(figsize=(5, 3))
pyplot.hist(egg_lengths,
            bins      = 20,
            color     = "lightsteelblue",
            edgecolor = "slategray")
pyplot.xlabel("egg length (mm)")
pyplot.ylabel("number of eggs")
pyplot.savefig("py_output/histogram_egg_lengths2.pdf", bbox_inches='tight')
pyplot.close()
```




The greater resolution achieved thanks to the larger number of bins now allows us to see more clearly that the distribution of egg lengths is asymmetric. More advanced histograms, for example comparing the distributions of multiple sample sets, will be discussed later in Section 5.7.1.

★ **Exercise 77:** Consider the following dataset of upper temperature limits (UTLs, in °C) determined for various amphibians [64], stored as an HTSV file (one row per individual, UTLs in the fifth column):

http://www.loucalab.com/archive/BioDataAnalysisPython
→ file "`amphibian_upper_temperature_limits_Pottier2022.htsv`".

Load the UTLs in the above file into a 1-dimensional `numpy` array, and display the distribution of UTLs as a histogram with 20 bins.

★ **Exercise 78:** Consider the following dataset of mammal body masses (gram) and basal metabolic rates (Watt) across multiple species [19], stored as an HTSV file (one row per mammal, masses in the first column, metabolic rates in the second column):

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`mass_metabolic_rate_Hatton2019_mammals.htsv`".

Load the above file into a `numpy` array, compute the mass-specific metabolic rates (basal metabolic rate divided by body mass), and show their distribution as a histogram. Try to select a reasonable number of bins that strikes a good balance between resolution and accuracy. Also show in the same figure the average mass-specific metabolic rate as a vertical line, for comparison.

*Hint: You can draw a vertical line using the `pyplot` function `axvline`.*

★★ **Exercise 79:** Thanks to modern genomic sequencing, we now know that even closely related bacterial strains within the same species can exhibit substantial differences in their gene content, that is, in the sets of genes present [65–67]. Consider the following gzipped HTSV file, listing information on the presence/absence of various genes in the genomes of multiple *Escherichia coli* strains:

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`Escherichia_coli_eggNOG_table_Louca2022.htsv.gz`".

Each row in this file corresponds to a different genome, the first column lists genome names, each of the remaining columns corresponds to a different gene, and each entry in those columns indicates whether a given gene was found in a given genome (1:present, 0:absent). Perform the following tasks:

1. Load the above HTSV file as a `pandas` dataframe. Compute for each genome the total number of genes found, and create a histogram showing the distribution of these numbers. *Hint: Since the first column and the first row contain genome and gene names, respectively, you will need to omit these after loading the file as a `numpy` array.*
2. For each unique pair of genomes, compute the fraction of shared genes (FSG) and create a histogram showing the distribution of all computed FSGs. Don't include self-pairs, and make sure to count each pair only once (i.e., for every two genomes A & B, only include one out of the two comparisons A-vs-B and B-vs-A). For any two genomes, the FSG is defined as the number of genes found in both genomes, divided by the average number of genes per genome. In other words, if genome A has $n$ genes, genome B has $m$ genes, and there are $k$ genes present in both genomes, then the FSG is defined as $\frac{k}{(n+m)/2}$.

   What are the typical FSGs between *E. coli* strains, based on your findings?

   *Hint: You could use two nested loops to iterate over all possible genome pairs, each time computing their FSG and storing the result in a numeric list.*

# 5 Somewhat more advanced python

## 5.1 Custom functions

So far we have encountered numerous functions, either provided by core `python` or by packages, which allowed us to perform a variety of operations without worrying about the underpinnings

of each function (i.e., without knowing how they actually work). In many situations, no function may be available yet that does precisely what we would like it to, in which case we may consider creating our own new function for the job. It is easy, and quite common, for a scientist to write their own custom functions, which can then be used throughout their analysis. For example, we may need to analyze a large number of time series in a specific way at various locations of our code; we could thus write a custom function specifically for this analysis, and then *call* the function whenever the analysis is to performed on a given time series. Using functions to represent complex computations allows us to write more organized, concise and readable code, fix mistakes in our code if there are any, and to abstracize analyses. Wrapping a chunk of code into a function also allows us to solve a problem (i.e., code up a computational task) once and for good and in a generic way, and not worry about the details of the solution ever again.

**Introductory example:** To define a new function we use the `def` keyword, followed by the name of our function, an optional comma-separated list of arguments inside parentheses, a colon, and the body of the function, which specifies the computations to be performed whenever the function is called. A function's arguments are like mechanical pieces in a car assembly line, and a function's body specifies how these pieces should be put together to return a car whenever the function is called. For example, the following code defines a function called `square`, which takes a single numeric argument ($x$) and returns its square ($x^2$):

```
def square(x):
    y = x**2
    return y
```

The line `def square(x):` is known as the header of the function, and it specifies the function's name (`square`) and the function's sole argument (`x`). The two subsequent lines make up the body of the function, and the `return` keyword tells `python` that the value of the variable `y` should be returned as a result. Observe that, similar to `if` statements and `for` loops, the body of a function must be indented relative to its header. Defining a function as above does not actually execute it; it just tells `python` what to do *when* the function is called. We can call our function `square` elsewhere in our code to compute the square of any numeric variable using the function's name, for example:

```
radius = 10
area = 3.14159 * square(radius)
print(area)
```

```
314.159
```

Each time we call `square` in our code, we are telling `python` to assign the passed value to `x` and execute the body of the function for the specific value of `x`. Within a function's body we can access the function's arguments using their names specified in the header, regardless of the variable names provided during the function call. For example, the body of our function `square` refers to the input number by `x` and not by `radius`. This convention makes sense, since at the time we define a function we don't necessarily know in what context the function will be used later on. Hence, each of the function calls below assign a different value to `x` and thus return a different result:

```
A = 2
print("The square of {0} is {1}".format(A,square(A)))
B = 3
print("The square of {0} is {1}".format(B,square(B)))
```

```
C = 5
print("The square of {0} is {1}".format(C,square(C)))
```

```
The square of 2 is 4
The square of 3 is 9
The square of 5 is 25
```

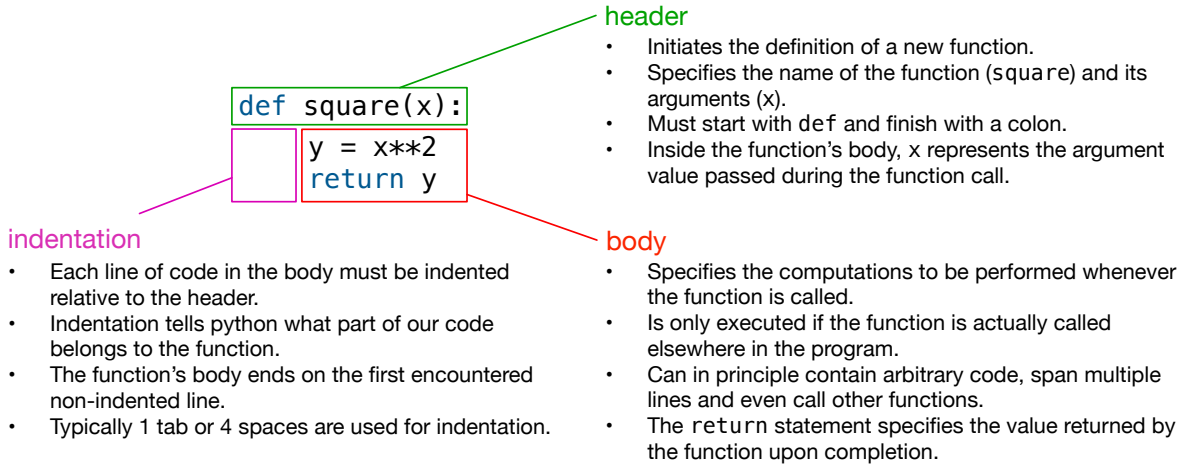
On the other hand, we cannot access the variables defined inside a function from outside, as these variables are only temporarily created during the execution of the function. In technical terms, such variables are called "local" to the function. For example, the following code would cause an error, because the local variable `y` is only temporarily created within the body of the function `square` and is not accessible from outside:

```
radius = 10
area = 3.14159 * square(radius)
print(y)
```

I should be noted that our function `square` assumes that the mathematical expression `x**2` inside its body is meaningful for the provided argument `x`. In principle, however, we could call the function with an argument for which squaring makes no sense, such as a string:

```
meaningless = square("shark")
```

The above code would generate an error at the point where `python` will attempt to raise `"shark"` to the power of 2, because such an operation is meaningless for strings. The responsibility is thus on us, the users of the function `square`, to only call it with meaningful arguments.




Initiates the definition of a new function.
Specifies the name of the function (`square`) and its arguments (x).
Must start with `def` and finish with a colon.
Inside the function's body, x represents the argument value passed during the function call.

Just like ordinary variables, we can name our functions and their arguments any way we like (but using only alphanumeric characters and underscores), although it is recommended to use concise informative names. Care must be taken not to accidentally use function names already used by another existing function (e.g., in a loaded package) or variable, as this will overwrite the existing function or variable and can lead to unexpected issues, for example if another part of your code relies on the overwritten function or variable. One way to avoid confusing functions with ordinary variables is to preferentially use verbs as function names instead of nouns (e.g., prefer `count_genes` over `number_of_genes`, and `analyze` over `analysis`).

**A function with multiple arguments:** In the example above we haven't really gained much by writing a new function for number squaring, since `radius**2` is just as concise and understandable as writing `square(radius)`. In realistic situations, however, function bodies commonly comprise many lines of code, and may utilize many other functions to accomplish their task. Let's consider a slightly more complex example. Let's define a function that takes as argument a single numeric list called `values`, as well as an integer variable called `rank`, and returns the element in the `values` list with the specified rank (e.g., the smallest value if `rank==0`, the 2nd-smallest value if `rank==1`, and so on):

```
def find_value_at_rank(values,rank):
    sorted_values = sorted(values)
    return sorted_values[rank]
```

Notice that the body of our function actually makes use of another function (`sorted`) to accomplish its task. We can now use our function `find_value_at_rank` to find elements in lists at any given rank, for example:

```
body_masses = [3, 2.6, 0.4, 6.8, 9, 8, 1.5, 7, 97, 71]
print("Third smallest mass:",find_value_at_rank(body_masses, 2))
print("Fifth smallest mass:",find_value_at_rank(body_masses, 4))
```

```
Third smallest mass: 2.6
Fifth smallest mass: 6.8
```

As with most functions provided by `python` packages, we can also pass arguments to our custom function by name, as follows:

```
print("Third smallest mass:",find_value_at_rank(values=body_masses, rank=2))
```

Passing arguments by name is very useful when a function takes many arguments, especially when some arguments are optional, as this removes any ambiguity in the meaning of the various arguments passed.

**Optional function arguments:** It is possible to declare some or all of a function's arguments as optional, which essentially means that they have some default value unless specified otherwise during the function call. We have already encountered optional function arguments in many examples in this book. For example, the arguments for line color and line thickness in the `pyplot` function `plot` were both optional, and if left unspecified some default values were used instead. Consider our earlier custom function `find_value_at_rank`, and suppose that we want to provide the option of sorting elements in reverse, so that `rank=0` returns the largest element (instead of the smallest), `rank=1` returns the second-largest element, and so on. We thus introduce a third, optional argument, which we call `rev` and which we assign the default value `False`:

```
def find_value_at_rank(values,rank,rev=False):
    sorted_values = sorted(values, reverse=rev)
    return sorted_values[rank]
```

Hence, if `find_value_at_rank` is called with `rev=False` or without specifying `rev` at all, the old behavior is retained; if the function is called with `rev=True`, then the ranking order is reversed:

```
body_masses = [3, 2.6, 0.4, 6.8, 9, 8, 1.5, 7, 97, 71]
print("Third smallest mass:",find_value_at_rank(values=body_masses, rank=2))
print("Third largest mass:",find_value_at_rank(values=body_masses, rank=2, rev=True))
```

```
Third smallest mass: 2.6
Third largest mass: 9
```

**Multiple return statements:** The above examples show that we use the `return` statement to specify the value that the function should return. It is also possible to define a function that does not return any value, for example a function that merely prints something to the screen. In that case, a `return` statement is not required. On the other hand, a function's body may include multiple `return` statements. In that case, the function's execution is considered complete as soon as one the `return` statements is encountered. Often, early `return` statements are used to handle problematic or unusual situations that prevent a function from executing properly to its full extent. For example, we could modify our earlier function `find_value_at_rank` such that it returns `NaN` (not-a-number) if the `values` list is empty, since there's no element available to return. In fact, we could make our function return `NaN` whenever `values` does not have more than `rank` elements, since again there's no element of the desired rank.

```
from numpy import NaN

def find_value_at_rank(values,rank,rev=False):
    if(len(values)<=rank):
        return NaN
    sorted_values = sorted(values, reverse=rev)
    return sorted_values[rank]
```

Let's see what happens when we request an element of excessively high rank:

```
print("Fiftieth smallest mass:",find_value_at_rank(values=body_masses, rank=49))
```

```
Fiftieth smallest mass: nan
```

**Returning multiple values:** Our previous function `find_value_at_rank` only computes the value of the element at a specific rank, but not its position (index) in the input list. Sometimes, both pieces of information may be desired. We can easily rewrite the function to return both the index and the value of the element at a given rank, as follows:

```
import numpy
def find_value_and_index_at_rank(values,rank,rev=False):
    if(len(values)<=rank):
        return NaN, NaN
    value_order = numpy.argsort(values)
    if(rev):
        # reverse the order
        value_order = value_order[::-1]
    index = value_order[rank]
    value = values[index]
    return value, index
```

Note that we use the `numpy` function `argsort` to determine the order of indices that would sort the input list, as explained in Section 4.15.6, and then pick the index placed at position `rank`. The `return` statement at the end of the function's body specifies the two variables to return under normal circumstances, while the conditional `return` statement at the beginning returns

`NaN, NaN` if the input list is too short. We can now use our new function to find both the value and index of the element at a specific rank in a list, for example:

```
body_masses = [3, 2.6, 0.4, 6.8, 9, 8, 1.5, 7, 97, 71]

mass, index = find_value_and_index_at_rank(body_masses, 2)
print("Third smallest mass (%g) found at index %d"%(mass,index))

mass, index = find_value_and_index_at_rank(body_masses, 2, rev=True)
print("Third largest mass (%g) found at index %d"%(mass,index))
```

```
Third smallest mass (2.6) found at index 1
Third largest mass (9) found at index 4
```

Note that we are expected to *collect* all of the function's returned values. For example, if we were to write:

```
results = find_value_and_index_at_rank(body_masses, 2)
```

then `results` would actually be a tuple containing two elements (the value and index of the 3-rd smallest body mass):

```
print(results)
```

```
(2.6, 1)
```

Hence, instead of

```
mass, index = find_value_and_index_at_rank(body_masses, 2)
```

we could also write:

```
results = find_value_and_index_at_rank(body_masses, 2)
mass    = results[0]
index   = results[1]
```

**Accessing outside variables from within a function:** Within a function's body, we can access any variables that would normally be accessible at the point where the function is called. This is useful for example if our function relies on previous computations (performed prior to the function call) or loaded data, and we don't want to pass all of those as additional arguments to the function. For example, suppose that we have a large dataset of species names, stored in the following text file:

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`animal_species_names_1000.txt`".

Suppose that we want to construct a function that takes a single string argument, `genus`, and that returns the number of times the given genus is found in the dataset.

```
with open("data/animal_species_names_1000.txt", "rt") as fin:
    dataset = fin.read()

def count_genus(genus):
```

```
    return dataset.count(genus+" ")
```

Observe that we are accessing the `dataset` variable from within the function's body, without actually passing it as an argument. We can now call our function with any arbitrary genus name as the sole argument:

```
for genus in ["Lelapia", "Eliurus", "Ascandra", "Grantia"]:
    N = count_genus(genus)
    print("%s occurs %d times"%(genus, N))
```

```
Lelapia occurs 4 times
Eliurus occurs 0 times
Ascandra occurs 14 times
Grantia occurs 42 times
```

**Calling functions within functions:** Since the body of a function can in principle contain arbitrary code, we can also call other functions (including custom ones) from within a function, and those other functions can themselves call other functions, and so on. It's helpful to think of a function like a baking recipe: A function calling other functions is like having a cake recipe that tells you to follow another recipe to make a glaze, another recipe to make the chocolate filling, and so on. This makes the cake's recipe more concise and readable, and reduces the duplication of recipes if other cakes also require a glazing or chocolate filling. Splitting computations into reasonable discrete logical steps encapsulated by individual functions is an essential aspect of any complex data analysis.

★ **Exercise 80:** Predict what the following code prints to the screen, without running it:

```
def first(x):
    y = x**2
    return y

def second(x):
    y = x*10
    return y

def third(x):
    return x+5

def fourth(x,y):
    z = x+y
    return z, first(x)+second(y)

print(third(second(first(2))))
print(fourth(first(3), second(4)))
print(first(3) + first(4) + second(5))
print(fourth(1,5))
```

★ **Exercise 81:** Define a new function (called `circle`) that takes a single argument ($r$), and returns the area ($\pi r^2$) and the circumference ($2\pi r$) of a circle with radius $r$. Use this function to compute and print out the surface areas and circumferences of three different circles, with radii $r = 1$ m, $r = 10$ m and $r = 1000$ m.

★ **Exercise 82:** Define a function that takes four arguments — `concentration`, `old_units`, `new_units` and `molar_mass` (in g/mol) — and which converts the concentration of a chemical compound dissolved in water at standard temperature and pressure between any of the following units (in any direction): `mg/L`, `g/L`, `mM` (millimoles per liter), `ppt` (parts per thousand w/w) and `ppm` (parts per million w/w). Use this function to convert 7 mg/L of dissolved oxygen ($O_2$) into the units mM, to convert 1 mM of dissolved methane ($CH_4$) into the units ppm w/w, and to convert 2 mg/L of dissolved hydrogen sulfide ($H_2S$) into the units ppt w/w. The function should return `NaN` if the old or new units are not in the above list.

*Hint: Keep in mind that the density of pure water at standard conditions is* $998.2\ \mathrm{g \cdot L^{-1}}$*. For this exercise, you may ignore the effects of the dissolved compound on the solution's density, i.e., you may assume that the solution is very dilute and thus its density is approximately equal to the density of pure water. Recall that* `NaN` *is defined in the* `numpy` *package.*

★ **Exercise 83:** This exercise's goal is to define a custom function for computing the rate of change of a biological variable measured over time, and to apply that function to COVID-19 incidence data. Perform the following tasks:

1. Specifically, define a function named `rate_of_change`, which takes two arguments (`times` and `values`), each assumed to be a 1-dimensional numeric `numpy` array. The argument `times` represents successive measurement times in ascending order, while `values` represents the variable's values at the corresponding times. Your function should compute the rate of change of the variable at each time point and return those rates of change as a new 1-dimensional `numpy` array. For every time point $n > 0$ the rate of change should be computed as:

   ```
   (values[n]-values[n-1])/(times[n]-times[n-1])
   ```

   while for $n = 0$ the rate of change should be set to `NaN`.

   *Hint: You can use the* `numpy` *function* `zeros` *or* `empty` *to create a 1-dimensional array of the same size as* `values`*, and then use a* `for` *loop to compute the rates of change and store them at the proper index. Alternatively, you may use* `numpy` *array operations and the* `numpy` *function* `diff` *to more succinctly compute the rates of change.*

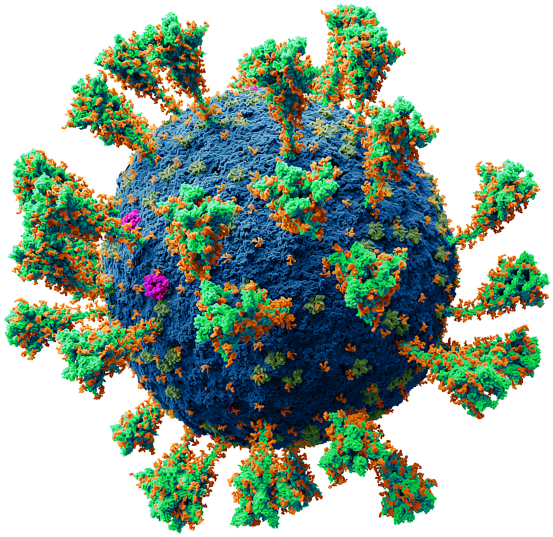
2. Consider the following weekly time series of cumulative confirmed COVID-19 cases and deaths in the US, stored as an HTSV file (one row per week):

   http://www.loucalab.com/archive/BioDataAnalysisPython
   → file "`WHO_COVID19_US_weekly_cumulative_counts.htsv`".

   Note that the first column lists dates (in the format YYYY.MM.DD), the second column lists times (in years), the third column lists cumulative confirmed COVID19 cases to date, and the fourth column lists cumulative confirmed COVID19-related deaths to date. Load the second and third column of this file as a `numpy` array, and plot the cumulative number of confirmed cases as a curve over time.

3. Use your previously defined function `rate_of_change` to compute the weekly rate of new COVID19 cases and plot it as a curve over time.

4. Use the function `rate_of_change` to compute the weekly rate of change of the previously computed weekly case rate (henceforth "acceleration"). In other words, compute how fast the weekly case rate changes over time. Plot the acceleration as a curve over time. Pay attention to show the proper units in the axis labels.



★★ **Exercise 84:** In the following we will construct a custom function that performs a simple type of time series smoothing called moving average. Smoothing time series (i.e., reducing fluctuations or noise) is a common preparatory step in scientific analyses, because many calculations yield unreliable results when the input data are too noisy [68–71]. Smoothing is also commonly used as part of exploratory data visualizations, because it can help emphasize biologically relevant trends. Starting with a time series $X_0, X_1, ...$, moving average smoothing constructs a new, smoother, time series $Y_0, Y_1, ..$ by replacing each point in the old time series with the average of multiple nearby points. The number of nearby points used for averaging is defined by the *window size* $W$, and typically an equal number of nearby points is taken towards the left and right (except at the beginning and end of the time series, where fewer points may be used). Here, we consider the following precise definition of the moving average: For any odd-numbered window size $W$, and for any index $n \in \{0, .., N-1\}$ (where $N$ is the size of the time series), compute the smoothened value $Y_n$ as the average of the values $X_{\max(0,n-(W-1)/2)}, ..., X_{\min(N-1,n+(W-1)/2)}$.

Perform the following tasks:

1. Define a new function named `moving_average` that accepts two arguments, `values` and `W`. The argument `values` should be assumed to be a 1-dimensional `numpy` array, containing the values of a numeric time series at increasing time points. The argument `W` should be assumed to be an odd integer, specifying the size of the smoothing window. The function should return a new 1-dimensional array of the same size as `values`, listing a moving-average-smoothened version of the input time series.

   *Hint: You could use the function* `numpy.zeros` *or* `numpy.empty` *to create a new 1-dimensional array of the same size as* `values`*, and then loop over all time points to compute and store the moving average in the appropriate index of the new array.*

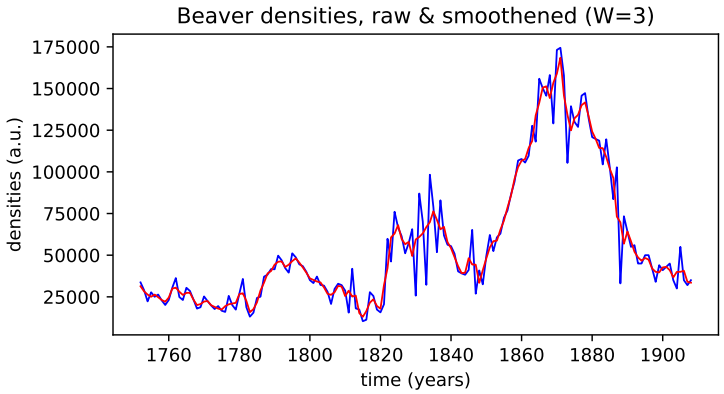
2. Consider the following dataset of annual animal population size estimates in the North-American tundra [35], stored in TSV file format (one row per year):

   http://www.loucalab.com/archive/BioDataAnalysisPython
   → file "`Hudson_Bay_fur_bearing_animals.tsv`".

   Note that the first column lists years, while the remaining columns list population size estimates for various animal species, including North American beavers (second column), American martens (third column) and so on (in arbitrary units). Load the first two columns of this file as a `numpy` array, and plot the beaver densities as a curve over time. Then use your

function `moving_average` (with `W=10`) to compute and plot into the same figure a smoothened version of the time series, using a different color.

*Hint: For `W=3`, the figure would look similar to the following:*

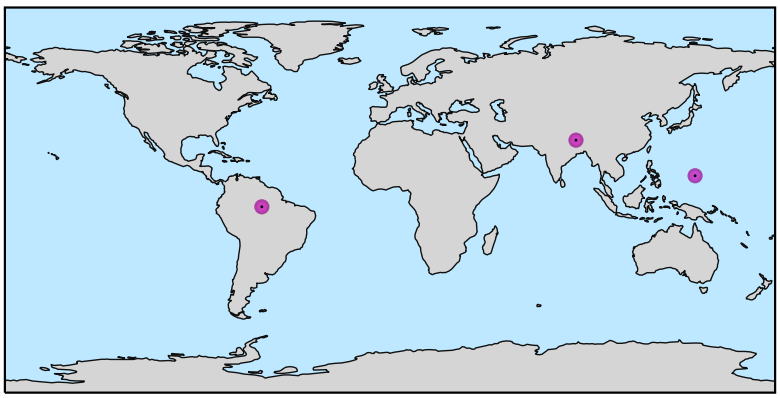



3. Repeat the previous task with a larger window size of `W=30`. How do the two smoothened curves (for `W=10` and `W=30`) differ?

★ **Exercise 85:** Consider the following gzipped TSV file, listing the coordinates of all major coastlines worldwide at roughly 50 m resolution (latitudes in the first column, longitudes in the second column):

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`coastline_coordinates_50m.tsv.gz`".

Load the contents of the TSV file as a `numpy` array. Then, define a new function that takes 2 numeric arguments, `latitude` and `longitude`, representing the coordinates of an arbitrary focal point on Earth's surface (in degrees), and that returns the great-circle distance of the focal point to the nearest coastline point (in km). Finally, use this function to compute the nearest distance to the coastline for each of the following locations (lat, lon): 11.373333, 142.591667 (Challenger Deep), 27.988056, 86.925278 (Mt. Everest) and -3.136667, -59.904722 (confluence between Rio Negro and Amazon river in Brazil, known as the meeting of waters). See the map below for reference:



*Hint: To compute the great-circle distance between any two points on Earth's surface, you can use the Vincenty formula introduced in Exercises 9 and 61. Try to use `numpy` array arithmetics and functions to evaluate Vincenty's formula and compute the distances of the focal point to all coastline points (i.e., without a loop), then return the minimum of those distances.*

## 5.2 List comprehension

We have seen that we can use the `map` function to apply another given function to every element of a list, for example, to convert a list of strings into a list of numbers (Section 4.10.2). Often, the computation that we wish to perform with each element may be more complicated than just applying a single function. Of course, in that case we could write a full-fledged `for` loop that iterates over all elements of the old list and stores the computed values into a new list, or we could define a new custom function that we then apply using `map`. In many cases, however, a third option is the most concise, elegant and straightforward: list comprehension. List comprehension essentially allows us to combine a `for` loop and the generation of a new list into a single statement. Suppose,, for example, that we have a list of Cetacean latin-binomial species names:

