Programming and Bioinformatics

Rory Jansen

Contents

Before you begin

This chapter serves to get the practicalities out of the way so you can start programming.

Install Python

TODO: update

In this course, we use a distribution of Python called *Anaconda*. Anaconda is simply an easy way of installing Python on Windows, macOS (Mac), and Linux.

To install Anaconda, head to this site. Click "Download" And select the graphical installer for your platform. Make sure to select **Python 3.7 version** (do *not* download version 2.7).

When the download has completed, you should follow the platform specific instructions:

- **For Windows:** Double-click the file you just downloaded and follow the instructions on the screen. Make a default installation. The installer will also ask you if you want to download and install a program called Visual Studio Code. Do that too.
- **For OSX:** Double-click the file you just downloaded and follow the instructions on the screen. Make a default installation. The installer will also ask you if you want to download and install a program called Visual Studio Code. Do that too.

Install the text editor

You will also need a *text editor*. A text editor is where you write your Python code and for this course, we will use one called *Sublime Text*. To install *Sublime Text*, go to this site. Again, you'll have to download the version of Sublime Text that fits your platform. At the top of the page, click **Windows 64 bit** if you're running Windows and **OSX** if you are on a Mac.

- **For Windows:** Double-click the `.exe` file you just downloaded and follow the instructions on the screen.

- **For OSX:** Double click the `.dmg` file you just downloaded. A small window should open showing you two icons: *Applications* and *Sublime Text*. Drag the *Sublime Text* icon onto the *Applications* icon. You can now close the window.

You should now have *Sublime Text* installed. If you open *Sublime Text*, you should see something like fig. 0.1:

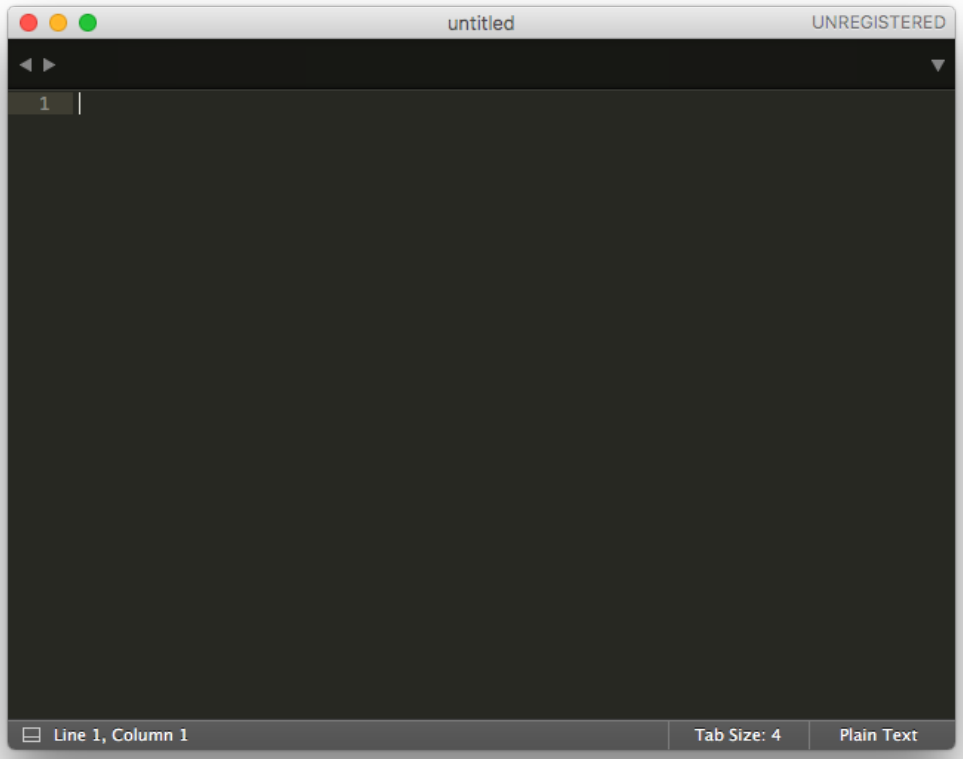


Figure 0.1: Sublime Text, screenshot

The terminal

The last thing you need is a tool to make Python run the programs you write. Fortunately, that is already installed. On OSX and Linux this is a program called *Terminal*. You can find it by typing "Terminal" in Spotlight Search. On Windows it is called the *Anaconda Prompt* and was installed along with Anaconda Python. You should be able to find it from Start menu.

What is *Anaconda Prompt* and this *Terminal* thing, you ask! Both programs are what we

call *terminal emulators*. They are programs used to run other programs, such as the ones you are going to write. I will informally refer to both *Terminal* and *Anaconda Prompt* as "the terminal" So if I write something like "open the terminal" later in the book you should open *Anaconda Prompt* if you are running Windows. If you are running OS X, you should open the *Terminal* application.

If you open *Anaconda Prompt* you should see something like fig. **??**. The OSX Terminal looks like fig. 0.3, possibly with a different background color.

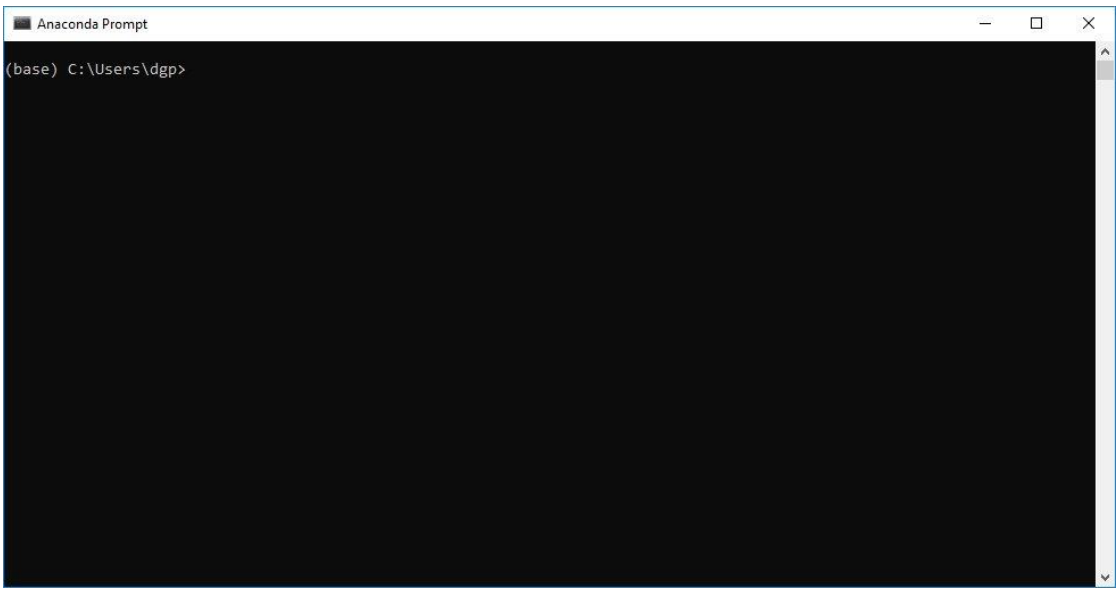


Figure 0.2: Screenshot of Anaconda Prompt

The terminal is a very useful tool. However, to use it you need to know a few basics. First of all, a terminal lets you execute commands on your computer. You simply type the command you want and then hit enter. The place where you type is called a prompt and it may look a little different depending on which terminal emulator you use. In this book we represent the prompt with the character `$`.

When you open the terminal you'll be located in a folder. You can see which folder you are in by typing `pwd` on OSX and `cd` on windows, and then press `Enter` on the keyboard. When you press `Enter` you tell the terminal to execute the command you just wrote. In this case, the command you typed simply tells you the path of the folder we are in. If I do it I get:

```
$ pwd
/Users/kasper/programming
```

If I had been on a windows machine it would have looked something like:

```
$ cd
```

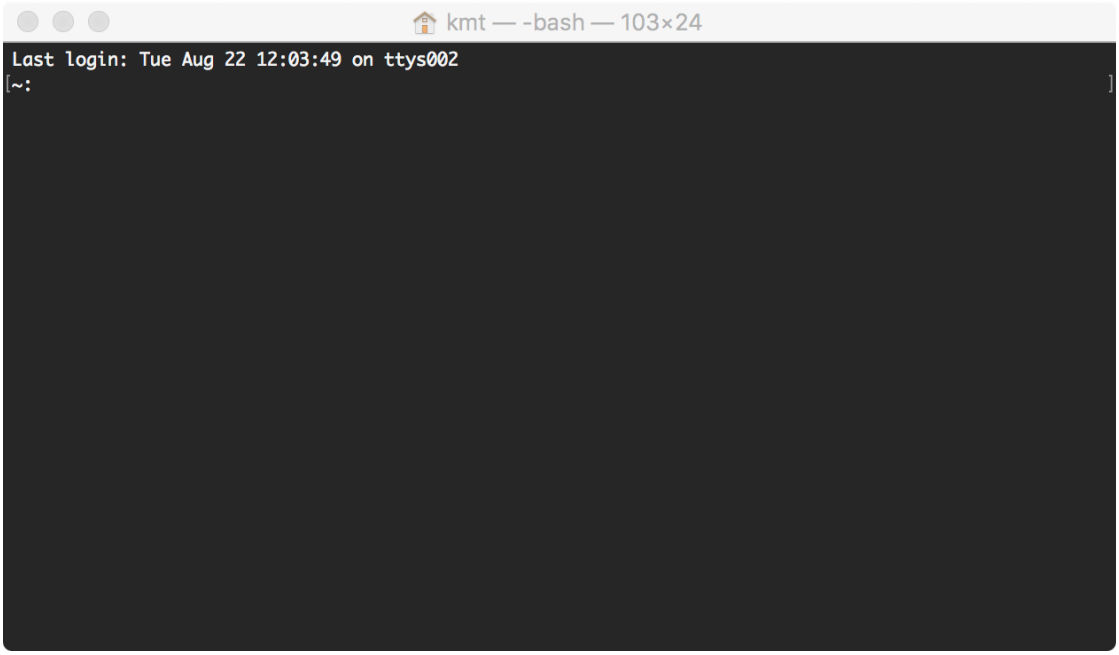


Figure 0.3: Screenshot of Terminal

```
C:\Users\kasper\programming
```

So right now I am in the folder `programming`. `/Users/kasper/programming` is the "full address" of the folder with dashes (or backslashes on windows) separating folders. So `programming` is a subfolder of `kasper` which is a subfolder of `Users`. That way you not only know which folder you are in but also where that folder is. Let us see what is in this folder. On OSX you type the `ls` command (l as in Lima and s as in Sierra). On windows you type `dir`. When I do that and press `Enter` I get:

```
$ ls
notes projects
```

It seems that there are two other folders, one called `notes` and another called `projects`. If you are curious about what's inside `notes` you can "walk" into the folder with the `cd` command. To use this command you must specify which folder we want to walk into (in this case `notes`). We do this by typing `cd`, then a space and the then name of the folder. This is the same OSX and Windows. When I press enter I get:

```
$ cd notes
$
```

It seems that nothing really happened, but if I run the `pwd` command (`cd` on Windows) now to see which folder I am in, I get:

```
$ pwd
```

```
/Users/kasper/programming/notes
```

Just to keep track of what is happening: before we ran the `cd` command we were in the directory `/Users/kasper/programming` folder, and now we're in `/Users/kasper/programming/notes`. This means that we can now use the `ls` command (`dir` on Windows) to see what is in the `notes` folder:

```
$ ls
$
```

Again it seems like nothing happened. Well, `ls` and `dir` do not show anything if the folder we are in is empty. So `notes` must be empty. Let us go back to where we came from. To walk "back" or "up" to `/Users/kasper/programming` we again use the `cd` command, but this time we do not tell it to go to a specific folder. Instead, we use the special name `..` to say that we wish to go to the parent folder called `programming`, i.e. the folder we just came from:

```
$ cd ..
$ pwd
/Users/kasper/programming
```

Now when we run the `pwd` (or `cd`) command we see that we are back where we started. Let us see if the two folders are still there:

```
$ ls
notes projects
```

They are!

Hopefully, you are now able to use navigate your folders and see what is in them. You will need this later to go to the folders with the code you write for the exercises and projects during the course.

Action	Windows	OSX
Show current folder	`cd`	`pwd`
List folder content	`dir`	`ls`
Go to subfolder "notes"	`cd notes`	`cd notes`
Go to parent folder	`cd ..`	`cd ..`

You are all set

Well done! You are all set to start the course. Have a cup of coffee and look forward to your first program. While you sip your coffee I need to you take an oath (one of three related to this course). Raise your right hand! (put your coffee in the left).

Oath 1: I swear never to copy and paste examples from this book into *Sublime Text*. I will *always* read the examples in the book and and *type* them into my editor.

This serves three purposes (as if one was not enough):

1. You will be fully aware of each and every bits of each example.
2. You will learn to write code correctly and without omissions and mistakes.
3. You will get Python "into your fingers". It sounds silly, but it *will* get into your fingers.

Appendix: Conda and PyMol

In bioinformatics, we install packages and programs for use in our analyses and pipelines. Sometimes, however, the versions of packages you need for one project conflicts with the versions you need for other projects that you work on in parallel. Such conflicts seem like an unsolvable problem. Would it not be fantastic if you could create a small world, insulated from the rest of your Anaconda installation. Then that small world could only contain the packages you needed for a single project. If each project had its own isolated world, then there would be no such conflicts. Fortunately, there is a tool that lets you do just that, and its name is Conda. The small worlds that Conda creates are called "environments," and you can create as many as you like, and then switch between them as you switch between your bioinformatics projects.

Conda also downloads and installs the packages for you and makes sure that the packages you install in each environment are compatible. It even makes sure that packages needed by packages (dependencies) are installed. Conda is truly awesome.

> Conda is an open source package management system and environment management system for installing multiple versions of software packages and their dependencies and switching easily between them.

Creating an environment with PyMol

When you install Anaconda, Conda makes a single base environment for you. It is called "base" and this is why it says "(base)" on your terminal.

Many of you take the course "Biomolecular Structure and Function" (let us call that BSF19) alongside this course. In BSF19, you need to install programs (e.g. PyMol) that may conflict with the packages you need for this course. So we will create an isolated Conda environment for BSF19 to avoid such conflicts. Conda is a program you run from the command line, just like `python` or `cd`. So open your terminal (i.e., the "Terminal" program if you are on a Mac and the "Anaconda Prompt" program if you are on Windows). Now copy/paste this line into the terminal and press return (enter):

```
conda create -n BSF19 -c schrodinger pymol python=3.7
```

This command runs the Conda program and tells it to create a new environment with name "BSF19" and to install pymol and python in that environment. Once you hit enter

Conda works for some time and then writes a long list of packages in your terminal. These are all the packages and dependencies required by python and PyMol in versions that all fit together. It looks something like this:

```
  readline           pkgs/main/osx-64::readline-7.0-h1de35cc_5
  requests           pkgs/main/osx-64::requests-2.22.0-py37_0
  rigimol            schrodinger/osx-64::rigimol-1.3-2
  setuptools         pkgs/main/osx-64::setuptools-41.0.1-py37_0
  sip                pkgs/main/osx-64::sip-4.19.8-py37h0a44026_0
  six                pkgs/main/osx-64::six-1.12.0-py37_0
  sqlite             pkgs/main/osx-64::sqlite-3.29.0-ha441bb4_0
  tk                 schrodinger/osx-64::tk-8.6.9-x11tk0_2000
  urllib3            pkgs/main/osx-64::urllib3-1.24.2-py37_0
  wheel              pkgs/main/osx-64::wheel-0.33.4-py37_0
  xz                 pkgs/main/osx-64::xz-5.2.4-h1de35cc_4
  zlib               pkgs/main/osx-64::zlib-1.2.11-h1de35cc_3
  zstd               pkgs/main/osx-64::zstd-1.3.7-h5bba6e5_0


Proceed ([y]/n)?
```

Just hit enter again to Proceed. Once you do that, Conda starts to download and install all the packages. It takes a bit of time.

At the end Conda writes:

```
Preparing transaction: done
Verifying transaction: done
Executing transaction: done
#
# To activate this environment, use
#
#     $ conda activate BSF19
#
# To deactivate an active environment, use
#
#     $ conda deactivate
```

Conda tells you both how to enter (activate it) an environment and how to leave it (deactivate it). To activate your BSF19 environment, you enter:

```
conda activate BSF19
```

and press enter. Voila, you are now in the BSF19 environment. Notice how the command prompt changed from "(base)" to "(BSF19)" to show that you are now in the BSF19 environment. To run PyMol that you installed in this environment, just type

```
pymol
```

and hit enter.

Now try to close PyMol. Then go back to your terminal and type

```
conda deactivate
```

Notice how it now again says "(base)" on your command prompt. That is because you are back in your base environment. Try to type `pymol` (and hit enter), you terminal will tell you that it could not find anything called `pymol`. This is the way it should be. That is because PyMols is installed in the BSF19 environment, *not* in the base environment. It illustrates how the base environment is entirely separate from the BSF19 environment you just made.

Starting PyMol

From now on, you can start PyMol by typing these commands into the terminal (Anaconda Prompt on Windows):

```
conda activate BSF19
```

(hit enter)

```
pymol
```

(hit enter)

Writing a program

Heads up: Before you head into this chapter and the rest of the book, it is essential that you have installed Python 3.7 (not Python 2.9), that you have installed the *Sublime Text* editor, and acquainted yourself with navigating a terminal. If not, go back to the previous chapter for instructions.

Hello World

Dive in and make your first program. Start by creating a new file in your editor (*Sublime Text*) and save it as `hello.py`. The `.py` file suffix tells your editor that this file contains Python code. As you will see, this makes your life a whole lot easier. Such a file with Python code is usually called a *script*, but we also call it a program.

Now write *exactly* this in the file (`hello.py`):

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

The editor colors your code a little differently, but that is not important. Save your file with the added code, and you have your first program! Of course, there is not much point in having a program if it just sits there on your computer. To run your program, do the following:

1. Open the terminal and navigate to the folder (directory) where you saved `hello.py` using the `cd` command. If you do not know how to do that, go back and read the previous chapter again.
2. Type `python hello.py` in the terminal and hit enter.

You should see something like fig. 0.1.

This is where you shout "it's alive!", toss your head back and do the insane scientist laugh.

Okay, what just happened? You wrote a program by creating a file and writing one line of code in it. Then you ran the program using Python and it wrote (printed) `Hello world` in the terminal. Do not worry about the parentheses and quotes for now and just enjoy your new life as a programmer.

Maybe you wonder why we write `print` and not `write` or something else? That actually

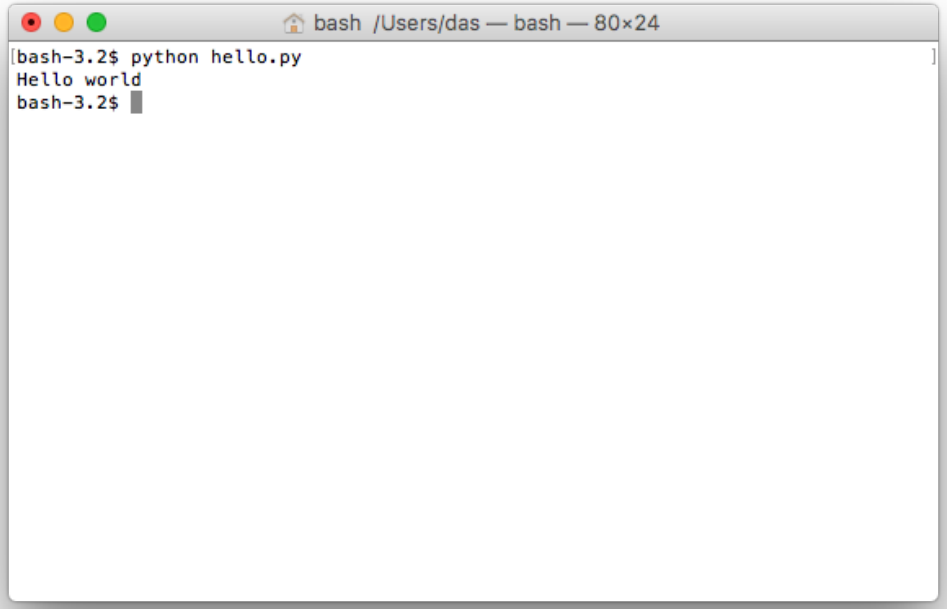


Figure 0.4: Hello world, first program

goes all the way back to the days when computers were big clunky things with no screens attached. They could only interact with the user by *printing* out things on a real physical paper printer. Back then, the output you see on the screen was printed onto a piece of paper that the programmer could then look at. These days print shows up in the terminal, but the story should help you remember that print sends text *out* of your program.

Now try to add another line of code like this:

```
print("Hello world")
print("I am your first program")
```

Save the file and run it again by typing `python hello.py` in your terminal and hit Enter.

You should see this:

```
$ python hello.py
Hello world
I am your first program
```

Now your program first prints `Hello world` and *then* it prints `I am your first program`. The *then*-part is important. That is how a Python program works. Python (the `python` you write in front of `hello.py`) reads your `hello.py` file and then executes the code, one line after the other until it reaches the end of the file.

This is essential, so read that paragraph again. Now read it once more. It may seem trivial, but it is absolutely fundamental always to remember that this is how Python runs your code. So here is oath 2:

> **Oath 2:** I swear always to remember that each line of code in my script is executed one after the other, starting from the first line in the script and ending at the last line.

When you write Python code, you always follow this workflow:

1. Change the code in the file.
2. Save the file.
3. Execute the code in the file using the terminal.
4. Start over.

Make sure you get the hang of this in the following exercises.

Important: The examples and exercises in this course are designed to work if you execute your scripts from the folder they are stored in. So you must navigate into the relevant folder before you execute your script. If your script is called `hello.py`, you must *always* execute it exactly like this: `python hello.py`. On some computers it is possible to just type `hello.py` without `python` in front of it. Do *not* do that. Do *not* "drag" the script file into the Terminal either.

Exercise 1
Try to swap the two lines of code in the file and run the program again. What does it show now?

Exercise 2
Try to make the program print a greeting to yourself. Something like this:

```
$ python hello.py
Hello Sarah!
```

-- if your name is Sarah, of course.

Exercise 3
Add more lines of code to your program to make it print something else. Can you make your program print the same thing ten times?

Error Messages

Did you get everything just right with your first program or did you get error messages when you executed your code with `python`? Maybe you wrote the following code (adding an extra closing parenthesis):

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

-- and then got an error like this:

```
$ python hello.py
  File "hello.py", line 1
    print('Hello world'))
                         ^
SyntaxError: invalid syntax
```

This is Python's way of telling you that the `hello.py` script has an error in line 1. If you write something that does not conform to the proper syntax for Python code then you will get a `SyntaxError`. Python will do its best to figure out where the problem is and point to it with a ^ character.

You will see many such error messages in your new life as a programmer, and it is important that you practice how to read them. At first, they will be hard to decipher, but once you get used to them, they will help you quickly identify where the problem is. If there is an error message that you do not understand the internet is your friend. Just paste the error message into Googles search field, and you will see that you are not the only one out there getting started on Python programming.

It is okay if you do not know how to fix the problem right now, but it is essential to remember that these error messages are Pythons way of helping you understand what you did wrong. You are entering a world of pain if you do not try to read and understand them every time.

Exercise 4

Try to break your new shiny program and make it give an error message when you run it. An easy way of doing this is to remove or change random characters from the program. Which error messages do you see? Do you see the same error message every time, or are they different? Try googling the error messages you get. Can you figure out why the change you made broke the program? How many different error messages can you produce?

Strings

You have used strings a lot in your first program. In programs, text values are called *strings*. A string is simply a piece of text, but we call it a string because it is a "string of characters". In Python, we represent a string like this:

```
"this is a string"
```

or like this:

```
'Hello world'
```

That is, we take the text we wish to use as our value, and we put quotes around it. We are allowed to use either double quotes (the first example) or single quotes (the second example). We can mix them like this:

```
print('this is "some text" with a quotes')
```

-- but not like this:

```
print("this is a broken string')
```

Can you see why, and how it is handy to have both single and double quotes? Well, if we did not have both, we could not have text with quotes in it. You just need to remember to use the same kind of quotes at each end of the string. Running the latter example gives an error message telling you that Python cannot find the quote that was supposed to end the string:

```
  File "broken.py", line 1
    print("this is a broken string')
                                    ^
SyntaxError: EOL while scanning string literal
```

Comments

You have already seen that Python reads and executes one line of code at a time until your program has no more lines of code.

However, we can make a line invisible to Python by putting a # symbol in front of it, like this:

```
# print("Hello world")
print("Greetings from your first program")
```

When you do that, Python simply does not read that line. It is not part of the program. Run the program again. You will notice that now only the second line is part of your program:

```
$ python hello.py
Greetings from your first program
```

This is useful in two ways:

1. It lets you disable certain lines of your code, by keeping Python from reading them. For example, you may want to see what happens if that line of code was not executed to understand how your program works.
2. It allows you to write notes in your code Python to help you remember how your code works.

Lines with a # in front are called "comments" because we normally use them to write comments about the code we write. If you ask for help about some problem, you will often hear your instructor say: try to "comment out" line 2. When your instructor says that, it simply means that you should add a # in front of line two to see what then happens.

Exercise 5

What happens if I put a # in the middle of a line of code? Try it out!

Exercise 6

Try this:

```
# Note to self: the lines below print stuff
print("Hello world")
print("Greetings from your first program")
```

and this:

```
print("Hello world") # actually, everything after a # is ignored
print("Greetings from your first program")
```

Exercise 7

Now try this:

```
print("Hello # world")
print("Greetings from your first program")
```

and this:

```
print("Hello world"#)
print("Greetings from your first program")
```

Did you expect the last example to work? Why? Why not? Which parts of each line are considered part of the program?

Dealing with values

This chapter is about values and variables, the two most central concepts in programming.

Math

Much programming is done to compute stuff. In Python the usual math operations are done using these arithmetic operators:

Operator	Operation
`+`	plus
`-`	minus
`/`	division
`//`	integer division
`*`	multiplication
`**`	exponentiation
`%`	modulo (remainder)

You are probably quite familiar with these -- except perhaps for integer division, exponentiation and modulo. Let us take some of the operators for a spin. Remember to carefully write the whole thing in an empty file in *Sublime Text*. Do *not* copy-paste. Then save the file as `mathandlogic.py` and run it from the terminal.

Note! Do not call your file `math.py`. It may bite you later. Just trust me on that one.

```
print("Four times two is",  4 * 2)
```

Notice how you can print more than one thing at a time if you put commas between the values you want to print? We can group computations using parentheses, just like in normal math. Try this:

```
print("10 / (2 + 3) is", 10 / (2 + 3))
print("(10 / 2) + 3 is", (10 / 2) + 3)
```

In addition to the regular math operators, there are a few extra operators that we call *comparison operators* because they are used to compare two values, e.g. two numbers.

Operator	Operation
<	less-than
>	greater-than
<=	less-than-or-equal
>=	greater-than-or-equal
==	equal
!=	not equal

Try this:

```
print("Is 5 greater than -2?", 5 > -2)
print("Is 5 greater or equal to -2?", 5 >= -2)
print("Is 5 less or equal to -2?", 5 <= -2)
print("Is 5 less than 7 - 2?", 5 < 7 - 2)
print("Is 5 equal to 7 - 2?", 5 == 7 - 2)
```

As you may have noticed running this code, comparing things using these operators we always produce either `True` or `False`. E.g. the following

```
print(5 < 7)
```

prints the value `True`, because 5 is actually smaller than 7. 'True' and 'False' are special values in Python that we can use (and print if we like) just like any other Python value:

```
print(True)
print(False)
```

Exercise 8

Try to write and run the code below. Compare each line to what is printed when you run ht code and make sure you understand why.

```
print("I have", 25 + 30 / 6, "of someting")
print("I have", 100 - 25 * 3 % 4, "of something else")

print("Is it true that 3 + 2 < 5 - 7?")
print(3 + 2.1 < 5.4 - 7)

print("3 + 2.1 is", 3 + 2.1)
```

```
print("5.4 - 7 is", 5.4 - 7)

print("Oh, that's why it's False.")
```

Exercise 9

Say the supermarket has chocolate bars for 2.80 kr. Write a small Python program that prints how many chocolate bars you can get for your 30 kr. For example, it could output something like this:

```
$ python chocolate.py
I can buy 10.7142857143 chocolate bars!
```

Exercise 10

We mentioned a special operator called *modulo*. Google it you do not remember what it does. How about *integer division*. Explain both to a fellow student, or to yourself out loud.

Exercise 11

You obviously cannot go buy 10.71 chocolate bars in a store. You will have to settle for 10. Can you change the program you made in exercise 9 to print the number of bars you can actually buy, and the change you then have left? Consider using the modulo and integer division operators.

Exercise 12

What happens if you try to run the following program?

```
print("What happens next?", 1 / 0)
```

If you get an error, why do you think you get that error? Does it make sense?

Exercise 13

You have probably heard of the Pythagorean theorem for computing the hypotenuse (the longest side) of a right-angled triangle. The Pythagorean theorem looks like this:

$$a^2 + b^2 = c^2$$

Here a is the length of the hypotenuse and b and c are the lengths of the two legs of the triangle. So if we have a triangle where $a = 5$ and $b = 2$ and we want to find c^2 we can do this in Python:

```
print("The squared length of the hypotenuse is:", 5**2 + 2**2)
```

Exercise 14

However, we are rarely interested in the *squared* length of the hypotenuse. Can you modify the code you wrote in exercise 13 so you compute c instead of c^2? Taking the square root of a number is the same as taking that number and exponentiating it to 0.5, so the square root of x is $x^{0.5}$. Do you know of a Python operator that does exponentiation?

Logic

Now you have done some math and even used Python to compare two numbers. However, there are three operators left that are special. These are the logical operators: `and`, `or` and `not`. Using these operators we can express more elaborate True/False statements than with the comparison operators alone.

Exercise 15

Go through the code below and see if you can figure out what each line does. Then write the code into your editor and run it to see what actually happens.

```
print(2 < 3)
print(10 < 12)
print(8 > 100)
print(2 < 3 and 10 < 12)
print(8 > 100 and 2 < 3)
print(8 > 100 or 2 < 3)
print(not 8 > 100 and 2 < 3)
print(not 8 > 100 and not 2 > 3)
```

Did it do what you expected? Can you explain each line?

Exercise 16

When exposed to the operators `and`, `or` and `not`, some values are considered true and others are considered false. What happens when you put `not` in front of someting that is considered true or false? Decide what you think and why before you write the code and try it out.

```
print(not True)
print(not False)
print(not 0)
print(not -4)
print(not 0.0000000)
print(not 3.14159265359)
print(not "apple")
print(not "")
```

From the code above, try to find out which values are values Python considers true and which it considers false.

Exercise 17

The logical operator `and` takes two values (the one to the left of the operator and the one to the right) and figures out whether *both* the left and the right expression is true. It actually boils down to this:

Left expression	Right expression	Result
`True`	`True`	`True`
`True`	`False`	`False`
`False`	`True`	`False`
`False`	`False`	`False`

Write some code to confirm that the table above is correct using Python. For example, to test the first case, do this:

```
print(True and True)
```

Exercise 18

Python will not do any more work than absolutely necessary to find out if a logical expression is true or not. That means that, if the value left of `and` is considered false by Python, then there is no reason look at the right value, since it is already established that they are not *both* considered true. In this case the expression reduces to the *left* value. I.e. `False and True` reduces to `False`.

If the value left of `and` is considered *true* by Python, then Python needs to look at the right value too to establish if they are *both* considered true. In this case the expression reduces to the *right* value. I.e. `True and False` reduces to `False`.

Use the same rationale to explain to yourself how the two last combinations in exercise 17 are evaluated.

Exercise 19

Like the `and` operator, the `or` operator also takes two values. However, the `or` operator tries to figure out whether *one* of the two values are true. Thus, the `or` operator boils down to this:

Left expression	Right expression	Result
`True`	`True`	`True`
`True`	`False`	`True`
`False`	`True`	`True`
`False`	`False`	`False`

Write some code to confirm that the table above is correct using Python. For example, to

test the first case, do this:

```
print(True or True)
```

Exercise 20

As with the `and` operator, Python will not do any more work than absolutely necessary when evaluating an expression with 'or'. So if the value left of `or` is considered true by Python, then there is no reason look at the right value, since it is already established that at least *one* of them are considered true. In this case the expression reduces to the *left* value. I.e. `True or False` reduces to `True`.

If the value left of `or` is considered *false* by Python, then Python stil needs to look at the right value to establish if at least one of them are considered true. In this case the expression reduces to the *right* value. I.e. `False or True` reduces to `True`.

Use the same rationale to explain to yourself how the two last combinations in exercise 19 are evaluated.

Exercise 21

Remember what you learned in {#sec:trueish_falseish} about which values are considered true and which are considered false. Combine that with what you learned in exercise 17 and exercise 19 about what logical expressions reduce to and see if you can figure out what is printed below and why. Decide what you think before you write the code and try it out.

```
print(True and 4)
print(0 and 7)
print(-27 and 0.5)
print(42 and 0)
print("apple" and "orange")
print("apple" and "")
print(42 or 0)
print("apple" and "")
print("apple" or "")
```

If you where surprised what was printed, maybe go back and have a look at exercise 17 and exercise 19 again.

Exercise 22

As you can see, logic also works on strings, and there is even an additional comparison operator that tests if something is a part of something else. That operator is called `in`. You can use it to test if one string is part of another string. Try this to figure out how it works:

```
print("Hell" in "Hello world")
print("Hello world" in "Hello world")
```

```
print("Hello world" in "Hell")
print("lo wo" in "Hello world")
print("Artichoke" in "Hello world")
```

Exercise 23
There is also an operator called `not in`. I guess you can imagine what that tests. Try it out.

Variables

> **Heads up:** By now you probably feel the first signs of brain-overload. If you do not take breaks between chapters your brain may overheat and explode -- we have seen that happen. One of the nice things about the brain is that it works when you rest. Archiving and understanding a lot of new information takes time, and force-feeding your brain will not help. The last part of this chapter is very important so now might be a good time for a good long break.

This section is about *variables* and this is where the fun begins. A variable is basically a way of assigning a name to a value. `8700000` is just a value, but if we assign a name to it then it gets a special meaning:

```
number_of_species = 8700000
print(number_of_species)
```

In this case, the variable `number_of_species` represents the estimated number of eukaryotic species on the planet, which is 8700000. So `8700000` is the *value* and `number_of_species` is the variable name. Write the code above into a file and run it. Notice how this lets us refer to the *value* using the *variable name*. What appears in the terminal when you do that? do you see `number_of_species` or `8700000`?

As you can see in the small program above, one of two different things happens when a variable name appears in Python code:

- **Assignment:** A value is assigned to the variable. This is what happens in the first line where `number_of_species` is *assigned* the value `8700000`.
- **Substitution:** The variable is substituted for its value. This is what happens in the second line where Python *substitutes* the variable name `number_of_species` for its value `8700000` and then prints that.