```
names = ["Eubalaena australis", "Balaenoptera musculus", "Lagenorhynchus cruciger",\
         "Megaptera novaeangliae", "Balaenoptera borealis", "Eubalaena glacialis",\
         "Balaenoptera brydei", "Balaenoptera physalus", "Eubalaena japonica"]
```

and that we wish to create a new list with only the generic name parts. We can achieve this with the tools that have already learned, for example a `for` loop:

```
genera = []
for name in names:
    genera += [name.split(" ")[0]]
```

Note that we are splitting each latin name at the space and are keeping the first part as generic name, which we then append to the `genera` list. This list will thus contain the genus names corresponding to the original binomial names:

```
print(genera)
```

```
['Eubalaena', 'Balaenoptera', 'Lagenorhynchus', 'Megaptera', 'Balaenoptera', '
    Eubalaena', 'Balaenoptera', 'Balaenoptera', 'Eubalaena']
```

The same outcome can be achieved more concisely using list comprehension:

```
genera = [name.split(" ")[0] for name in names]
```

Notice that list comprehension still includes an internal `for` loop that defines an iterator variable (`name`) and the sequence over which to iterate (`names`), but the loop's body has now been compressed into a single expression (`name.split(" ")[0]`), each outcome of which defines an element in the newly generated list. In plain language, the above statement tells `python` to:

> *evaluate the expression* ***`name.split(" ")[0]`*** *separately for each element* ***`name`*** *in the list* ***`names`****, use the results to form a new list, and store that list in a variable called* ***`genera`****.*

The expression evaluated in a list comprehension can in principle be more complex and include multiple operations. For example, suppose that we wanted to create a list that lists latin binomial names, but with the genus name abbreviated to the first letter. This can be achieved through a simple modification of our previous code:

```
abbreviations = [name[0]+". "+name.split(" ")[1] for name in names]
```

Observe that we are taking the first character of each name, as well as the second part obtained

from a split at the space, and concatenate these two with an ". " in between. Let's check that everything worked as intended:

```
print(abbreviations)
```

```
['E. australis', 'B. musculus', 'L. cruciger', 'M. novaeangliae', 'B. borealis', 'E.
    glacialis', 'B. brydei', 'B. physalus', 'E. japonica']
```

**List comprehension with conditional inclusion:** List comprehension also allows us to conditionally include/exclude elements from the new list, using an optional `if` statement after the `for` statement. For example, suppose that we only wanted to keep abbreviated names for species in the *Eubalaena* genus, omitting all others. This can be achieved as follows:

```
abbreviations = [n[0]+". "+n.split(" ")[1] for n in names if n.startswith("Eubalaena")]
```

Note that we replaced `name` with the truncated `n` for the sole purpose of fitting everything into one line. In plain language, the above code tells `python` to:

> *evaluate the expression `n[0]+". "+n.split(" ")[1]` separately for each element `n` in the list `names`, but only if `n` starts with "`Eubalaena`", use the results to form a new list, and store the list in a variable called `abbreviations`.*

We thus obtain the following shorter list, only containing abbreviations of the *Eubalaena* species:

```
print(abbreviations)
```

```
['E. australis', 'E. glacialis', 'E. japonica']
```

**Using list comprehension for filtering:** List comprehension with conditional inclusion of elements can also be quite useful for filtering an existing list, i.e., keeping a subset of its elements without modifying them in any other way. For example, suppose that we have a list of tree heights measured in different forest patches, stored as a list of (sub)lists (one sublist per forest patch), such as the following:

```
tree_heights = [[0.5, 0.8, 0.3, 1.1, 1.4], [0.1, 0.76, 2.3], [1.8, 2.3], [0.53, 0.05],\
                [2, 0.9, 2.2, 3.1], [0.6], [5, 7.9, 3, 1.8], [7.1, 6.5, 1.1, 4]]
```

Suppose that we wish to only analyze those forest patches within which at least 4 trees were measured, so as to avoid inaccurate results for forest patches with insufficient data. To create a new list only comprising sublists of forest patches with at least 3 measurements, we can use list comprehension as follows:

```
tree_heights_filtered = [patch for patch in tree_heights if (len(patch)>=4)]
```

Let's check the outcome:

```
print(tree_heights_filtered)
```

```
[[0.5, 0.8, 0.3, 1.1, 1.4], [2, 0.9, 2.2, 3.1], [5, 7.9, 3, 1.8], [7.1, 6.5, 1.1, 4]]
```

**Iterating over multiple corresponding lists:** Similarly to regular `for` loops, we can also loop over multiple corresponding lists in a list comprehension using the keyword `zip`. For example, suppose that we have another list of the same length as `tree_heights`, specifying the corresponding forest patch's area (in $km^2$):

```
areas = [12, 5.5, 8.3, 67, 19, 20.2, 16.1, 7.5]
```

Suppose that we wish to only keep height measurements from forest patches with areas above 10 $km^2$. This can be achieved with the following list comprehension, which iterates concurrently over `tree_heights` and `areas`:

```
tree_heights_filtered = [patch for patch,area in zip(tree_heights,areas) if (area>10)]
```

Let's check the outcome:

```
print(tree_heights_filtered)
```

```
[[0.5, 0.8, 0.3, 1.1, 1.4], [0.53, 0.05], [2, 0.9, 2.2, 3.1], [0.6], [5, 7.9, 3, 1.8]]
```

★ **Exercise 86:** Suppose that we have two numeric lists of the same length, listing average body masses for various species and corresponding global population sizes, respectively, for example:

```
body_masses = [0.4, 5.6, 2.3, 9.0, 2.4, 3.5, 70, 91, 64, 76]
pop_sizes   = [101, 30, 35, 982, 74, 378, 2984, 17, 559, 308]
```

The following `for` loop creates a new list, specifying the global biomass for each species:

```
global_biomasses = []
for body_mass, pop_size in zip(body_masses, pop_sizes):
    if(body_mass>1):
        global_biomasses += [body_mass * pop_size]
print(global_biomasses)
```

```
[168.0, 80.5, 8838.0, 177.6, 1323.0, 208880, 1547, 35776, 23408]
```

Rewrite the above loop as a list comprehension, to generate the same list `global_biomasses`.

★ **Exercise 87:** Write a single list comprehension statement that creates a list of all powers of 2, from 1 ($= 2^0$) until and including 1024 ($= 2^{10}$).

★ **Exercise 88:** In this exercise we will learn to use list comprehension to elegantly omit comment lines from a loaded text file. Specifically, consider the following small dataset of cetacean species names, in the form of a plain text file (one line per species):

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`species_with_comments.txt`".

Let's have a look at the first few lines of this file:

```
# Cetacean binomial species names (one line per species)
# This non-exhaustive list also includes extinct species
```

```
# Source: Wikipedia, July 21, 2022
#
# Family Balaenidae, extant
Balaena mysticetus
Eubalaena glacialis
Eubalaena japonica
Eubalaena australis
#
# Family Balaenidae, extinct
Balaena affinis
```

Notice that some lines start with a #, which indicates that these lines are comment lines and not actual species names. Read the entire contents of this file as a list of strings (one string per line), then use list comprehension with conditional inclusion to only keep non-comment lines. Print out the total number of non-comment lines loaded.

★ **Exercise 89:** Suppose that we have a list of latitudes (in degrees), such as the following:

```
latitudes = [44.539, 39.984, 56.711, 29.081, -87.23, -1.893, 21.783, -48.772, 86.325]
```

Perform the following tasks:

1. Write a code that uses list comprehension to compute the distance from the equator corresponding to each latitude, thus obtaining a new list of the same length. Distances should be computed in km, and rounded to the nearest km.

   *Hint: You may assume that the Earth is a sphere of radius* $R = 6371$ km*; thus, the distance to the equator is* $\pi R\,|\varphi|\,/180$*, where* $\varphi$ *is the latitude in degrees.*

2. Modify your list comprehension code to only include locations between the polar circles, i.e., between the latitudes -66.56°and 66.56°.

★ **Exercise 90:** Suppose that we have a list of tree heights measured in different forest patches, stored as a list of (sub)lists (one sublist per forest patch), such as the following:

```
tree_heights = [[0.5, 0.8, 0.3, 1.1, 1.4], [0.1, 0.76, 2.3], [1.8, 2.3], [0.53, 0.05],\
                [2, 0.9, 2.2, 3.1], [0.6], [5, 7.9, 3, 1.8], [7.1, 6.5, 1.1, 4]]
```

Perform the following tasks:

1. Write a code that uses list comprehension to compute the total number of trees measured.

   *Hint: You can first create a list containing the number of trees per patch, and then use the* ***numpy*** *function* ***sum*** *to sum up the elements of that list.*

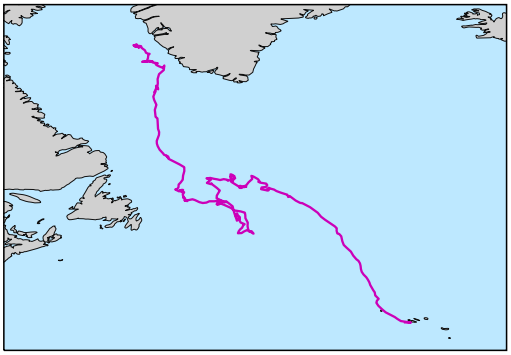
2. Write a code that uses list comprehension to compute and print the mean tree height for each forest patch, thus yielding a new numeric list that has as many elements as there are forest patches.

   *Hint: You can use the* ***numpy*** *function* ***mean*** *to compute the mean of each sublist.*

★★ **Exercise 91:** Consider the following two datasets:

http://www.loucalab.com/archive/BioDataAnalysisPython
→ files "`coastline_coordinates_50m.tsv.gz`"
and "`sei_whale_route_Silva2013.htsv`".

The first file lists the coordinates of all major coastlines worldwide at roughly 50 m resolution in TSV format (one row per coastline point, latitudes in the first column, longitudes in the second column). The second file lists the migratory route traversed by a sei whale in the Northern Atlantic, recorded by the Azores Great Whales Satellite Telemetry Program [72], in HTSV format (one row per recording, dates and times in the first column, time elapsed since the first recording in the second column, latitudes in the third column, longitudes in the fourth column). All coordinates are given in degrees and times elapsed are measured in hours. The map below shows the whale's route for reference.



Perform the following tasks:

1. Load the coastline coordinates as a `numpy` array. Define a custom function that takes 2 numeric arguments (`latitude` and `longitude`) representing the coordinates of an arbitrary focal point on Earth's surface, and which returns the great-circle distance of the focal point to the nearest coastline point (in km, henceforth "coastal distance"). Next, load the elapsed times and coordinates of the whale's route (second, third and fourth columns in the file) as a `numpy` array. Use your custom function inside a list comprehension to compute for each point of the route the point's coastal distance. Finally, compute and print out the minimum, maximum and average of the route's coastal distances. In other words, determine the smallest, largest and average distance of the whale from the nearest coast. The average should weigh each recorded point equally, i.e., you don't need to account for the somewhat variable time intervals between recordings.

   *Hint: To compute the great-circle distance between any two points on Earth's surface, you can use the Vincenty formula introduced in Exercises 9 and 61. Also see Exercise 85. Pay attention to properly convert coordinates between degrees and radians, as needed. For example,* ***`numpy`****'s trigonometric functions expect angles in radians. Try to use* ***`numpy`****'s array arithmetics inside your custom function when computing the distances to all coastline points, for efficiency.*

2. Compute for each pair of consecutive points in the route ("route segment") their great-circle distance, thus obtaining an array listing the approximate lengths of all route segments (in km). Also compute the time differences between consecutive points in the route, thus obtaining an array listing the durations of all route segments (in hours). Divide the segment lengths by their durations, to estimate the travel speed of the whale in each route segment (in km/hour). Print out the minimum, maximum and average speed of the whale.

3. Create a scatterplot comparing the coastal distances (horizontal axis) to the travel speeds (vertical axis) along the whale's route (one point plotted per route segment). For each route segment, use the coastal distance of the segment's initial location. Do you see a strong correlation between coastal distance and travel speed?

## 5.3 Boolean arrays

We have seen previously that we can use `numpy` arrays in mathematical expressions to efficiently apply calculations to entire datasets (Section 4.15.5). Here we will learn to evaluate logical conditions and perform boolean algebra using entire arrays, in a similar fashion. This may be useful, for example, for determining which records in a time series dataset satisfy a certain condition, for filtering a dataset based on specific quality criteria, or for replacing all missing/erroneous data points with some default value. Central to our discussion will be boolean arrays, i.e., `numpy` arrays consisting entirely of boolean variables (`True` or `False`). Most of what we know from numeric arrays also applies to boolean arrays. For example, the following code defines a 1-dimensional boolean array:

```
B = numpy.array([True, False, False, False, True, False])
```

The following code constructs and prints a $3 \times 4$ boolean array whose elements are all `False`:

```
B = numpy.zeros([3,4], dtype=bool)
print(B)
```

```
[[False False False False]
 [False False False False]
 [False False False False]]
```

Note that here we used the fact that `False` corresponds to 0, hence constructing an array willed with zeros and demanding that its data type be `bool` yields the desired boolean array.

In most practical scenarios a boolean array is created as the result of a logical condition, evaluated over all elements of one or more arrays. Suppose, for example, that we have a dataset of optical density measurements for multiple cell cultures [73], in the form of a 1-dimensional numeric array:

```
OD = numpy.array([0.63, 0.2, -0.04, 0.98, 0.75, 0.53, 0.09, -0.02, 1.02, 0.56])
```

While negative optical densities are theoretically implausible, they may occasionally be obtained in practice if cell counts are very low or due to procedural failures. We can easily determine which elements in `OD` are negative as follows:

```
negative = (OD<0)
```

Upon execution of the above statement, `negative` will be a 1-dimensional boolean array of the same length as `OD`, whose elements are the result of evaluating the logical condition $x < 0$ for every element $x$ in `OD`:

```
print(negative)
```

```
[False False  True False False False False  True False False]
```

Observe that `negative` contains the value `True` in those exactly positions where `OD` has negative values. More complex logical conditions can also be applied using `numpy`'s boolean algebra. For example, to determine which elements in `OD` are within the interval [0.2, 0.8], we can write:

```
in_interval = (OD>=0.2) & (OD<=0.8)
```

Note that we used the symbol "&" instead of the familiar boolean operator `and`. This is because `numpy` arrays use a distinct set of symbols for logical operations, which however map one-to-one to `python`'s standard `and` ("&"), `or` ("|") and `not` ("~"). Each of the two logical conditions `(OD>=0.2)` and `(OD<=0.8)` is evaluated for each individual element in `OD`, thus yielding two temporary boolean arrays of the same length as `OD`. These temporary boolean arrays are then compared via the logical operator `&` element-wise, thus yielding the boolean array `in_interval`:

```
print(in_interval)
```

```
[ True  True False False  True  True False False False  True]
```

Observe that `in_interval` contains the value `True` in exactly those positions where `OD` has values between 0.2 and 0.8. Alternatively, to determine which optical densities are extremely low or extremely high, i.e., either below 0.2 or above 0.8, we could write:

```
extreme = (OD<0.2) | (OD>0.8)
print(extreme)
```

```
[False False  True  True False False  True  True  True False]
```

The fact that `True` and `False` can also be interpreted as numbers 1 and 0, respectively, means that we can also use `numpy`'s arithmetic capabilities to compute basic statistical properties of boolean arrays, such as the number and fraction of `True` elements. For example, to count how many cell cultures had optical densities below 0.2 or above 0.8, we can use the function `numpy.sum`:

```
print("Number of extreme ODs:",numpy.sum(extreme))
```

```
Number of extreme ODs: 5
```

To compute the fraction of cell cultures with optical densities below 0.2 or above 0.8, we can use the function `numpy.mean`:

```
print("Fraction of extreme ODs:",numpy.mean(extreme))
```

```
Fraction of extreme ODs: 0.5
```

Of course, we could have also determined the above number and fraction using a loop, for example as follows:

```
Nextreme = 0
for density in OD:
    if((density<0.2) or (density>0.8)):
        Nextreme += 1
fraction_extreme = Nextreme/len(OD)
print("Number of extreme ODs:",Nextreme)
print("Fraction of extreme ODs:",fraction_extreme)
```

```
Number of extreme ODs: 5
Fraction of extreme ODs: 0.5
```

In many situations, however, we can write more concise and elegant code using boolean array arithmetics (as the above example demonstrates).

**Indices of elements satisfying a condition:** To obtain the indices (positions) of the `True` elements in a boolean array, we can use `numpy`'s function `where`. This function returns the indices of `True` elements, separately for each axis (although in the above examples our boolean arrays only have one axis). Let's try it out using our previously generated boolean array `negative`:

```
negative_indices = numpy.where(negative)[0]
```

Hence, `negative_indices` will be an integer array, listing the indices of the `True` elements in `negative`, or equivalently, the indices of the negative elements in `OD`:

```
print(negative_indices)
```

```
[2 7]
```

Note that for technical reasons we appended `[0]` to the return value of the function `numpy.where`, in order to extract the desired indices as an integer array (if you are curious to see what the full return value of `numpy.where` looks like, simply omit the `[0]` and re-run the code). For brevity, we could have also nested the logical expression `OD<0` into `numpy.where` without defining the intermediate boolean array `negative`, for example:

```
print("Indices of negative ODs:",numpy.where(OD<0)[0])
```

```
Indices of negative ODs: [2 7]
```

Similarly:

```
print("Indices of extreme ODs:",numpy.where((OD<0.2) | (OD>0.8))[0])
```

```
Indices of extreme ODs: [2 3 6 7 8]
```

**Multidimensional arrays:** The above procedures can also be applied to multidimensional arrays, although the behavior of `numpy.where` is a bit more complex. For example, suppose that we have a 2-dimensional array listing the relative population sizes of multiple bacterial species over time (one row per time point, one column per bacterial species, see for example Exercise 63):

```
abundances = numpy.array([[0.097,   0.35,   0.37],
                          [0.16,    0.26,   0.29],
                          [0.14,    0.09,   0.37],
                          [0.15,    0.21,   0.28],
                          [0.077,   0.28,   0.24],
                          [0.078,   0.35,   0.23],
                          [0.023,   0.44,   0.40]])
```

Suppose that we want to find all elements below a certain threshold, say 0.1. We can achieve this easily with an appropriate logical condition applied to the entire array:

```
below = (abundances<0.1)
print(below)
```

```
[[ True False False]
 [False False False]
 [False  True False]
 [False False False]
 [ True False False]
 [ True False False]
 [ True False False]]
```

Further, we can use `numpy.where` to obtain the indices of elements below the threshold:

```
below_indices = numpy.where(below)
print(below_indices)
```

```
(array([0, 2, 4, 5, 6]), array([0, 1, 0, 0, 0]))
```

Observe that `numpy.where` returns two 1-dimensional integer arrays, the first one listing the row indices and the second one listing the corresponding column indices of the elements in `abundances` that are below 0.1. Hence, the first such element is located in row 0 and column 0, the next such element is located in row 2 and column 1, and so on. In total, there are 5 elements below 0.1.

★ **Exercise 92:** Suppose that we have measured the ages (in years) in a population of trees, and stored those in the form of a 1-dimensional `numpy` array, such as the following:

```
ages = numpy.array([13.5, 14, 18.3, 9.2, 0.19, 6.7, 8.3, 20.4, 15.2, 4.5, 8.6, 19.8])
```

Write a code that uses boolean array arithmetics to compute and print the fraction of trees aged between 10 and 20 years. Do not use a loop or list comprehension.

★ **Exercise 93:** Consider the following HTSV file, listing the proportions of 3 bacterial species in a wastewater treatment plant, measured at weekly intervals (one row per week, one column per species) [43]:

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`WWTP_AOB_abundances_Ofiteru2010.htsv`".

Perform the following tasks:

1. Load this file as a `numpy` array, and use boolean array arithmetics to compute and print the fraction of weeks at which at least one species had abundance above 0.4. Do not use a loop or list comprehension.

   *Hint: You can use the function* ***`numpy.max`*** *(applied along axis 1) to first determine the weeks in which the maximum abundance of any species was above 0.4, then use* ***`numpy.mean`*** *to compute the fraction of those weeks.*

2. Use boolean array arithmetics to compute and print the fraction of weeks at which the most abundant species was more than 10 times more abundant than the rarest species. Do not use a loop or list comprehension.

## 5.4 Boolean indexing of arrays

We have seen that we can access and modify individual elements and even rather complex parts of `numpy` arrays using positional indexing, i.e., by providing a range or sequence of integer indices specifying the desired locations along each axis (Section 4.15.3). Here we will learn about another powerful indexing method called boolean indexing. In boolean indexing, we specify a subset of elements in an array by providing a suitably shaped boolean array, i.e., an array containing only the values `True` and `False`, where `True` denotes elements to be kept and `False` denotes elements to be omitted. This is where the tools for creating boolean arrays, discussed earlier in Section 5.3, become particularly useful. As a recurring example, let's consider the following 1-dimensional array listing optical density measurements of bacterial cultures:

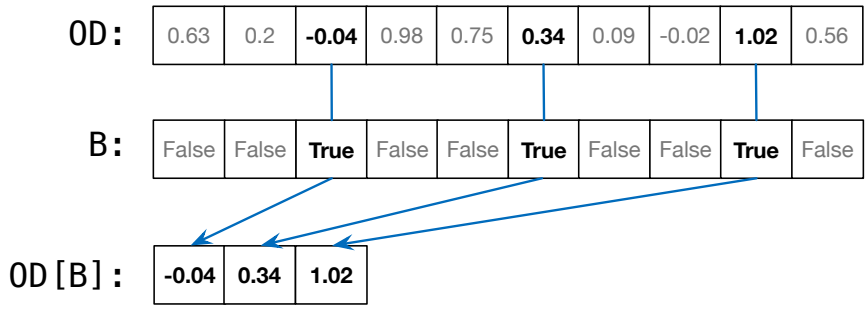
```
OD = numpy.array([0.63, 0.2, -0.04, 0.98, 0.75, 0.34, 0.09, -0.02, 1.02, 0.56])
```

To extract a subset of this array using boolean indexing, instead of a range/sequence of positional indices we specify a sequence of booleans of the same length as `OD`, for example as follows:

```
B = [False, False, True, False, False, True, False, False, True, False]
extracted = OD[B]
print(extracted)
```

```
[-0.04  0.34  1.02]
```

Observe that `extracted` only includes those elements of `OD` at the corresponding positions of which `B` was `True`.




If the provided list of booleans comprised only `False`, then we would have extracted an empty array:

```
B = [False, False, False, False, False, False, False, False, False, False]
extracted = OD[B]
print(extracted)
```

```
[]
```

In practice, we often wish to extract elements from an array based on some logical condition. Suppose, for example, that we want to only keep non-negative (i.e., biologically meaningful) optical densities. We learned recently how we can construct a suitable boolean array encoding this condition:

```
B = (OD>=0)
print(B)
```

```
[ True  True False  True  True  True  True False  True  True]
```

Hence, to extract all non-negative elements in `OD` we can write:

```
OD_nn = OD[B]
```

We can also avoid the definition of the intermediate variable `B` and instead write more concisely:

```
OD_nn = OD[OD>=0]
```

Let's look at the contents of `OD_nn`:

```
print(OD_nn)
```

```
[0.63 0.2  0.98 0.75 0.34 0.09 1.02 0.56]
```

Voila! The array `OD_nn` now only contains the non-negative optical density measurements.

**Filtering rows and columns:** Boolean indexing can also be applied to multidimensional arrays, such as to extract a subset of rows or columns. For example, suppose that we have a 2-dimensional array listing the relative population sizes of multiple bacterial species over time (one row per time point, one column per bacterial species, see for example Exercise 63):

```
abundances = numpy.array([[0.097,   0.35,   0.37],
                          [0.16,    0.26,   0.29],
                          [0.14,    0.09,   0.37],
                          [0.15,    0.21,   0.28],
                          [0.077,   0.28,   0.24],
                          [0.078,   0.35,   0.23],
                          [0.023,   0.44,   0.40]])
```

The following code extracts and prints the first two columns of the array (i.e., corresponding to the first two bacterial species) using boolean indexing:

```
print(abundances[:,[True, True, False]])
```

```
[[0.097 0.35 ]
 [0.16  0.26 ]
 [0.14  0.09 ]
 [0.15  0.21 ]
 [0.077 0.28 ]
 [0.078 0.35 ]
 [0.023 0.44 ]]
```

Similarly, the following code extracts and prints the last two rows of the array:

```
print(abundances[[False,False, False, False, False, True, True],:])
```

```
[[0.078 0.35  0.23 ]
 [0.023 0.44  0.4  ]]
```

Similarly to the 1-dimensional case, we can use boolean indexing to filter an array's rows or

columns based on some logical condition. Suppose, for example, that we only wanted to keep those species in our dataset whose average abundance exceeds a certain threshold, say, 0.2. We can compute the average abundance of each species using the `numpy` function `mean`, applied along the first axis:

```
averages = numpy.mean(abundances, axis=0)
print(averages)
```

```
[0.10357143 0.28285714 0.31142857]
```

We can then determine which columns of the `abundances` array have an average value above 0.2 using a logical condition, which yields a boolean array with as many elements as there are species:

```
B = (averages>0.2)
print(B)
```

```
[False  True  True]
```

Finally, we can use the boolean array `B` to extract the desired columns of the `abundances` array:

```
abundances_filtered = abundances[:,B]
print(abundances_filtered)
```

```
[[0.35 0.37]
 [0.26 0.29]
 [0.09 0.37]
 [0.21 0.28]
 [0.28 0.24]
 [0.35 0.23]
 [0.44 0.4 ]]
```

As usual, we could have avoided the intermediate variables `averages` and `B` by nesting the above statements for conciseness:

```
abundances_filtered = abundances[:,(numpy.mean(abundances, axis=0)>0.2)]
print(abundances_filtered)
```

```
[[0.35 0.37]
 [0.26 0.29]
 [0.09 0.37]
 [0.21 0.28]
 [0.28 0.24]
 [0.35 0.23]
 [0.44 0.4 ]]
```

**Replacing elements with a single value based on a logical condition:** Boolean indexing can also be used to modify an arbitrary subset of elements in an array. The simplest case is one where we wish to replace all elements satisfying a certain condition with a single common value. For example, suppose that we want to replace all negative elements in our earlier example array, `OD`, with the value 0. This may be necessary, for example, if downstream statistical analyses fail

when confronted with (physically implausible) negative optical densities. We can easily perform this replacement as follows:

```
OD[OD<0] = 0
```

Here, the expression `OD<0` results in a temporary boolean array of the same length as `OD`, specifying for each element whether it is negative or not. This boolean array is then used to specify the subset of elements in `OD` that are to be replaced by 0. Let's print the modified array to confirm our expectations:

```
print(OD)
```

```
[0.63 0.2  0.   0.98 0.75 0.34 0.09 0.   1.02 0.56]
```

It is worth noting that we could have achieved the same outcome using many alternative methods already discussed in this book. Indeed, we could have instead used a simple loop:

```
for k in range(len(OD)):
    if(OD[k]<0):
        OD[k] = 0
```

Alternatively, we could have used list comprehension to generate a list of indices corresponding to the negative values:

```
OD[[k for k in range(len(OD)) if (OD[k]<0)]] = 0
```

That said, using boolean indexing to modify elements of an array often allows for a more concise and elegant solution than loops or list comprehensions. Boolean indexing can also be used for replacing elements in multidimensional arrays. Suppose that we had a 2-dimensional array of optical densities, `multiOD`, listing optical densities of different cell cultures exposed to different experimental treatments (one column per treatment):

```
multiOD = numpy.array([[0.2,    0.35,   0.67],
                       [0.24,   0.64,   0.49],
                       [0.98,   0.07,   0.56],
                       [-0.02,  -0.12,  0.01],
                       [0.07,   0.23,   -0.08]])
```

To replace all negative elements in `multiOD` with 0, we can use the same syntax as in the 1-dimensional case:

```
multiOD[multiOD<0] = 0
print(multiOD)
```

```
[[0.2  0.35 0.67]
 [0.24 0.64 0.49]
 [0.98 0.07 0.56]
 [0.   0.   0.01]
 [0.07 0.23 0.  ]]
```

**Replacing elements with multiple corresponding values:** The situation is a bit more complex if we want to replace multiple elements with multiple corresponding values (rather than

with a single value). In that case, the right hand side of the assignment must be consistent with the indexed subset on the left hand side in terms of shape (we have encountered this requirement before in the case of positional indexing, Section 4.15.3). Suppose, for example, that we measured optical densities using two distinct procedures, with the first procedure generally being more accurate than the second procedure, but also more complicated and prone to failure. Missing values or failed measurements are often represented by `NaN`, which is defined in the `numpy` package and stands for "not a number".