That is basically it, but let us take the example a bit further and create another variable that we assign the value `1200000` to. That is the number of species discovered so far. Now, lets add this to the program and use it to compute the number of species we have yet to identify. Start by reading the code below super carefully. Remember that a variable is *either* assigned a value or replaced with the value it represents. For each

occurrence of the variables below, determine if they are being assigned a value or if they are substituted for their value.

```
number_of_species = 8700000
number_discovered = 1200000
number_unidentified = number_of_species - number_discovered
print(number_unidentified)
```

Now write the code into a file and run it. Take some time to let it sink in that variables are extremely useful for two reasons:

1. Variables give meaning to a value. Without the variable name, the value 1200000 could just as well be the number of people in the city of Prague. However, by giving the value a meaningful name, it becomes clear what the value is meant to represent.
2. We can assign new values to variables (that is why they are called *variables*). That way we can change the value of `number_discovered` as new species are discovered.

Your variable names can be pretty much anything, but they have to start with a letter or an underscore (_) and the rest of the name has to be either letters, numbers or underscores. Just to be clear: a space is *not* any of those things, so do not use spaces in variable names. Above all, be careful in your choice of variable names. Variable names are case sensitive, meaning that `count` and `Count` are two different variables. Stick to lower case variable names. That makes your code easier to read.

Exercise 24

For each occurrence of the variables below, determine if they are being assigned a value or if they are substituted for their value.

```
breeding_birds = 4
print(breeding_birds)
breeding_birds = 5
print(breeding_birds)
```

Exercise 25

For each occurrence of the variables below, determine if they are being assigned a value or if they are substituted for their value.

```
breeding_birds = 4
print(breeding_birds)
breeding_birds = breeding_birds + 1
print(breeding_birds)
```

Exercise 26

What happens if you take the first example in this section swap the two lines? So going from this:

```
number_of_species = 8700000
print(number_of_species)
```

to this:

```
print(number_of_species)
number_of_species = 8700000
```

Explain to yourself what happens in each line of each version of the program. What kind of error do you get and why? Remember Oath 2!

Exercise 27

Write the following code in a file, save it and run it.

```
income = 45000
taxpercentage = 0.43
tax_amount = tax_percentage * income
income_after_tax = income - tax_amount
print('Income after tax is', income_after_tax)
```

You should get an error that looks at lot like this one:

```
Traceback (most recent call last):
  File "tax.py", line 3, in <module>
    tax_amount = tax_percentage * income
NameError: name 'tax_percentage' is not defined
```

It says that the error is on line 3. Can you figure out what is wrong? Hopefully, you will now appreciate how much attention to detail is required when programming. Every tiny, little symbol or character in your code is *essential*.

Exercise 28

With a correlation coefficient 0.99, the number of divorces per 1000 people in Maine USA is almost completely explained by the US per capita consumption of margarine. Not related to programming at all, but worth thinking about.

Year	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009
Divorce rate per 1000	5	4.7	4.6	4.4	4.3	4.1	4.2	4.2	4.2	4.1
Margarine consumption	8.2	7	6.5	5.3	5.2	4	4.6	4.5	4.2	3.7

Different types of values

By now you probably have a pretty good idea about what a value in Python is. So far you have seen text like `'Banana'`, integers like `7` and numbers with a fractional part like `4.25`.

In Python, a text value is a *type* of value called a *string* which Python denotes `str` (abbreviation for "string"). So `'Banana'` is a string, and so is `'Banana split'`. There are two types of numbers in Python. Integers (like 7, 42, and 3) are called `int`. Numbers with a fractional part (like 3.1254 and 4.0) are that are called `float` (abbreviation for "floating-point number").

As I mentioned earlier, `True` and `False` are Python values too. They are called booleans or `bool`, named after an English mathematician called George Boole famous for his work on logic.

So the different *types* of values we know so far are:

Name	Type in Python	Examples
String	`str`	`"hello"`, `'9'`
Integer	`int`	`0`, `2721`, `9`
Floating-point	`float`	`1.0`, `4.4322`
Boolean	`bool`	`True`, `False`
None	`NoneType`	`None`

In case you did not notice, I added a special type at the end that can only have the value `None`. I may sound a little weird, but in programming, we sometimes need a value that is nothing, or `None`. More about that later.

When you do computations in Python it is no problem to mix integers and floating-point numbers. Try this:

```
print("What is 0.5 * 2?", 0.5 * 2)
print("What is 3 / 2?", 3 / 2)
```

As you can see we can also make computations using only integers that result in floating-point numbers. Some of the math operators not only work on numbers, they also work on strings. That way you can add two strings together. It is no longer math of course -- but quite handy.

```
fruit = 'Ba' + 'na' + 'na'
print(fruit)
```

Exercise 29

What do you think is printed here?

```
main_course = 'Duck a la Banana\n'
dessert = 'Banana split\n'
menu = main_course + dessert
print(menu)
```

Can you figure out what the special character `'\n'` represents?

Exercise 30

If you try to combine different types of values in ways that are not allowed in Python, you will get an error. Try the following weird calculations, and read the each error message carefully.

```
x = 3 - '1.5'
print(x)
```

```
x = None - 4
print(x)
```

Exercise 31

Write these two examples and compare the resulting values of `x`

```
x = '9' + '4'
print(x)
```

```
x = 9 + 4
print(x)
```

Exercise 32

Write these two examples. What happens in each case?

```
x = '72' * 3
```

```
x = '72' * '3'
```

Exercise 33

Will this work? Use what you have learned from the other exercises and try to predict what will happen here. Then write the code and try it out.

```
x = 'Ba' + 'na' * 2
print(x)
```

Exercise 34

Sometimes you may need to change a string to a number. You can do that like this:

```
some_value = "42"
other_value = int(some_value)
```

Try to convert between values from and to strings, integers and floats using `int`, `float` as in the example above. You will notice that only meaningful conversions work. E.g. this will not work: `number = int('four')`.

Having completed the above exercises you should take note of the following four important points:

1. All Python values have a type. So far you know about strings, integers, floating-points, and booleans.
2. Math operators let you do cool things like concatenating two strings by adding them together.
3. The flip side of that cool coin is that Python will assume you know what you are doing if you add two strings (`'4' + '4'` is `'44'` *not* `8`) or multiply a string with an integer (`'4' * 4` is `'4444'` *not* `16`).
4. You can change the type of a value. E.g `"4"` to `4` or `1` to `1.0`.
5. Python will throw a `TypeError` if you try to combine types values of values in ways that are not allowed. Remember this!

General exercises

Each chapter in the book ends with a set of general exercises that are ment to give you an opportunity to combine what you have learned so far. In this case, they are meant to train your familiarity with the following topics:

- Strings
- Math
- Logic
- Types of values
- Variables

Exercise 35

What happens if you try to run the following program?

```
print("What happens now?", 1 / )
```

If you get an error, why do you think you get that error?

Exercise 36

What happens if you try to run the following program?

```
print("What happens now?", 1 / 3
```

If you get an error, why do you think you get that error? Can you fix it?

Exercise 37

Determine, for each occurrence of the variable x below, where it is being assigned a value and when it is substituted for its value:

```
x = 1
x = x + 1
x = x + 1
x = x + 1
print(x)
```

Then figure out what is printed and why (remember oath 2). What value does x represent at each occurrence in the code?

Exercise 38

Comparison operators also work with strings. Consider this code:

```
print("apples" == "pears")
```

What is printed here? Write the code and see for yourself once you think you know. If you were wrong, make sure you understand why.

Exercise 39

Consider this code:

```
print('aaaaaa' < 'b')
print('a' < 'b')
print('aa' < 'ab')
print('99' > '100')
print('four bananas' > 'one banana')
```

What is printed here? Write the code and see for yourself once you think you know. By what rule does Python decide if one string is smaller than an other? If you have looked something up in an encyclopedia recently, you may have a clue. Also try to google "ASCII table".

Exercise 40

Consider this code:

```
print('banana' < 'Banana')
```

What is printed here? Write the code and see for yourself once you think you know.

Exercise 41

Do you think it is allowed to use relational operators on values of different types? Try these out and see for yourself:

```
print('Banana' > 4)

print('42' == 43) # this one is dangerous...

print(4 in '1234')
```

Practice reading this kind of error (`TypeError`).

Exercise 42

Can you use the `in` operator to test if this mini gene is part of the DNA string?

```
mini_gene = 'ATGTAG'
dna_string = 'GCTATGTAGGTA'
```

Exercise 43

Say you have two strings `"4"` and `"2"`. What happens if you add them like this: `"4" + "2"`. Can you convert each one to integers so you get `6` when you add them? (have a look at exercise 34 if you do not remember).

Exercise 44

What happens if you run this code? Do you get an error? Do you remember why?

```
1value = 42
```

Exercise 45

What happens if you run this code?

```
print('Hi')
print('Hi')
print('Hi')
```

Compare to what happens when you run this code:

```
print('Hi\nHi\nHi')
```

What does that tell you about what `\n` represents? It also tells you what is added at the end every time you print something.

The order of events

This chapter is about how Python interprets (or evaluates) the code you write. It has a few fancy long words that may seem foreign to you. Do not let that throw you off. They are all just fancy names for something quite simple.

Precedence of Operators

Fear not. Precedence is just a nasty word for something we have already talked about. Precedence simply specifies that some things are done before other things -- or more correctly, that some operations are performed before others. You already know that multiplication is done before addition. Another way of saying that is that multiplication *takes precedence* over addition. The expression below obviously reduces to 7 in two steps:

$$1 + 3 * 2$$

First, 3 * 2 reduces to 6 and then 1 + 6 reduces to 7. If we wanted to add 1 and 3 first we would need to enforce this by adding parentheses:

$$(1 + 3) * 2$$

This is because multiplication operator (`*`) has a higher precedence than the addition operator (`+`). Here is the list of the most common operators and their precedence in Python:

Level	Category	Operators
Highest	exponent	`**`
	positive / negative	`+x`, `-x`
	multiplication	`*`, `/`, `//`, `%`
	addition	`+`, `-`
	relational	`!=`, `==`, `<=`, `>=`, `<`, `>`, `in`, `not in`
	logical	`not`
	logical	`and`
Lowest	logical	`or`

Sometimes a statement contains adjacent operators with the same precedence. In this case Python evaluates the expression left to right. I.e. This expression evaluates to `1.0`:

```
2 / 4 * 2
```

and this evaluates to `4.0`:

```
2 / 2 * 4
```

Exercise 46
Look at each expression in the exercises below and use the table above to decide if it evaluates to `True` or `False`. Then write the code and test if you were right. If not figure out why.

```
2 + 4 * 7 == 2 + (4 * 7)
```

Exercise 47
Does this reduce to `True` or `False`?

```
4 > 3 and 2 < 1 or 7 > 2
```

Exercise 48
Does this reduce to `True` or `False`?

```
4 > 3 and (2 < 1 or 7 > 2)
```

Exercise 49
Does this reduce to `True` or `False`?

```
2 * 4 ** 4 + 1 == (2 * 4) ** (4 + 1)
```

Statements and Expressions

To be able to talk concisely about programming (and to receive more useful help from your instructors) you need a bit of vocabulary. Statements and expressions are two such words that you need to know. Distinguishing between statements and expressions will help us talk about the code we write.

- A **statement** is a line of code that performs an action. Python evaluates each statement in turn until it reaches the end of the file (remember oath 2?). `print(y + 7)` is a statement and so is `x = 14`. They each represent a full line of code and they each perform an action.
- An **expression** is any piece code that reduces to one value. `y + 7` is an expression and so is `4 + 7 * 14 - x` and `4 > 5`.

We will talk more about how expressions are handled by Python in the next section, but right now it is important that you understand that statements *do something* while

expressions are things that reduces to a value. Hopefully, this distinction will be clearer when you have completed the following exercises.

Exercise 50

Did you notice in the above examples that `print(y + 7)` is a statement and `y + 7` is an expression? Yes, expressions can be part of statements. In fact they very often are. Take a look at this code:

```
x = 5
y = 20
z = (x + y) / 2 + 20
print(z * 2 + 1)
h = 2 * x - 9 * 48
print(h)
```

Write down the code on a piece of paper. Now mark all statements and all expressions. Expression are often made up of smaller expressions. E.g. `(x + y)`, `2 + 20`, and `(x + y) / 2 + 20` are all expressions. Discuss with a fellow student. Do you agree?

Exercise 51

Consider the following code:

```
greeting = 'Hello' + ' '
print(greeting, 'You')
```

How many statements are there in this piece of code? How many expressions?

Substitution and Reduction

Although *substitution* and *reduction* may not sound like you new best friends, they truly are! If you remember to think about your Python code in terms of substitution and reduction, then programming will make a lot of sense. Understanding and using these simple rules you will allow to read and understand any expression. If you do not, you may get by for a while -- only to find yourself in big big trouble when things start to get more complicated.

You should remember, from the section on variables in the previous chapter, that variables in Python in are *either* assigned a value *or* substituted for the value they represent.

In the first two lines of code below the variables `x` and `y` are each assigned a value. Now consider the last line in the example:

```
x = 4
y = 3
z = x * y + 8
```

Here `x` is *substituted* by the value `4` and `y` is *substituted* by the value `3`. So now the expression after the equals sign reads `4 * 3 + 8`. Because we multiply before we add, `4 * 3` *reduces* to `12` so that the expression now reads `12 + 8`. Finally, this *reduces* to the value `20`. The very *last* thing that happens is that the variable `z` is assigned the value `20`.

You should do these steps every time you see an expression. You may think that this is overdoing things a bit, but it is *not*. This kind of explicit thinking is what programming is all about and it will become increasingly important in the course, so make sure you make it a habit while it still seems trivial. Then, over time, it will become second nature.

Now raise your right hand and read the third and last oath out loud:

> **Oath 3:** I hereby solemnly swear to consciously consider every single substitution and reduction in every Python expression that I read or write from this moment on.

This was the last of the three oaths but it is the most important one. You can take your hand down now.

> **Heads up!** You may not realize at this point, but the last two subsections are the most important ones in the book. Go back and read them many times as you proceed through the course. If you explicitly think in terms of substitution and reduction you will have no trouble. If you do not, you are entering a mine field with snowshoes on.

Exercise 52

Do the substitution and reduction steps with pen and paper, then run it to check yourself by iserting a `print` statement at the end.

```
x = 7
y = 4 + x
2 + x * x**2 + y - x
```

Exercise 53

Do the substitution and reduction steps with pen and paper, then run it to check yourself by iserting a `print` statement at the end.

```
a = 4
b = a
c = 2
c = a + b + c
```

Exercise 54

Do the substitution and reduction steps with pen and paper, then run it to check yourself by iserting a `print` statement at the end.

```
x = 1
x = x
```

Exercise 55
Do the substitution and reduction steps with pen and paper, then run it to check yourself by iserting a `print` statement at the end.

```
microsatellite = "GTC" * 41
```

Exercise 56
Do the substitution and reduction steps with pen and paper, then run it to check yourself by iserting a `print` statement at the end.

```
mini_gene = "ATG" + "GCG" + "TAA"
```

What did you do first here? Does the order of additions matter?

Exercise 57
Do the substitution and reduction steps with pen and paper, then run it to check yourself by iserting a `print` statement at the end.

```
x = 4
y = x + x
```

Exercise 58
Do the substitution and reduction steps with pen and paper, then run it to check yourself by iserting a `print` statement at the end.

```
x = 4
x = x + 1
```

Exercise 59
Do the substitution and reduction steps with pen and paper, then run it to check yourself by iserting a `print` statement at the end.

```
x = 4
x += 1
```

Compare the final value of `x` to that in exercise 58. Can you see what `+=` is a short hand for? Nifty, right?

General exercises

The following exercises are meant to train your familiarity with the topics we have treated so far -- in this case especially:

- Substitution

- Reduction
- Assignment
- Simple precedence rules
- Comparison operators
- Logical operators
- Distinction between text and numbers

Read each exercise and think hard about the questions before you code anything. *Then* write the code and try it out. Remember that it is crucial that you type it in -- as super boring as it may be (remember oath one). This trains your accuracy and attention to detail and it builds programming into your brain. Play around with each bit of code. Make small changes and see how it behaves.

There is a reason why there are lots of questions in this exercise but no answers. You are supposed to find them yourself -- also if it takes you quite a while. That is the way you build understanding. Some of the questions may seem trivial but do them anyway. If you only understand these concepts superficially they will come back and bite you in the ass when things get more complicated.

Exercise 60

Consider this code:

```
1.2 * 3 + 4 / 5.2
```

What does that expression evaluate to? Try to explicitly make all the substitutions and reductions on a piece of paper before you write and run the code.

Exercise 61

Consider this code:

```
1.2 * (3 + 4) / 5.2
```

What does that expression evaluate to? Try to explicitly make all the substitutions and reductions on a piece of paper before you write and run the code.

Exercise 62

Consider this code:

```
10 % 3 - 2
```

What does that expression evaluate to? Try to explicitly make all the substitutions and reductions on a piece of paper before you write and run the code.

Exercise 63

Consider this code:

```
11 % (7 - 5)**2
```

What does that expression evaluate to? Try to explicitly make all the substitutions and reductions on a piece of paper before you write and run the code.

Exercise 64

Consider this code:

```
a = 5
x = 9
banana = 7
x + 4 * a > banana
```

What does the last expression evaluate to? Try to explicitly make all the substitutions and reductions on a piece of paper before you write the code. What happens if you write and run the code? Why?

Exercise 65

Consider this code:

```
dance = 'can'
dance = dance + dance
print("Do the", dance)
```

What is printed? Try to explicitly make all the substitutions and reductions on a piece of paper before you write the code. What happens if you write and run the code? Why?