```
OD1 = numpy.array([0.63, 0.2,  NaN,  0.98, 0.75, 0.34, 0.09, NaN,  NaN,  0.53])
OD2 = numpy.array([0.59, 0.23, 0.07, 0.91, 0.70, 0.36, 0.01, 0.15, 0.99, 0.54])
```

One approach to reducing these data gaps would be to use the measurements from the first procedure (`OD1`) wherever available, but replace the few missing data points with measurements from the second (less accurate) procedure (`OD2`). This can easily be achieved using boolean indexing:

```
missing = numpy.isnan(OD1)
OD1[missing] = OD2[missing]
```

Here, we first use the `numpy` function `isnan` to determine which elements of `OD1` are `NaN` and which ones aren't, thus obtaining a boolean array:

```
print(missing)
```

```
[False False  True False False False False  True  True False]
```

We then use this boolean array to replace all `NaN` elements of `OD1` with the corresponding elements extracted from `OD2`. Let's look at the modified `OD1`:

```
print(OD1)
```

```
[0.63 0.2  0.07 0.98 0.75 0.34 0.09 0.15 0.99 0.53]
```

It is also possible to use boolean indexing to replace multiple elements with multiple corresponding values in multidimensional arrays. For example, if `OD1` and `OD2` were multidimensional arrays of the same shape, then the above approach would work just as in the 1-dimensional case. Similarly to positional indexing, boolean indexing is a deep topic with many nuances not discussed here. For more details see this official documentation, but also feel free to experiment by yourself.

★ **Exercise 94:** Predict the full output of the following code, without running it.

```
latitudes = numpy.array([150, 79, 23, 198, 1090, 600, 341, 589, 200, 890])
zones = numpy.zeros(latitudes.shape, dtype=int)
zones[latitudes<100] = 0
zones[(latitudes>=100) & (latitudes<500)] = 1
zones[(latitudes>=500)] = 2
print(zones)
```

★ **Exercise 95:** Suppose that we have a 2-dimensional numeric `numpy` arrays of the same shape, `coverages` and `ground`, with both arrays listing estimated tree coverages on a geographic

grid. The first array lists estimates for all grid points based on remote (satellite) sensing, while the second array lists more reliable estimates based on ground surveys for a small subset of grid points. Missing estimates are represented by `NaN`, as in the example below:

```
coverages = numpy.array([[0.4,  0.9,  0.5,  0.15],
                         [0.98, 0.2,  0.35, 0.24],
                         [0.55, 0.13, 0.8,  0.7]])

ground = numpy.array([[NaN,   0.94, NaN,  NaN],
                      [0.87,  NaN,  NaN,  NaN],
                      [NaN,   NaN,  0.75, NaN]])
```

Write a code that replaces all elements in `coverages` with their more reliable counterparts in `ground`, whenever the latter are not `NaN`. In other words, if a ground survey-based coverage estimate is available for a grid point, it should replace the remote sensing-based value in `coverages`. Your code should print out the modified `coverages` array.

★ **Exercise 96:** Consider the following TSV file, specifying physical properties of various forest patches:

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`forest_patch_properties.tsv`".

Each row in the above file represents a different forest patch, the first column lists soil pH values, the second column lists average yearly precipitations (in mm), and the third column lists patch areas (in $km^2$). Consider a specific tree species that can grow in a soil pH range of 4–6 and an annual precipitation range of 200–350 mm. Write a code that uses boolean indexing and array arithmetics to compute and print the number of forest patches in which this species can grow, as well as the total area of those patches. Do not use any loop or list comprehension.

★ **Exercise 97:** Consider the following dataset of adult body masses (gram) and basal metabolic rates (Watt) for various eukaryotic and prokaryotic species [19], stored as an HTSV file:

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`mass_metabolic_rate_Hatton2019.htsv`".

Note that each row corresponds to a different individual (some species are represented multiple times), the first column lists major taxon names (e.g., "Mammal", "Protist", "Plant", "Bird", "Prokaryote" and so on), the fourth column lists body masses and the fifth column lists basal metabolic rates. Perform the following tasks:

1. Load the first column of the above file as a 1-dimensional string array (`dtype=str`), and load the fourth and fifth columns as a 2-dimensional numeric array (`dtype=float`). For each major taxon, compute and print the average body mass and average basal metabolic rate (averages should weigh all entries in the dataset equally, counting each species as many times as it is represented). For any given major taxon, use boolean indexing and the `numpy` array function `mean` to compute these averages.

   *Hint: To determine the unique set of major taxa in the dataset, you can use a* `set` *as described in Section 4.11. You can use a* `for` *loop to iterate over each unique major taxon. For each major taxon, determine which records are associated with that taxon as a boolean array.*

2. Create a scatterplot showing body masses on the horizontal axis and corresponding metabolic rates on the vertical axis across all birds, reptiles, protists and prokaryotes, coloring each of these four taxa with a different color. Both plot axes should be on a logarithmic scale. Do include a legend that shows the color of each taxon. Make the points slightly transparent to ensure the visibility of all taxa.

   *Hint: See Section 4.16.1 on creating scatterplots.*

## 5.5 Pandas dataframes

We have seen that we can use `numpy` arrays to store multidimensional data, such as the values of multiple morphological traits across multiple animal species. `Numpy` arrays, however, have two important shortcomings: First, they are primarily designed to store data of a single type, for example only numbers or only strings. Second, they don't provide a a convenient way to store row and column names, and hence indexing is primarily done based on integer position[1]. Both of these shortcomings are resolved by another important data structure: dataframes, defined in the package `pandas`. The name `pandas` stands for "panel data", which is a technical term for multidimensional datasets (think, e.g., of flight departure/arrival panels at airports, or financial panels at the stock exchange). `Pandas` dataframes (henceforth "dataframes" for brevity) provide a convenient toolset for representing and working with tables exhibiting named rows and columns. In `pandas` terminology, rows and columns are uniquely "labeled", and labels can be in the form of numbers, text (strings), dates, and many more, although labels are commonly strings. In contrast to `numpy` arrays, a strong distinction is made between integer positions and labels. This distinction is conceptually analogous to a distinction between student seat positions and student names. `Pandas` dataframes can also represent higher-than-two-dimensional data, although here we will focus on 2-dimensional data, which cover the majority of practical situations.

**Figure 5.1:** In contrast to `numpy` arrays, in `pandas` dataframes a distinction is made between the integer positions of rows and columns on the one hand, and row and column labels on the other hand. This distinction is analogous to the distinction made between student seat positions and student names.

[1] This is not entirely true, since `numpy` also provides "structured arrays" that can contain *field* names analogous to column names. Structured `numpy` arrays, however, are not recommended for serious work with labeled tabulated data.

### 5.5.1 Creating dataframes

One of the simplest ways to create a new dataframe is with the `DataFrame` function in the `pandas` module. The two main arguments are the data (split by row, provided as argument `data`) and the column names (provided as argument `columns`):

```
import pandas
df = pandas.DataFrame(data = [[18, 6, 60], [23, 6, 70], [16, 5.9, 65], [31, 5.5, 80]],
                      columns = ["temperature", "pH", "humidity"])
```

Observe that we specified 4 rows as a list of 4 sublists (one sublist per row), with each sublist containing 3 elements corresponding to the columns "`temperature`", "`pH`" and "`humidity`". To quickly inspect the contents of this new dataframe, we can use our familiar `print` function:

```
print(df)
```

```
   temperature   pH  humidity
0           18  6.0        60
1           23  6.0        70
2           16  5.9        65
3           31  5.5        80
```

By default, the rows of `df` are labeled using unique integers in ascending order (shown above on the far left). However, we can also label rows using custom names when creating the dataframe, by providing a list of strings as argument `index`. The following modified code creates a dataframe with string-valued row labels:

```
df = pandas.DataFrame(data = [[18, 6, 60], [23, 6, 70], [16, 5.9, 80], [31, 5.5, 40]],
                      columns = ["temperature", "pH", "humidity"],
                      index = ["agricultural", "forest", "marsh", "savanna"])
```

Let's inspect our new dataframe, now including row names:

```
print(df)
```

```
             temperature   pH  humidity
agricultural           18  6.0        60
forest                 23  6.0        70
marsh                  16  5.9        80
savanna                31  5.5        40
```

As we will learn later on, we can henceforth refer to individual rows by their label, rather than their integer position. It is also common to add columns to an existing dataframe after it was created, for example as part of downstream calculations. Suppose, for example, that we wanted to add another column called "`precipitation`" to the dataframe `df`, with the new column's elements specified by a numeric list like the following:

```
[98, 200, 578, 180]
```

To add such a new column, we can write:

```
df["precipitation"] = [98, 200, 578, 180]
```

Instead of a list on the right hand side, we could have also specified the column's values as a `numpy` array. Let's inspect our extended dataframe:

```
print(df)
```

```
              temperature   pH  humidity  precipitation
agricultural           18  6.0        60             98
forest                 23  6.0        70            200
marsh                  16  5.9        80            578
savanna                31  5.5        40            180
```

Observe that now `df` includes the additional column `precipitation` with the specified values.

★ **Exercise 98:** Write a `python` code that creates the following dataframe, listing molar nutrient ratios in the foliage of different forest types [74]:

```
            C_to_N  C_to_P
overall       43.6  1334.0
broadleaf     35.1   922.3
coniferous    59.5  1231.8
tropical      35.5  2456.9
```

Then, amend your code to add another column called "`N_to_P`", containing the following numerical values: 27.8, 28.2, 21.7, 43.4 (in this order).

### 5.5.2 Loading dataframes from a file

In almost all practical cases dataframes are loaded from a file, such as an HTSV file (a TSV file with a header containing column names) or a Microsoft Excel file. Consider, for example, the following dataset of monthly primary productivity measurements by the Hawaii Ocean Time Series program [32, 33]:

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`HOTS_monthly_pp.htsv`".

Let's have a look at the first few lines.

```
# Time is measured in fractional years.
# Original data downloaded on June 6, 2021, from:
#   https://hahana.soest.hawaii.edu/hot/hot-dogs/ppextraction.html
#
time    chlA    phaeopigments   light12
1989.5  0.332748    0.263504    3.27627
```

```
1989.58 0.346524    0.264326    3.65575
1989.67 0.308802    0.372161    5.02991
```

Notice that the first non-comment line is a header row containing column names. To load the above file as a dataframe variable, we can use the `pandas` function `read_csv`:

```
ppdata = pandas.read_csv("data/HOTS_monthly_pp.htsv", delimiter="\t", comment="#")
```

Let's see how many rows and columns our new dataframe has, using the familiar `shape` method:

```
print(ppdata.shape)
```

```
(367, 4)
```

Hence, we see that the dataframe `ppdata` has 367 rows and 4 columns. Let's inspect its contents using the `print` function, which for larger dataframes displays a reasonably sized portion:

```
print(ppdata)
```

```
        time      chlA  phaeopigments  light12
0    1989.50  0.332748       0.263504  3.27627
1    1989.58  0.346524       0.264326  3.65575
2    1989.67  0.308802       0.372161  5.02991
3    1989.75  0.197382       0.289890  1.85003
4    1989.83  0.158318       0.211617  1.16877
..       ...       ...            ...      ...
362  2019.63  0.190008       0.213719  1.44859
363  2019.71  0.217403       0.310110  1.39022
364  2019.80  0.240528       0.442706  1.33955
365  2019.88  0.261264       0.465853  2.18828
366  2019.96  0.282000       0.489000  3.03700

[367 rows x 4 columns]
```

In contrast to `numpy` arrays loaded using the `genfromtxt` function, our dataframe includes the column names found in the header row as labels. To obtain a list of all column labels in the dataframe we can write:

```
column_labels = list(ppdata.columns)
print(column_labels)
```

```
['time', 'chlA', 'phaeopigments', 'light12']
```

Note that `ppdata.columns` returns an "index object", which *contains* information about column labels, rather than a mere list of column labels; this object is then converted to a list of column names using the `list` function.

**Loading row labels:** As we have learned earlier, dataframes can also have row labels. These can either be defined in retrospect (as we saw in Section 5.5.1) or loaded directly from an input file. For example, consider the following HTSV file containing abundance estimates for 6 cetacean species based on 3 separate surveys [75]:

http://www.loucalab.com/archive/BioDataAnalysisPython
→ file "cetacean_abundances_Branch2001.htsv".

Let's look at the file's contents:

```
species survey1 survey2 survey3
Blue whale  440 550 1100
Fin whale   2100    2100    5500
Sperm whale 5400    10000   8300
Humpback whale  7100    9200    9300
Killer whale    91000   27000   25000
Southern bottlenose whale   NA  72000   54000
```

To load this HTSV file as a dataframe, with species names serving as row labels, we use the `pandas.read_csv` function with the argument `index_col`, as follows:

```
cetdata = pandas.read_csv("data/cetacean_abundances_Branch2001.htsv",
                          delimiter="\t", comment="#", index_col="species")
```

Let's confirm that the dataframe loaded correctly, with species names used as row labels:

```
print(cetdata)
```

```
                          survey1  survey2  survey3
species
Blue whale                  440.0      550     1100
Fin whale                  2100.0     2100     5500
Sperm whale                5400.0    10000     8300
Humpback whale             7100.0     9200     9300
Killer whale              91000.0    27000    25000
Southern bottlenose whale     NaN    72000    54000
```

Observe that `cetdata` does not include species as a separate column, since that column has been used for defining row labels. To extract all row labels of a dataframe as a list, we can use the `index` method, as follows:

```
row_labels = list(cetdata.index)
print(row_labels)
```

```
['Blue whale', 'Fin whale', 'Sperm whale', 'Humpback whale', 'Killer whale', 'Southern
    bottlenose whale']
```

In section 5.5.3 below, we will learn how to use column and row labels to access specific parts of a dataframe.

★ **Exercise 99:** Consider the following HTSV file, listing daily reported COVID-19 cases and deaths in the US:

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`WHO_COVID19_US_daily_counts.htsv`".

Load this file as a `pandas` dataframe, while setting the row names to the contents of the column "`date`". Print out the number of loaded rows and columns. Also print out the column labels.

### 5.5.3 Dataframe indexing

Accessing portions of dataframes, such as specific rows, columns or even individual elements, has some similarities but also important differences to `numpy` arrays. Essentially the dataframe indexing paradigm promotes a shift away from integer position-based indexing and towards a more semantics-based indexing. As before, in the following we focus on 2-dimensional dataframes, i.e., shaped like a regular spreadsheet. As examples, we consider the dataframe listing primary productivities, `ppdata`, and the dataframe listing cetacean abundances, `cetdata`, both discussed in the previous section.

**Positional indexing:** One of the simplest ways to access specific rows and/or columns in a dataframe is the positional indexing method `iloc[]`, which is analogous to the `[]` brackets in `numpy` arrays (Section 4.15.3). Indeed, we can provide two arguments to `iloc[]`, the first one specifying the integer position(s) of the desired row(s) and the second one specifying the integer position(s) of the desired column(s). We can use colons (`:`) to denote the full range of available rows or columns, we can provide lists of integers to specify arbitrary subsets of rows or columns, and we can use negative integers to denote positions counted backwards from the end. For example, to extract the first column from `ppdata`, we can write:

```
first_column = ppdata.iloc[:,0]
```

Similarly, to extract the last row we can write:

```
last_row = ppdata.iloc[-1,:]
```

The variables `first_column` and `last_row` will each be a `pandas` "series object", which is `pandas`' standard way of representing one-dimensional data. We can confirm this using the `type` function:

```
print(type(first_column))
```

```
<class 'pandas.core.series.Series'>
```

```
print(type(last_row))
```

```
<class 'pandas.core.series.Series'>
```

Let's also check their contents:

```
print(first_column)
```

```
0      1989.50
1      1989.58
2      1989.67
```

```
3      1989.75
4      1989.83
        ...
362    2019.63
363    2019.71
364    2019.80
365    2019.88
366    2019.96
Name: time, Length: 367, dtype: float64
```

```
print(last_row)
```

```
time             2019.960
chlA                0.282
phaeopigments       0.489
light12             3.037
Name: 366, dtype: float64
```

Observe that the elements of `first_column` are labeled according to the row labels of the original dataframe. `Pandas` series generally behave very similarly to 1-dimensional `numpy` arrays: They can be included in mathematical expressions (as we will learn later), they can be used to replace existing columns, they can be used as input to `pyplot`'s plotting functions, we can access their elements via integer positions, and so on. If we nevertheless wish to convert a `pandas` series into a 1-dimensional `numpy` array, we can use the `to_numpy` method, for example:

```
first_column_np = ppdata.iloc[:,0].to_numpy()
```

Let's confirm that `first_column_np` is indeed a `numpy` array:

```
print(type(first_column_np))
```

```
<class 'numpy.ndarray'>
```

To extract multiple columns or rows from a dataframe, we can specify their positions as a list of integers, just as we would for `numpy` arrays. For example, the following code extracts the first and third column of `ppdata`:

```
two_columns = ppdata.iloc[:,[0,2]]
```

Note that in contrast to the single-column extraction earlier, `two_columns` will itself be a dataframe, that is, a dataframe with two columns and as many rows as there were in `ppdata`:

```
print(two_columns)
```

```
        time  phaeopigments
0    1989.50       0.263504
1    1989.58       0.264326
2    1989.67       0.372161
3    1989.75       0.289890
4    1989.83       0.211617
..       ...            ...
362  2019.63       0.213719
```

```
363  2019.71       0.310110
364  2019.80       0.442706
365  2019.88       0.465853
366  2019.96       0.489000

[367 rows x 2 columns]
```

We can also obtain `two_columns` as a `numpy` array using the `to_numpy` method:

```
two_columns = ppdata.iloc[:,[0,2]].to_numpy()
```

The method `iloc` can also be applied to `pandas` series objects, such as those obtained when extracting a single row or column from a dataframe. For example, the following code extracts the first 3 elements of the `first_column` series as a new (shorter) series and prints it to the screen:

```
sub_first_column = first_column.iloc[0:3]
print(sub_first_column)
```

```
0    1989.50
1    1989.58
2    1989.67
Name: time, dtype: float64
```

**Label-based indexing:** One advantage of dataframes over `numpy` arrays is that dataframes natively support row and column labeling and, consequently, label-based indexing. Dataframe row and column labels can be of a variety of datatypes, including but not limited to numbers, strings, dates and so on, although strings are the most common way of labeling columns. To specify a subset of rows and/or columns by their labels, we can use the function `loc` (note the name distinction between `iloc` and `loc`). Similarly to `iloc`, we can provide two arguments to specify the desired row(s) and the desired column(s), respectively. For example, the following code extracts the column "`chlA`" from `ppdata`:

```
chlA_column = ppdata.loc[:,"chlA"]
```

As with positional column extractions seen earlier, `chlA_column` will be a "series object". To instead obtain `chlA_column` as a `numpy` array we can use `to_numpy`, as follows:

```
chlA_column = ppdata.loc[:,"chlA"].to_numpy()
```

To extract multiple columns, for example "`chlA`" and "`light12`", we can simply provide their labels as a list, as follows:

```
two_columns = ppdata.loc[:,["chlA","light12"]]
```

Note that in contrast to the single-column extraction, `two_columns` will itself be a dataframe, that is, a dataframe whose two columns are "`chlA`" and "`light12`":

```
print(two_columns)
```

```
         chlA  light12
0    0.332748  3.27627
```

```
1    0.346524  3.65575
2    0.308802  5.02991
3    0.197382  1.85003
4    0.158318  1.16877
..        ...      ...
362  0.190008  1.44859
363  0.217403  1.39022
364  0.240528  1.33955
365  0.261264  2.18828
366  0.282000  3.03700

[367 rows x 2 columns]
```

We can also obtain `two_columns` as a `numpy` array if needed, using `to_numpy`:

```
two_columns = ppdata.loc[:,["chlA","light12"]].to_numpy()
```

We can use a similar approach to access specific rows in `ppdata`, with one difference: Since we did not explicitly specify any names for the rows of `ppdata`, they were by default integer-labeled in the order in which they were originally loaded from the file. For example, the following code extracts the row with label 0 (which is the first loaded row):

```
first_row = ppdata.loc[0,:]
```

while the next code extracts the rows with labels 0, 1, 2 (i.e., the first 3 loaded rows):

```
first_few_rows = ppdata.loc[[0,1,2],:]
```

Similarly to column extractions, `first_row` will be a `pandas` series and `first_few_rows` will be a dataframe, and we can convert both of them into `numpy` arrays if needed using the `to_numpy` method. Note that the integers passed to `loc` are strictly speaking not treated as positional specifiers, because a row's label (integer or not) need not necessarily correspond to its position in the dataframe. For example, suppose that we extract the rows with labels 0, 2, 4 and 6 as a new dataframe:

```
subdata = ppdata.loc[[0,2,4,6],:]
```

The rows of this new dataframe will retain their original labels, namely 0, 2, 4 and 6:

```
print(subdata)
```

```
      time      chlA  phaeopigments  light12
0  1989.50  0.332748       0.263504  3.27627
2  1989.67  0.308802       0.372161  5.02991
4  1989.83  0.158318       0.211617  1.16877
6  1990.00  0.176706       0.184435  1.24872
```

This demonstrates that row labels — even if integer-valued — need not at all reflect the row positions (the same also applies to columns). Hence, the following request to extract and print the row with label 1 from `subdata` will fail even though there are 4 rows available, because none of the available rows have label 1:

```
print(subdata.loc[1,:])
```

This behavior makes sense if one remembers that nearly anything (not just integers) can be used as a row or column label in dataframes. For example, we can label the rows of the `ppdata` dataframe based on the loaded sampling times (column name "`time`"), as follows:

```
ppdata.set_index("time", inplace=True)
```

The argument `inplace` tells `python` to modify `ppdata` in place, rather than returning a modified copy of it. Let's confirm that rows are now labeled based on `time`:

```
print(ppdata)
```

```
             chlA  phaeopigments  light12
time
1989.50  0.332748       0.263504  3.27627
1989.58  0.346524       0.264326  3.65575
1989.67  0.308802       0.372161  5.02991
1989.75  0.197382       0.289890  1.85003
1989.83  0.158318       0.211617  1.16877
...           ...            ...      ...
2019.63  0.190008       0.213719  1.44859
2019.71  0.217403       0.310110  1.39022
2019.80  0.240528       0.442706  1.33955
2019.88  0.261264       0.465853  2.18828
2019.96  0.282000       0.489000  3.03700

[367 rows x 3 columns]
```

Observe that `ppdata` now has one less column than before, because the column `time` was used to define the row labels. Thus, rows can hereafter be specified using sampling times, for example:

```
print(ppdata.loc[[1991.5,1993,2000.4],:])
```

```
            chlA  phaeopigments   light12
time
1991.5  0.210075       0.195771  1.812540
1993.0  0.223125       0.289635  0.915043
2000.4  0.168876       0.210755  1.906730
```

If we want to ensure that rows are properly sorted according to their labels (e.g., in increasing times, in this example), we can use the dataframe method `sort_index`:

```
ppdata.sort_index(axis=0, inplace=True)
```

As another example, consider the previously discussed dataframe `cetdata`, listing cetacean abundances. Since we explicitly used the species names listed in the input file as row labels, we can specify rows using species names. For example, to extract and print the rows corresponding to species "`Blue whale`" and "`Killer whale`", we can write:

```
subdata = cetdata.loc[["Blue whale", "Killer whale"],:]
print(subdata)
```

```
              survey1  survey2  survey3
species
```

```
Blue whale      440.0      550     1100
Killer whale  91000.0    27000    25000
```

For more details and examples using the `loc` function see the official documentation here.

**Indexing shortcuts:** It is worth mentioning that dataframes provide various syntactical shortcuts to label-based indexing. Notably, if we only want to specify a subset of columns (i.e., keeping all rows), we can omit the `loc` keyword and first colon, and use solely square brackets. For example, the following code extracts the column "`chlA`" from `ppdata` as a `pandas` series:

```
chlA_column = ppdata["chlA"]
```

while the following code extracts the columns "`chlA`" and "`light12`" as a dataframe:

```
two_columns = ppdata[["chlA","light12"]]
```

Such shortcuts, which also exist for row indexing, are widely used in practice, however it can be easy to get confused about which shortcut refers to row and column indexing. Further, if column labels happen to be integers, it may not be obvious to novices whether `[]` uses positional or label-based indexing. If in doubt, or simply for the sake of clarity, you may prefer to stick to the full `iloc` and `loc` syntaxes discussed earlier.

**Boolean indexing:** Similarly to `numpy` arrays, we can also specify subsets of rows or columns in dataframes using boolean indexing, i.e., by passing to `iloc` or `loc` a boolean array of the same length as the dimension being indexed. This boolean array may in turn be constructed based on some logical condition. Boolean indexing with dataframes is conceptually similar to `numpy` arrays, although more pitfalls exist because of the more complex label-based indexing of dataframes. Here we provide a rather narrow and painless introduction through examples, keeping in mind that the approaches presented here are optimized for ease of understanding rather than for code elegance. Consider our previously discussed dataframe `cetdata`, and suppose that we wanted to extract the first and third row using boolean indexing. We thus need to specify a list or array of booleans with as many elements as there are rows in `cetdata`, listing the value `True` only in the first and third position:

```
B = [True, False, True, False, False, False]
subdata = cetdata.iloc[B, :]
```

Hence, `subdata` will be a dataframe with two rows, corresponding to the `True` elements in `B`:

```
print(subdata)
```

```
             survey1  survey2  survey3
species
Blue whale     440.0      550     1100
Sperm whale   5400.0    10000     8300
```

In practice, boolean arrays are rarely typed in by hand, and are instead created automatically as the result of some logical condition. For example, the following code uses boolean indexing to extract the rows for only those cetaceans with abundances between 5000 and 20000 in the third survey:

```
# create the proper boolean array
B = ((cetdata["survey3"]>=5000) & (cetdata["survey3"]<=20000)).to_numpy()
```

```
# use the boolean array for the first axis in iloc, to extract the proper rows
subdata = cetdata.iloc[B,:]
```

Note that the expression `(cetdata["survey3"]>=5000) & (cetdata["survey3"]<=20000)` yields a boolean series (`pandas`' analogue to `numpy` boolean arrays), which is then converted to a `numpy` boolean array using the `to_numpy` method, which is then passed as an argument to `iloc` to specify which rows to keep. Let's confirm our expectation:

```
print(subdata)
```

```
                survey1  survey2  survey3
species
Fin whale        2100.0     2100     5500
Sperm whale      5400.0    10000     8300
Humpback whale   7100.0     9200     9300
```

In the above example, we could have also skipped the conversion to a `numpy` array and directly used the boolean series for indexing; in this case we would need to use `loc` instead of `iloc`, because `iloc` intentionally does not accept boolean `pandas` series.

```
B = ((cetdata["survey3"]>=5000) & (cetdata["survey3"]<=20000))
subdata = cetdata.loc[B,:]
```

Using a boolean series with `loc` as above comes with an important pitfall: `loc` matches booleans to rows according to labels, not positions, which can cause an error or yield wrong results if the boolean series follows a different labeling scheme (e.g., if it was created from a logical condition involving a distinct dataframe). Of course, we could have also achieved the same outcome without boolean indexing, for example using list comprehension and positional indexing:

```
# extract cetacean abundances in 3rd survey as numpy array
abundances3 = cetdata["survey3"].to_numpy()

# create a list of positional indices, specifying the rows to keep
keep_rows = [a for a in range(cetdata.shape[0]) if (abundances3[a]>=10000)]

# filter the dataframe by passing the desired positional indices to iloc
subdata = cetdata.iloc[keep_rows,:]
print(subdata)
```

```
                          survey1  survey2  survey3
species
Killer whale              91000.0    27000    25000
Southern bottlenose whale     NaN    72000    54000
```

But as with `numpy` array boolean indexing, so `pandas` dataframe boolean indexing can help us write more concise and intuitive code. Boolean indexing with `iloc` and `loc` also works for subsetting `pandas` series, such as those obtained when extracting a single row or column; in that case, the extracted portion will again be in the form of a `pandas` series. For example, to extract and print the abundances from the third survey that were between 5000 and 20000, we can write:

```
# extract the survey3 column as a series
survey3_data = cetdata["survey3"]
```

```
# extract a subset from survey3_data using boolean indexing
survey3_subdata = survey3_data.loc[B]
```

Note that the above can also be achieved by combining label-based and boolean indexing into a single `loc` call:

```
survey3_subdata = cetdata.loc[B,"survey3"]
```

Let's inspect the result:

```
print(survey3_subdata)
```

```
species
Fin whale          5500
Sperm whale        8300
Humpback whale     9300
Name: survey3, dtype: int64
```

★ **Exercise 100:** Predict the output of the following `python` code, without actually running it. You may use pen and paper for aid.

```
df = pandas.DataFrame(data = [[18, 6, 60], [23, 6, 70], [16, 5.9, 65], [31, 5.5, 80]],
                      columns = ["temperature", "pH", "humidity"])
print(df,"\n")

subdf = df.loc[1:,["pH", "humidity"]]
print(subdf,"\n")

subdf2 = subdf.loc[[1,2], :]
print(subdf2,"\n")

subdf3 = subdf.iloc[[1,2], 1]
print(subdf3)
```

★ **Exercise 101:** Consider the following HTSV file containing abundance estimates for 6 cetacean species based on 3 separate surveys [75], discussed in Section 5.5:

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`cetacean_abundances_Branch2001.htsv`".

Load this dataset as a `pandas` dataframe. Use dataframe label-based indexing to extract the subset corresponding to Humpback whales and Killer whales, and only listing abundance estimates based on the first two surveys. Print out the resulting smaller dataframe.

★ **Exercise 102:** Consider the following HTSV file, listing the coordinates of a Galapagos albatross tracked during the summer 2008, while repeatedly traveling back and forth between the Galapagos islands and the Peruvian coast [44]:

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`albatross_foraging_routes_Dodge2013.htsv`".

Note that the first column lists dates and times (in format YYYY.MM.DD hh:mm), the second

column lists latitudes (in degrees) and the third column lists longitudes (in degrees). Load the above file as a `pandas` dataframe. Then, add two new columns named "`date`" and "`time`", listing the dates (in format YYYY.MM.DD) and times (in format hh:mm), respectively. Print out the first 10 rows of the final dataframe.

★ **Exercise 103:** Consider the following HTSV file, listing daily reported COVID-19 cases and deaths in the US:

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`WHO_COVID19_US_daily_counts.htsv`".

Note that the first column lists dates, the second column lists times measured in years (including fractions), the third column lists daily cases, and the fourth column lists daily deaths. Load this file as a `pandas` dataframe, and generate a scatterplot that shows daily cases (horizontal axis) versus daily deaths (vertical axis), across all days of the years 2020–2022 (one point per day). Color points differently for each of the three years. Ensure that scatterpoints are reasonably sized and opaque so that each year's data are visible.

### 5.5.4 Math with dataframes

When it comes to mathematical calculations, dataframes can typically be treated like `numpy` arrays. Indeed, dataframes can be written into arithmetic expressions just like `numpy` arrays, in which case these expressions are evaluated in an element-wise fashion. For example, consider the cetacean abundance estimates from the previous example, stored in the `cetdata` dataframe. Suppose that we wanted to convert all abundances from individuals to thousands (i.e., divide all abundances by 1000). This could be achieved as follows:

```
cetdata_new = cetdata / 1000
```

The new variable `cetdata_new` represents a dataframe consisting of the elements of `cetdata` divided by 1000:

```
print(cetdata_new)
```

```
                          survey1  survey2  survey3
species
Blue whale                   0.44     0.55      1.1
Fin whale                    2.10     2.10      5.5
Sperm whale                  5.40    10.00      8.3
Humpback whale               7.10     9.20      9.3
Killer whale                91.00    27.00     25.0
Southern bottlenose whale     NaN    72.00     54.0
```

Similarly, many mathematical functions from the `numpy` package, such as `exp`, `log`, `sin`, `cos` and so on, can be applied to entire dataframes. For example, suppose that we want to compute the natural logarithm of all cetacean abundances (a common modification done in ecological analyses [76, 77]). We can achieve this as follows:

```
log_cetdata = numpy.log(cetdata)
```

The new dataframe `log_cetdata` will thus contain the log-transformed abundances:

```
print(log_cetdata)
```

```
                           survey1    survey2    survey3
species
Blue whale                6.086775   6.309918   7.003065
Fin whale                 7.649693   7.649693   8.612503
Sperm whale               8.594154   9.210340   9.024011
Humpback whale            8.867850   9.126959   9.137770
Killer whale             11.418615  10.203592  10.126631
Southern bottlenose whale       NaN  11.184421  10.896739
```

One caveat is that dataframe-level mathematical calculations such as the above are only possible if the calculation is meaningful for all columns; for example, dataframes containing string data will yield an error if included in a mathematical expression. To restrict mathematical calculations to a subset of columns or rows, we can use indexing to specify that subset. For example, to only log-transform the abundances from the third survey and store the results in place (i.e., replacing the original data), we can write:

```
cetdata["survey3"] = numpy.log(cetdata["survey3"])
```

Let's inspect the modified dataframe to confirm our expectation:

```
print(cetdata)
```

```
                         survey1  survey2    survey3
species
Blue whale                 440.0      550   7.003065
Fin whale                 2100.0     2100   8.612503
Sperm whale               5400.0    10000   9.024011
Humpback whale            7100.0     9200   9.137770
Killer whale             91000.0    27000  10.126631
Southern bottlenose whale    NaN    72000  10.896739
```

Indeed, the contents of columns `survey1` and `survey2` remained unchanged, but the values in column `survey3` have been log-transformed. If we instead wanted to store the log-transformed abundances as a new column (say, "`survey3_log`"), we would simply specify a different column name on the left-hand-side of the assignment:

```
cetdata["survey3_log"] = numpy.log(cetdata["survey3"])
print(cetdata)
```

```
                         survey1  survey2    survey3  survey3_log
species
Blue whale                 440.0      550   7.003065     1.946348
Fin whale                 2100.0     2100   8.612503     2.153215
Sperm whale               5400.0    10000   9.024011     2.199889
Humpback whale            7100.0     9200   9.137770     2.212416
Killer whale             91000.0    27000  10.126631     2.315169
Southern bottlenose whale    NaN    72000  10.896739     2.388464
```

It should be noted that `numpy` functions for computing summary statistics, such as `sum`, `mean`, `median` and `std`, work somewhat differently when applied to a dataframe than what we are used to with `numpy` arrays. Specifically, if the `axis` argument is not specified for these functions, they apply separately to each dataframe column, yielding a series object with as many elements as

there were columns in the input dataframe. For example, `mean` computes the mean along each column:

```
print(numpy.mean(ppdata))
```

```
chlA             0.209393
phaeopigments    0.323930
light12          1.982823
dtype: float64
```

Similarly, `std` computes the standard deviation along each column:

```
print(numpy.std(ppdata))
```

```
chlA             0.046690
phaeopigments    0.127129
light12          0.999133
dtype: float64
```

If in doubt, or simply for the sake of clarity, is recommended to explicitly specify the `axis` argument even if unnecessary. For example, to compute the mean along each column:

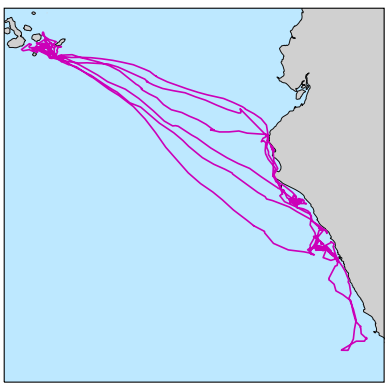
```
print(numpy.mean(ppdata, axis=0))
```

```
chlA             0.209393
phaeopigments    0.323930
light12          1.982823
dtype: float64
```

★ **Exercise 104:** Consider the following HTSV file, listing the coordinates of a Galapagos albatross tracked during the summer 2008, while repeatedly traveling back and forth between the Galapagos islands and the Peruvian coast [44]:

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`albatross_foraging_routes_Dodge2013.htsv`".

Note that the first column lists dates and times (in format YYYY.MM.DD hh:mm), the second column lists latitudes (in degrees) and the third column lists longitudes (in degrees). The map below gives an overview of the recorded routes.



Perform the following tasks:

1. Load the above file as a `pandas` dataframe, and compute for each recorded point the great-circle distance (km) to the Galapagos islands (use coordinates lat -0.5, lon -90.5 as a representative point for the Galapagos). Store those distances as a new column in the dataframe, named "`Gdistance`". Print out the first 10 rows of your extended dataframe.

   *Hint: To compute the great-circle distance between any two points on Earth's surface, you can use the Vincenty formula used in Exercises 9, 61 and 85.*

2. Compute the number of days on which the albatross was recorded within a radius of 300 km from the above representative point. How does this number compare with the total number of days covered by the dataset?

   *Hint: Keep in mind that each date may be represented multiple times in the dataset, but should only be counted at most once. Recall that this can easily be achieved using* `set()`*.*

★ **Exercise 105:** Consider the following HTSV file listing moulting patterns and shell quality of American lobsters captured in Atlantic Canada [78]:

> `http://www.loucalab.com/archive/BioDataAnalysisPython`
> → file "`captured_lobsters_Atlantic_Koepper2022.htsv`".

Perform the following tasks:

1. Load this file as a dataframe, and compute and print out the average lobster size (column "`Size`", in mm) and average water depth (column "`Water_depth_m`", in m).

2. Repeat the previous task separately for each year, i.e., compute and print out the average lobster size and average water depth separately for each of the years represented in the dataset.

   *Hint: Years are listed in column "*`Year`*". You can use the function* `set` *to automatically determine the unique years in the dataset, and boolean indexing to extract the subset of records for any given year.*

3. Create a histogram showing the distribution of lobster sizes in the entire dataset (regardless of year).

### 5.5.5 Numpy arrays or pandas dataframes?

Both `numpy` arrays and `pandas` dataframes are widely used in scientific computing for storing and working with multidimensional data, albeit with rather different foci. `Numpy` arrays should generally be preferred for homogeneous data (e.g., a 3-dimensional matrix listing the distribution of nitrogen across the 3 spatial dimensions in the ocean, or a 2-dimensional matrix listing pairwise occurrence correlations between a large number of species), as well as intensive numerical calculations (e.g., solving large linear systems of equations, simulating the diffusive dispersal of thousands of animals, integrating over a large number of simulations of a population model). For very large numerical datasets, `numpy` arrays tend to be more computationally efficient than

`pandas` dataframes. `Pandas` should generally be preferred for labeled ("spreadsheet") data, such as multiple morphological traits measured across multiple species, especially when the data are heterogeneous (e.g., combinations of text and numbers) and not too massive. Dataframes also include tools for conveniently generating basic but decent plots (see here for examples). All of this said, keep in mind that `pandas` dataframes make heavy use of `numpy` arrays, so being familiar with both is essential for serious data analysis. Further, most common `python` functions for working with `numpy` arrays (e.g., for statistical analyses or plotting) can be used in a similar manner with `pandas` dataframes and series with very few or zero adjustments.

★★ **Exercise 106:** Consider the following dataset of intestinal bacterial community compositions determined in multiple mice over the course of several days [79]:

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`mouse_microbiomes_OTU_table_Marino2014.htsv`".

Note that each row in the above HTSV file corresponds to a separate intestinal sample (i.e., a unique combination of mouse ID and sampling day), the first column lists sample IDs, the second column lists mouse IDs, the third column lists the sampling day, and the remaining columns correspond to individual bacterial operational taxonomic units (OTUs, which are analogous to bacterial species [80, 81]). Each entry in one of the OTU columns specifies the number of sequencing reads matched to a specific OTU in a specific sample, and is proportional to the OTU's abundance in that sample. Perform the following tasks:

1. Load the above HTSV file as a `pandas` dataframe. Compute the proportion of each OTU in each sample, by dividing the number of reads matched to the OTU in that sample by the total number of reads (summed over all OTUs) in that sample. Thus, for each sample the sum of OTU proportions should be equal to 1. Then compute the average proportion of each OTU across all samples, and print out the names of the 5 most abundant OTUs (i.e., with the 5 highest average proportions) in order of descending average proportion.

   *Hint: Use* ***`numpy.argsort`*** *to determine the order of OTUs in terms of their average proportions, as discussed in Section 4.15.6.*

2. Plot the proportions of the most abundant OTU (i.e., with the highest average proportion) as curves over time, separately for each mouse. Hence, you should create a single figure, with as many different curves as there were mice, each showing the proportions of the same OTU over the course of the experiment. Color each curve differently, and include a legend in the plot. Do the OTU's proportions follow a similar trend over time in all mice?

## 5.6 Computations with randomness

While computers are well-appreciated for their ability to perform precise mathematical calculations, there are many situations where we actually want to deploy randomness (aka. stochasticity) in a computation. For example, randomness is often used for testing statistical hypotheses and estimating confidence intervals, by generating hypothetical random data from a probabilistic null model [82–84]. Randomness is also used for simulating mathematical stochastic models [85–87], such as those describing the spread of an infectious disease or models describing random genetic changes in a population. Randomness is also needed for various methods of statistical inference, e.g., Markov Chain Monte Carlo methods in Bayesian statistics [88–90], and certain numerical optimization algorithms (e.g., simulated annealing). Lastly, randomness can also help in experimental design, such as setting up randomized clinical trials.


At the heart of randomized computations are random number generators (RNGs), i.e., algorithms that generate random numbers upon demand. Indeed, virtually any randomization task encountered in practice (including, for example, randomly assigning subjects to experimental treatments) can be reduced to the problem of generating random numbers. All respectable programming languages thus include one or more RNGs. In `python`, random number generation functionalities are provided in the `random` module of the `numpy` package, although the `random` package also provides basic functionalities. In the following we will focus on `numpy`'s RNG. To generate random numbers (and other derived types of randomness), we need to explicitly create an RNG object, as follows:

```
import numpy
rng = numpy.random.default_rng()
```

In `numpy`'s language, `rng` is called a "Generator" object, although for simplicity we will refer to it as our RNG. In the following passages, we will learn how to use this object to achieve randomness.

### 5.6.1 Continuously distributed random numbers

One of the simplest functions for random number generation is the RNG's `uniform` method, which returns one or more random numbers drawn uniformly from within a given interval $[a, b]$. Let's try it out for $a = 0$ and $b = 100$:

```
# generate and print a single random number
print(rng.uniform(0,100))

# generate and print another random number
print(rng.uniform(0,100))

# generate and print another random number
print(rng.uniform(0,100))

# generate and print another random number
print(rng.uniform(0,100))
```

```
53.33324905659117
82.13639104915376
62.90745238390686
55.548264430027025
```

Observe that each time we call `uniform` we get a different value, with no obvious relationship to the previous values. We can generate a lot of these numbers at once using the optional `size` argument:

```
N = 10000
random_numbers = rng.uniform(0,100,size=N)
```

Let's check how unique they are:

```
Nunique = len(set(random_numbers))
print("{0} out of {1} numbers are unique".format(Nunique,N))
```

```
10000 out of 10000 numbers are unique
```

We can also scatterplot these numbers to further examine their randomness:

```
from matplotlib import pyplot
pyplot.figure(figsize=(10, 3))
pyplot.scatter(range(N), random_numbers, s=3, alpha=0.7)
pyplot.xlabel("index n")
pyplot.ylabel("random number[n]")
pyplot.xlim([0,N-1])
pyplot.ylim([0,100])
pyplot.savefig("py_output/random_numbers_uniform_scatter.pdf",
               bbox_inches='tight')
pyplot.close()
```

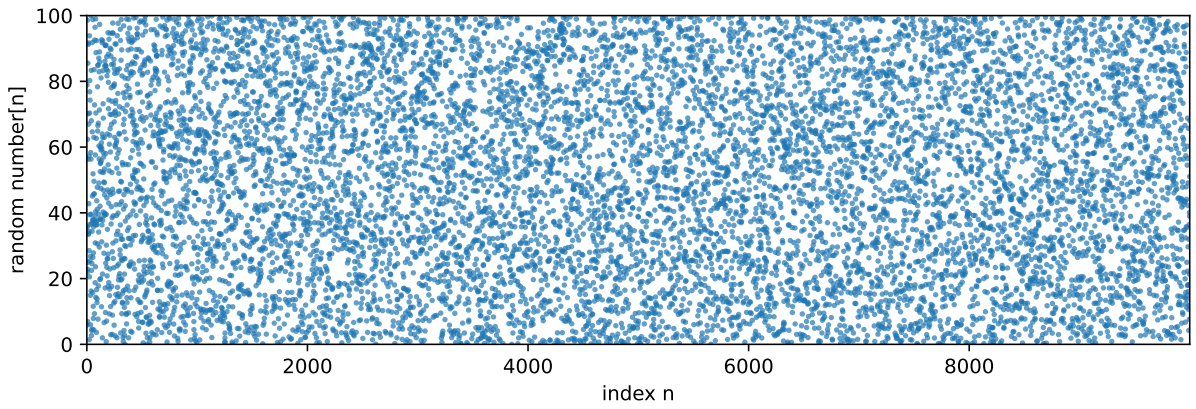
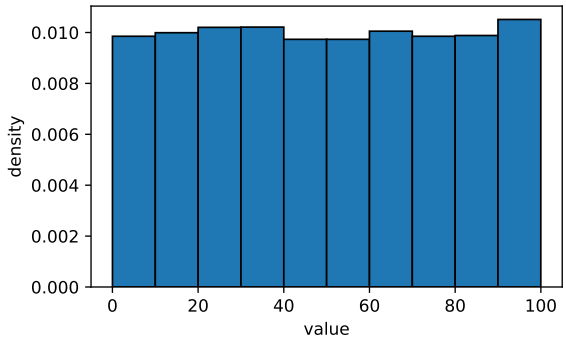



**Figure 5.2:** Random numbers generated with `numpy.random.uniform`.

Based on the above scatterplot, it seems that there is no obvious trend, periodicity or other simple pattern that would make us doubt the randomness of these numbers. Lastly, let's look at the distribution of the generated numbers using a histogram:

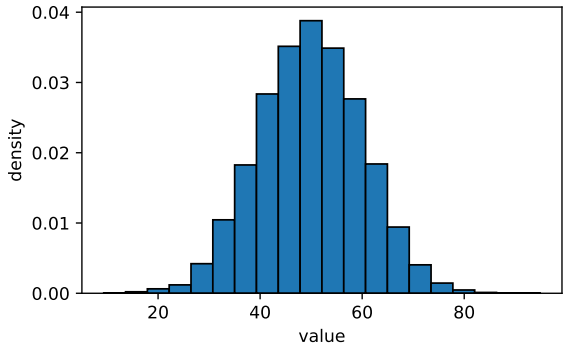
```
pyplot.figure(figsize=(5, 3))
pyplot.hist(random_numbers, density=True, edgecolor="black")
pyplot.xlabel("value")
pyplot.ylabel("density")
pyplot.savefig("py_output/random_numbers_uniform_hist.pdf",
               bbox_inches='tight')
pyplot.close()
```




We see that these numbers are quite uniformly (i.e., evenly) distributed in the interval $[a, b]$, as expected.

Another important probability distribution in the sciences is the normal (aka. Gaussian or "bell curve") distribution. To generate normally distributed random numbers we can use the function `normal` in the `numpy` module `random`, which requires that we specify the mean and standard deviation, and optionally how many number we want. As an example, let's generate several normally distributed random numbers with mean 50 and standard deviation 10, and create a histogram of their distribution:

```
N = 10000
random_numbers = rng.normal(50,10,size=N)
pyplot.figure(figsize=(5, 3))
pyplot.hist(random_numbers, bins=20, density=True, edgecolor="black")
pyplot.xlabel("value")
pyplot.ylabel("density")
pyplot.savefig("py_output/random_numbers_normal_hist.pdf", bbox_inches='tight')
pyplot.close()
```




Observe that in contrast to the uniformly drawn numbers, the normally drawn numbers are concentrated around the mean and values far from the mean are rare. The normal distribution is widely used in the sciences, partly because it approximately resembles the distribution of many physical/biological quantities and partly due to its useful mathematical properties [91–93]. Many more functions exist in the `random` package for drawing random numbers from other continuous probability distributions, such as the gamma distribution (`gamma`) and the exponential distribution (`exponential`). For a complete list see the official documentation.

★ **Exercise 107:** Write an `if` statement that, when executed, will either print out the message "`gene active`" (at probability 0.7) or the alternative message "`gene inactive`" (at probability 0.3). As a source of randomness, use only the function `numpy.random.uniform`.

### 5.6.2 Example: Testing hypothesis on bird nest associations

Let's see how random number generation can be used in practice. Consider the following dataset of nesting locations for two bird species in a rectangular forest patch ($10 \times 5$ km):

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`bird_nesting_sites.htsv`".

Let's inspect the first few lines in this HTSV file:

```
# Nest locations (X & Y) for two bird species
# in a rectangular (10 x 5 km) forest patch
species X   Y
species1    2.712   0.296
species1    1.376   3.998
species1    6.891   0.347
```

We would like to examine whether there is any association between the nesting locations of the two species, specifically, whether the first species preferentially nests close to (or away from) the second species. We begin by loading the data as a `pandas` dataframe:

```
import pandas
import numpy

data = pandas.read_csv("data/bird_nesting_sites.htsv", delimiter="\t", comment="#")
N1 = numpy.sum(data["species"]=="species1")
N2 = numpy.sum(data["species"]=="species2")

print("Found %d nests for species1, %d nests for species2"%(N1,N2))
```

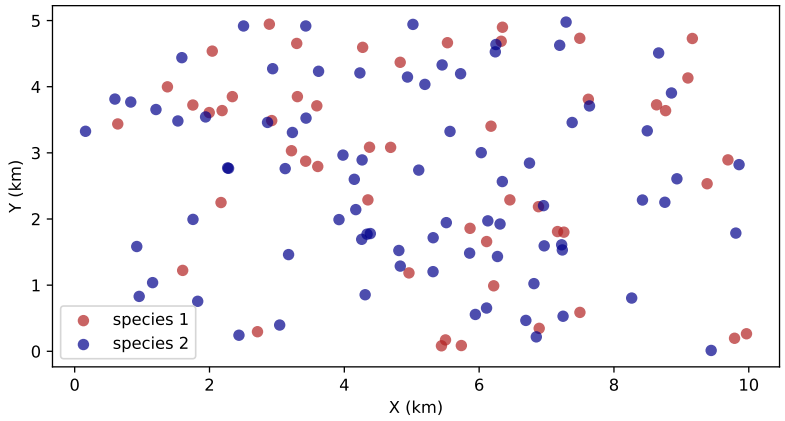
```
Found 50 nests for species1, 80 nests for species2
```

We then extract the nest coordinates for the two species as two separate 2-dimensional `numpy` arrays for convenience:

```
coordinates1 = data.loc[data["species"]=="species1", ["X","Y"]].to_numpy()
coordinates2 = data.loc[data["species"]=="species2", ["X","Y"]].to_numpy()
```

Let's visualize the loaded nest locations as a scatterplot, coloring points by species:

```
from matplotlib import pyplot
W = 10
H = 5
pyplot.figure(figsize=(4*W/H, 4))
pyplot.scatter(coordinates1[:,0], coordinates1[:,1],
               linewidth=1, color="firebrick", alpha=0.7, label="species 1")
pyplot.scatter(coordinates2[:,0], coordinates2[:,1],
               linewidth=1, color="darkblue", alpha=0.7, label="species 2")
pyplot.xlabel("X (km)")
pyplot.ylabel("Y (km)")
pyplot.legend()
pyplot.savefig("py_output/bird_nest_scatter.pdf", bbox_inches='tight')
pyplot.close()
```




To measure the potential association between the nesting sites of the two species, we consider the following quantity (aka. "test statistic"): $D$ shall denote the shortest distance of a species 1 nest to any of the species 2 nests, averaged over all species 1 nests. In other words, if $d_{ij}$ denotes the distance between species 1 nest $i$ and species 2 nest $j$, then $D$ is computed as:

$$D = \frac{1}{N_1} \sum_{i=1}^{N_1} \min_{j \in \{1,..,N_2\}} d_{ij}. \tag{5.1}$$

Let's define a custom function for computing this mean shortest distance, as we will need to perform this calculation more than once:

```
def compute_D(XY1, XY2):
    shortest = [numpy.min(numpy.sqrt((XY1[r,0]-XY2[:,0])**2 + (XY1[r,1]-XY2[:,1])**2))
                        for r in range(XY1.shape[0])]
    return numpy.mean(shortest)
```

Observe that the function takes two arguments, `XY1` (the nest coordinates for species 1) and `XY2` (the nest coordinates for species 2). Applying this function to our dataset yields:

```
D = compute_D(coordinates1,coordinates2)
print("D = %.3f km"%(D))
```

```
D = 0.323 km
```

We see that, on average, species 1 nests are only 0.323 km away from the nearest species 2 nest, which seems rather small considering the overall dimensions of the forest patch. However, what if this small $D$ is just coincidental, and not actually driven by a preference of species 1 to nest close to species 2? How *small* is *small enough* to reliably conclude that species 1 indeed preferentially nests near species 2? One of the most common ways to formally answer this question is known as "null model testing", i.e., computing the probability that such a low $D$ could also have been caused by pure chance. In our case, the null model corresponds to the scenario where species 1 chooses its nesting sites completely independently of species 2. However, this does not yet fully specify *how* species 1 chooses its nesting sites, so we need to make our null model more precise. In the following, we consider the null model in which species 1 chooses its nesting sites independently of species 2 and uniformly randomly within the forest. We thus need to calculate how probable it would be for $D$ to be as small as observed, if this null model was indeed correct.

Calculating this probability with analytical formulas is rather hard, but fortunately we can easily simulate hypothetical data from the null model and estimate $D$ from the hypothetical data. Specifically, we can re-assign random new coordinates (chosen uniformly within the forest

patch) to all species 1 nests, recompute $D$, and repeat this process a large number of times. If the $D$ from the null model data is very rarely as low as the $D$ from our real data, then we would have evidence that species 1 does prefer to nest near species 2. Let's start by writing a custom function that returns random coordinates for all species 1 nests according to our null model:

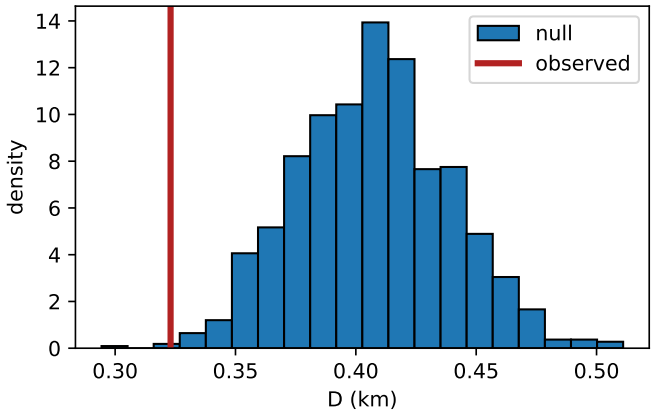
```
rng = numpy.random.default_rng()
def sim_null_model():
    XY1      = numpy.zeros([N1,2]) # preallocate space
    XY1[:,0] = rng.uniform(0,W,size=N1) # draw X-coordinates
    XY1[:,1] = rng.uniform(0,H,size=N1) # draw Y-coordinates
    return XY1
```

Nest, we use a list comprehension to simulate the null model 1000 times (using `sim_null_model`) and compute the corresponding $D$ values (using `compute_D`):

```
null_D = [compute_D(sim_null_model(), coordinates2) for n in range(1000)]
```

Observe that we are passing to `compute_D` the simulated coordinates for species 1 nests, and the true coordinates for species 2 nests. Let's have examine the null model's $D$ values using a histogram, and also include the $D$ observed in the data as a red vertical line for reference:

```
pyplot.figure(figsize=(5,3))
pyplot.hist(null_D, bins=20, edgecolor="black", density=True, label="null")
pyplot.axvline(D, color="firebrick", linewidth=3, label="observed")
pyplot.xlabel("D (km)")
pyplot.ylabel("density")
pyplot.legend()
pyplot.savefig("py_output/bird_nest_null_hist.pdf", bbox_inches='tight')
pyplot.close()
```




Observe that the null model's $D$ values are very rarely as small as (or smaller than) the observed $D$. In fact, we can estimate the probability of this happening by counting the fraction of simulations for which the null model's $D$ was at or below the observed $D$:

```
P = numpy.mean(null_D<=D)
print("P = %.4f"%(P))
```

```
P = 0.0020
```

This probability $P$ is called the "statistical significance" of the observed $D$ under the null model;

a smaller $P$ value means that the null model is less likely to generate the observed data [94, 95]. A $P$ value below the threshold 0.05 (as seen here) is conventionally considered strong evidence that the null model must be rejected. In our case, this suggests that species 1 does not nest independently of species 2, but indeed tends to nest near species 2.

Note that while our results suggest that species 1 prefers to nest near species 2, they do not say anything about the strength of this effect. In other words, we only know that the preference to nest near species 2 is strong enough to be statistically detectable with the available sample sizes, but we haven't determined how much it actually contributes to the overall nesting choices of species 1. Defining and assessing the strength of this contribution would involve a more sophisticated statistical or mechanistic model, which is beyond the scope of this book.

### 5.6.3 Random vs pseudorandom numbers

It is worth mentioning that the numbers generated above are actually not really random, because they are generated by a mathematically deterministic (thus predictable) algorithm (see e.g., exercise 108 below). Indeed, with careful investigation it is possible to detect regularities and relationships between these numbers that shouldn't exist if they were truly random and independent. They are thus more correctly referred to as *pseudorandom* numbers, although most people still refer to them as "random" for simplicity. Constructing algorithms for generating numbers close to true randomness is hard, and a topic of active research [96]. For the vast majority of scientific applications, however, pseudorandom numbers generated by respectable packages such as `numpy` are sufficiently close to true randomness that we rarely need to worry about this distinction. One useful property of the deterministic nature of pseudorandom numbers is that we can precisely replicate their sequence with every re-execution of our code, if needed. This is useful, for example, if we are testing and revising our code, and would like to exactly repeat the previous computation for comparison.

Ensuring exact replication of all pseudorandom numbers generated by an RNG is achieved initializing the RNG with a "seed", which can be any arbitrary integer:

```
rng = numpy.random.default_rng(12345)
```

All random numbers henceforth generated with the `rng` object are entirely determined by the provided seed (`12345` in this case). Let's generate a few of them:

```
print(rng.uniform(0,100))
print(rng.uniform(0,100))
print(rng.uniform(0,100))
```

```
22.733602246716966
31.675833970975287
```

```
79.73654573327342
```

Now re-create the `rng` object with the same seed as earlier:

```
rng = numpy.random.default_rng(12345)
```

and generate some more random numbers:

```
print(rng.uniform(0,100))
print(rng.uniform(0,100))
print(rng.uniform(0,100))
```

```
22.733602246716966
31.675833970975287
79.73654573327342
```

Observe that the sequence of numbers generated in the second round matches the ones from the first round. If we instead want our code to generate different random numbers each time it is executed, we need to use a different seed each time, ideally drawn from a source of true randomness. One simple source of "true" randomness that is commonly used to seed RNGs is the computer's internal clock time, although many modern systems employ more sophisticated methods. To seed `python`'s RNG using clock time or another system-provided source of randomness, we simply omit the `seed` argument when calling `default_rng` (as done in the previous section):

```
rng = numpy.random.default_rng()
```

In summary, for code development/testing purposes it is generally recommended to seed the RNG with a fixed integer, while for the finalized computations the RNG should be seeded randomly.

★ **Exercise 108:** In this exercise we will write and study our own simple pseudorandom number generator using the LCG algorithm. The purpose is to illustrate how a deterministic (i.e., in principle fully predictable) mathematical sequence can appear to yield random data by various statistical measures. The LCG algorithm is parameterized by three positive integers, the *modulus* $m$, the *multiplier* $a$, the increment $c$, and the initial value $X_0$ known as *seed*. For any current value $X_n$, the LCG algorithm defines the next value $X_{n+1}$ as follows:

$$X_{n+1} = (aX_n + c) \bmod m. \tag{5.2}$$

Here, "mod" denotes the modulo operator, i.e., the remainder of integer division, which in `python` is implemented using the `%` character. The above algorithm yields a sequence of seemingly random (but strictly speaking fully predictable) integers between 0 and $m-1$. In practice, it is more useful to generate non-integer random numbers within the interval [0,1], which can easily be achieved in retrospect by dividing the generated values $X_n$ by $m-1$, i.e., using $Y_n := X_n/(m-1)$ as pseudorandom numbers within [0,1]. While the optimal choice of the parameters $m$, $a$ and $c$ is a complicated matter [96, 97], for simplicity here we consider the values used by ZX81 (a home computer released back in 1981): $m = 2^{16}+1$, $a = 75$ and $c = 74$. The seed can be drawn from a "truly" random process, for example based on the computer's current clock time, but for this exercise we will just fix it at the arbitrary value $X_0 = 1234$. Perform the following tasks:

1. Use the LCG algorithm, with the parameters and seed provided above, to generate the first $10^4$ values of the corresponding pseudorandom number sequence, i.e., $X_1, X_2, ..., X_{10^4}$. Divide

each $X_n$ by $m-1$ to obtain a sequence $Y_1, .., Y_{10^4}$ within the interval [0,1]. Plot the resulting $Y_1, Y_2, ..$ over their index as a scatterplot, i.e., with the horizontal axis representing the indices 1,2,3,.. and the vertical axis showing the $Y_1, Y_2, Y_3, ..$ (similar to Fig. 5.2). Do you see any simple pattern, such as an overall trend or periodicity?

2. Create a histogram showing the distribution of the generated $Y_1, Y_2, ...$ over the interval [0,1]. Does the distribution seem uniform, or are there clear biases towards certain values?

### 5.6.4 Random shuffling

Apart from drawing random values from continuous probability distributions, we also often need to generate random data from a discrete distribution. As another example, cross-validating a mathematical model or estimating confidence intervals via bootstrapping often requires that we extract random subsets of our data. Lastly, null hypothesis testing often requires that we randomly "shuffle" (aka., "permute") our data.

Suppose that we have a set of 10 laboratory rats that we wish to evenly and randomly assign to two experimental treatments (5 rats per treatment):

```
rats = ["A1", "A2", "B1", "B2", "C2", "C3", "C4", "E1", "E2", "E6"]
```

One approach is to randomly shuffle (aka. permute) the above list of rat names, and then assign the first 5 rats to treatment 1 and the remaining 5 rats to treatment 2. Randomly shuffling a list can be done using the RNG's `permutation` method:

```
rats_shuffled = rng.permutation(rats)
```

Note that `permutation` leaves the original list unchanged and returns a shuffled copy of it, which we assign to the new variable `rats_shuffled`:

```
print(rats_shuffled)
```

```
['E2' 'A1' 'B2' 'B1' 'C2' 'E6' 'C3' 'C4' 'A2' 'E1']
```

The `permutation` method can also be applied similarly to 1-dimensional `numpy` arrays as well as `pandas` series, such as those representing individual dataframe columns (Section 5.5.3). We can now split the shuffled list into two equal parts (one part per treatment), being confident that we have not introduced any biases in doing so:

```
NR = len(rats) # total number of rats
print("Treatment 1:",rats_shuffled[0:int(NR/2)])
print("Treatment 2:",rats_shuffled[int(NR/2):])
```

```
Treatment 1: ['E2' 'A1' 'B2' 'B1' 'C2']
Treatment 2: ['E6' 'C3' 'C4' 'A2' 'E1']
```

Virtually any type of data (e.g., numbers, strings, lists of lists, etc) can be shuffled using `permutation`, as long as it is in the form of a list or array.

Instead of shuffling the rate names themselves, we could have also created a shuffled list of positional indices that point to the rat names:

```
shuffled_indices = rng.permutation(range(NR))
```

Note that `permutation` can also accept a single integer $n$ as argument, which is equivalent to permuting the sequence $0, 1, .., n-1$, in other words we could have also written:

```
shuffled_indices = rng.permutation(NR)
```

Either way, we obtain the following shuffled list of integers, each referring to a different rat:

```
print(shuffled_indices)
```

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

We can now split this list of integers into two halves, corresponding to an assignment of the rats to the two treatments:

```
print("Treatment 1:",[rats[r] for r in shuffled_indices[0:int(NR/2)]])
print("Treatment 2:",[rats[r] for r in shuffled_indices[int(NR/2):]])
```

```
Treatment 1: ['C2', 'C4', 'B2', 'A1', 'B1']
Treatment 2: ['E6', 'A2', 'E2', 'C3', 'E1']
```

This latter approach can be useful, for example, if we have a database of rat records in the form of a `pandas` dataframe (one row per rat), and we wish to determine which rows to assign to each treatment. The distinction between shuffling rat names and shuffling a list of positional indices referring to rats is analogous to the distinction between `numpy.sort` and `numpy.argsort` discussed in Section 4.15.6.

★ **Exercise 109:** Consider the following numeric list:

```
L = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

Write a code that randomly shuffles (permutes) the first 10 elements and randomly shuffles the last 10 elements, without changing the relative position between the first 10 and last 10 elements. In other words, after reshuffling, the first 10 elements should still be 1,..,10 (in random order) and the last 10 elements should still be 11,..,20 (in random order). Your code should print out the shuffled list.

★ **Exercise 110:** Suppose that we have a 2-dimensional numerical `numpy` array listing pairwise geographic distances between various sampling locations, such as the following:

```
distances = numpy.array([[0,    6.99,  5.05, 6.80,  5.16],
                         [6.99, 0,     5.78, 10.63, 6.32],
                         [5.05, 5.78,  0,    4.93,  0.54],
                         [6.80, 10.63, 4.93, 0,     4.40],
                         [5.16, 6.32,  0.54, 4.40,  0]])
```

Write a code that randomly shuffles the rows and columns of this matrix in the same way. For example, if the first row moves to the third position, then the first column should also move to the third position. Your code should print out the shuffled matrix.

*Hint: First determine the shuffling order that would apply to both rows and columns, as a 1-dimensional integer array, then re-order the rows and columns of the distances array based on*

*the integer array.*

**Context:** A common question in biology is whether a correlation exists between two distance/dissimilarity measures, for example between the pairwise geographic distances and compositional differences of microbial communities [45, 98, 99], or between geographic distance and genetic divergence [100, 101], or between bioacoustic and genetic divergence [102]. The statistical tests deployed for assessing the significance of such correlations commonly involve random permutations of the rows and columns of the distance matrixes, as practiced in this exercise.

### 5.6.5 Example: Permutation null model testing

Spotted hyenas usually live in multi-family female-dominated clans, in which the social status of a female is largely inherited from its mother [103]. Males generally disperse into new clans upon adulthood, where they remain at the lowest social status. The social status, in turn, influences the priority of access to food. In the following, we will see how we can use random shuffling to statistically examine whether family association and sex indeed affect the order of access to food, in a hypothetical hyena clan. Suppose that we observed a clan of spotted hyenas consuming a recent kill, and recorded the order in which individuals gained access to the carcass:

```
hyenas = ["XF5", "XF2", "YF5", "YF3", "XF4", "XM3", "XF7", "ZM4", "YF1", "YM2", "WM6"]
```

Here, the elements of the list `hyenas` represent hyena names in the order in which they gained access, with the first character denoting the family of a hyena (e.g., `XF5` and `XF7` are closely related), and the second character denoting the sex of a hyena (`F`:female, `M`:male). Based on the above data, it appears that hyenas from family `X` tended to gain access earlier compared to family `Y`, and that females tended to gain access earlier than males. To test this suspicion a bit more quantitatively, we can look at the mean access orders of hyenas from families `X` and `Y` as well as the mean access orders of females and males. So let's define a new function for computing the mean access order of a certain group of hyenas, defined based on either the first or second character in their names:

```
import numpy
NH = len(hyenas)
def mean_access_order(names, position, char):
    return numpy.mean([n for n, name in enumerate(names) if name[position]==char])
```

We can now apply our new function to compute the mean order of the various groups:

```
mean_order_X = mean_access_order(hyenas, 0, "X")
mean_order_Y = mean_access_order(hyenas, 0, "Y")
mean_order_F = mean_access_order(hyenas, 1, "F")
mean_order_M = mean_access_order(hyenas, 1, "M")
print("Mean access orders: X=%.2f, Y=%.2f, F=%.2f, M=%.2f"\
      %(mean_order_X,mean_order_Y,mean_order_F,mean_order_M))
```

```
Mean access orders: X=3.20, Y=5.50, F=3.43, M=7.75
```

We indeed see that the mean access order of family `X` was lower than of family `Y`, and that the mean access order of females was lower than males. However, this does not in and of itself imply that members of family `X` truly have priority over those of family `Y`, since it is in principle possible that these observed differences were just coincidental and specific to this particular kill. In other words, if members of both families `X` and `Y` had equal social status and the order of

access to the kill was purely determined by chance, one of the two could still end up having a lower mean access order than the other. A similar reasoning also applies to the differences observed between females and males. To more reliably conclude that status differences between families or between sexes indeed drove these differences in access order, we need to show that they would be unlikely to result from pure chance. This brings us to null hypothesis testing, similarly to what we discussed earlier in Section 5.6.2.

Let's focus on the comparison between the two families `X` and `Y` (we will deal with the comparison between females and males later). The null hypothesis that we need to consider is one where access order is statistically independent of family association. One way to implement such a null hypothesis as a concrete stochastic model is to randomly shuffle (permute) the order of `hyenas`, thus breaking any links between access order and family association. In other words, after shuffling, any effects that family association had on access order disappears. Such null models are generally known as "permutation models" and they are commonly encountered in biology, especially in situations where the distribution of the data (even under the null hypothesis) is unknown [84, 104–106]. We can repeat the shuffling many times, each time re-computing the difference in mean access order between families `X` and `Y`, and then compare the distribution of these hypothetical differences to the true difference observed in the data.

```
diff_XY = mean_order_Y - mean_order_X

NR = 1000 # number of random shufflings to perform
null_diffs_XY = numpy.zeros(NR) # pre-allocate space
rng = numpy.random.default_rng()
for r in range(NR):
    # randomly shuffle the hyena order
    null_hyenas = rng.permutation(hyenas)
    # recompute difference in mean access order between X & Y
    null_mean_order_X  = mean_access_order(null_hyenas, 0, "X")
    null_mean_order_Y  = mean_access_order(null_hyenas, 0, "Y")
    null_diff_XY = null_mean_order_Y - null_mean_order_X
    null_diffs_XY[r] = null_diff_XY
```

Let's plot the distribution of the null-model-based differences (`null_diffs_XY`) as a histogram, and also show the difference observed in the data (`diff_XY`) as a vertical red line for comparison:

```
from matplotlib import pyplot
pyplot.figure(figsize=(5,3))
pyplot.hist(null_diffs_XY, bins=15, edgecolor="black", density=True, label="null")
pyplot.axvline(diff_XY, color="firebrick", linewidth=3, label="observed")
pyplot.xlabel("difference in mean access order Y-X")
pyplot.ylabel("density")
pyplot.legend()
pyplot.savefig("py_output/hyena_access_order_differences_XY.pdf", bbox_inches='tight')
pyplot.close()
```

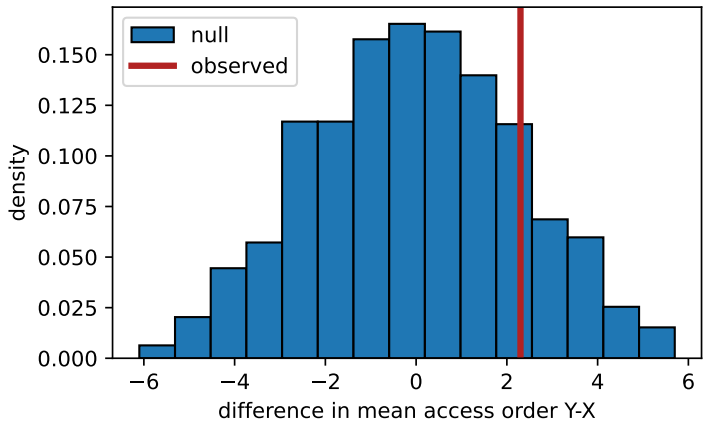



**Figure 5.3:** Histogram of the differences in mean access order generated by the null model, compared to the difference observed in the data (red vertical line).

We see that the observed difference in mean access order between `X` and `Y` is within the typical range expected under the null model. We can in fact estimate the probability that the null model would lead to a difference at least as extreme (in either direction) as seen in our data:

```
P = numpy.mean(numpy.abs(null_diffs_XY)>=numpy.abs(diff_XY))
print("Statistical significance P = %.3f"%(P))
```

```
Statistical significance P = 0.339
```

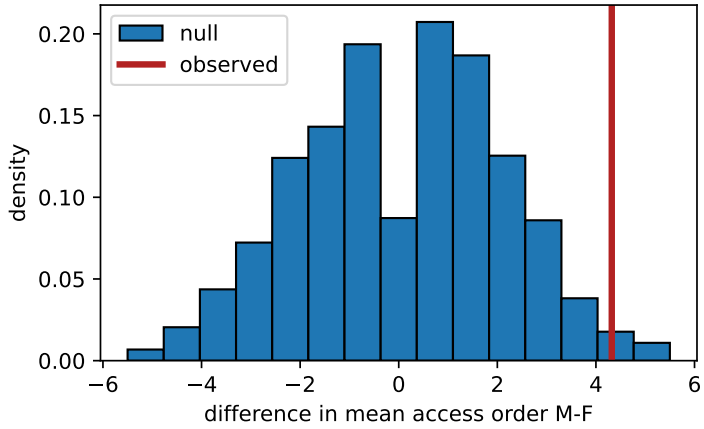
Hence, our null model is not unlikely to generate as strong of a difference between the two families as observed. We thus cannot reject the null hypothesis that family status was independent of access order on the grounds of this difference alone. In other words, if we want evidence that social status differences between families `X` and `Y` caused the observed differences in access order, we need to look elsewhere.

What about the difference between females and males? We can perform a similar permutation model analysis as above, to examine whether the observed difference would be likely under a null model where sex is independent of access order.

```
diff_FM = mean_order_M - mean_order_F

null_diffs_FM = numpy.zeros(NR) # pre-allocate space
for r in range(NR):
    null_hyenas = rng.permutation(hyenas)
    null_mean_order_F  = mean_access_order(null_hyenas, 1, "F")
    null_mean_order_M  = mean_access_order(null_hyenas, 1, "M")
    null_diff_FM = null_mean_order_M - null_mean_order_F
    null_diffs_FM[r] = null_diff_FM

pyplot.figure(figsize=(5,3))
pyplot.hist(null_diffs_FM, bins=15, edgecolor="black", density=True, label="null")
pyplot.axvline(diff_FM, color="firebrick", linewidth=3, label="observed")
pyplot.xlabel("difference in mean access order M-F")
pyplot.ylabel("density")
pyplot.legend()
pyplot.savefig("py_output/hyena_access_order_diffs_FM.pdf", bbox_inches='tight')
pyplot.close()
```

```
P = numpy.mean(numpy.abs(null_diffs_FM)>=numpy.abs(diff_FM))
print("Statistical significance P = %.3f"%(P))
```

```
Statistical significance P = 0.041
```




We see that it is rather improbable to generate as extreme of a difference in mean access order under the null model as that observed in the data. The statistical significance $P$, in particular, is below the conventional threshold of 0.05, suggesting that we should reject the null model. This is evidence that females did indeed have some priority over males in their access to the carcass. That said, while our analysis shows that this priority effect is sufficiently strong to be statistically detectable in our dataset, it says nothing about how strong this effect really is (e.g., other factors may also play a partial role). Defining and assessing the strength of this priority effect would require the formulation of a concrete statistical or mechanistic (non-null) model, which we haven't done here.

★ **Exercise 111:** Consider the following HTSV file listing the genome sizes of several species in the bacterial order Enterobacterales (one species per row):

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`data/Enterobacterales_genome_sizes.htsv`".

The column `genome` lists genome IDs, the column `genome_size` lists genome sizes (number of basepairs), the column `animal_associated` specifies whether the species is estimated to be animal-associated (`True` or `False`), and the column `taxonomy` lists each species' taxonomic classification. For context, recall that *Enterobacterales* comprises a variety of human pathogens and commensals, including *Escherichia coli*, *Yersinia pestis* and *Enterobacter cloacae*. In the follow-

ing we will use a permutation test to determine whether animal-associated Enterobacterales have statistically significant smaller genomes than non-animal-associated ones [107–109]. This type of investigation falls under the general category of "two-sample hypothesis testing" [110–112]. Perform the following tasks:

1. Load the above HTSV file as a `pandas` dataframe, such that the column `animal_associated` is loaded as booleans rather than strings. Compute and print out the mean genome size separately for animal-associated (AA) and non-animal-associated (NAA) Enterobacterales. Which of the two groups has a larger mean genome size?

   *Hint: To specify the data type of specific columns when loading a `pandas` dataframe from a file with `read_csv`, use the optional argument `dtype`. In this particular case, you can use `dtype={"animal_associated":bool}`.*

2. Let $D$ denote the difference between the mean genome sizes of AA and NAA Enterobacterales observed in the data. Examine the statistical significance of the observed $D$ under the null hypothesis that AA and NAA have equal mean genome sizes, using a permutation null model defined as follows: Under the null model, the elements in the `genome_size` column are randomly permuted, thus breaking any correlation between genome size and animal-association. For each iteration of the null model (i.e., after each permutation), re-compute the difference between the mean genome sizes of AA and NAA prokaryotes. Perform 1000 random permutations, and store the corresponding 1000 differences in a numeric array. Show the distribution of these differences as a histogram, and include the $D$ observed in the data as a vertical line for comparison (similar to Fig. 5.3).

3. Compute and print out the statistical significance, aka. $P$-value, of the observed $D$ under the null model, estimated based on the fraction of null models that generated a mean genome size difference at least as strong (in absolute terms) as observed in the data. Based on your finding, should the null hypothesis be rejected?

### 5.6.6 Random choice

Another common randomization task is choosing random values from a finite pool of possible values. For example, when designing a randomized trial we may need to assign subjects to a discrete set of alternative experimental treatments. Random choice may be done with or without replacement. Without replacement, each value in the pool can be chosen at most once, while with replacement each value in the pool may in principle be chosen multiple times. Choosing without replacement (aka., subsampling) is analogous to a raffle, in which a set of unique ticket numbers is drawn out of a container. Choosing with replacement is analogous to drawing one ticket number at a time, reading out the winner, and then replacing the ticket number back into the container prior to the next draw (thus, allowing for the possibility that some players can win multiple times). Both of these operations can be performed with the RNG method `choice`. Similarly to the `permutation` function, `choice` can be applied to virtually any type of data, including strings, number etc, as long as the pool to choose from is provided in the form of a list, a 1-dimensional `numpy` array, a `pandas` series or similar.