Exercise 66

Consider this code:

```
foo = 30
bar = 50
baz = bar + foo
print(baz)
bar = 10
print(baz)
```

There are two print statements. The first print statement prints `80`. But what about the second print statement? Does that print `80` or `40`? Find out and make sure you understand why it prints what it prints.

Exercise 67

Consider this code:

```
1 == '1'
```

and this:

```
1 == 1.0
```

What does this reduce to? Try to print it and see once you think you know. If you were wrong, make sure you figure out why.

Exercise 68

Consider this code:

```
a = '1'
b = '2'
c  = a + b
print(a, b, c)
print(a + b == 3)
```

What is printed here? Write the code and see for yourself once you think you know. If you were wrong, make sure you understand why.

Exercise 69

Consider this code:

```
a = 1
b = 2
c  = a + b
print(a, b, c)
print(a + b == 3)
```

What is printed here? Compare to exercise 68. Write the code and see for yourself once you think you know. If you were wrong, make sure you understand why.

Exercise 70

Consider this code:

```
x = 4
print(x + 2 and 7)
print(x + 2 or 7)
x = -2
print(x + 2 and 7)
print(x + 2 or 7)
```

What is printed here? Write the code and see for yourself once you think you know. If you were wrong, make sure you understand why.

Exercise 71

In the code below I have shuffled the statements. Put them in the right order to make the code print `100`.

```
x = x + 4
print(x)
x = x * 5
```

```
x = x * x
x = 4
```

Exercise 72

In the code below I have shuffled the statements. Put them in the right order to make the code print the string `"Banana"`.

```
y = 'n'
x = 'B' + y + x
print(x)
x = 'a'
y = (x + y) * 2
```

Exercise 73

Remind yourself of the different types of Python values you know. E.g. one of them is integer (`int`). Make a list.

Exercise 74

You already know about several types of data values in Python. Two of them are integers called `int`, and decimal numbers (or floating points) called `float`. When you use an operator like `+` or `>` it produces a value. No matter what you put on either side of `>` in it produces a boolean value (`bool`), `True` or `False`. For other operators what type of value that is produced depends on which values the operator work on. Try this and see if you print an integer or a float (`8` or `8.0`):

```
x = 4
y = 2
result = x * y
print(result)
```

Now try to replace `4` with `4.0`. What type is result now?. Try to also replace `2` with `2.0`. What type is result now? Can you extract a rule for what the `*` operator produces depending on the what types the two values have?

Exercise 75

In exercise 74 you investigated what types of values the `*` operator produce. Redo that exercise with the operators: `+`, `-`, `/`, `**`, `//`, and `%`. What are the rules for what is returned if both values are integers, one values is a float, or both values are floats?

Exercise 76

Make a list of all the operators you know so far in order of precedence (without looking in the notes). Then check yourself.

Exercise 77

What does his expression reduce to?

```
3 > 2
```

Exercise 78

What does his expression reduce to? Do all the reduction steps in your head.

```
2 - 4 * 5 - 2 * 9
```

Exercise 79

What does his expression reduce to? Do all the reduction steps in your head.

```
3 > 2 and 2 - 4 * 5 - 2 * 9
```

Exercise 80

What does his expression reduce to? Do all the reduction steps in your head.

```
0 and 1 or 2
```

Exercise 81

What does his expression reduce to? Do all the reduction steps in your head.

```
4 and 1 or 2
```

Exercise 82

What is the value of `results` once the code below has run?

```
x = 7
y = 13
z = x + y
x = 0
result = x + y + z
```

Exercise 83

What is the value of `results` once the code below has run?

```
x = 5
y = x + 1
x = y + 1
y = x + 1
result = x + y
```

Exercise 84

In the code below I have shuffled the statements. Put them in the right order to make the code print `9`. To do that you must think about which values each variable will in each statement depending on the how you order the statements.

```
x = x + 1
y = 5
y = y - 1
print(y)
x = 1
y = y * x
```

Exercise 85

In the code below I have shuffled the statements. Put them in the right order to make the code print `'Mogens'`

```
c = b
print(c)
a = b + a
b = 'og'
b = c + a
c = 'M'
a = 'ens'
```

Exercise 86

Make three exercises that requires the knowledge of programming so far. Have your fellow students solve them.

Controlling execution

This chapter is about how you make your program do different things under different circumstances. Making functionality dependent on data is what makes programs useful.

If-statement

The small programs you have written so far all run the exact same sequence of statements (lines). Imagine if you could control which statements were run depending on the circumstances. Then you would be able to write more flexible and useful programs. Cue the music - and let me introduce: the "if-statement".

Write the following carefully into a file. It is a small program to monitor bus passenger status. Notice the colon ending the if-statements. Also, note that the lines below each if-statement are indented with exactly four spaces. While you write the, try to figure out what the if-statement does. Then run the code and see what happens.

```
bus_seats = 32
passengers = 20
bags = 20

print(passengers, "people ride the bus")

if bus_seats >= passengers:
    print('Everyone gets to sit down, no complaints')

if bus_seats < passengers:
    print('Some passengers standing, annoyed')

if bus_seats >= passengers + bags:
    print("Smiles, everyone has room for bags")

if bus_seats < passengers / 3:
    print("General dissatisfaction, some swearing")
```

Try to change the values of `bus_seats`, `passengers` and `bags` and see how the program

adapts.

You have probably realized that the if-statements control which print statements that are evaluated. A statement nested under an if-statement is only evaluated if the expression between the `if` keyword and the `:` reduces to `True`. If the expression between the `if` and `:` reduces to a value other than `True` or `False` then Python will interpret zero and empty values (like `0` and `''`) as `False` and all other non-zero and non-empty values as `True`.

Exercise 87

Which of the following letters are printed: A, B, C, D, E, F, G. Make up your mind before you write and run the code.

```
if 0:
    print('A')

if "Banana":
    print('B')

if 3.14159265359:
    print('C')

if False:
    print('D')

if 9 > 5 and 4 < 7:
    print('E')

if '':
    print('F')

if False or "banana":
    print('G')
```

Exercise 88

What happens if you forget to write the `:` in the if-statement?

```
if 4 > 2
    print('Hi!')
```

Exercise 89

What happens if you do not indent the code under the if-statement?

```
if 4 > 2:
print('Hi!')
```

FAQ - Frequently Asked Qustions

Q: Isn't "If" a poem by Rudyard Kipling?
A: Yes.

Else-statement

Sometimes you not only want your program to something if an expression reduces to `True`, you also want it to do something *else* if it is `False`. It is as simple as it looks:

```
cookies = 3

if cookies > 0:
    print("Yum yum, I wonder if we have some milk too...")
else:
    print("WHO HAS TAKEN THE LAST COOKIES!?")
```

Remember to put a `:` after the `else` keyword. Write the code and change the value of `cookies` to `0` -- if you dare.

Exercise 90

Test your understanding about which expressions that reduce to a `True` or `False` value. Write the code below and then see how it responds to different values of `x`. Try to come up with other variations yourself.

```
x = 0.0
# x = '0'
# x = '   '
# x = ''
# x = not 0
# x = 'zero'

if x:
    print('x is substituted with True in the if-statement')
else:
    print('x is substituted with False in the if-statement')
```

FAQ - Frequently Asked Questions

Q: Is "Else" a poem by Rudyard Kipling?
A: No.

Exercise 91

What do you think this code prints? Notice how you can nest if and else-statements under other if and else-statements. This way you can make your program only include

some statements when certain combinations of conditions are met. Just remember that the code below each `if` or `else` is indented by four spaces. Try to change the True/False values of `milk` and `cookies`.

```
milk = False
cookies = True
if milk:
    if cookies:
        status = 'good times'
    else:
        status = 'what ever'
else:
    if cookies:
        status = 'sad times'
    else:
        status = 'desparate times'

print(status)
```

Blocks of code

In the examples above some lines are indented more than others, and you probably already have some idea of how this is interpreted by Python. Indentation defines blocks of code. Whether each block of code is evaluated when your code runs, is controlled by the `if` and `else` statements.

- All statements in a block of code have the same indentation. That is, they line up vertically.
- A block of code begins by a line that is indented more than the one before it.
- A block ends when it is followed by a line that is less indented.

This way, a block can be nested inside another block by indenting it further to the right as shown in fig. 0.5. Compare the example in fig. 0.5 to the code example above. Note how a colon at the end of a statement means "this applies the block of code below". Make sure you understand which print statements that are controlled by which `if` and `else` statements.

Elif-statement

Say you need to test a number of mutually exclusive scenarios. E.g. if a base is equal to A, T, C or G. You can do that like in the example below, but it is very verbose and shifts your code further and further to the right.

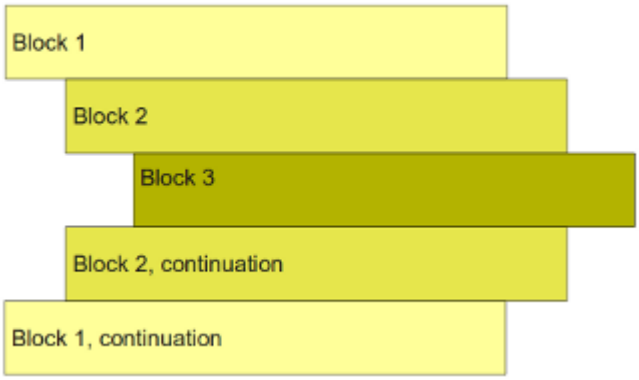


Figure 0.5: The amount of indentation defines blocks of code

```
base = 'G'

if base == 'A':
    print('This is adenine')
else:
    if base == 'T':
        print('This is thymine')
    else:
        if base == 'C':
            print('This is cytosine')
        else:
            print('This is guanine')
```

This is where an `elif` statement can be helpful. It is basically short for "else if". If you compare to the example below the correspondence is hopefully obvious.

```
base = 'G'

if base == 'A':
    print('This is adenine')
elif base == 'T':
    print('This is thymine')
elif base == 'C':
    print('This is cytosine')
else:
    print('This is guanine')
```

Exercise 92

You can use logical operators (`and`, `or`, `not`) in the expressions tested in an if-statement. Can you change the program from exercise 91 so that there are no nested if-statements --

in a way so the program still does exactly the same? You can use `if`, `elif` and `else` and test if e.g. both `milk` and `cookies` are true using `and`.

Exercise 93

In the snippet of code below there are three blocks with three statements in each. Which statements belong to which block?

```
x = 5
if x > 4:
    y = 3
    if x < 1:
        x = 2
        y = 7
        z = 1
    x = 1
z = 4
```

Exercise 94

Can you see four blocks of code?

```
x = 5
if x > 4:
    y = 3
    if x < 1:
        x = 2
        y = 7
    else:
        x = 1
        y = 9
z = 4
```

General exercises

Exercise 95

Will this print `You are a super star!`?

```
if -4 and 0 or 'banana' and not False:
    print("You are a super star!")
```

Exercise 96

Will this print `You are a super star!`?

```
if -1 + 16 % 5 == 0 :
    print("You are a super star!")
```

Exercise 97

Assign values to two variables `x` and `y`. Then write some code that prints `OK` if and only if `x` is smaller than five *and* `y` is larger than five. Do it using *two* `if` statements:

```
x = 3 # or someting else
x = 7 # or someting else

# rest of code here...
```

Now solve the same problem using only *one* `if` statement.

Exercises 98

Assign values to two variables `x` and `y`. Then write some code that prints `OK` if and only if `x` is smaller than five *or* `y` is larger than five. Do it using *two* `if` statements:

```
x = 3 # or someting else
x = 7 # or someting else

# rest of code here...
```

Now solve the same problem using only *one* `if` statement and *one* `elif` statement.

Exercises 99

Assign values to two variables `x` and `y`. Then write some code that prints `OK` if either `x` or `y` is zero, but not if both are zero (this is tricky one).

```
x = 3 # or someting else
x = 7 # or someting else

# rest of code here...
```

Organising your code

This chapter is about how you can organize your code into chunks that you can call upon to perform a well defined tasks in your program.

Functions

Buckle down for the most powerful and useful thing in programming. Functions! Functions serve as mini-programs that perform small well-defined tasks in your program. I have started to write a song about functions:

```
print("Functions are super, Functions are cool")
print("When writing a program they are a great tool")
print("La la dim du da da di")
print("Skubi dubi dumdi di")
print("Bing di dubi dum da di")

print("Functions are used to package some code")
print("They are not so strange that your head will explode")
print("La la dim du da da di")
print("Skubi dubi dumdi di")
print("Bing di dubi dum da di")
```

I am going to add a lot more verses and I do not want to have to write the entire chorus every time. So what would be more natural than to make a function named `chorus` that takes care of that for us? That way we can write our song the way lyrics with a chorus are usually written:

```
def chorus():
    line1 = "La la dim du da da di"
    line2 = "Skubi dubi dumdi di"
    line3 = "Bing di dubi dum da di"
    return line1 + '\n' + line2 + '\n' + line3

print("Functions are super, Functions are cool")
print("When writing a program they are a great tool")
print( chorus() )
```

```
print("Functions are useful to wrap up some code")
print("They are not so strange that your head will explode")
print( chorus() )
```

First, let us break down the function definition in the top part of this code:

1. We *define* a function with the `def` keyword (which is short for "define" in case you wonder).
2. After `def` we write the name of the function. We call the function `chorus`. We could name it something else, but like good variable names, good function names can help you remember what your code does.
3. After the name you put two parentheses, `()`.
4. Then a colon, `:`.
5. The statements that are part of the function are nested under the `def` statement and are indented with four spaces exactly like we do under if-statements.
6. The return-statement ends the function. The expression after the `return` keyword reduces to a *value* that the function then returns.

When Python runs this code, each line is executed one by one starting from the first line (remember oath two?). So in this case python first executes the *definition* of the `chorus` function. The only thing that has happened after Python has executed the first five lines of code is that it has assigned the name `chorus` to the four indented statements. So Python now "knows" about the `chorus` function (like it "knows" about a variable `x` after we do `x = 4`).

To *use* the function, we "call" it by writing its name followed by parentheses: `chorus()`. When it comes to functions, "use", "call" and "run" means the same thing. As you can see, we call the function twice in the rest of the code. Each time we do, the following happens:

1. When a function is called each statement in the definition is executed one after the other. If you look at the function definition, you can see that our `chorus` function has four statements.
2. The first statement assigns a string value to the variable `line1`.
3. The second statement assigns a string value to the variable `line2`.
4. The third statement assigns a string value to the variable `line3`.
5. The fourth statement is a return-statement. The expression after the `return` keyword in the final statement reduces to a *value* and this value is what the function call is substituted for. In this case, that value is the following string:

```
"La la dim du da da di\nSkubi dubi dumdi di\nBing di dubi dum da di"
```

So the key properties of functions are:

- A function names a piece of code (some statements) just like variables name values like strings and numbers.

- We call a function by writing the function name followed by parentheses: `chorus()`. Just writing the function name will not call the function.
- When a function is *called* it is substituted by the *value* that the function *returns* -- exactly like a variable in an expression is substituted by its value. It is absolutely crucial that you remember this.

Exercise 100

Now that we have a `chorus` function, that part is out of the way and we can concentrate on our song without having to worry about remembering how many "la la"s it has and so on. Try to change the "lyrics" in the chorus a little bit. Notice how you only need to make the change in one place to change all the choruses in the song -- cool right? Without the function, you would have to rely on correctly changing the code in many different places.

Exercise 101

Try to delete the return-statement in the `chorus` function (the last line in the function) and run the code again. You should see something like this:

```
Functions are super, Functions are cool
When writing a program they are a great tool
None
Functions are used to wrap up some code
They are not so strange that your head will explode
None
```

It seems that the function call (`chorus()`) is now substituted with `None`. How can that be when we did not return anything? The reason is that when you do not specify a `return` statement the function returns `None` by default. This is to honour the rule that variables and a function calls are substituted by a value, and `None` is simply the value that Python uses to represent "nothing".

> **A short history about nothing:** None is basically a value denoting the lack of value. As you just saw it is used to represent that no value is returned from a function. It can also be assigned to a variable as a placeholder value until another value is assigned:
>
> ```
> x = None
> x = 4
> ```
>
> Also, `None` is considered false in a logical context:
>
> ```
> print(not None)
> ```

Exercise 102

Try this variant to the chorus function. Go through the code *slowly* and repeat all the steps in the breakdown of what happens when a function is called. Remember to also

do each substitution and reduction carefully.

```
def chorus():
    line1 = 'La la'
    line2 = 'Du bi du'
    return line1 + '\n' + line2
```

Do the same for this variant:

```
def chorus():
    line1 = 'La la'
    line2 = 'Du bi du'
    chrous_text = line1 + '\n' + line2
    return chrorus_text
```

and for this variant:

```
def chorus():
    return "La la\nDu bi du"
```

Exercise 103

What do you think happens if you move the definition of `chorus` to the bottom of your file? Decide what you think will happen and why (maybe you remember what happens when you try to use a variable in an expression before you have defined it?). Then try it out.

```
print("Functions are super, Functions are cool")
print("When writing a program they are a great tool")
print( chorus() )

print("Functions are useful to wrap up some code")
print("They are not so strange that your head will explode")
print( chorus() )

def chorus():
    line1 = "La la dim du da da di"
    line2 = "Skubi dubi dumdi di"
    line3 = "Bing di dubi dum da di"
    return line1 + '\n' + line2 + '\n' + line3
```

Make sure you understand how the error you get relates to the way Python runs your script (remember oath two?).

Exercise 104

Consider the code below. Do all the substitution and reduction steps in your head. Remember that each function call is substituted by the value that the function returns. Then run it.

```
def lucky_number():
    return 7

x = lucky_number()
y = lucky_number()
twice_as_lucky = x + y
print(twice_as_lucky)
```

Now change the code to that below. The code makes the same computation but in fewer steps. Do all the substitution and reduction steps again.

```
def lucky_number():
    return 7

twice_as_lucky = lucky_number() + lucky_number()
print(twice_as_lucky)
```

Now change the code to that below. The code makes the same computation but in fewer steps. Do all the substitution and reduction steps again.

```
def lucky_number():
    return 7

print(lucky_number() + lucky_number())
```

Functions can take arguments

The functions we have written so far are not very flexible because they return the same thing every time they are called. Now write and run this beauty:

```
def square(number):
    squared_number = number**2
    return squared_number

result = square(3)
print(result)
```

Notice how we put a variable (`number`) between parentheses in the function *definition*. This variable is assigned the value that we put between the parentheses (`3`) when we *call* the function. So when we call the function like this: `square(3)` -- then this implicitly happens: `number = 3`.

Here is another example:

```
def divide(numerator, denominator):
    result = numerator / denominator
```

```
    return result

division_result = divide(44, 77)
print(division_result)
```

When the function call `divide(44, 77)`, these two things implicitly happen: `numerator = 44` and `denominator = 77`.

Take note of the following three important points: 1. The *values* that we pass to the function in the function call, like `3`, `44` and `77`, are called *arguments*. It is crucial to remember that it is *values* and not variables that are passed to functions. 2. The *variables* in the definition line of a function, like `number`, `numerator` and `denominator`, are called *parameters*. They hold the *values* passed to the function when it is called (the arguments). 3. You can define functions with any number of *parameters* as long as you use the same number of *arguments* when you call the function.

Exercise 105

Try to call your `divide` function like this `divide(77, 44)`. What does it return and what do you learn from that?

Exercise 106

Try to call your `divide` function like this `divide(44)`. Do you get an error, and what do you learn from that?

Exercise 107

Try to call your `divide` function like this `divide(44, 77, 33)`. Do you get a different error message, and what do you learn from that?

Exercise 108

Read this code and do all substitution and reduction steps from beginning to end.

```
def square(x):
    return x ** 2

result = square(9) + square(5)
print("The result is:", result)
```

Now replace the line `return x ** 2` with `print(x ** 2)`. What is printed now? and why?

Exercise 109

As described above, a `return` statement ends the function by producing the value that replaces the function call. If a function has more than one `return` statement, then the function ends when the first one is executed.

```
def assess_number(x):
    if x < 3:
        return 'quite a few'
    if x < 100:
        return 'a lot'
    return 'really a lot'

nr_apples = 2
print(nr_apples, "apples is", assess_number(nr_apples))
```

What happens when x is 2, 3, 50, 200? Think about it first.

Functions and variables

Now you know about functions, you know how you pass *values* into them and you know how to make them return a *value* back. It is all a little mind-boggling, but you will get used to it once you do a lot of exercises.

Every time we call a function, a new fresh mini-program springs into existence the way it is described in the function definition. This temporary little world only exists from the function is called and until it returns its value. It did not exist before it was called and it does not exist after it returns a value. This means that variables defined in your functions are temporary and private to the function and that they are not available to code *outside* the function. Running the following example should help you understand this:

```
def make_greeting():
    name = 'Dan'
    message = greeting + " " + name
    return message

name = 'Kasper'
greeting = 'Hello'
print("Message:", make_greeting() )
print(name)
```

Notice the following:

- The message produced by the function greets Dan not Kasper.
- From the last print-statement, you can see that the `name` variable we defined in the first line of the script still has the value `'Kasper'`, no matter what happens in the temporary life of the function.
- Python could not find the `greeting` variable in the function but found it in the main script.

You should learn two rules from this example:

1. All variables that you assign values to in a function (including the function parameters) are *private* to the function. If a variable in a function has the same name as a variable in the main script (like `name` above) then these are two *separate* variables that just happen to have the same name.
2. When you use variables like `greeting` and `name` in the function, then Python checks if the variables have been defined in the function. If that is not the case, then it will look in the main script too. In this case, it finds `name` in the function and `greeting` in the main script. It is good practice to make your functions "self-contained", in the sense that Python should not have to look outside the function for variables.

Exercise 110

Consider the following example:

```
def double(z):
    return z * 2

x = 7
result = double(x)
print(result)
```

When the function is called (`double(x)`) the `x` is substituted by its value `7`. That *value* is passed as the argument and assigned to the function parameter `z` (`z = 7`). `z` is a private function variable and does not exist before or after the function call. Does this change in any way if we use the variable name `x` instead of `z` like below?

```
def double(x):
    return x * 2

x = 7
result = double(x)
print(result)
```

Do *all* substations and reductions for each line of code from top to bottom. Keep the sequence of events in mind and remember that a function definition is merely a template describing a mini-world that is created anew everytime the function is called.

Builtin functions

So far we have only talked about functions you write yourself, but Python also has built-in functions that are already available to you. They work just like a function you would write yourself. You already know the `print` function quite well, and that is an example of a function that prints something but returns `None`. There are many other useful builtin function, but for now, I will just tell you about another two: Those are `len`

and `type`.

Exercise 111
Try these examples:

```
x = 'Banana'
print("The value of variable x is of type", type(x))
print("The value of variable x has length", len(x))
```

As you can see, `type` returns the type of the value passed as the argument, and `len` returns the length of the value passed as the argument. The `type` function is handy in case you wonder what type a value has, but it is not a function we will use in this course. The `len` function, however, is your new best friend. You will see why soon enough.

Exercise 112
Try to change the value of the `x` in exercise 111 to an integer or a float and see what happens when you run it. Do you get an error? Does it make sense that not all types of values can meaningfully be said to have a length?

Exercise 113
What happens if you pass an empty string (`""`) as the argument to the `len` function?

General exercises

The following exercises treat the areas we have worked on in this and previous chapters. They are meant to train your familiarity with if-statements and functions. Remember that the purpose of the exercises is not to answer the questions but to train the chain of thought that allows you to answer them. Play around with the code for each example and see what happens if you change it a bit.

Exercise 114
Consider this function definition that takes a single number as the argument:

```
def square(n):
    return n**2
```

What does it do? What does it return? What number does `square(2)` then represent?

Below I have used it in some expressions that are printed. Make sure you understand what each expression evaluates to. Do the explicit substitutions and replacements on paper before you run it. Remember that we can substitute a function call (like `square(2)`) for the value it returns, just like we can substitute a variable `x` for the value it points to.

```
print(square(3))
print(square(2 + 1))
print(square(2) * 2 + square(3))
```

```
print(square(square(2)))
print(square(2 * square(1) + 2))
```

Exercise 115

What does this function do? How many parameters does it have? How many statements does the function have? What does the function print? Which value does it return?

```
def power(a, b):
    print("This function computes {}**{}".format(a, b))
    return a**b

print(power(4, 2))
```

Try (possibly strange) variations of the code like the ones below to better understand the contribution of each line of code. What is the difference between return and print? What happens when the Python gets to a `return` statement in a function? What happens when the function does not have a `return` statement?

Variation 1:

```
def power(a, b):
    print("This computes", a, "to the power of", b)
    print(a**b)

result = power(4, 2)
print(result)
```

Variation 2:

```
def power(a, b):
    print("This computes", a, "to the power of", b)
    return a**b

result = power(4, 2)
print(result)
```

Variation 3:

```
def power(a, b):
    print("This computes", a, "to the power of", b)
    a**b

print(power(4, 2))
```

Variation 4:

```
def power(a, b):
    return a**b
    print("This computes", a, "to the power of", b)

print(power(4, 2))
```

Exercise 116

Define a function called `diff`, with two parameters, `x` and `y`. The function must return the difference between the values of `x` and `y`.

```
Example:
def diff(x, y):
    ...

diff(8, 2) # should return 6
diff(-1, 2) # should return -3
```

Save the value returned from the function in a variable and test if the function works correctly by comparing the result to what you know is the true difference (using ==).

Exercise 117

Define a function called `all_equal` that takes five arguments and returns `True` if all five arguments have the same value and `False` otherwise. The function should work with any input, for example:

```
all_equal("Dan", "Dan", "Dan", "Dan", "Dan")
all_equal(0, 0, 0, 0, 0)
all_equal(0.5, 0.5, 0.5, 0.5, 0.5)
all_equal(True, True, True, True, True)
```

Hint: You test equality with `a == b`. Now think back to what you learned about logic. Which operator can you use to test if `a == b` *and* `b == c`?

Exercise 118

Define a function called `is_even` which takes one argument and returns `True` if (and only if) this is an even number and `False` otherwise (remember the modulo operator?).

```
is_even(8) # should return True
is_even(3) # should return False
```

Exercise 119

Define a function called `is_odd` which takes one argument and returns `True` if (and only if) the argument is an odd number and `False` otherwise.

```
is_odd(8) # should return False
is_odd(3) # should return True
```

Can you use the `is_even` you defined in exercise 118 to complete this exercise? How? Why is that a good idea?

Exercise 120

Here is a function that should return `True` if given an uppercase (English) vowel, and `False` otherwise:

```
def is_uppercase_vowel(c):
        c == 'A' or c == 'E' or c == I or c == 'O' or c == 'U'

char = 'A'
if is_uppercase_vowel(char):
    print(char, "is an uppercase vowel")
else:
    print(char, "is NOT an uppercase vowel")
```

Now type the code exactly as shown and run it. Do you get what you expect? Does the code work? If not, try to figure out why. Try print the value that the function returns. Does that give you any hints about the cause of the problem?

Exercise 121

Define a function called `is_nucleotide_symbol` which takes one argument and returns `True` if this is either A, C, G, T, a, c, g or t, and `False` in any other case.

Name your parameter something sensible like `symbol`.

```
is_nucleotide_symbol("A") # should return True
is_nucleotide_symbol("B") # should return False
is_nucleotide_symbol("Mogens") # should return False
is_nucleotide_symbol("") # should return False
```

Exercise 122

Define a function called `is_base_pair` which takes two parameters, `base1`, `base2`, and returns `True` if `base2` is the complementary of `base1`, and `False` otherwise.

```
is_base_pair("A", "G") # should return False
is_base_pair("A", "T") # should return True
is_base_pair("T", "A") # should return True
is_base_pair("Preben", "A") # should return False
```

Can you use the function you defined in exercise 121 to complete this exercise? How? Why is that a good idea?

Exercise 123

Did you find the bug in exercise 120? You were supposed to find that the function did not have a return value. This makes the function return `None` by default. Do you think the `None` value is considered true or false in an if-statement?

Exercise 124

Define a function called `celcius2fahrenheit` that converts from celsius to Fahrenheit. You can do this because you know the linear relationship between the two. On the figure below you can see that the slope is `9 / 5` and the intercept is `32`. The function should have one parameter `celsius`. Inside the function, you should define the variables `slope` and `intercept` and give them the appropriate values. Then you can calculate the conversion to Celcius using these variables and return the result.

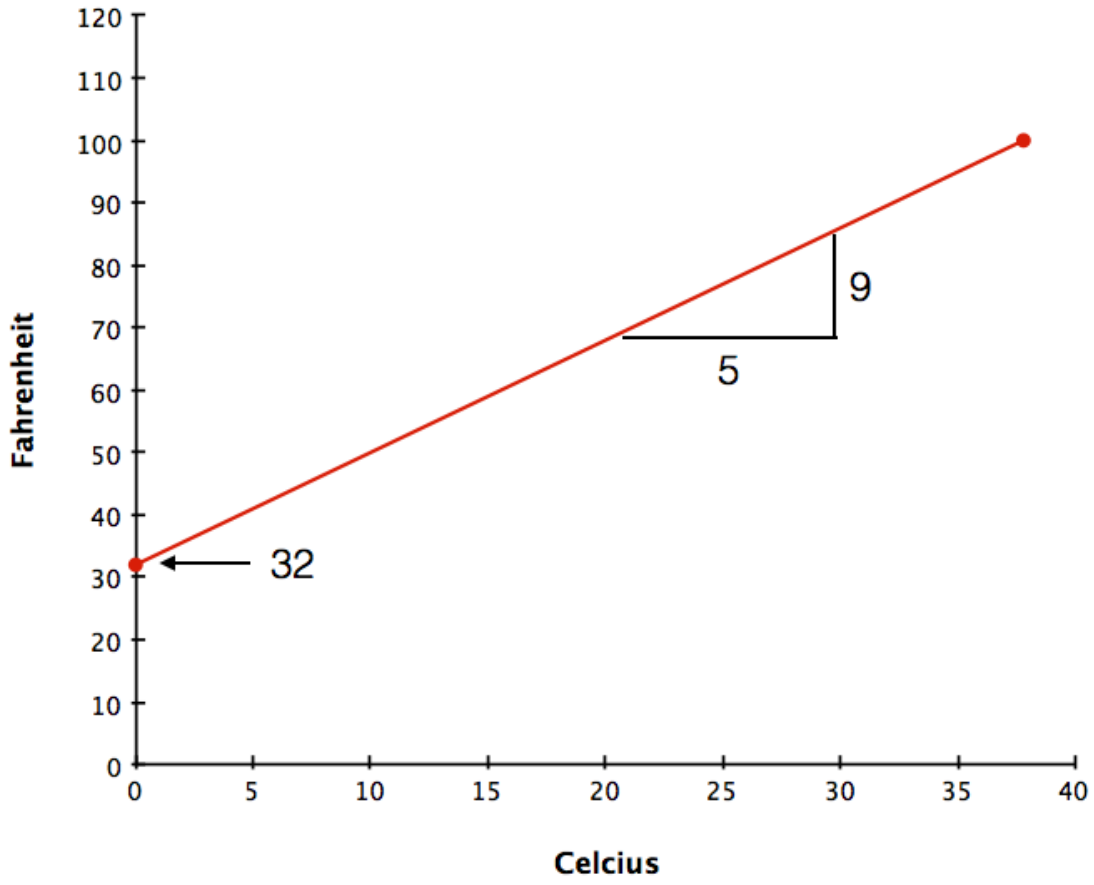


Figure 0.6: Temperature conversion

Exercise 125

Try to change your conversion function so it takes three arguments, corresponding to celsius, slope and intercept so you can call it like this to convert 27 degrees celsius:

`conversion(37, 9 / 5, 32)`. Now you have a function that can do any linear conversion that you can put inside another function like this:

```
def celcius2fahrenheit(celsius):
    return conversion(celsius, 9 / 5, 32)
```

Exercise 126

Now try to extend this to a different problem: It has been found that the height and weight of a person are related by a linear equation with slope = 0.55 and intercept = -25. Define a function called `predict_weight` which takes just one argument, the height of a person, and returns the estimated weight of the person.

Exercise 127

By now you know that some of the words in your code have specific purposes. `def` defines functions, `return` returns value from a function, `and` is a logical operator etc. Here is a list of the ones you will see in this course: `and`, `assert`, `break`, `continue` `def`, `del`, `elif`, `else`, `False`, `for`, `from`, `if`, `import`, `in`, `is`, `not`, `or`, `pass`, `return`, `while`, `True`, `None` (you can see a full list here)

These words are reserved for their special purposes in Python and you will not be allowed to assign values to them. Try this to see for yourself:

```
None = 4
```

or this:

```
and = 2
```

FAQ - Frequently Asked Questions

Q: Can function names be anything?
A: Just about. The rules that apply to variable names also apply to function names. Good function names are lower case with underscores (_) to separate words, like in the examples above.

Python values are objects

This chapter introduces the notion of an object, one of the most central aspects of Python. Once you get over the surprise, you will find it super useful that Python values are actually objects.

Methods

In Python, each value (like integers, floating-points, and strings) not only holds data but is also packaged with a lot of useful functionality relevant to that particular *type* of value. When a value is packaged with relevant functionality and meta information, programmers call it an *object*, and in Python *all* values are objects.

The attached functionality comes in the form of *methods*. You can think of methods as functions that are packaged together with the value. String values have a method called `capitalize`. Try it out:

```
x = "banana".capitalize()
print(x)
```

To call the method on the string value, we connect string and the method call with a dot. So to call a method on a value you do the following:

1. Write the value (or a variable name that substitutes for a value).
2. Then write a `.`.
3. Then write the name of the method.
4. Then write two parentheses to call it. If the method takes any arguments other than the value it belongs to, then you write those additional values between the parentheses with commas in between, just as when you call a function.

You can see that the method call itself looks just like a function call and in many ways calling a method works much like calling a function. The difference is that when we call a function we say: "*Hey function, capitalize this string!*". When we call a method we say: "*Hey string, capitalize yourself!*"

So why do we need methods? Can we not just do with functions? We can, but it turns out that it is very handy to have some relevant and ready-to-use functionality packaged

together with the data it works on. You will start to appreciate that sooner than you think.

When we put a method call after a *variable* like below, the variable is first substituted for its value and then the method is called on its value. Consider the second line of this example:

```
x = "banana"
print(x.capitalize())
```

Here `x` is first substituted by `"banana"` and *then* the method is called on that value, like this: `"banana".capitalize()`.

Now write and run these examples:

```
message = "Methods Are Cool"
print(message)

shout = message.upper()
print(shout)

whisper = message.lower()
print(whisper)

new_message = message.replace("Cool", "Fantastic")
print(new_message)
```

You can see what these methods do. For example: `upper` returns an uppercased copy of the string.

Exercise 128

Write and run the following code. What do you think it does?

```
line = '\n\tSome text\n'
print(">{}<".format(line))

line = line.strip()
print(">{}<".format(line))
```

Make sure you do the substitution and reduction steps in your head. Be especially careful about the third line of code. Also, what do you think the special `\t` character is?