As an example, consider our previous list of rat names, and suppose that we wanted to choose a random subset of 3 rats for our next experiment. This can be achieved as follows:

```
chosen_rats = rng.choice(rats, size=3, replace=False)
print(chosen_rats)
```

```
['A2' 'E6' 'B1']
```

Observe that we specified the pool of values to choose from as the first argument to `choice`, we specified the number of requested values using the `size` argument, and indicated that we want values to be drawn without replacement by setting `replace=False`. The above code yields a new string array consisting of 3 randomly chosen rat names, while leaving the input list unchanged. Choosing without replacement has many other applications. For example, we may want to choose and analyze a small subset of a larger dataset for exploratory purposes, prior to investing large computational resources to analyze the entire dataset.

As another example, suppose that we would like to test (benchmark) a new computational pipeline for analyzing arbitrary DNA sequences. Thus, we would like to generate random DNA sequences that we can feed into our pipeline for testing. Generating a random DNA sequence means repeatedly choosing among the four nucleotides "A", "C", "G" and "T" with replacement. Replacement is needed, since each nucleotide may appear multiple times in a DNA sequence. This task can easily be achieved using the `choice` method:

```
DNA = rng.choice(["A","C","G","T"], size=50, replace=True)
```

Hereafter, `DNA` will be a 1-dimensional `numpy` array consisting of 50 individual characters, each chosen randomly and independently among "A", "C", "G" and "T":

```
print(DNA)
```

```
['A' 'A' 'A' 'C' 'G' 'T' 'A' 'A' 'G' 'G' 'A' 'C' 'C' 'C' 'T' 'T' 'A' 'C'
 'G' 'G' 'C' 'C' 'G' 'T' 'C' 'T' 'T' 'C' 'T' 'G' 'T' 'G' 'T' 'A' 'C' 'G'
 'C' 'T' 'G' 'T' 'A' 'C' 'C' 'T' 'G' 'T' 'T' 'A' 'G' 'T']
```

To convert `DNA` into a single string of 50 characters, we can use the string method `join`. This method concatenates a list or array of strings with a specific delimiter (in this case, we concatenate with the empty string `""`):

```
DNA = "".join(DNA)
print(DNA)
```

```
AAACGTAAGGACCCTTACGGCCGTCTTCTGTGTACGCTGTACCTGTTAGT
```

It is also possible to draw values from a pool with non-equal probabilities. Indeed, it is well-known that in natural DNA sequences the four nucleotides rarely appear at equal proportions. As a case in point, nucleotide proportions in the genome of *Mycobacterium lepraemurium* strain Hawaii are around A:0.155, C:0.345, G:0.345 and T:0.155 [12]. To generate random DNA sequences according to these probabilities, we can specify the latter using the optional argument `p`:

```
probabilities = [0.155, 0.345, 0.345, 0.155]
DNA = "".join(rng.choice(["A","C","G","T"], size=50, replace=True, p=probabilities))
print(DNA)
```

```
ACCATGGCCCGTACCCCTGGGGGCTTCGAGGGCGCAGTCGGGGCCAGCTA
```

There is of course no guarantee that the nucleotide proportions in the generated sequence will be exactly equal to the specified probabilities since, after all, nucleotides were randomly chosen. For long sequences, however, the obtained nucleotide proportions will approximately resemble the specified probabilities.

★ **Exercise 112:** Write a code that generates and prints out a random sequence of 50 amino acids (as a single string), by choosing each amino acid randomly and independently with replacement from the standard amino acid "alphabet": A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y.

★ **Exercise 113:** In this exercise we will use random choice to examine how random mass species extinctions affect higher taxa (in this case, genera). Consider the following dataset of over one million animal species names, in the form of a gzipped text file (one latin binomial species name per line):

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`animal_species_names.txt.gz`".

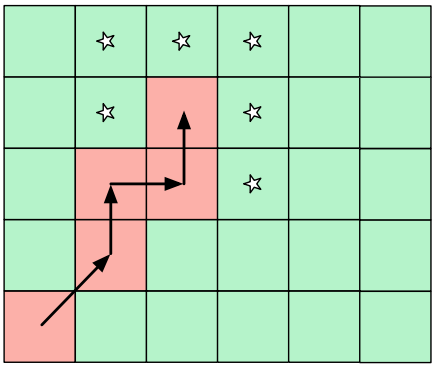
Perform the following tasks:

1. Load all species names from the file as a list of strings (one string per species). Make sure to omit any comment lines, i.e., lines starting with `#`. Print out the number of species loaded.
2. Write a custom function (named `count_genera`) that accepts a single argument, `species`, in the form of a list of strings representing binomial species names, and which returns the number of unique genera in the list. Apply this function to your loaded dataset to compute and print out the number of genera loaded.
3. To simulate a hypothetical mass extinction event, randomly choose 20% of species for survival and create a new list comprising only those species (the remaining 80% of species are thus assumed to have gone "extinct"). All species should have equal probability of going extinct. Using your function `count_genera`, determine and print out the number and fraction of genera remaining in the new list (i.e., "surviving"). How does the fraction of surviving genera compare to the fraction of surviving species?
4. To simulate a second extinction event shortly after the first one, repeat the previous task with the new species list, i.e., only keeping 20% of the previously surviving 20% of species. How does the fraction of surviving genera compare to the fraction of surviving species this time?

★★ **Exercise 114:** DNA sequences can contain intriguingly long repetitions of short patterns, some potentially resulting from interesting biological processes (e.g., transposons) and others occurring by mere coincidence. For example, the genome sequence of bacterium *M. lepraemurium* strain Hawaii [12] contains a suspiciously long contiguous segment of 25 cytosines ("CCCCCCCCCCCCCCCCCCCCCCCCC"). In this exercise we will learn how to examine whether such non-random looking patterns are indeed non-random with a simple hypothesis test. Generate 100 random nucleotide sequences, each comprising $4 \times 10^6$ characters chosen randomly and independently from the pool "A", "C", "G", "T" and at probabilities A:0.155, C:0.345, G:0.345 and T:0.155. Note that these parameters correspond approximately to the properties of the aforementioned *M. lepraemurium* genome. For each integer $n \in \{1, .., 25\}$, determine the number of generated sequences that exhibit a contiguous mono-nucleotide segment of length $\geq n$ (i.e., exhibiting $n$ contiguous adenines, or $n$ contiguous cytosines, etc). Plot these numbers as a bar plot, i.e., such that the horizontal axis denotes the lengths $n = 1, .., 25$ and the vertical axis represents the number of sequences exhibiting a contiguous mono-nucleotide segment of length $\geq n$. Based on your finding, is it probable or improbable that a contiguous mono-nucleotide segment as long as the one mentioned earlier appears in a bacterial genome of comparable size?

*Hint: Note that the above computation can take a few minutes to complete. It is thus recommended to first develop and test the code's correctness with shorter sequences (e.g., $10^5$ basepairs), prior to running it for full-length sequences.*

★★ **Exercise 115:** In the following we will construct and examine a simple stochastic simulation model for the movement of a single foraging herbivore across patches of grass. While the model considered here is rather simplistic, more realistic stochastic simulation models are indeed used by ecologists to understand animal movement [113–115]. We consider an enclosed foraging area in the form of a rectangular grid, each grid cell corresponding to a small patch of grass. Suppose that once per day the herbivore moves to a new random adjacent patch that it had not yet visited before, at equal probabilities. In other words, each day the animal randomly chooses one of the 8 adjacent patches to move next to, but excluding any previously visited (thus consumed) patches. The following figure illustrates a hypothetical possible path of the animal on a $5 \times 6$ grid:



Red boxes indicate previously visited patches, green boxes indicate patches not yet visited. As a next move, the animal will randomly choose among the 6 possible patches indicated by the stars. In the following, we consider a grid of $R = 20$ rows and $C = 30$ columns. Perform the following tasks:

1. Construct a custom function named `simulate` with no arguments that uses a `for` loop to simulate a random hypothetical path until a move is no longer possible (i.e., the animal is exclusively surrounded by previously visited patches), whichever comes first. The animal should always start in the lower-left corner of the grid. The function should return two lists, representing the vertical and horizontal coordinates of the patches visited along the path, respectively (i.e., the integer positions of rows and columns). For example, the lower-left patch has coordinates (0,0), while the one to its immediate right has coordinates (0,1). Use the function to simulate a single path, and visualize the path as a curve in space. Do show the entire grid when plotting the curve within it, i.e. don't just show the segment visited by the herbivore.

   *Hint: Since the grid only comprises $R \times C$ patches, no path of the animal can possibly be longer than that. Hence, your loop can be set to iterate for up to $R \times C + 1$ times, and exit once a "dead end" is reached. You can use an auxiliary* `numpy` *boolean array to keep track of which patches have already been visited, and are thus impermissible for the next move.*

2. Call the function `simulate` 1000 times using a loop, and determine the average number of patches visited by the animal per simulation. In other words, determine on average how many patches an animal visits before reaching a dead-end. How does this number compare to the total number of available patches?

## 5.7 More plotting

### 5.7.1 Multiple histograms

One of the most common uses of histograms is to visually compare two or more distributions, for example measurements of the same variable taken under two different experimental treatments, or a morphological trait among different cohorts of a population. To plot multiple histograms

side by side, we can pass the samples for each histogram as a separate array or list, all bundled together into a larger list. As an example we consider our previous dataset of gull egg lengths, but separated by the year in which each egg was laid. The data are available in the following HTSV file:

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`gull_egg_and_chick_properties_Yurlov2022.htsv`".

Let's begin by loading the egg lengths and the corresponding years as a 2-dimensional numeric array, with years in the first column and egg lengths in the 2nd column:

```
import numpy
data = numpy.genfromtxt("data/gull_egg_and_chick_properties_Yurlov2022.htsv",
                        delimiter="\t", usecols=[0,9])[1:,:]
```

We consider eggs from years 1993 and 2001, which make up a large portion of this dataset. To split the egg lengths by year we can use boolean indexing (Section 5.4), as follows:

```
lengthsA = data[data[:,0]==1993,1]
lengthsB = data[data[:,0]==2001,1]
```

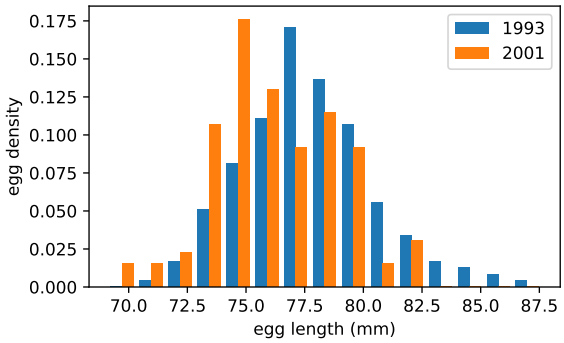
To plot the two histograms side by side, we will need to pass both dataset as a single list, whose elements are the two arrays `lengthsA` and `lengthsB`:

```
combined_lengths = [lengthsA, lengthsB]
```

Since the two sample sets may not be of equal size (for example, there may have been more eggs in one year than in another), it is generally advised to normalize the total area of each histogram if the goal is to compare the underlying probability distributions. This can be achieved by setting the optional argument `density` to `True`. Further, as in other situations where multiple datasets are plotted int the same figure, it is recommended to label each histogram and include a legend in the figure.

```
from matplotlib import pyplot
pyplot.figure(figsize=(5, 3))
pyplot.hist(combined_lengths,
            bins      = 15,
            density   = True,
            label     = ["1993", "2001"])
pyplot.xlabel("egg length (mm)")
pyplot.ylabel("egg density")
pyplot.legend() # add a legend
pyplot.savefig("py_output/histogram_egg_lengths3.pdf", bbox_inches='tight')
pyplot.close()
```

We thus obtain the following figure:



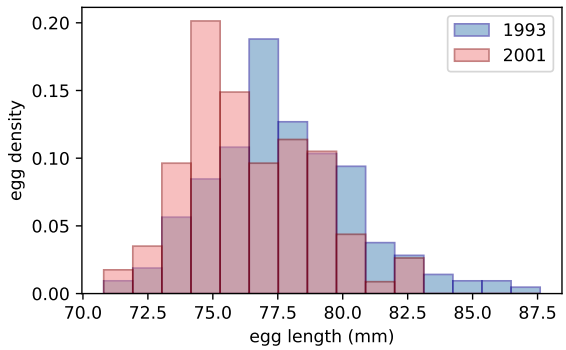

Note that `hist` alternates between the bars of the two histograms, and puts some spacing between consecutive bins to facilitate their visual distinction. Having both histograms interleaved, allows us to see that the 2001 histogram is somewhat shifted towards the left compared to the 1993 histogram, suggesting that eggs tended to be smaller in 2001 compared to 1993.

**Overlapping histograms:** An alternative approach would be to not alternate between histograms, but instead have both histograms overlap with each other. To achieve this, we can simply call `hist` twice, once for each histogram. In that case, it is recommended to make each histogram a bit transparent using the optional argument `alpha`, so that both histograms are fully visible. It is also recommended to use the exact same set of bins (not just the same number of bins) for both histograms; since by default `hist` re-determines the bin bounds for each histogram, we need to explicitly specify the desired bins the second time we call `hist`. Fortunately, the bins used by the first `hist` call are accessible as part of the function's return value (specifically, the 2nd element of the return value). So let's give it a try:

```
pyplot.figure(figsize=(5, 3))

# plot the first histogram, and "collect" its return value
results = pyplot.hist(lengthsA,
                      bins      = 15,
                      density   = True,
                      color     = "steelblue",
                      edgecolor = "darkblue",
                      alpha     = 0.5,
                      label     = "1993")

# plot the second histogram with a different color,
# specifying the bins to be the same as those in the previous histogram
pyplot.hist(lengthsB,
            bins      = results[1],
            density   = True,
            color     = "lightcoral",
            edgecolor = "darkred",
            alpha     = 0.5,
            label     = "2001")

# add axis labels
pyplot.xlabel("egg length (mm)")
pyplot.ylabel("egg density")

# add legend
pyplot.legend()

# save and close figure
```

```
pyplot.savefig("py_output/histogram_egg_lengths4.pdf", bbox_inches='tight')
pyplot.close()
```

We thus obtain the following figure:




Which method we use for showing two histograms together (interleaved vs overlapping) is ultimately our choice and largely a matter of aesthetics.

★ **Exercise 116:** Consider the following HTSV file listing the genome sizes of several prokaryotic species (one species per row):

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`data/RefSeq_prokaryote_genome_sizes.htsv`".

The column `genome` lists genome IDs, the column `genome_size` lists genome sizes (number of basepairs), the column `animal_associated` specifies whether the species is estimated to be animal-associated (`True` or `False`), and the column `taxonomy` lists each species' taxonomic classification. Load this file as a `pandas` dataframe, and show the distribution of genome sizes as histograms, separately for animal-associated (AA) and non-animal-associated prokaryotes. The histograms should overlap (not alternate), as demonstrated in Section 5.7.1.

### 5.7.2 Box plots

A common task in science is to visually compare the values of a focal variable measured multiple times in multiple discrete groups, for example in different populations or under different experimental conditions. For example, we may have measured a morphological trait in multiple individuals from various closely related plant species, and would like to see whether there are differences in the trait between the species, taking into account the general spread and potential overlap between them [116]. As another example, we may want to visually compare pathogen densities measured on plants under different treatments, taking into account the variability seen within any given treatment [117]. Showing the distributions of the measured variable as histograms, separately for each group, is one possible approach (Section 5.7.1). However histograms alone make it hard to see specific quantitative differences (e.g., means), and visualizations can become quite cluttered when more than two overlapping histograms are shown. An alternative powerful visualization, which we will learn about here, is the boxplot (aka. "box-whisker plot"). In box plot, the distribution of the focal variable is visually summarized separately for each group in terms of the median, the first and third quartiles (i.e., 25th and 75th percentiles, as a measure of the spread), and the minimum and maximum, although some slight variations exist in practice. Optionally, data outliers or even the full original data may be shown as scatterpoints.

As an example, consider the following HTSV file, listing adult body masses and basal metabolic rates for various species [19]:

http://www.loucalab.com/archive/BioDataAnalysisPython
→ file "mass_metabolic_rate_Hatton2019.htsv".

Let's begin by loading the data as a `pandas` dataframe and printing out the first few rows for verification. We only load the three columns listing major taxon names, body masses (gram) and metabolic rates (Watt), as we won't need the other data in the file.

```
import numpy
import pandas
data = pandas.read_csv("data/mass_metabolic_rate_Hatton2019.htsv",
                       delimiter="\t", comment="#",
                       usecols = ["Major_taxon","Mass_g", "Metabolism_W"])
print(data.iloc[0:5,:])
```

```
  Major_taxon  Mass_g  Metabolism_W
0      Mammal    70.0      0.472222
1      Mammal    34.0      0.227778
2      Mammal   116.4      0.750000
3      Mammal    26.1      0.086111
4      Mammal     6.9      0.044444
```

We are specifically interested in the mass-specific metabolic rates (MSMR, in Watt/gram), which we can easily compute and add as a new column to the dataframe:

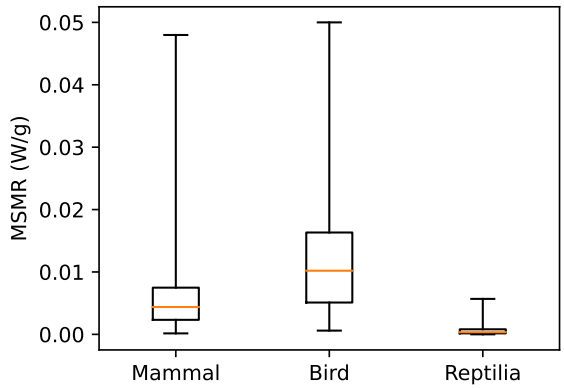
```
data["MSMR"] = data["Metabolism_W"]/data["Mass_g"]
```

To plot a separate box for each major taxon (mammals, birds, reptiles), we first construct a list whose elements correspond to the various taxa, each in the form of a 1-dimensional `numpy` array or `pandas` series. List comprehension works perfectly for this task:

```
taxa = ["Mammal", "Bird", "Reptilia"]
grouped_MSMRs = [data["MSMR"].loc[data["Major_taxon"]==taxon] for taxon in taxa]
```

Henceforth, `grouped_MSMRs[0]` contains the MSMRs of mammals, `grouped_MSMRs[1]` contains the MSMRs of birds and `grouped_MSMRs[2]` contains the MSMRs of reptiles. We are now ready to create a boxplot using the function `boxplot` in the `pyplot` module:

```
from matplotlib import pyplot

# initialize a new figure
pyplot.figure(figsize=(4, 3))

# draw the boxplot
pyplot.boxplot(x = grouped_MSMRs, labels = taxa, whis = (0,100))

# add a vertical axis label
pyplot.ylabel("MSMR (W/g)")

# save and close figure
pyplot.savefig("py_output/boxplot_mass_specific_metabolic_rates1.pdf",
               bbox_inches='tight')
pyplot.close()
```

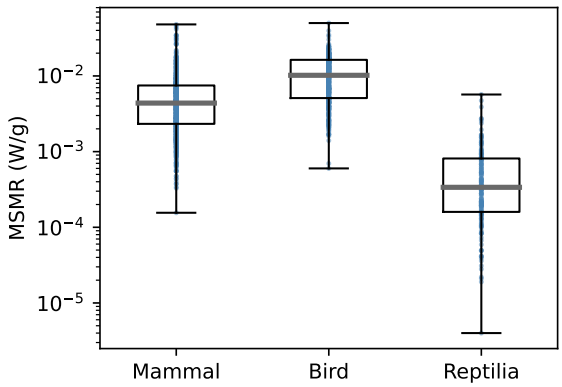



For each group (mammals, birds, reptiles), the box spans the 2nd and 3rd quartiles of the MSMRs, the vertical lines (called "whiskers") indicate the full range of the MSMRs (minimum to maximum), and the small horizontal orange line inside the box represents the median of the MSMRs. The box and whiskers yield insight into the *spread* of the MSMRs. For example, we can clearly see that the MSMRs of birds have the greatest median and greatest spread (in terms of the interquartile range) of all considered groups. We can also see that reptiles tend to have much lower MSMRs than mammals and birds, likely because reptiles are ectotherms while mammals and birds are endotherms.

In the above `boxplot` call, the `x` argument specifies the data to be visualized by the boxplot, `labels` specifies the group names to be displayed beneath each box, and `whis` specifies the range to be spanned by the whiskers (in our case, from the 0-th to 100-th percentile). Many other optional arguments can be used to customize the figure, described in the official documentation. For example, we can use the `medianprops` argument to improve the thickness and color of the median line, and we can use the `widths` argument to change the widths of the boxes. In addition, it is common to overlay the raw data for each group as scatterpoints. This can be achieved by calling the familiar `scatter` function, separately for each group, keeping in mind that the horizontal box positions are by default 1, 2 and so on. Lastly, it can be beneficial to show the vertical axis on a logarithmic scale instead of a linear one, especially when the data span multiple orders of magnitude, as in this case. So let's give it another try, incorporating all of the above improvements.

```
pyplot.figure(figsize=(4, 3))

# draw boxplot
pyplot.boxplot(x           = grouped_MSMRs,
               labels      = taxa,
               widths      = 0.5,
               whis        = (0,100),
               medianprops = {"linewidth":2.5, "color":"dimgrey"} )

# show raw data as scatterpoints
for g, group_data in enumerate(grouped_MSMRs):
    X = numpy.full(len(group_data),g+1)
    pyplot.scatter(X, group_data, s=2, alpha=0.5, color="steelblue")

# use log scale for vertical axis
pyplot.yscale('log')

pyplot.ylabel("MSMR (W/g)")
pyplot.savefig("py_output/boxplot_mass_specific_metabolic_rates2.pdf",
               bbox_inches='tight')
```

```
pyplot.close()
```




Thanks to the logarithmic scale, we can now more clearly see the full range of each group's MSMRs. However, due to the large data sizes the scatterpoints seem rather cluttered, and thus it is hard to see their distributions. A simple trick for resolving this issue is to show each scatterpoint at a slightly different horizontal location. Specifically, we can add a small random horizontal displacement to each data point, a technique called "jittering". Let's see how this looks in practice:

```
pyplot.figure(figsize=(4, 3))

# draw boxplot
pyplot.boxplot(x            = grouped_MSMRs,
               labels       = taxa,
               widths       = 0.5,
               whis         = (0,100),
               medianprops  = {"linewidth":2.5, "color":"dimgrey"} )

# show raw data as jittered scatterpoints
rng = numpy.random.default_rng()
for g, group_data in enumerate(grouped_MSMRs):
    X = numpy.full(len(group_data),g+1) + rng.normal(0, 0.07, size=len(group_data))
    pyplot.scatter(X, group_data, s=2, alpha=0.5, color="steelblue")

pyplot.yscale('log')
pyplot.ylabel("MSMR (W/g)")
pyplot.savefig("py_output/boxplot_mass_specific_metabolic_rates3.pdf",
               bbox_inches='tight')
pyplot.close()
```

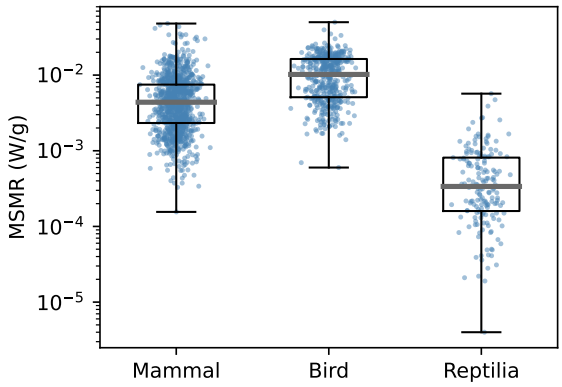



**Figure 5.4:** Boxplot of mass-specific metabolic rates for various species, separated by major taxon (one box per taxon, one point per species). Data from [19].

We can now better see how the data are distributed in each group. For further examples of boxplots in the scientific literature see e.g. [118, Fig. 3], [116, Fig. 1] and [119, Figs. 2a and 4b].

★ **Exercise 117:** Consider the following TSV file listing average adult body masses for various primate species, separated by sex [34]:

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`data/primate_body_masses.tsv`".

The first column lists species names, the second column lists male body masses, and the third column lists female body masses. Create a boxplot showing the distribution of the male and female body masses (one box per sex). The whiskers should span the entire data range (minimum to maximum), and the vertical axis should be on a logarithmic scale. Also overlay the raw data as jittered scatterpoints, similarly to Fig. 5.4.

★ **Exercise 118:** Consider the following dataset of various morphological, ecological and morphological traits for nearly all bird species [60] (one row per species):

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`AVONET_bird_traits.htsv`".

This HTSV file includes many columns, including `Wing.Length` (listing average wing length in mm) and `Habitat` (listing the main habitat type, e.g., Forest, Grassland etc). Create a boxplot that shows the distributions of wing lengths, separately for each habitat type (one box per habitat). Make sure to exclude "`NA`" as habitat type in your figure, since `NA` merely indicates for some species that information on the habitat is not available. The whiskers should span the

entire data range (minimum to maximum), and the vertical axis should be on a linear scale. Also overlay the raw data as jittered scatterpoints, similarly to Fig. 5.4. In which 3 habitats do birds have the highest median wing lengths?

*Hint: Use a set to determine the unique habitat types in this dataset, and the set method* `difference` *to exclude the* `NA` *habitat. Use the optional argument* `keep_default_na=False` *to load* `NA` *entries in the* `habitat` *column as "*`NA`*" strings, as this will make it easier to omit those afterwards. You can use list comprehension to create the appropriate data list to be passed to the* `boxplot` *function.*

### 5.7.3 Heatmaps

Another common visualization type encountered throughout biology are heatmaps, which color-code the values of a numeric variable across two separate discrete axes. For example, a heatmap can be used to show the abundances of different species (axis 1) at various locations or time points (axis 2) [120, 121], or the expression levels of multiple genes (axis 1) under different treatments or in different tissues (axis 2) [122–124].

As an example, let's consider the following dataset of relative abundances (proportions) of microbial species associated with different metabolic functions, estimated for multiple permafrost and bovine gut samples:

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`FAPROTAX_function_table.htsv`".