Exercise 129

The string methods you have tried so far have all returned a new string. Try this example:

```
'ATGACGCGGA'.startswith('ATG')
```

and this

```
'ATGACGCGGA'.endswith('ATG')
```

What do the methods do and what do they return?

Using the Python documentation

Now that you are well underway to becoming a programmer, you should know your way around the Python documentation. Especially the part called the Python standard library. There is a *lot* of details in there that we do not cover in this course. These are mainly are tools and techniques that for writing more efficient, extensible, robust and flexible code. The parts we cover in this course are is the minimal set that will allow you to write a program that can do *anything*.

Exercise 130

There is also a string method that returns a secret agent:

```
print('7'.zfill(3))
```

If you do not believe me, you can look it up in the Python documentation.

Exercise 131

Try to find the documentation for the `append` method for lists.

String formatting

You have already tried string formatting in exercise 128. String formatting is a simple but powerful technique that lets you generate pretty strings from pre-computed values. You may have noticed that every time we print a floating-point number, a lot of decimals are shown. Not very pretty if you are only interested in two decimals anyway. To format a string you use the `format` method (surprise). `format` replaces occurrences of `{}` with the arguments passed to it. Like this:

```
taxon = "genus:{}, species:{}".format('Homo', 'sapiens')
```

Exercise 132

What happens if you try this?

```
question = "Was {} {} Swedish?".format('Carl', 'Linneaus')
```

and this?

```
question = "Was {} Swedish?".format('Carl', 'Linneaus')
```

and this?

```
question = "Was {} {} Swedish?".format('Carl Linneaus')
```

In the two last examples the number of `{}` was not the same as the number of arguments to `format`. What happens when there are too few and what happens when there are too many?

Exercise 133
Consider this code:

```
s = "{} is larger than {}".format(4, 3)
print(s)
```

What will happen if you run this code? Write the code and see for yourself once you think you know. If you were wrong, make sure you understand why.

Exercise 134
Consider this code:

```
language = 'Python'
invention = 'sliced bread'
s = '{} is the best thing since {}'.format(language, invention)
print(s)
```

What will happen if you run this code? Write the code and see for yourself once you think you know. If you were wrong, make sure you understand why.

Exercise 135
Consider this code:

```
my_template = '{} is the best thing since {}'
language = 'Python'
print(my_template.format(language, 'sliced bread'))
print(my_template.format(language, 1900 + 89))
```

What will happen if you run this code? Do the substitution and reduction steps in your head.

Exercise 136
Think back to exercise 9 where you calculated how many cookies you could buy for 30 kr. The bars are 2.80 kr. So your program looked something like this:

```
nr_bars = 30 / 2.8
print('I can buy', nr_bars, 'chocolate bars!')
```

and it ran like this:

```
$ python chocolate.py
I can buy 10.7142857143 chocolate bars!
```

String formatting lets you rewrite the program like this:

```
nr_bars = 30 / 2.8
message = "I can buy {} chocolate bars!".format(nr_bars)
print(message)
```

Try to replace `{}` with `{:.2f}`. `format` reads the stuff between the curly brackets and uses as directions for how to format the value it inserts. Try it out and see what happens if you change the number `2` to `3`, `4`, `5` or `10`.

Exercise 137

See if you can find the documentation for the `format` function in the Python documentation. It can do wondrous things, for this course we will only try to control the number of digits and padding with spaces. Look at the examples below. Maybe you can figure out how it works

```
pi = 3.14159265359
print("*{}*".format(pi))
print("*{:.3f}*".format(pi))
print("*{:.6f}*".format(pi))
print("*{:>5.3f}*".format(pi))
print("*{:>10.3f}*".format(pi))
```

Bonus info: How do you think python can figure out that adding strings with is supposed to differently than adding numbers? The answer is that all values you can add with the + operator has a secret method called `__add__` that defines how it should works for that type of value:

```
s1 = "11"
s2 = "22"
n1 = 11
n2 = 22
print(s1 + s2)
print(s1.__add__(s2))
print(n1 + n2)
print(n1.__add__(n2))
```

This was just to show how Python does this. Just like thorns or yellow and black stripes in nature means "hands off" -- double underscores (__) is Python's way of saying "do not use this!"

Indexing and slicing strings

Another feature of string objects is that they allow you to extract individual parts of the string.

Each character in a string is identified by its *index* in the string. To access a character in a list you write brackets after the string and between those brackets you specify the index of the character you want. The first character has index 0, the second has index 1 and so on.

```
codon = 'ATG'
print("first base is", codon[0])
print("second base is", codon[1])
print("third base is", codon[2])
```

You may wonder why the index of the first character is zero and not one. That is simply the convention in programming. Over time you will start to find this useful rather than annoying. You should think of the index as the offset from the start of the string.

That also means that the index is not the length of the string, but the length minus one:

```
amino_acids = 'ARNDCQEGHILKMFPSTWYV'
last_index = len(amino_acids)-1
print("Last amino acid is", amino_acids[last_index])
```

If you want a sub-string from a string (we call that a *slice*) you specify a start index and an end index separated by a colon:

```
print(amino_acids[1:4])
```

When you run that you can see that `amino_acids[1:4]` is substituted for `'RND'`, so the slicing operation produces a sub-string of `amino_acids`. You may wonder why the value at index 4 is not in the resulting sub-string. That is another programming convention: intervals are "ends exclusive". So when you specify an interval with a start index of 1 and an end index of 4 it represents all the characters starting from 1 and up to, *but not including*, 4. So the slice `1:4` corresponds to the characters at indexes 1, 2 and 3. The reason programmers handle intervals in this way is that it makes it easier to write clear and simple code as you will see in the exercises.

Exercise 138

What do this expression reduces to?

```
"Donald Trump"[7]
```

Exercise 139

What is printed here? Do all the substitution and reduction steps and compare to the exercise above.

```
s = "Donald Trump"
print(s[7])
```

Exercise 140

What is printed here? Do all the substitution and reduction steps -- and do it *twice*. Next week you will be happy you did.

```
dna = 'TGAC'
i = 0
print(dna[i])
i = 1
print(dna[i])
i = 2
print(dna[i])
```

Exercise 141

What do you think happens here? Make up your mind and try out the code below:

```
s = "Donald Trump"
s[7] = 'D'
```

Did you see that coming? Strings are *immutable*, which means that you cannot change them once you have made them. If you want `"Donald Drump"` you need to produce a new string with that value. Try to figure out how to do that with the `replace` method of strings.

Exercise 142

When you do not specify the start and/or the end of a slice Python will assume sensible defaults for the start and end indexes. What do you think they are? Make up your mind and try out the code below:

```
s = 'abcdefghijklmnopqrstuvxyz'
print(s[:11])
print(s[11:])
print(s[:])
```

Exercise 143

Try to find the documentation for how slicing of strings work.

Exercise 144

What do you think happens when you specify an index that does not correspond to a value in the list:

```
s = 'abcdefghijklmnopqrstuvxyz'
print(s[99])
```

Read and understand the error message.

Exercise 145

Do you think you also get an error when you specify a slice where the end is too high? Try it out:

```
s = 'abcdefghijklmnopqrstuvxyz'
print(s[13:99])
```

I guess that is worth remembering, right?

Exercise 146

Which character in a string named `s` does this expression reduce to?

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

Exercise 147

Because intervals are "ends exclusive" we can compute the length of a slice as `end - start`:

```
dna = "ATGAGGTCAAG"
start = 1
end = 4
print("{} has length {}".format(dna[start:end], end-start))
```

Figure out what this code would look like if ends were included in intervals.

Exercise 148

Another advantage of "ends exclusive" intervals is that you only need one index to split a string in two:

```
s[:idx] + s[idx:] == s
```

Figure out what indexes you would need to use to split a sequence in two if ends were included in intervals.

Exercise 149

Did you look up the details of how slicing works? If so you should be able to explain what happens here:

```
s = 'abcdefghi'
print(s[::-1])
```

General exercises

Exercise 150

Will this print `You are a super star!`?

```
if 'na' * 2 == "Banana"[2:]:
    print("You are a super star!")
```

Exercise 151

Will this print `You are a super star!`?

```
if "{}s".format('Banana'[1:].capitalize()) == 'Ananas':
    print("You are a super star!")
```

Exercise 152

Write a function called `even_string` that takes a string argument and returns `True` if the length of the string is an even number and `False` otherwise. E.g. `even_string('Pear')` should return `True` and `even_string('Apple')` should return `False`.

Exercise 153

Look at the code below and decide what is printed at the end. Then write the code and test your prediction. If you are wrong, make sure you figure out why.

```
def enigma(x):
    if x == 4:
        return x

result = enigma(5)
print(result)
```

Structuring data

This chapter is about lists and dictionaries, which are types of Python values, which are containers for other values. Using them and combining them lets you build relationships between values, which is what data structures are.

Lists

Often when you have some data, the order of things is important. The order of characters is important for the meaning of text in a string. Sometimes we want to specify the order of other things than characters -- like numbers or strings. A list is useful when the relative order of items in the list has some meaning. It could be a grocery list where you have listed the things to buy in the order you get to them in the supermarket. When you print a list it nicely shows all the values it contains.

```
grocery_list = ["salad", "canned beans", "milk", 'beer', 'candy']
print(grocery_list)
```

Unlike strings that are sequences of characters, lists can have *any* kind of values in them, and you can mix different types of values any way you like. Here is a list that contains an integer, a boolean, a string and a list:

```
mixed_list = [42, True, 'programming', [1, 2, 3] ]
```

By now you have probably guessed that you make a list with two square brackets. A list can have values with commas in-between. A list is a *container* of other values and the value of the list itself does not depend on the values it contains. This makes sense. Otherwise, an empty list would not have a value:

```
my_list = []
```

You can add single values to the end of a list using the append method of lists:

```
desserts = []
print(desserts)
desserts.append('Crepe suzette')
print(desserts)
desserts.append('Tiramisu')
```

```
print(desserts)
desserts.append('Creme brulee')
print(desserts)
```

If you have a list you want to add to the end of another list you use the `extend` method:

```
cheeses = ['Gorgonzola', 'Emmentaler', 'Camembert']
desserts.extend(cheeses)
print(desserts)
```

Notice how `append` and `extend` *modifies* the existing list instead of producing a new list.

Exercise 154

Do you think this will work?

```
cheeses = ['Gorgonzola',
           'Emmentaler',
           'Camembert']
print(cheeses)
```

surprised? Code inside parentheses, brackets and braces can span several lines, which can sometimes make your code easier to read.

Exercise 155

If you want to test if a value is in a list you use the `in` operator. Try this:

```
print('Tiramisu' in desserts)
print('Meatloaf' in desserts)
```

Exercise 156

You can concatenate two lists to produce a new joined list. Make sure you figure out how this works before you try it out. Then experiment with changing the lists. Can you concatenate two empty lists?

```
some_list = [1, 2, 3]
another_list = [7, 8, 9]
merged_list = some_list + another_list
print(merged_list)
```

Exercise 157

What do you think is printed here? Make sure you figure out how you think this works before you try it out. What does the `append` method return?

```
my_list = []
x = my_list.append(7)
print(x)
```

Indexing and slicing lists

Now you know how to make lists, but to work with the values in lists you must also know how to access the individual values that a list contains. Fortunately, indexing lists works just like indexing strings: Each value in a list is identified by an *index* exactly like each character in a string:

```
numbers = [7, 4, 6, 2, 8, 1]
print("first value is", numbers[0])
print("second value is", numbers[1])
print("third value is", numbers[2])
```

Notice that the function `len` can also compute the length of a list. So you also get the last value in list like this:

```
numbers = [7, 4, 6, 2, 8, 1]
last_index = len(numbers)-1
print("Last element is", numbers[last_index])
```

If you want a sub-list of values from a list (we also call that a *slice*) you specify a start index and an end index separated by a colon, just like with strings:

```
print(numbers[1:4])
```

When you run that you can see that `numbers[1:4]` is substituted for `[4, 6, 2]`, so the slicing operation produces a new list of the specified values.

Exercise 158

What do these two expressions reduce to?

```
[11, 12, 13, 14, 15, 16, 17][2]
```

Exercise 159

What is printed here? Do all the substitution and reduction steps and compare to the exercise above.

```
l = [11, 12, 13, 14, 15, 16, 17]
print(l[2])
```

Exercise 160

What is printed here? Do *all* the substitution and reduction steps -- and do it *twice*. Next week you will be happy you did.

```
numbers = [1,2,3]
i = 0
print(number[i])
i = 1
print(number[i])
```

```
i = 2
print(number[i])
```

Exercise 161

What do you think happens here? Make up your mind and try out the code below:

```
l = [11, 12, 13, 14, 15, 16, 17]
l[4] = "Donald"
print(l)
```

Were you surprised what happened? Compare to exercise 141. Lists are not immutable like strings and you can replace values by assigning a new value to an index in the list.

Exercise 162

With your knowledge of slicing, what do you think is printed below:

```
l = [11, 12, 13, 14, 15, 16, 17]
print(l[:3])
print(l[3:])
print(l[:])
```

Exercise 163

What do you think happens when you specify an index that does not correspond to a value in the list:

```
l = [11, 12, 13, 14, 15, 16, 17]
print(l[7])
```

Read and understand the error message. Does it ring a bell?

Exercise 164

Do you think you also get an error when you specify a slice where the end is too high? Try it out:

```
l = [11, 12, 13, 14, 15, 16, 17]
print(l[4:99])
```

I guess that is also worth remembering.

Exercise 165

Which value in a list named `l` does this expression reduce to?

```
l[len(l)-1]
```

Exercise 166

If you do not like Emmentaler you can just delete it. What do you think the `del` keyword does?

```
cheeses = ['Gorgonzola', 'Emmentaler', 'Camembert']
print(cheeses)
del cheeses[1]
print(cheeses)
```

Exercise 167

Because intervals are "ends exclusive" we can compute the length of a slice as `end - start`:

```
l = [7, 4, 6, 2, 8, 1]
start = 1
end = 4
print("{} has length {}".format(l[start:end], end-start))
```

Think about what this code would look like if ends were included in intervals.

Exercise 168

Another advantage of "ends exclusive" intervals is that you only need one index to split a list in two:

```
l[:idx] + l[idx:] == l
```

If ends were included in intervals this would not be as simple.

Exercise 169

Do *all* the substitution and reduction steps in your head (or on paper) before you write any of the following code. Think carefully and make up your mind what you think will be printed below. Remember that the *value* of a list is a container that holds other *values* in it. Then write the code and see if you were right. If you were not, make sure you figure out what led you to the wrong conclusion.

```
x = 'A'
y = 'B'
z = 'C'
lst = [x, y, z]
print(lst)

x = 'Preben'
print(lst) # what is printed here?

lst[0] = 'Mogens'
print(lst) # what is printed here?
```

Exercise 170

Do you remember this trick from string slicing?

```
l = [1, 2, 3, 4, 5]
print(l[::-1])
```

Exercise 171

You can produce a list by splitting a long string into smaller parts. Think: "*Hey string, split yourself on this smaller string*". Try these variations to figure out how it works

```
"Homo sapiens neanderthalensis".split(" ")
"Homo sapiens neanderthalensis".split('en')
'ATGCTCGTAACGACACTGCACTACTACAATAG'.split('')
"1, 2, 3, 5, 3, 2, 5, 3".split(', ')
"1,2,3,5,3,2,5,3".split(',')
'ATGCTCGTAACGACACTGCACTACTACAATAG'.split()
"Homo sapiens neanderthalensis".split()
```

Notice that the method has a default behavior when no argument is passed to it.

Exercise 172

You can produce a string by joining the elements of a list (if all the elements are strings of course). Think: "*Hey string, put yourself in between all the strings in this list*".

```
"-".join(['Homo', 'sapiens', 'neanderthalensis'])
"...".join(['Homo', 'sapiens', 'neanderthalensis'])
"".join(['A', 'T', 'G'])
```

Notice how you can join something on an empty string. This is a very useful technique; for example if you want to turn a list of characters into a string.

Dictionaries

Lists are useful for storing values when the order of the values is important but lists have one drawback: you can only access a value in a list using the index of the value.

A dictionary, called `dict` in Python, is a much more flexible data type. Like a list, a dictionary is a container for other values, but dictionaries do not store values in sequence. They work more like a database that lets you store individual *values*. When you store a value you assign it to a *key* that you can then use to access the stored value. Now create your first dictionary:

```
person = {'name': 'Donald Trump', 'age': 70, 'job': 'President'}
```

This dictionary has three values (`'President'`, `'Donald Trump'` and `70`) and each value is associated with a key. Here `'age'` is the key for the value `70`. So when defining a dictionary you should note the following:

1. You make a dictionary using braces.

2. Between you braces you put key-value pairs separated by a colon.
3. The key-value pairs are separated by commas.
4. To make an empty dictionary you just write the braces with nothing between them: `{}`.

To access a value in the dictionary you its key in square brackets after the dictionary:

```
"{} is a {} year old {}".format(person['name'], person['age'], person['job'])
```

Here we used strings as keys, but you can also use many types of values as keys (Python will give you an error if you try to use a type that is not allowed):

```
misc_dict = {42: "Meaning of life", "pi": 3.14159, True: 7}
```

A dictionary stores key-value pairs but do not keep track of their order. So when you print a dictionary the order of the key-values pairs is arbitrary.

If you have a dictionary you can add key-value pairs in this way:

```
person['age'] = 71
person['hair'] = 'uniquely combed'
print(person)
```

Notice that if you assign a value (`71`) to a key that is already in the dictionary (`'age'`), then the old value (`70`) is replaced.

Exercise 173

What does this expression evaluate to?

```
{'name': 'Donald Trump', 'age': 70, 'job': 'President'}['name']
```

Exercise 174

Assuming the definition of the `person` dictionary above, what does this expression evaluate to? Compare to the expression in the previous exercise.

```
person['name']
```

Exercise 175

The *in* operator also works with dictionaries. Look at what these expressions reduce to and then try to figure out what *in* does when applied to a dictionary:

```
'name' in person
'age' in person
'job' in person
70 in person
'President' in person
'Donald Trump' in person
```

Exercise 176

Write and run this code with different values of `key` and read any error messages.

```
key = 3
# key = 'banana'
# key = 3.14159
# key = True
# key = {}
# key = []
d = {}
d[key] = 7
```

Are any of the values not allowed as keys?

Exercise 177

Do you think this will work?

```
person = {'name': 'Donald Trump',
                'age': 70,
                'job': 'President'}
print(person)
```

General exercises

Start by makeing dictionaries for (some of) the Trump family:

```
donald = {'name': 'Donald Trump', 'age': 70, 'job': 'President' }
melania = {'name': 'Melania Trump', 'age': 70, 'job': 'First lady' }
tiffany = {'name': 'Tiffany Trump', 'age': 23, 'job': 'Internet personality' }
ivanka = {'name': 'Ivanka Trump', 'age': 35, 'job': 'Top aide' }
```

Exercise 178

What do you think the following code produces? Do *all* of the substitution and reduction steps in your head, and only then try out the code.

```
donald['child'] = tiffany
melania['husband'] = donald

print(melania)
print(melania['husband']['child'])
```

Exercise 179

A dictionary can contain *any* kind of Python values, even lists or dictionaries. Consider the code below where we add a list of ex-wives to the Trump persona. Can you see why we need to check if the `'ex-wives'` key before we add to the list of ex-wives?

```
donald = {'name': 'Donald Trump', 'age': 70, 'job': 'President' }

if 'ex-wives' not in donald:
    donald['ex-wives'] = []
donald['ex-wives'].append('Marla Maples')
donald['ex-wives'].append('Ivana Trump')

print(donald)
```

Exercise 180

I case you wonder what the *type* of value a list is, or a dictionary, try this:

```
print("A list has type:", type([]))
print("A dictionary has type:", type({}))
```

Now the types `list` and `dict` are your friends too.

Exercise 181

Lists can also contain any kind of value. Consider this example. What do you think the following code produces? Do *all* the substitution and reduction steps in your head, and only then try out the code.

```
trump_family = [donald, melania, ivanka, tiffany]
print(trump_family)
print(trump_family[1]['job'])
```

Exercise 182

Write and run this code

```
amino_acids = {}
amino_acids['ATG'] = 'met'
amino_acids['TCT'] = 'ser'
amino_acids['TAC'] = 'tyr'

codon = 'TCT'
print("{} encodes {}".format(codon, amino_acids[codon]))
```

Bonus info: For each type of value the interpretation of length is different. In a string it is the number of characters, in a list is the number of values in the list and in a dictionary, it is the number of key-value pairs. How do you think Python knows which interpretation of length to use when the `len` function is called? This is where objects shine. `len(x)` just returns the value that `x.__len__()` returns. So the `len` function is defined roughly like this:

```
def len(x):
    return x.__len__()
```

Similarly, the `in` operator call a secret `__contains__` method.

Glueing values in sequence

Tuples

A tuple is a sequence of values just like a list. However, unlike a list, the elements of a list can not be changed. You cannot append to a tuple either. Once a tuple is made it is immutable (or unchangeable). To make a tuple you just use round brackets instead of square brackets:

```
fruits = ("apple", "banana", "cherry")
```

It may seem strange that Python has both tuples and lists. One reason is that whereas lists are more flexible, tuples are more efficient. We will not use tuples a lot, but you need to know what they are.

You can do most of the operations on a tuple that you can also do on a list. The following exercises should be easy if you remember how to do the same thing on lists:

Exercise 183

Find the number of elements in the `fruits` tuple using the `len` function.

Exercise 184

Extract the second element of the `fruits` tuple (`"banana"`) using indexing.

Exercise 185

Try to change the second element of the `fruits` tuple to `"apple"` and see what happens. It should be someting like this:

```
Traceback (most recent call last):
  File "script.py", line 2, in <module>
    fruits[3] = "apple"
TypeError: 'tuple' object does not support item assignment
```

You cannot change elements of a tuple because they are immutable (once made, they stay that way).

Tuple assignment

Python lets you assign a tuple of values to a tuple of variables like this:

```
father, mother, son = ("Donald", "Ivana", "Eric")
```

It does the same as the following three assignments:

```
father = "Donald"
mother = "Ivana"
son = "Eric"
```

When a tuple is made, the values are "packed" in sequence:

```
family = ("Donald", "Ivana", "Eric")
```

Using the same analogy, values can be "unpacked" using tuple assignment:

```
father, mother, son = family
```

The only requirement is that the number of variables equals the number of values in the tuple.

Once in a while, it is useful to swap the values of two variables. With conventional assignment statements, we have to use a temporary variable. For example, to swap a and b:

Exercise 186

Try this and read the error message:

```
family = ("Donald", "Ivana", "Eric")
father, mother = family
```

Exercise 187

Try this and read the error message:

```
family = ("Donald", "Ivana", "Eric")
father, mother, son, daughter = family
```

Compare to the error message in the previous exercise.

Exercise 188

Say want to swap the values of two variable `a` and `b`. To do that you would need to keep one of the values in an extra variable like this:

```
temp = a
a = b
b = temp
```

Can you come up with a simple and pretty way of swapping `a` and `b` in one statement, using what you have learned in this chapter? Maybe it occurs to you before you realize how it works, so make sure you can connect your solution to the rules of tuples and tuple assignment.

Iteration over values

This chapter is about how you repeat the same code for many different values -- and the many reasons why this is useful.

The for-loop

Programs often need to do repetitive things. Consider this example below where x is assigned a value that is then printed:

```
x = 1
print(x)
x = 5
print(x)
x = 3
print(x)
x = 7
print(x)
```

You can see that we do the same thing four times with the only difference that the variable x takes a new value each time. Now carefully write the alternative version below and compare what is printed to what was printed in the above example.

```
for x in [1, 5, 3, 7]:
    print(x)
```

It should be exactly the same. What you just wrote is called a *for-loop*. It is called a for-loop because it does something *for* each of many values -- in this case for each value in our list.

The statements nested under the for-loop are run as many times as there are values in our list, and every time they are run x is assigned a new value. The first time the statements are run, x is assigned the *first* value in the list. The second time they are run x is assigned the *second* value in the list. This continues until x has been assigned all the values in the list.

The semantics of a for-loop is as follows:

1. First, you write `for`.

2. Then you write the name of the variable that will be assigned a new value for each iteration of the loop (`x` in the above case).
3. Then you write `in`.
4. Then you write the name of an *iterable* or an expression that reduces to one. In the above case, it was the list `[1, 5, 3, 7]`.
5. The statements nested under the for loop are indented with four spaces just like with if-statements. These statements are executed once for every value in the *iterable*.

What is an *iterable*, you may ask? Actually, it is any kind of value that knows how to serve one value at a time until there are none left. Only objects that have an `__iter__` method can do this. If you try to iterate over a value that does not have an `__iter__` method you will get an error. Try the code below and see how Python complains that "'int' object is not iterable":

```
for x in 4:
    print(x)
```

Try these variations of the for-loop above and notice how the rules 1-5 apply in each case:

```
for x in [1, 5, 3, 7]:
    print(x)

list_of_numbers = [1, 5, 3, 7]
for x in list_of_numbers:
    print(x)

for x in [1, 5] + [3, 7]:
    print(x)
```

In each case, the expression after `in` reduces to the *value* `[1, 5, 3, 7]`, which then serves as the *iterable*.

Not only lists are iterable. Strings are too. Their '**iter**' method of a string tell it that it should serve once character at a time. Try this:

```
for character in 'banana':
    print(character)
```

Neat, right?.

In programming, you very often need to iterate over integer values, and sometimes quite a few (like the 250 million bases of the human chromosome one). It would be quite annoying if you had to manually make long lists of integers, so Python provides a builtin function called `range` that helps you out. It returns a special *iterator* value that lets you iterate over a specified range of numbers. Try the two examples below and

compare what is printed:

```
total = 0
for number in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]:
    total += number
print(total)

total = 0
for number in range(10):
    total += number
print(total)
```

You can see that using `range` works just like using a list of numbers, but the cool thing about range is that it does not return a list. It just serves one number at a time until it is done. This is also why you will not see a list if you try to print what `range` returns:

```
number_iterator = range(10)
print(number_iterator)
```

The `range` function needs three values to know which values to iterate over: "start", "end" and "step". If you do not give it all three arguments it will assume sensible defaults. Try this:

```
for i in range(0, 10, 1):
    print(i)

for i in range(0, 10):
    print(i)

for i in range(10):
    print(i)
```

You can see that the first and the last arguments default to 0 and 1. If you give it two arguments it will assume that they are "start" and "end". If you only give it one argument it will assume that it is the "end".

Exercise 189

What do you think the third argument to `range` specifies? Try these variations and see if you can figure it out:

```
for i in range(0, 10, 1):
    print(i)

for i in range(0, 10, 2):
    print(i)
```

```
for i in range(0, 10, 3):
    print(i)
```

Check the documentation once you have made up your mind.

Exercise 190
What will happen here:

```
for x in []:
    print(x)
```

and here:

```
for x in range(0):
    print(x)
```

and here:

```
for x in range(10, 10):
    print(x)
```

Exercise 191
The two examples below print the same. Make sure you understand why. Write and experiment with the code on your own.

```
list_of_words = ['one', 'two', 'three']

# example 1
for word in list_of_words:
    print(word)

# example 2
list_length = len(list_of_words)
for index in range(list_length):
    print(list_of_words[index])
```

Exercise 192
Finish the code below so all the even numbers go into one list and all the odd numbers go into the other (hint: remember the modulo operator?)

```
numbers = [4, 9, 6, 7, 4, 5, 3, 2, 6]
even = []
odd = []
for n in numbers:
    # your code here ...
```

Exercise 193

You can put any statements under the for-loop. Here it includes an if-statement that lets you generate a list of all the a characters in banana (in case you need that).

```
result = []
fruit = 'banana'
for character in fruit:
    if character == 'a':
        result.append(character)
print(result)
```

Now change the code so you instead get the *indexes* of the 'a' characters: `[1, 3, 5]`. Here are some hints:

1. You need a for-loop over a list of numbers.
2. `range(len(fruit))` may be relevant numbers :-).
3. `fruit[1]` substitutes for `'a'`.

Exercise 194

Imagine you want to throw a big party and that you have a rented a place with room for 100 people. Now you want to start inviting people. What kind of error do you get here and why?

```
friends = ["Mogens", "Preben", "Berit"]
invited = []
for index in range(100):
    invited.append(friends[index])
```

Exercise 195

You can also put a for-loop under another for loop, and the rules for *each* for loop are still those explained above: The statements nested under the for loop are indented with four spaces just like with if-statements. These statements are executed once for every value in the iterable. Think carefully about what you think is printed in the example below before you try it out.

```
for i in range(3):
    for j in range(3):
        print(i, j)
```

Make sure you understand i, j pairs are printed in the order they are.

Creating data structures

In this section, we will further train your ability to create such data structures using iteration.

Exercise 196

Imagine you want to count how many times each nucleotide appears in a DNA string like this one: `'ATGCCGATTAA'`. One way to proceed an account of this is in the form of a dictionary where the keys represent the different values we want to count (in this case `'A'`, `'T'`, `'C'` and `'G'`). The values associated with each key is the number of times we have seen that key (nucleotide). So we want to end up with a dictionary like this one (not necessarily with key-value pairs in this order):

```
{'G': 2, 'C': 2, 'T': 3, 'A': 4}
```

Remind yourself how you assign a value to an existing key in a dictionary. Here is some code to get you going:

```
dna = 'ATGCCGATTAA'
counts = {'G': 0, 'C': 0, 'T': 0, 'A': 0}
for base in dna:
    # Your code here...
```

Exercise 197

In exercise 196 we started by initializing a dictionary with a key for each nucleotide and the number zero for each key to show that we had not seen any of those nucleotides yet. *Then* we iterated over the values we wanted to count using the for-loop. This approach of cause only works if know which values we will encounter in the iteration. When counting nucleotides, this works because we know there are only four different nucleotides.

Often, however, we are faced with one or both of the following problems:

- The number of different values to count is very large. If we were to count English words, we would have to initialize a dictionary with more than 170.000 key value pairs (which would be somewhat impractical). If we were to count numbers this would be impossible (as there are infinitely many of those).
- We do not know what values we are going to count. It goes without saying that we cannot initialize a dictionary with keys if we do not know what they are.

We can solve this problem by only adding keys for the values we actually see in the iteration. To do this we need to change our approach from before in two ways:

1. We start with an *empty* dictionary.
2. For every value we iterate over, we check if that value is a key in our dictionary. If it is not, then we need to add it and pair it with the value `0`. You can test if a key is not in a dictionary using the `not in` operator:

```
if number not in counts:
    counts[number] = 0
```

Now use these hints to complete the code below so that it counts how many times each number appears in `number_list`.

```
counts = {}
number_list = [13, 51, 3, 51, 6, 42, 3]
for number in number_list:
    # Your code here...
```

When you are done you should have a dictionary like this (possibly with key-value pairs in a different order):

```
{3: 2, 42: 1, 51: 2, 13: 1, 6: 1}
```

Exercise 198

The counting technique you developed in exercise 197 lets you count pretty much anything that can be a key in a dictionary. Try using the same approach to count the number of each thing in this list:

```
stuff = ['sofa', 42, 42, 3.14159, 'sofa', 'Dragon']
```

Exercise 199

Instead of counting values, we sometimes need to split the values we iterate over into categories. Here we build a dictionary where the keys are the first letter of each word we iterate over, and the value associated with each key is a list of all the words that start with that letter. So if we iterate over the words in this list:

```
names = ['apricot', 'banana', 'ananas', 'apple', 'cherry']
```

we should end up with this dictionary (not necessarily with key-value pairs in that order):

```
{'c': ['cherry'], 'a': ['apricot', 'ananas', 'apple'], 'b': ['banana']}
```

Now figure out how to reorder and indent the statements below to produce code that performs this task:

```
names = ['apricot', 'banana', 'ananas', 'apple', 'cherry']
words[first_letter].append(fruit)
first_letter = fruit[0]
for fruit in names:
words = {'a': [], 'b': [], 'c': []}
```

Exercise 200

Produce the same dictionary as in exercise 199 by reordering and indenting the following statements:

```
first_letter = fruit[0]
words = {}
words[first_letter].append(fruit)
if first_letter not in words:
names = ['apricot', 'banana', 'ananas', 'apple', 'cherry']
words[first_letter] = []
for fruit in names:
```

Compare the solution to that in exercise 199. How is it different? Is it more genetic? How does the difference relate to the difference between exercise 196 and exercise 197?

Exercise 201

Decide what you think the code below does and why you think so. Do *every* step in your head including all the substitutions and reductions. Then write the code carefully and run it.

```
nr_list = [10, 20, 30]
combinations = []
for a in nr_list:
    for b in nr_list:
        pair = [a, b]
        combinations.append(pair)
```

The `combinations` list becomes:

```
[[10, 10], [10, 20], [10, 30],
 [20, 10], [20, 20], [20, 30],
 [30, 10], [30, 20], [30, 30]]
```

Here I broke over three lines to make it fit on the page. You should print your own `combinations` list to make sure you got the code right.

Exercise 202

The code in exercise 201 printed all combinations of numbers in the list -- including those where the two numbers are the same. Change the code above so these pairs are *not* printed. You should end up with this list:

```
[[10, 20], [10, 30], [20, 10], [20, 30], [30, 10], [30, 20]]
```

Exercise 203

Can you solve the same task as in exercise 202, by modifying the code below? Begin by making sure you understand why it produces the same list as that in exercise 201. If you have trouble with that, then have another look at exercise 195.

```
nr_list = [10, 20, 30]
combinations = []
```

```
for i in range(len(nr_list)):
    for j in range(len(nr_list)):
        pair = [nr_list[i], nr_list[j]]
        combinations.append(pair)
```

Exercise 204

The code in the exercise above printed all combinations of *different* numbers in the list. But you can see that each pair of numbers still appear twice if you do not take their order into account (e.g. `[10, 30]` are the same two numbers as `[30, 10]`). Change the code you wrote for the previous exercise so these pairs are not printed. You should end up with a list like this:

```
[[10, 20], [10, 30], [20, 30]]
```

Hint: the easiest way to do it is to use the value of `i` to change the range of numbers you iterate over in the second for-loop.

Exercise 205

Sometimes programmers (like you) work with matrices of numbers like the one below:

```
[[0, 1, 2, 3, 4],
 [0, 1, 2, 3, 4],
 [0, 1, 2, 3, 4],
 [0, 1, 2, 3, 4],
 [0, 1, 2, 3, 4]]
```

Here I wrote the list in a nice way so you can see that a matrix is really just *a list of lists*. When you print it looks like this:

```
[[0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4]]
```

Do you think you can write some code that produces this matrix? If you let out a sigh just now, then reread the sections on lists and for-loops. You may think absorbed all the information you could when you read it the first time, but with your practice since then, you may be able to understand it at a deeper level the second or third time you read it.