Each row in this HTSV file corresponds to a different metabolic function, each column to a different sample, and each entry lists the proportion of a given function in a given sample. We begin by loading this dataset as a `pandas` dataframe:

```
import pandas
data = pandas.read_csv("data/FAPROTAX_function_table.htsv",
                       delimiter = "\t", comment = "#",
                       index_col = "function")
print("Found %d functions x %d samples"%(data.shape[0],data.shape[1]))
```

```
Found 90 functions x 18 samples
```

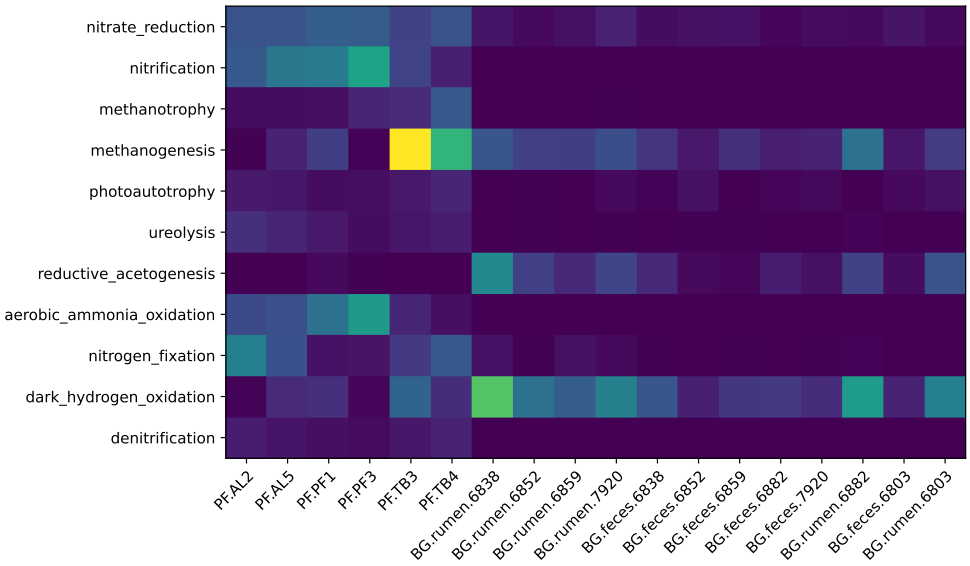
Note that the sample names are stored as the dataframe's column labels, while the function names are stored as the dataframe's row labels. Since this is a large table, for simplicity let's focus on a smaller subset of particularly interesting functions:

```
functions = ["nitrate_reduction", "nitrification", "methanotrophy",
             "methanogenesis", "photoautotrophy", "ureolysis",
```

```
            "reductive_acetogenesis", "aerobic_ammonia_oxidation",
            "nitrogen_fixation", "dark_hydrogen_oxidation","denitrification"]
data = data.loc[functions]
```

To visualize this table as a basic heatmap, we can use the function `imshow` in the `pyplot` module:

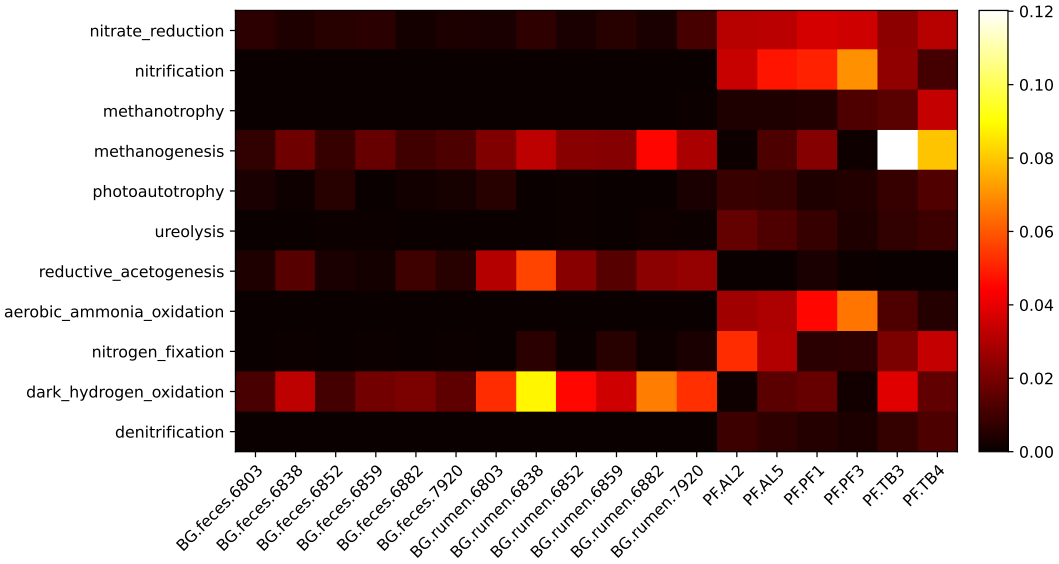
```
from matplotlib import pyplot

# initialize a new figure
# choose the width & height according to the data shape
pyplot.figure(figsize=(0.8*data.shape[0], 0.8*data.shape[1]))

# draw the heatmap
pyplot.imshow(data)

# add x-ticks indicating the sample names, rotated by 45 degrees
pyplot.xticks(ticks         = range(data.shape[1]),
              labels        = list(data.columns),
              rotation      = 45,
              rotation_mode = "anchor",
              ha            = "right")

# add y-ticks indicating the function names
pyplot.yticks(ticks = range(data.shape[0]), labels = list(data.index))

# save and close figure
pyplot.savefig("py_output/heatmap_prokaryotic_functions1.pdf",
               bbox_inches='tight')
pyplot.close()
```




In the above heatmap, a brighter color indicates a higher abundance, however it is unclear which color corresponds to which abundance. It is thus recommended to add a "color bar" next to the figure, which is essentially a legend clarifying the correspondence between colors and values. This can be achieved with the `colorbar` function, however it requires a bit of technical tweaking because by default the color bar's size does not match that of the heatmap. Further, observe that samples are fully ordered according to type, with rumen and fecal samples alternating occasionally. It is generally recommended to order rows and/or columns in a biologically meaningful

way, whenever possible. Lastly, we can specify an alternative color map using the argument `cmap` (for a list of common color map options see here). Let's give it another try, incorporating the above suggestions:

```
# import the function make_axes_locatable,
# needed for properly placing & sizing the color bar
from mpl_toolkits.axes_grid1 import make_axes_locatable

# sort samples (dataframe columns) alphabetically
data = data.iloc[:,numpy.argsort(list(data.columns))]

# initialize a new figure
pyplot.figure(figsize=(0.8*data.shape[0], 0.8*data.shape[1]))

# draw the heatmap, and store it in the variable heatmap
heatmap = pyplot.imshow(data, cmap="hot")

# add x- and y-ticks, representing sample & function names
pyplot.xticks(ticks         = range(data.shape[1]),
              labels        = list(data.columns),
              rotation      = 45,
              rotation_mode = "anchor",
              ha            = "right")
pyplot.yticks(ticks = range(data.shape[0]), labels = list(data.index))

# add a color bar to the right of the heatmap
# we need to create an additional axis to the right of the figure
cax = make_axes_locatable(pyplot.gca()).append_axes("right", size="4%", pad="3%")
pyplot.colorbar(heatmap, cax=cax)

# save and close figure
pyplot.savefig("py_output/heatmap_prokaryotic_functions2.pdf",
               bbox_inches='tight')
pyplot.close()
```




Observe that clear differences exist between samples in terms of the abundances of various functions, especially between the BG (bovine gut) and PF (permafrost) samples. To better see

which functions have similar distributions across samples, we can re-order the heatmap's rows so that similarly distributed functions appear close to each other, a process called "clustering". A common clustering method is "hierarchical clustering". We won't discuss the theory behind hierarchical clustering in this book, but a good overview is available on wikipedia or here [125]. For our purposes, it suffices to know that hierarchical clustering yields a tree-like relationship between rows based on their similarity, and that this tree-like structure defines a particular row order:

```
from scipy.cluster import hierarchy
row_tree  = hierarchy.ward(data)
row_order = hierarchy.leaves_list(hierarchy.optimal_leaf_ordering(row_tree,data))
```

Hereafter, `row_order` will be a 1-dimensional integer array that specifies the order of rows consistent with the hierarchical clustering. Let's create a new heatmap, with rows ordered accordingly.

```
# sort samples (columns) alphabetically
# and functions (rows) based on the previously computed clustering
data = data.iloc[row_order,numpy.argsort(list(data.columns))]

# create heatmap, exactly as before
pyplot.figure(figsize=(0.8*data.shape[0], 0.8*data.shape[1]))
heatmap = pyplot.imshow(data, cmap="hot")
pyplot.xticks(ticks         = range(data.shape[1]),
              labels        = list(data.columns),
              rotation      = 45,
              rotation_mode = "anchor",
              ha            = "right")
pyplot.yticks(ticks = range(data.shape[0]), labels = list(data.index))
cax = make_axes_locatable(pyplot.gca()).append_axes("right", size="4%", pad="3%")
pyplot.colorbar(heatmap, cax=cax)
pyplot.savefig("py_output/heatmap_prokaryotic_functions3.pdf",
               bbox_inches='tight')
pyplot.close()
```

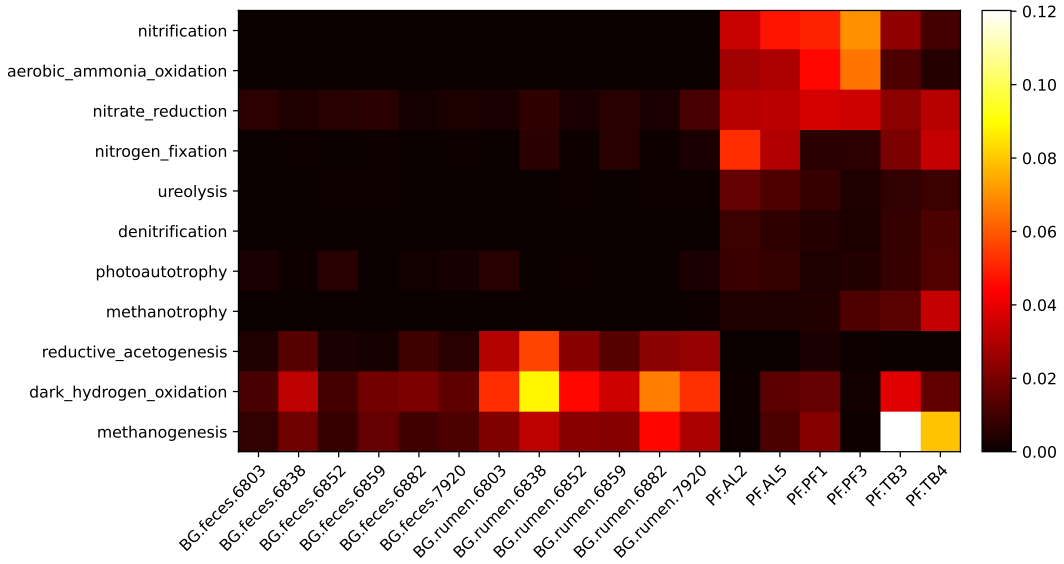



**Figure 5.5:** Heatmap showing the relative abundances of various microbial metabolic functions (rows) in samples from different environments (columns). Rows have been ordered using hierarchical clustering, so that similarly distributed functions are listed close to each other in the heatmap.

We can now see much more clearly which microbial metabolic functions tend to be distributed similarly across environments. For example, methanogenesis, reductive acetogenesis and dark hydrogen oxidation are particularly prevalent in the bovine rumen.

★ **Exercise 119:** Consider the following dataset of gut microbial community surveys in mice, performed on multiple days during a gut colonization experiment [79]:

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`mouse_microbiomes_OTU_table_Marino2014.htsv`".

Each row in this HTSV file corresponds to a different sample (mouse-day combination), the first column (`SampleID`) lists sample IDs (mouse+day), the second column (`mouse`) lists the mouse ID, the third column (`day`) lists the day and the remaining columns correspond to individual bacterial operational taxonomic units (OTUs), with each entry in those columns specifying the number of sequencing reads matched to a given OTU in a given sample. Create a heatmap that shows the relative abundances (proportions) of the first 20 OTUs (i.e., based on the order in which they are listed in the file) in all mice on the first and 20-th day of the experiment (omit any samples from other days). The rows in the heatmap should correspond to samples and the columns to OTUs. Order samples first by day and then by mouse (i.e., first show all day-1 samples, then all day-20 samples), and order OTUs using hierarchical clustering, as demonstrated in Section 5.7.3 (Fig. 5.5).

*Hint: To compute the proportion of each OTU in a sample, divide its number of read by the total number of reads in that sample (i.e., summed over all OTUs; see e.g. Exercise 106). Note that the function* `hierarchy.ward` *in the* `scipy` *package, introduced in Section 5.7.3, expects an array whose rows are to be clustered. To instead cluster the columns, you will need to pass a transposed version of your data as an argument to that function (you can use the array method* `transpose` *for this).*

## 5.8 Advanced string operations (regular expressions)

For more complex string searches and replacements, such as finding or replacing substrings bracketed between two keywords, we can use the `re` package (`re` stands for regular expression). Regular expressions ("regex") provide a powerful and widely used language for pattern matching in text. Regular expressions are used, for example, to find patterns in DNA sequences. Countless regex tutorials exist online, although keep in mind that some subtle differences exist between different regex implementations (e.g., in `python` vs. other programming languages). To learn more about regular expressions in `python` see this GfE document. In the following we will only discuss a few basic examples.

### 5.8.1 Replacing contiguous sequences of characters

Suppose that we have a DNA sequence in which some nucleotides are unknown. Unknown nucleotides can result from sequencing errors, and are commonly denoted with an `N` character, for example as follows:

```
DNA = "TATCGGCATNNNNTAATCCGACTTTCCCNNNNNNNNNNTAGGCAACGGCTAANNTTACTGCAACAGTAGCNA"
```

Suppose that we wanted to replace all contiguous sequences of `N` (no matter how long) in the above DNA sequence with single dashes, i.e., to get the following sequence:

```
TATCGGCAT-TAATCCGACTTTCCC-TAGGCAACGGCTAA-TTACTGCAACAGTAGC-A
```

To achieve this, we can use the `sub` function in the `re` package to achieve this, as follows:

```
new_DNA = re.sub("N+","-",DNA)
```

Here the first argument specifies that we are looking for segments of `N` of arbitrary length, the 2nd argument specifies that we want to replace any such segment with `-`, while the 3rd argument specifies that we want to perform this operation on the string variable `DNA`. The function then returns a modified copy of the string `DNA` with all `N` segments replaced by `-`, which we assign to a new variable `new_DNA` while leaving the original string unchanged.

### 5.8.2 Finding/replacing bracketed substrings

Suppose that we want to find the segment in a DNA sequence that is contained between two specific target sequences, representing a forward and a reverse PCR primer:

```
DNA     = "TATCGGCCCCTACTAATCCGACTTTCCCTTCACGAGCTAAGGCAACGGCTAAATTTACTGCAACAGTA"
forward = "TACTAATCC"
reverse = "ATTTACTGC"
```

To find the segment between the two primer sequences, we can use the `search` function in the `re` package, as follows:

```
import re
segment = re.search(forward+"(.*)"+reverse, DNA).group(1)
```

In the above code, we provide two arguments to the function `re.search`: The first argument specifies that we want to find text bracketed by the `forward` and `reverse` strings, while the 2nd argument specifies the string variable within which we want to perform the search. Note that `re.search` returns multiple pieces of information, such as the location of the bracketed segment

as well as its actual content; we use `.group(1)` to extract the actual content of the segment, thus obtaining:

```
GACTTTCCCTTCACGAGCTAAGGCAACGGCTAA
```

If instead of finding the segment contained between the two primers we wanted to completely omit it and the primers from the sequence, we could use the function `sub` instead of `search`:

```
new_DNA = re.sub(forward+".*"+reverse, "", DNA)
```

Observe that we provide 3 arguments to the function `sub`: The first argument specifies the pattern that we are searching for, the 2nd argument specifies that we want to replace the pattern with an empty string (thus effectively erasing it), and the 3rd argument specifies the string variable on which we want to perform this operation. Once the above code has executed, the `new_DNA` variable will contain the following modified sequence:

```
TATCGGCCCCAACAGTA
```

★ **Exercise 120:** In this exercise we will analyze the genome sequence of a strain of *Staphylococcus aureus*. The genome can be downloaded from the official NCBI database (accession number `GCF_000698045.1`) as well as from the following location:

> http://www.loucalab.com/archive/BioDataAnalysisPython
> → file "`Staphylococcus_aureus_A69_GCF_000698045.1.fasta`".

Perform the following tasks:

1. After downloading the above fasta file, load its entire contents into a single string variable (e.g., as shown in Section 4.5), and use a regular expression to remove all metadata lines, i.e., lines starting with a `>` character. Report the total number of remaining characters.

   *Hint: You can use the function* `sub` *in the* `re` *package, as described in Section 5.8.2, to remove anything between a* `>` *character and a newline character (*`\n`*).*

2. Count the total number of nucleotides, for example using the string member function `count` discussed in Section 4.7.2.

   *Hint: Make sure to not count the newline characters remaining in the string.*

3. Compute the total number of known nucleotides (`A`, `C`, `G` and `T`), as well as the fraction of known nucleotides (i.e., the ratio of known over total nucleotides).

4. Compute the fraction of known nucleotides that are either guanines (`G`) or cytosines (`C`). This fraction is commonly known as the GC-content of the genome, and is an important property in systematics, ecology and molecular biology [126–128].

# 6 Appendix: Useful specialized `python` functions

The following is a list of personal favorite specialized functions that often come in handy in practice. To use them you will need to look up their corresponding user manual and examples online, but it is already useful to simply know that these functions exist.

`system`
Execute simple shell commands (i.e., normally ran in the terminal or command line) from within python.

`requests.get`
Download a file over HTTP from the internet and store its contents in a `python` string. For example, the following code will download the mitochondrial genome sequence of the fruit fly *Neoceratitis asiatica* from the NCBI database, and store it in the `genome` variable:

```
import requests
URL = "https://www.ebi.ac.uk/ena/browser/api/fasta/MF434829.1?download=true"
genome = requests.get(URL).text
```

Of course, you could have just downloaded the file using your web browser. But what if you wanted to download 10,000 mitochondrial genomes? A simple loop, calling `requests.get` with each iteration, will generally be more efficient.

`time.strftime`
Get the current date and/or time as a string. For example, the code:

```
print(time.strftime("%Y.%m.%d") + " " + time.strftime("%H:%M:%S"))
```

prints the current date (year.month.day) and time (hours:minutes:seconds), as follows:

```
2022.10.07 14:57:51
```

## References


[1] Noble, D., 2002. The rise of computational biology. Nature Reviews Molecular Cell Biology 3:459–463.

[2] Hufsky, F., Lamkiewicz, K., Almeida, A., Aouacheria, A., Arighi, C., Bateman, A., Baumbach, J., Beerenwinkel, N., Brandt, C., Cacciabue, M., Chuguransky, S., Drechsel, O., Finn, R.D., Fritz, A., Fuchs, S., Hattab, G., Hauschild, A.C., Heider, D., Hoffmann, M., Hölzer, M., Hoops, S., Kaderali, L., Kalvari, I., von Kleist, M., Kmiecinski, R., Kühnert, D., Lasso, G., Libin, P., List, M., Löchel, H.F., Martin, M.J., Martin, R., Matschinske, J., McHardy, A.C., Mendes, P., Mistry, J., Navratil, V., Nawrocki, E.P., O'Toole, Á.N., Ontiveros-Palacios, N., Petrov, A.I., Rangel-Pineros, G., Redaschi, N., Reimering, S., Reinert, K., Reyes, A., Richardson, L., Robertson, D.L., Sadegh, S., Singer, J.B., Theys, K., Upton, C., Welzel, M., Williams, L., Marz, M., 2020. Computational strategies to combat COVID-19: useful tools to accelerate SARS-CoV-2 and coronavirus research. Briefings in Bioinformatics .

[3] Marx, V., 2013. The big challenges of big data. Nature 498:255–260.

[4] Truskinger, A., Cottman-Fields, M., Eichinski, P., Towsey, M., Roe, P., 2014. Practical analysis of big acoustic sensor data for environmental monitoring, in: 2014 IEEE Fourth International Conference on Big Data and Cloud Computing, pp. 91–98.

[5] Dolinski, K., Troyanskaya, O.G., 2015. Implications of big data for cell biology. Molecular Biology of the Cell 26:2575–2578.

[6] Chi, M., Plaza, A., Benediktsson, J.A., Sun, Z., Shen, J., Zhu, Y., 2016. Big data for remote sensing: Challenges and opportunities. Proceedings of the IEEE 104:2207–2219.

[7] López-López, P., 2016. Individual-based tracking systems in ornithology: Welcome to the era of big data. Ardeola 63:103–136.

[8] Hahn, A.S., Konwar, K., Louca, S., Hanson, N.W., Hallam, S.J., 2016. The information science of microbial ecology. Current Opinion in Microbiology 31:209–216.

[9] Lewis, K.P., Vander Wal, E., Fifield, D.A., 2018. Wildlife biology, big data, and reproducible research. Wildlife Society Bulletin 42:172–179.

[10] Pal, S., Mondal, S., Das, G., Khatua, S., Ghosh, Z., 2020. Big data in biology: The hope and present-day challenges in it. Gene Reports 21:100869.

[11] Kowarski, K.A., Moors-Murphy, H., 2021. A review of big data analysis methods for baleen whale passive acoustic monitoring. Marine Mammal Science 37:652–673.

[12] Andrej, B., P., H.T., Charlotte, A., Enrique, B.V., Iris, E.G., Oscar, R.E., C., S.A., T., C.S., Roland, B., L., S.C., 2017. Insights from the genome sequence of *Mycobacterium lepraemurium*: Massive gene decay and reductive evolution. mBio 8:e01283–17.

[13] Kowalski, N.L., Dale, R.H.I., Mazur, C.L.H., 2010. A survey of the management and development of captive African elephant (*Loxodonta africana*) calves: birth to three months of age. Zoo Biology 29:104–119.

[14] Holmes, S., 2003. Bootstrapping phylogenetic trees: Theory and methods. Statistical Science 18:241–255.

[15] Ullah, M., Wolkenhauer, O., 2010. Stochastic approaches in systems biology. WIREs Systems Biology and Medicine 2:385–397.

[16] Kroese, D.P., Brereton, T., Taimre, T., Botev, Z.I., 2014. Why the Monte Carlo method is so important today. WIREs Computational Statistics 6:386–392.

[17] Fieberg, J.R., Vitense, K., Johnson, D.H., Gray, A., 2020. Resampling-based methods for biologists. PeerJ 8:e9089.

[18] Blattner, F.R., Plunkett, G., Bloch, C.A., Perna, N.T., Burland, V., Riley, M., Collado-Vides, J., Glasner, J.D., Rode, C.K., Mayhew, G.F., Gregor, J., Davis, N.W., Kirkpatrick, H.A., Goeden, M.A., Rose, D.J., Mau, B., Shao, Y., 1997. The complete genome sequence of *Escherichia coli* K-12. Science 277:1453–1462.

[19] Hatton, I.A., Dobson, A.P., Storch, D., Galbraith, E.D., Loreau, M., 2019. Linking scaling laws across eukaryotes. Proceedings of the National Academy of Sciences 116:21616–21622.

[20] Sadegh, N.M., Anh, N., Margaret, K., Alexandra, S., S., P.M., Craig, P., Jeff, C., 2018. Automatically identifying, counting, and describing wild animals in camera-trap images with deep learning. Proceedings of the National Academy of Sciences 115:E5716–E5725.

[21] Schneider, S., Greenberg, S., Taylor, G.W., Kremer, S.C., 2020. Three critical factors affecting automated image species recognition performance for camera traps. Ecology and Evolution 10:3503–3517.

[22] Whytock, R.C., Świeżewski, J., Zwerts, J.A., Bara-Słupski, T., Koumba Pambo, A.F., Rogala, M., Bahaa-el din, L., Boekee, K., Brittain, S., Cardoso, A.W., et al., 2021. Robust ecological analysis of camera trap data labelled by a machine learning model. Methods in Ecology and Evolution 12:1080–1092.

[23] Gibbons, W.J., Andrews, K.M., 2004. PIT tagging: Simple technology at its best. BioScience 54:447–454.

[24] Catarinucci, L., Colella, R., Mainetti, L., Mighali, V., Patrono, L., Sergi, I., Tarricone, L., 2012. An innovative animals tracking system based on passive UHF RFID technology, in: SoftCOM 2012, 20th International Conference on Software, Telecommunications and Computer Networks, pp. 1–7.

[25] Catarinucci, L., Colella, R., Mainetti, L., Patrono, L., Pieretti, S., Secco, A., Sergi, I., 2014. An animal tracking system for behavior analysis using radio frequency identification. Lab Animal 43:321–327.

[26] Sinha, S., 2017. The Fibonacci numbers and its amazing applications. International Journal of Engineering Science Invention 6:7–14.

[27] Ghose, M., Bahadur, B., 2019. Fibonacci sequence: A general account, in: Asymmetry in Plants. CRC Press, pp. 193–218.

[28] Livio, M., 2008. The golden ratio: The story of phi, the world's most astonishing number. Crown.

[29] Iosa, M., Morone, G., Paolucci, S., 2018. Phi in physiology, psychology and biomechanics: The golden ratio between myth and science. Biosystems 165:31–39.

[30] Bergeron, F., Reutenauer, C., 2019. Golden ratio and phyllotaxis, a clear mathematical link. Journal of Mathematical Biology 78:1–19.

[31] O'Leary, N.A., Wright, M.W., Brister, J.R., Ciufo, S., Haddad, D., McVeigh, R., Rajput, B., Robbertse, B., Smith-White, B., Ako-Adjei, D., Astashyn, A., Badretdin, A., Bao, Y., Blinkova, O., Brover, V., Chetvernin, V., Choi, J., Cox, E., Ermolaeva, O., Farrell, C.M., Goldfarb, T., Gupta, T., Haft, D., Hatcher, E., Hlavina, W., Joardar, V.S., Kodali, V.K., Li, W., Maglott, D., Masterson, P., McGarvey, K.M., Murphy, M.R., O'Neill, K., Pujar, S., Rangwala, S.H., Rausch, D., Riddick, L.D., Schoch, C., Shkeda, A., Storz, S.S., Sun, H., Thibaud-Nissen, F., Tolstoy, I., Tully, R.E., Vatsan, A.R., Wallin, C., Webb, D., Wu, W., Landrum, M.J., Kimchi, A., Tatusova, T., DiCuccio, M., Kitts, P., Murphy, T.D., Pruitt, K.D., 2016. Reference sequence (RefSeq) database at NCBI: current status, taxonomic expansion, and functional annotation. Nucleic Acids Research 44:D733.

[32] Karl, D.M., Lukas, R., 1996. The hawaii ocean time-series (hot) program: Background, rationale and field implementation. Deep Sea Research Part II: Topical Studies in Oceanography 43:129–156.

[33] Karl, D.M., Church, M.J., 2014. Microbial oceanography and the hawaii ocean time-series programme. Nature Reviews Microbiology 12:699–713.

[34] Smith, R.J., Cheverud, J.M., 2002. Scaling of sexual dimorphism in body mass: A phylogenetic analysis of Rensch's rule in primates. International Journal of Primatology 23:1095–1135.

[35] Poland, H., 1892. Fur-bearing Animals in Nature and in Commerce. Gurney & Jackson.

[36] Krebs, C.J., 2011. Of lemmings and snowshoe hares: the ecology of northern Canada. Proceedings of the Royal Society of London B: Biological Sciences 278:481–489.

[37] Slatkin, M., 1984. Ecological causes of sexual dimorphism. Evolution 38:622–630.

[38] Shine, R., 1989. Ecological causes for the evolution of sexual dimorphism: a review of the evidence. The Quarterly Review of Biology 64:419–461.

[39] Owens, I.P., Hartley, I.R., 1998. Sexual dimorphism in birds: why are there so many different forms of dimorphism? Proceedings of the Royal Society of London. Series B: Biological Sciences 265:397–407.

[40] Gosler, A.G., Johnson, J.A., Newton, I., Burnham, W.A., Burnham, K.K., 2012. The history and range expansion of peregrine falcons in the Thule area, Northwest Greenland. volume 353. Museum Tusculanum Press.

[41] Ghani, A.C., Donnelly, C.A., Cox, D.R., Griffin, J.T., Fraser, C., Lam, T.H., Ho, L.M., Chan, W.S., Anderson, R.M., Hedley, A.J., Leung, G.M., 2005. Methods for estimating the case fatality ratio for a novel, emerging infectiousdisease. American Journal of Epidemiology 162:479–486.

[42] Luo, G., Zhang, X., Zheng, H., He, D., 2021. Infection fatality ratio and case fatality ratio of covid-19. International Journal of Infectious Diseases 113:43–46.

[43] Ofiţeru, I.D., Lunn, M., Curtis, T.P., Wells, G.F., Criddle, C.S., Francis, C.A., Sloan, W.T., 2010. Combined niche and neutral effects in a microbial wastewater treatment community. Proceedings of the National Academy of Sciences 107:15345–15350.

[44] Dodge, S., Bohrer, G., Weinzierl, R., Davidson, S.C., Kays, R., Douglas, D., Cruz, S., Han, J., Brandes, D., Wikelski, M., 2013. The environmental-data automated track annotation (Env-DATA) system: linking animal tracks with environmental data. Movement Ecology 1:3.

[45] Louca, S., Jacques, S.M.S., Pires, A.P.F., Leal, J.S., Srivastava, D.S., Parfrey, L.W., Farjalla, V.F., Doebeli, M., 2016a. High taxonomic variability despite stable functional structure across microbial communities. Nature Ecology & Evolution 1:0015.

[46] Brown, J., Gillooly, J., Allen, A., Savage, V., West, G., 2004. Toward a metabolic theory of ecology. Ecology 85:1771–1789.

[47] Kanda, M., Fujiwara, N., Xu, X., Shindo, K., Nagamine, T., Ikeda, A., Shibasaki, H., 1996. Pain-related and cognitive components of somatosensory evoked potentials following CO2 laser stimulation in man. Electroencephalography and Clinical Neurophysiology/Evoked Potentials Section 100:105–114.

[48] Smith, C.H., 1983. Spatial trends in canadian snowshoe hare, *Lepus americanus*, population cycles. Canadian Field-Naturalist 97:151–160.

[49] Sinclair, A.R., Gosline, J.M., Holdsworth, G., Krebs, C.J., Boutin, S., Smith, J.N., Boonstra, R., Dale, M., 1993. Can the solar cycle and climate synchronize the snowshoe hare cycle in Canada? Evidence from tree rings and ice cores. American Naturalist :173–198.

[50] Krebs, C.J., Boutin, S., Boonstra, R., Sinclair, A., Smith, J., Dale, M.R., Martin, K., Turkington, R., 1995. Impact of food and predation on the snowshoe hare cycle. Science 269:1112–1115.

[51] Monod, J., 1942. Recherches sur la croissance des cultures bacteriennes. Hermann & cie, Paris.

[52] Healey, F., 1980. Slope of the Monod equation as an indicator of advantage in nutrient competition. Microbial Ecology 5:281–286.

[53] Owens, J., Legan, J., 1987. Determination of the Monod substrate saturation constant for microbial growth. FEMS Microbiology Letters 46:419–432.

[54] Wicht, H., 1996. A model for predicting nitrous oxide production during denitrification in activated sludge. Water Science and Technology 34:99–106.

[55] Ahring, B.K., Westermann, P., 1984. Isolation and characterization of a thermophilic, acetate-utilizing methanogenic bacterium. FEMS Microbiology Letters 25:47–52.

[56] Ahring, B.K., Westermann, P., 1987. Kinetics of butyrate, acetate, and hydrogen metabolism in a thermophilic, anaerobic, butyrate-degrading triculture. Applied and Environmental Microbiology 53:434–439.

[57] Louca, S., 2021a. Dynamical Modeling in Biology. Self-published and free.

[58] Weisberg, S., 2005. Applied Linear Regression. Wiley Series in Probability and Statistics, Wiley.

[59] Welham, S., Gezan, S., Clark, S., Mead, A., 2014. Statistical Methods in Biology: Design and Analysis of Experiments and Regression. Taylor & Francis. 1 edition.

[60] Tobias, J.A., Sheard, C., Pigot, A.L., Devenish, A.J.M., Yang, J., Sayol, F., Neate-Clegg, M.H.C., Alioravainen, N., Weeks, T.L., Barber, R.A., Walkden, P.A., MacGregor, H.E.A., Jones, S.E.I., Vincent, C., Phillips, A.G., Marples, N.M., Montaño-Centellas, F.A., Leandro-Silva, V., Claramunt, S., Darski, B., Freeman, B.G., Bregman, T.P., Cooney, C.R., Hughes, E.C., Capp, E.J.R., Varley, Z., Friedman, N.R., Korntheuer, H., Corrales-Vargas, A., Trisos, C.H., Weeks, B.C., Hanz, D.M., Töpfer, T., Bravo, G.A., Remeš, V., Nowak, L.,

Carneiro, L.S., Moncada R., A.J., Matysioková, B., Baldassarre, D.T., Martínez-Salinas, A., Wolfe, J.D., Chapman, P.M., Daly, B.G., Sorensen, M.C., Neu, A., Ford, M.A., Mayhew, R.J., Fabio Silveira, L., Kelly, D.J., Annorbah, N.N.D., Pollock, H.S., Grabowska-Zhang, A.M., McEntee, J.P., Carlos T. Gonzalez, J., Meneses, C.G., Muñoz, M.C., Powell, L.L., Jamie, G.A., Matthews, T.J., Johnson, O., Brito, G.R.R., Zyskowski, K., Crates, R., Harvey, M.G., Jurado Zevallos, M., Hosner, P.A., Bradfer-Lawrence, T., Maley, J.M., Stiles, F.G., Lima, H.S., Provost, K.L., Chibesa, M., Mashao, M., Howard, J.T., Mlamba, E., Chua, M.A.H., Li, B., Gómez, M.I., García, N.C., Päckert, M., Fuchs, J., Ali, J.R., Derryberry, E.P., Carlson, M.L., Urriza, R.C., Brzeski, K.E., Prawiradilaga, D.M., Rayner, M.J., Miller, E.T., Bowie, R.C.K., Lafontaine, R.M., Scofield, R.P., Lou, Y., Somarathna, L., Lepage, D., Illif, M., Neuschulz, E.L., Templin, M., Dehling, D.M., Cooper, J.C., Pauwels, O.S.G., Analuddin, K., Fjeldså, J., Seddon, N., Sweet, P.R., DeClerck, F.A.J., Naka, L.N., Brawn, J.D., Aleixo, A., Böhning-Gaese, K., Rahbek, C., Fritz, S.A., Thomas, G.H., Schleuning, M., 2022. AVONET: morphological, ecological and geographical data for all birds. Ecology Letters 25:581–597.

[61] West, G.B., Brown, J.H., Enquist, B.J., 1997. A general model for the origin of allometric scaling laws in biology. Science 276:122–126.

[62] Molina, N., van Nimwegen, E., 2008. The evolution of domain-content in bacterial genomes. Biology Direct 3:51.

[63] Yurlov, A.K., Yurlova, N.I., Garyushkina, M.Y., Selivanova, M.A., Doi, H., 2022. Long-term observation of the egg and chick size in the nests of *Larus ichthyaetus* in Lake Chany, Russia. Scientific Data 9:372.

[64] Pottier, P., Lin, H.Y., Oh, R.R.Y., Pollo, P., Rivera-Villanueva, A.N., Valdebenito, J., Yang, Y., Amano, T., Burke, S., Drobniak, S.M., Nakagawa, S., 2022. A comprehensive database of amphibian heat tolerance. Scientific Data 9:600.

[65] Nina, S., Karen, G., K., M.T., Gavin, S., Lucy, T., Stanley, F., 2000. A whole-genome microarray reveals genetic diversity among *Helicobacter pylori* strains. Proceedings of the National Academy of Sciences 97:14668–14673.

[66] Welch, R.A., Burland, V., Plunkett, G., Redford, P., Roesch, P., Rasko, D., Buckles, E.L., Liou, S.R., Boutin, A., Hackett, J., Stroud, D., Mayhew, G.F., Rose, D.J., Zhou, S., Schwartz, D.C., Perna, N.T., Mobley, H.L.T., Donnenberg, M.S., Blattner, F.R., 2002. Extensive mosaic structure revealed by the complete genome sequence of uropathogenic *Escherichia coli*. Proceedings of the National Academy of Sciences 99:17020–17024.

[67] Konstantinidis, K.T., Tiedje, J.M., 2005. Genomic insights that advance the species definition for prokaryotes. Proceedings of the National Academy of Sciences 102:2567–2572.

[68] Kettunen, J., Ravaja, N., Keltikangas-Järvinen, L., 2000. Smoothing facilitates the detection of coupled responses in psychophysiological time series. Journal of Psychophysiology 14:1–10.

[69] Mann, M.E., 2008. Smoothing of climate time series revisited. Geophysical Research Letters 35:L16708.

[70] Rojo, J., Rivero, R., Romero-Morte, J., Fernández-González, F., Pérez-Badia, R., 2017. Modeling pollen time series using seasonal-trend decomposition procedure based on loess smoothing. International Journal of Biometeorology 61:335–348.

[71] Roques, L., Klein, E.K., Papaïx, J., Sar, A., Soubeyrand, S., 2020. Using early data to estimate the actual infection fatality ratio from COVID-19 in France. Biology 9:97.

[72] Silva, M.A., Prieto, R., Jonsen, I., Baumgartner, M.F., Santos, R.S., 2013. North Atlantic blue and fin whales suspend their spring migration to forage in middle latitudes: building up energy reserves for the journey? PloS one 8:e76507.

[73] Beal, J., Farny, N.G., Haddock-Angelli, T., Selvarajah, V., Baldwin, G.S., Buckley-Taylor, R., Gershater, M., Kiga, D., Marken, J., Sanchania, V., Sison, A., Workman, C.T., Pehlivan, M., Roige, B.B., Aarnio, T., Kivisto, S., Koski, J., Lehtonen, L., Pezzutto, D., Rautanen, P., Bian, W., Hu, Z., Liu, Z., Liu, Z., Ma, L., Pan, L., Qin, Z., Wang, H., Wang, X., Xu, H., Xu, X., El Moubayed, Y., Dong, S., Fang, C., He, H., He, H., Huang, F., Shi, R., Tang, C., Tang, C., Xu, S., Yan, C., Bartzoka, N., Kanata, E., Kapsokefalou, M., Katopodi, X.L., Kostadima, E., Kostopoulos, I.V., Kotzastratis, S., Koutelidakis, A.E., Krokos, V., Litsa, M., Ntekas, I., Spatharas, P., Tsitsilonis, O.E., Zerva, A., Annem, V., Cone, E., Elias, N., Gupta, S., Lam, K., Tutuianu, A., Mishler, D.M., Toro, B., Akinfenwa, A., Burns, F., Herbert, H., Jones, M., Laun, S., Morrison, S., Smith, Z., Peng, Z., Ziwei, Z., Deng, R., Huang, Y., Li, T., Ma, Y., Shen, Z., Wang, C., Wang, Y., Zhao, T., Lang, Y., Liang, Y., Wang, X., Wu, Y., Aizik, D., Angel, S., Farhi, E., Keidar, N., Oser, E., Pasi, M., Kalinowski, J., Otto, M., Ruhnau, J., Cubukcu, H., Hoskan, M.A., Senyuz, I., Chi, J., Sauter, A.P., Simona, M.F., Byun, S., Cho, S., Kim, G., Lee, Y., Lim, S., Yang, H., Xin, T., Yaxi, Z., Zhao, P., Han, W., He, F., He, Y., Li, N., Luo, X., Boxuan, C., Jiaqi, H., Liangjian, Y., Wanji, L., Xinguang, C., Xinyu, L., Wu, Z., Xi, Y., Yang, X., Yang, Y., Yang, Z., Zhang, Y., Zhou, Y., Peng, Y., Yadi, L., Yang, S., Yuanxu, J., Zhang, K., Abraham, D., Heger, T., Leach, C., Lorch, K., Luo, L., Gaudi, A., Ho, A., Huang, M., Kim, C., Kugathasan, L., Lam, K., Pan, C., Qi, A., Yan, C., Schaaf, K., Sillner, C., Coates, R., Elliott, H., Heath, E., McShane, E., Parry, G., Tariq, A., Thomas, S., Chen, C.W., Cheng, Y.H., Hsu, C.W., Liao, C.H., Liu, W.T., Tang, Y.C., Tang, Y.H., Yang, Z.E., Jian, L., Li, C., Lin, C., Ran, G., Run, Z., Ting, W., Yong, Z., Yu, L., Lind, A.C., Norberg, A., Olmin, A., Sjolin, J., Torell, A., Trivellin, C., Zorrilla, F., Vries, P.G.d., Cheng, H., Peng, J., Xiong, Z., Altarawneh, D., Amir, S.S., Hassan, S., Vincent, A., Costa, B., Gallegos, I., Hale, M., Sonnier, M., Whalen, K., Elikan, M., Kim, S., You, J., Rambhatla, R., Viswanathan, A., Tian, H., Xu, H., Zhang, W., Zhou, S., Jiamiao, L., Jiaqi, X., Craw, D., Goetz, M., Rettedal, N., Yarbrough, H., Ahlgren, C., Guadagnino, B., Guenther, J., Huynh, J., He, Z., Liu, H., Liu, Y., Qu, M., Song, L., Yang, C., Yang, J., Yin, X., Zhang, Y., Zhou, J., Zi, L., Jinyu, Z., Kang, X., Xilei, P., Xue, H., Xun, S., Babu, P., Dogra, A., Thokachichu, P., Faurdal, D., Jensen, J.H., Mejlsted, J., Nielsen, L., Rasmussen, T., Denter, J., Husnatter, K., Longo, Y., Luzuriaga, J.C., Moncayo, E., Moreira, N.T., Tapia, J., Dingyue, T., Jingjing, Z., Wenhao, X., Xinyu, T., Xiujing, H., DeKloe, J., Astles, B., Baronaite, U., Grazulyte, I., iGEM Interlab Study Contributors, Aachen, Aalto-Helsinki, AHUT_China, Aix-Marseille, ASTWS-China, Athens, Austin_LASA, Austin_UTexas, Baltimore_BioCrew, BCU, BFSUICC-China, BGIC-Global, BGU_Israel, Bielefeld-CeBiTec, Bilkent-UNAMBG, BioIQS-Barcelona, BioMarvel, BIT, BIT-China, BJRS_China, BNDS_CHINA, BNU-China, BOKU-Vienna, BostonU, British_Columbia, Calgary, Cardiff_Wales, CCU_Taiwan, CDHSU-CHINA, Chalmers-Gothenburg, CIEI-BJ, CMUQ, CO_Mines, ColumbiaNYC, Cornell, CPU_CHINA, CSU_CHINA, CSU_Fort_Collins, Delgado-Ivy-Marin, DLUT_China, DLUT_China_B, DNHS_SanDiego, DTU-Denmark, Duesseldorf, Ecuador, ECUST, Edinburgh_OG, Edinburgh_UG, Emory, 2020. Robust estimation of bacterial cell count from optical density. Communications Biology 3:512.

[74] McGroddy, M.E., Daufresne, T., Hedin, L.O., 2004. Scaling of c:n:p stoichiometry in forests worldwide: implications of terrestrial redfield-type ratios. Ecology 85:2390–2401.

[75] Branch, T., Butterworth, D., 2001. Estimates of abundance south of 60ˆ os for cetacean species sighted frequently on the 1978/79 to 1997/98 iwc/idcr-sower sighting surveys. Journal of Cetacean Research and Management 3:251–270.

[76] Cassie, R.M., 1962. Frequency distribution models in the ecology of plankton and other organisms. The Journal of Animal Ecology 31:65–92.

[77] Shoemaker, W.R., Locey, K.J., Lennon, J.T., 2017. A macroecological theory of microbial biodiversity. Nature Ecology & Evolution 1:0107.

[78] Koepper, S., Scott-Tibbetts, S., Lavallée, J., Revie, C.W., Thakur, K.K., 2022. Fisheries dataset on moulting patterns and shell quality of American lobsters *H. americanus* in Atlantic Canada. Scientific Data 9:385.

[79] Marino, S., Baxter, N.T., Huffnagle, G.B., Petrosino, J.F., Schloss, P.D., 2014. Mathematical modeling of primary succession of murine intestinal microbiota. Proceedings of the National Academy of Sciences 111:439–444.

[80] Mysara, M., Vandamme, P., Props, R., Kerckhof, F.M., Leys, N., Boon, N., Raes, J., Monsieurs, P., 2017. Reconciliation between operational taxonomic units and species boundaries. FEMS Microbiology Ecology 93:fix029.

[81] Louca, S., Mazel, F., Doebeli, M., Parfrey, W.L., 2019. A census-based estimate of Earth's bacterial and archaeal diversity. PLOS Biology 17:e3000106.

[82] Efron, B., 1982. The jackknife, the bootstrap and other resampling plans. SIAM.

[83] Efron, B., Tibshirani, R., 1986. Bootstrap methods for standard errors, confidence intervals, and other measures of statistical accuracy. Statistical Science 1:54–75.

[84] Farine, D.R., Carter, G.G., 2022. Permutation tests for hypothesis testing with animal social network data: Problems and potential solutions. Methods in Ecology and Evolution 13:144–156.

[85] Wilkinson, D.J., 2009. Stochastic modelling for quantitative description of heterogeneous biological systems. Nature Reviews Genetics 10:122–133.

[86] Maarleveld, T.R., Olivier, B.G., Bruggeman, F.J., 2013. Stochpy: A comprehensive, user-friendly tool for simulating stochastic biological processes. PLOS ONE 8:e79345.

[87] Wilkinson, D.J., 2018. Stochastic modelling for systems biology. Chapman and Hall/CRC.

[88] Larget, B., Simon, D.L., 1999. Markov chain monte carlo algorithms for the bayesian analysis of phylogenetic trees. Molecular Biology and Evolution 16:750–759.

[89] Silvestro, D., Schnitzler, J., Zizka, G., 2011. A bayesian framework to estimate diversification rates and their variation through time and space. BMC Evolutionary Biology 11:311.

[90] Hartig, F., Calabrese, J.M., Reineking, B., Wiegand, T., Huth, A., 2011. Statistical inference for stochastic simulation models – theory and application. Ecology Letters 14:816–827.

[91] Maddison, W.P., 1991. Squared-change parsimony reconstructions of ancestral states for continuous-valued characters on a phylogenetic tree. Systematic Biology 40:304–314.

[92] Bailey, N., 1995. Statistical Methods in Biology. Cambridge low price editions, Cambridge University Press.

[93] Jaynes, E., Jaynes, E., Bretthorst, G., Press, C.U., 2003. Probability Theory: The Logic of Science. Cambridge University Press.

[94] Robinson, D.H., Wainer, H., 2002. On the past and future of null hypothesis significance testing. The Journal of Wildlife Management 66:263–271.

[95] Mogie, M., 2004. In support of null hypothesis significance testing. Proceedings of the Royal Society of London. Series B: Biological Sciences 271:S82–S84.

[96] Steele, G., Vigna, S., 2021. Computationally easy, spectrally good multipliers for congruential pseudorandom number generators. arXiv .

[97] L'ecuyer, P., 1999. Tables of linear congruential generators of different sizes and good lattice structure. Mathematics of Computation 68:249–260.

[98] Martiny, J.B.H., Bohannan, B.J., Brown, J.H., Colwell, R.K., Fuhrman, J.A., Green, J.L., Horner-Devine, M.C., Kane, M., Krumins, J.A., Kuske, C.R., et al., 2006. Microbial biogeography: putting microorganisms on the map. Nature Reviews Microbiology 4:102–112.

[99] Louca, S., Parfrey, L.W., Doebeli, M., 2016b. Decoupling function and taxonomy in the global ocean microbiome. Science 353:1272–1277.

[100] Diniz-Filho, J.A.F., Soares, T.N., Lima, J.S., Dobrovolski, R., Landeiro, V.L., Telles, M.P.d.C., Rangel, T.F., Bini, L.M., 2013. Mantel test in population genetics. Genetics and molecular biology 36:475–485.

[101] Louca, S., 2021b. The rates of global bacterial and archaeal dispersal. ISME Journal 16:159–167.

[102] Velásquez, N.A., Marambio, J., Brunetti, E., Méndez, M.A., Vásquez, R.A., Penna, M., 2013. Bioacoustic and genetic divergence in a frog with a wide geographical distribution. Biological Journal of the Linnean Society 110:142–155.

[103] Watts, H.E., Holekamp, K.E., 2007. Hyena societies. Current Biology 17:R657–R660.

[104] Hemerik, J., Goeman, J., 2014. Exact testing with random permutations. arXiv preprint arXiv:1411.7565 .

[105] Anderson, M.J., 2017. Permutational Multivariate Analysis of Variance (PERMANOVA), in: Balakrishnan, N., Colton, T., Everitt, B., Piegorsch, W., Ruggeri, F., Teugels, J. (Eds.), Wiley StatsRef: Statistics Reference Online. John Wiley & Sons, pp. 1–15.

[106] Kalyuzhny, M., 2020. Null models for community dynamics: Beware of the cyclic shift algorithm. Global Ecology and Biogeography 29:1085–1093.

[107] Mira, A., Ochman, H., Moran, N.A., 2001. Deletional bias and the evolution of bacterial genomes. Trends in Genetics 17:589–596.

[108] Moran, N.A., McLaughlin, H.J., Sorek, R., 2009. The dynamics and time scale of ongoing genomic erosion in symbiotic bacteria. Science 323:379–382.

[109] McCutcheon, J.P., Moran, N.A., 2012. Extreme genome reduction in symbiotic bacteria. Nature Reviews Microbiology 10:13–26.

[110] Wald, A., Wolfowitz, J., 1940. On a test whether two samples are from the same population. The Annals of Mathematical Statistics 11:147–162.

[111] Berger, V.W., Zhou, Y., 2014. Kolmogorov–Smirnov test: Overview, in: Encyclopedia of Statistics in Behavioral Science. American Cancer Society.

[112] MacFarland, T.W., Yates, J.M., 2016. Mann–Whitney U Test, in: MacFarland, T.W., Yates, J.M. (Eds.), Introduction to Nonparametric Statistics for the Biological Sciences Using R. Springer International Publishing, Cham, pp. 103–132.

[113] Carter, J., Finn, J.T., 1999. MOAB: a spatially explicit, individual-based expert system for creating animal foraging models. Ecological Modelling 119:29–41.

[114] Tang, W., Bennett, D.A., 2010. Agent-based modeling of animal movement: A review. Geography Compass 4:682–700.

[115] McLane, A.J., Semeniuk, C., McDermid, G.J., Marceau, D.J., 2011. The role of agent-based models in wildlife ecology and management. Ecological Modelling 222:1544–1556.

[116] Louca, S., Jacques, S.M.S., Pires, A.P.F., Leal, J.S., González, A.L., Doebeli, M., Farjalla, V.F., 2017. Functional structure of the bromeliad tank microbiome is strongly shaped by local geochemical conditions. Environmental Microbiology 19:3132–3151.

[117] Berg, M., Koskella, B., 2018. Nutrient- and dose-dependent microbiome-mediated protection against a plant pathogen. Current Biology 28:2487–2492.e3.

[118] Taylor, G.T., Muller-Karger, F., Thunell, R.C., Scranton, M., Astor, Y., Varela, R., Troccoli-Ghinaglia, L., Lorenzoni, L., Fanning, K.A., Hameed, S., et al., 2012. Ecosystem response to global climate change in the southern Caribbean Sea. Proceedings of the National Academy of Sciences 109:19–315.

[119] Thompson, L.R., Sanders, J.G., McDonald, D., Amir, A., Ladau, J., Locey, K.J., Prill, R.J., Tripathi, A., Gibbons, S.M., Ackermann, G., Navas-Molina, J.A., Janssen, S., Kopylova, E., Vázquez-Baeza, Y., González, A., Morton, J.T., Mirarab, S., Zech Xu, Z., Jiang, L., Haroon, M.F., Kanbar, J., Zhu, Q., Jin Song, S., Kosciolek, T., Bokulich, N.A., Lefler, J., Brislawn, C.J., Humphrey, G., Owens, S.M., Hampton-Marcell, J., Berg-Lyons, D., McKenzie, V., Fierer, N., Fuhrman, J.A., Clauset, A., Stevens, R.L., Shade, A., Pollard, K.S., Goodwin, K.D., Jansson, J.K., Gilbert, J.A., Knight, R., Consortium, T.E.M.P., 2017. A communal catalogue reveals Earth's multiscale microbial diversity. Nature 551:457–463.

[120] Yang, H., Huang, X., Fang, S., Xin, W., Huang, L., Chen, C., 2016. Uncovering the composition of microbial community structure and metagenomics among three gut locations in pigs with distinct fatness. Scientific Reports 6:27427.

[121] Zeb, F., Wu, X., Chen, L., Fatima, S., ul Haq, I., Chen, A., Xu, C., Jianglei, R., Feng, Q., Li, M., 2020. Time-restricted feeding is associated with changes in human gut microbiota related to nutrient intake. Nutrition 78:110797.

[122] Gregory Alvord, W., Roayaei, J.A., Quiñones, O.A., Schneider, K.T., 2007. A microarray analysis for differential gene expression in the soybean genome using bioconductor and r. Briefings in Bioinformatics 8:415–431.

[123] Ovaska, K., Laakso, M., Hautaniemi, S., 2008. Fast gene ontology based clustering for microarray experiments. BioData Mining 1:11.

[124] Brunskill, E.W., Aronow, B.J., Georgas, K., Rumballe, B., Valerius, M.T., Aronow, J., Kaimal, V., Jegga, A.G., Grimmond, S., McMahon, A.P., Patterson, L.T., Little, M.H., Potter, S.S., 2008. Atlas of gene expression in the developing kidney at microanatomic resolution. Developmental Cell 15:781–791.

[125] Andreopoulos, B., An, A., Wang, X., Schroeder, M., 2009. A roadmap of clustering algorithms: finding a match for a biomedical application. Briefings in Bioinformatics 10:297–314.

[126] Oliver, J., Marín, A., 1996. A relationship between GC content and coding-sequence length. Journal of Molecular Evolution 43:216–223.

[127] Hildebrand, F., Meyer, A., Eyre-Walker, A., 2010. Evidence of selection upon genomic gc-content in bacteria. PLOS Genetics 6:e1001107.

[128] Šmarda, P., Bureš, P., Horová, L., Leitch, I.J., Mucina, L., Pacini, E., Tichý, L., Grulich, V., Rotreklová, O., 2014. Ecological and evolutionary significance of genomic gc content diversity in monocots. Proceedings of the National Academy of Sciences 111:E4096–E4102.

# Index