The code below produces the matrix above. There are several tricky parts that you need to make sure you understand. In line three we add an empty list to the list of lists. In line five we add the value of `j` to the list at index `i` in the list of lists. Go over this many times in your head and with pen and paper.

```
matrix = []
for i in range(5):
    matrix.append([])
    for j in range(5):
        matrix[i].append(j)
```

Exercise 206

If you understand how you created the matrix in exercise 205, you should be able to produce the matrix below using a small modification to the code from exercise 205.

```
[[1, 1, 1, 1, 1],
 [2, 2, 2, 2, 2],
 [3, 3, 3, 3, 3],
 [4, 4, 4, 4, 4],
 [5, 5, 5, 5, 5]]
```

Exercise 207

Now produce this matrix:

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

Exercise 208

Can you write some code that produces this matrix?:

```
[[ 0, -1, -2, -3, -4],
 [-1,  0,  0,  0,  0],
 [-2,  0,  0,  0,  0],
 [-3,  0,  0,  0,  0],
 [-4,  0,  0,  0,  0]]
```

General exercises

> **Heads up:** By now you have learned a lot and the general exercises, which serve to keep it all current, get more complicated. But remember: even though the code may mix lists, for-loops, and functions, the *rules* for lists, for-loops and functions are *not* mixed. The separate simple rules for a list, a for-loop and a function are *still* the same. If you get confused, it is time revisit the sections about each separate topic. You will probably have to do that many times during the course.

Exercise 209

Write a function, `square_numbers`, that takes a list of numbers as argument and returns a new list with the numbers squared.

```
# write your function definition here ...
```

```
numbers = [1, 5, 3, 7]

# then you can call it like this:
squared = squared_numbers(numbers)
```

Exercise 210

Write a function `count_characters`, which takes a string argument and returns a dictionary with the counts of each character in the string. When you call the function like this:

```
count_characters('banana')
```

it must return (not necessarily with key-value pairs in that order):

```
{'n': 2, 'b': 1, 'a': 3}
```

The technique you should use is the one you learned in exercise 197. Here we just iterate over a string of characters instead of over a list of numbers. Here is a bit of code to help you along...

```
def count_characters(text):
    counts = {}
    # fill in the missing code ...

    return counts
```

Exercise 211

Use the function you made in the previous exercise to construct the following data structure:

```
{ 'banana': {'b': 1, 'a': 3, 'n': 2},
  'apple': {'a': 1, 'e': 1, 'p': 2, 'l': 1},
  'ananas': {'a': 3, 's': 1, 'n': 2} }
```

from this list:

```
['banana', 'ananas', 'apple']
```

Here is some code to help you along:

```
my_database = {}
for word in ['banana', 'ananas', 'apple']:
    my_database[word] =  # you figure this out...
```

Once you are done, what value do you think `my_database['banana']` represents? I.e. what will it reduce to if used in an expression? And what value does `my_database['banana']['a']` represent?

Exercise 212

Read the code below and make sure you understand each single step before you write any of it. Revisit previous sections if you must, go look in the Python documentation. *Then* write and run the code - and enjoy that it was exactly what you expected.

```
def get_words(text, search_string):
    hits = []
    for word in text.split():
         if search_string in word:
            hits.append(word)
    return hits

s = 'eenie meenie minie moe'
nie_words = get_words(s, 'nie')
m_words = get_words(s, 'm')

print(' '.join(nie_words))
print(' '.join(m_words))
```

Exercise 213

This larger will take you through some of the most common string manipulations. A palindrome is a string that is spelled the same way backward and forwards.

Write a function, `is_palindrome`, which takes one argument:

- A string.

The function must return:

- `True` if the string argument is a palindrome and `False` otherwise.

Example usage:

```
is_palindrome('abcba')
```

should return `True` and

```
is_palindrome('foo')
```

should return `False`

One approach to this is to run through s from the first to the middle character and for each character check if the character is equal to the character at the same index from the right rather than the left. Remember that the first character of a string is at index `0` and the last at index `-1`, the second character is at index `1` and the second last at index `-2` and so forth.

Since you need to run through the string from the first to the middle character you first need to figure out how many characters that corresponds to. Say your palindrome is

`"ACTGTCA"`, then the number of indexes you need to loop over with a for loop is:

```
s = "ACTGTCA"
nr_indexes = len(s)//2
```

Figure out how to make `range()` return indexes you can use to access the characters in the first half of the sequence. Then make a for loop where you iterate over the indexes you get from `range()`. Try to make the for-loop print out the first half of the characters, just to make sure you are using the right indexes.

Once you get this far you need to compare each character from the first half of the corresponding ones starting from the other end of the palindrome. Figure out how to change each index used for the first half to the corresponding index for the other half so you can compare the relevant pairs. (You need to compare index `0` with `-1`, `1` with `-2` and so on...)

Now try to make the for-loop print both the character from the first half and the corresponding character from the other end. If you got the indexes right you will see that the A print with the A from the other end, the C with the C and so on.

Write an if-statement in the for-loop that tests if the two corresponding characters are the same. If the string is a palindrome, then each pair is identical. So as soon as you see a pair which is not identical, you know it is not a palindrome and you can let your function return `False` like this:

```
if left_character != right_character:
    return False
```

Remember that the function ends as soon as it encounters a `return` statement.

If all pairs pass the test, it means that the string is a palindrome, and the function should return `True` when exiting the for-loop.

Testing your code

This chapter is about how you figure out if the code you wrote actually solves the problem in the way you intended. You will be surprised how often that is not the case -- even for seasoned programmers.

Why test your code?

There are tons of reasons why you should test your code. Here are what I think are the two most important ones:

1. **Makes you think:** Testing forces you to slow down and think about exactly what the code is supposed to do. By deciding what tests to do before you start coding, you try to anticipate errors and cases that are not covered by the way you want to solve the problem at hand. The notion of falsification is important in science and in coding too. The idea is that you should try to prove that your idea is wrong and only consider it valid only if this process fails. So testing motivates you to think about ways to break their code, thereby helping you solve your programming problem in a way that is general and robust so that it does exactly what you expect it to do.
2. **Gives you peace of mind:** Testing increases your confidence that a function you have written actually works as it is supposed to so that you can now stop thinking about how it is implemented and focus on using it as a component in solving a larger problem. Having set up a series of tests also allows you to change and improve the implementation of your function without having to worry that it stops working they way it is supposed to. As long as it passes all the tests is should be ok.

Testing of code is a *big* thing in programming. Professional consistently test their code. In time you will too, but in this course, you will only do the very basic testing yourself. Instead, you will have access to readymade testing suites made especially for each of your programming projects.

Basic testing

Say that you are asked to make a function that takes a string argument and returns `True` if that string is a palindrome and `False` otherwise (hypothetical example). Then you start thinking about which strings should make the function should return `True` and which should return `False`. Once you have defined you `is_palindrome` function you can set up some fairly obvious tests like this:

```
print(is_palindome('123321') == True)
print(is_palindome('ATGGTA') == True)
print(is_palindome('ATGATG') == False)
print(is_palindome('XY') == False)
```

But if you keep thinking, maybe you come up with more tests to cover all the different types of cases you may encounter:

```
print(is_palindome('12321') == True) # uneven length
print(is_palindome('121') == True) # uneven length
print(is_palindome('AA') == True)
print(is_palindome('A') == True) # single char
```

The project testing utility

To keep you focused on the programming part, each programming project comes with a ready-made suite of tests of the functions you are asked to implement. So for each function, you can run tests to make sure it implements the behaviour it is supposed to.

Each project comes with two files. They have their names for a good reason, so do not change them. In the project about translating DNA that you will do, they are called `translationproject.py` and `test_translationproject.py`.

To be able to test your functions, you *must* write your code in the file called `translationproject.py`. To run your code you type this in the Terminal as usual:

```
python translationproject.py
```

To test the functions as you complete each one, you can run the test script `test_translationproject.py` like this:

```
python test_translationproject.py
```

The code in `test_translationproject.py` reads your code in `translationproject.py` and performs a series of tests of each function. When you run the test script four things may happen depending on the state of your code:

Case 1: If you did not yet implement all the functions, the test script will remind you (once for each test) that you did not implement the functions with the names required.

```
ATTENTION! A test of "translate_codon" was skipped because this function was not defined.
Functions not correctly defined are marked as FAILED. Please make sure this is what you intended!
```

If you have implemented a function but misspelled its name, you will also get this type of reminder. The reminders are meant as a safeguard to ensure that you do not hand in the assignment with missing or misspelled function definitions.

Case 2: If a test for one the functions you have written fails, the testing is aborted and the script prints some information to help you understand what the problem could be. Say you wrote the function `translate_codon` wrongly so that it always returns `M` for some reason:

```
def translate_codon(codon):
    return 'M' # crazy
```

then you would get this message:

```
FAILED TEST CASE: test_translate_codon_2

MESSAGE:
    The call:

        translate_codon('TAA')

    returned:

        'M'

    However, it should return:

        '*'

======================================================================
Ran 4 tests in 0.001s

FAILED (failures=1)
```

It is now left to you to figure out why your function returns the wrong value when called with these arguments.

Case 3: If you defined all functions correctly and they all work the way they are supposed to, then the test script just prints:

```
Ran 14 tests in 0.140s

OK
```

General exercises

Exercise 214

Inspect the code below and figure out why it does not print that you are a super star. Test the function using various input and figure out the mistake.

```
def even_number(x):
    if x % 2:
        return False

if even_number(4):
    print('You are a super star!')
```

Working with files

This chapter covers the bare necessities of how to make your program read data from a file on your computer and how to make it create a file that it can write results to.

Writing files

To interact with a file on your harddisk you need to know the name of the file and whether you want to write to it or read from it. Then you can use the builtin function `open` to create a file object that lets you read or write to that file. The `open` function takes two arguments: The first is a string, which gives the name of the file. The second argument is also a string and should be `'w'` for "write" if you want to write to the file or `'r'` for "read" if you want to read from the file. To keep things simple we will assume that the file you want to open is always in the same folder (directory) as the Python script that calls the `open` function.

Exercise 215

Try to write the code below and run it:

```
f = open('workfile.txt', 'w')
f.write("First line\n")
f.write("Second line\n")
f.close()
```

Now open the `workfile.txt` in *SublimeText* and see what is in it now. It should contain:

```
First line
Second line
```

Lets break down what happened:

1. You used the `open` builtin function to open a file called "workfile.txt" in *writing* mode using the `'w'` as the second argument.
2. You then wrote the string `"First line\n"` to the file using the `write` method of the file object.
3. You wrote another string `"Second line\n"` to the file using the `write` method of the file object.

4. You closed the file using the `close` method of the file object.

Note that if you open a file for writing, a file with that name is created. If a file of that name already exist, it is overwritten.

Exercise 216
Close workfile.txt in *SublimeText* again and change your program above to this (removing the `\n` characters):

```
f = open('workfile.txt', 'w')
f.write("First line")
f.write("Second line")
f.close()
```

What do you think the content of workfile.txt is now? Decide before you open workfile.txt in *SublimeText* again and have a look. What do you think the `\n` character represents?

Exercise 217
Close workfile.txt in *SublimeText* and change your program above to this:

```
f = open('workfile.txt', 'w')
f.write("First line\nSecond line\n")
f.close()
```

Can you see how that is equivalent to what you did before? Open workfile.txt in *SublimeText* again and have a look.

Exercise 218
You can also make `print` write to a file instead of the terminal. That way your output ends up in the file instead of the terminal. To make print write to a file, you need to use the `file` keyword argument to give print the file object that represents the file you want to write to (`file=f` below). Try to write the code below and run it:

```
f = open('workfile.txt', 'w')
print("First line", file=f)
print("Second line, file=f")
f.close()
```

Compare the the code to that in exercise 215. Notice how the strings we print do end with a newline. This is because the default behaviour for `print` is to add a newline to the end of what it prints - just like when you print to the terminal.

Reading files

When you want to read a from an existing file you give the `open` function the name of that file and specify `'r'` for reading as second argument. If the file you name does not

exist, Python will tell you that it does not exist (it is nice like that). Before you head into the next rest of this section make sure you redo exercise 215 so the content of workfile.txt is:

```
First line
Second line
```

Exercise 219

```
f = open('workfile.txt', 'r')
file_content = f.read()
print(file_content)
f.close()
```

Lets break down what happened:

1. You used the `open` builtin function to open a file called "workfile.txt in *reading* mode using the `'r'` as the second argument.
2. You then read the content of the file using the `read` method, which return the contents as a string.
3. You printed the string.
4. You closed the file using the `close` method of the file object.

Exercise 220

Try to read from the file after you close it:

```
f = open('workfile', 'r')
f.close()
file_content = f.read()
```

Do you get an error? Which one? Do you understand why?

Exercise 221

You can use the `readline` method to read one line at a time. What do you think happens if you run this code:

```
f = open('workfile.txt', 'r')
line = f.readline()
print(line)
line = f.readline()
print(line)
line = f.readline()
print(line)
```

Once you decide, try it out. What is printed in the last print statement? The thing is, the file object keeps track of how much of the file it has read. Once it is at the end you can read as much as you like -- there is nothing left. If you want to start reading from

the top of the file you can close and open the file again. Try to insert the following two statements at various places in the code above and see what happens?

```
f.close()
f = open('workfile.txt', 'r')
```

Exercise 222

Look at the code below and figure out for yourself what it does:

```
input_file = open('workfile.txt', 'r')
output_file = open('results.txt', 'w')

for line in input_file:
    line = line.upper()
    output_file.write(line)
```

Then run it and open results.txt in *SublimeText* and see what it produced.

Were you surprised that the file object can be an iterator in a for-loop? Just like stings can iterate over characters, lists can iterate over values and dictionaries can iterate over keys, file objects can iterate over the lines in the file.

Try to modify your code to use the `print` function instead of the `write` method (see exercise 218).

General exercises

Exercise 223

Write a function `read_file` that takes the name of a file as argument. The file should read the content of the file and return it. Like:

```
content_of_file = read_file('some_file.txt')
```

Exercise 224

Write a function that takes the name of two files as arguments. The file should read the content of the first file and write it to the second file.

```
copy_file('some_file.txt', 'other_file.txt')
```

Project: Translating open reading frames

This chapter is about translating DNA into protein. If bacteria can do it, so can you.

In this project you will write the code needed to translate an open reading frame (ORF) on a DNA sequence into into the corresponding sequence of amino acids.

You need two project files for the project, which you can download from Blackboard:

- **translationproject.py** is an empty file where you must write your code.
- **test_translationproject.py** is the test program that lets you test the code you write in translationproject.py.

The file `translationproject.py` is for your code. As usual `test_translationproject.py` is the script you use to test the functions you write for this project.

In this project you will need a data structure that pairs each codon to the amino acid it encodes. This is an obvious use of a dictionary and at the top of `translationproject.py` I have defined such a dictionary you can use. Defining it outside the functions means that it is visible inside all your functions (unless you define *another* variable called `codon_map` inside a function). Defining variables globally to your program sometimes make sense if some value can be considered a *constant* in your program and is *never* changed.

Heads up: It is normally very bad programming style to access variables outside functions in this way because it may have all kinds of unexpected side effects across function calls. So make it a rule for yourself that code *inside* a function should never to access variables *outside* the function. The reason we define `codon_map` globally in this project is to help you understand that when Python cannot find a variable inside a function, it looks outside the function to find it. In this project functions will find `codon_map` in this way. However, as I already said, you should *never* do this yourself. The chance that you make an unexpected mistake is overwhelming.

Translating a single codon

Write a function, `translate_codon` that takes one argument:

1. A string, which is a codon.

The function should return:

- A string of length one (one character). If the string argument is a valid codon then this letter should the be an amino acid letter specified by the `codon_map` dictionary. Note that stop codons are represented by a star (`'*'`). If the string argument is *not* a valid codon, the function must return `'?'`.

Example usage:

```
translate_codon('ACG')
```

should return

```
'T'
```

Before you start coding you should always outline for yourself intuitively what you need to do to complete the task at hand. In this case want to translate, or map, between a three letter string, codon, and the corresponding one letter string for the amino acid that the codon corresponds to. Notice that the keys in the `codon_map` dictionary are in upper case, so you must make sure that the keys *you* use are also in upper case. You can translate codon into an upper case version of itself using the upper() method.

Try this out first:

```
codon = 'TTG'
amino_acid = codon_map[codon]
print(amino_acid)
```

Now write the function so it uses the string parameter as a key to look up the corresponding amino acid letter and returns this letter. Before you go on, make a function that does only that.

Before you are completely done you need to make your function handle the situation when the argument to the function is not a key in the `codon_map` dictionary. Use an if-else construct to handle the two cases. The boolean expression must test if the function argument is a key in `codon_map`. Remember that you can use the `in` operator to do this.

Splitting an open reading frame into codons

To translate an entire open reading frame into the corresponding amino acid sequence, you need to split the ORF sequence into codons. When we have done that we can translate each codon using the function `translate_codon` you just wrote.

Write a function, `split_codons`, that takes one argument:

1. A string, which is an ORF sequence

The function must return:

- A list of strings. Each string must have length 3 and must represent the-non overlapping triplets in the same sequence as they appear in the string given as argument.

Example usage:

```
split_codons('ATGTATGCCTGA')
```

should return

```
['ATG', 'TAT', 'GCC', 'TGA']
```

Divide the problem into simpler tasks like above. You need to loop over the sequence to perform operations on it. Start by writing a function that prints each character:

```
def split_codons(orf):
    for i in range(len(orf)):
        print(orf[i])
```

Now try to figure out how you can modify the function to make it move over the sequence in jumps of three. Look at the documentation for the `range` function to see how you can make it iterate over numbers with increments of three like this: 0, 3, 6, 9, 12, ... Modify your function so that it now prints every third character.

What you want is obviously not every third character. You want three characters. I.e. every third character *and* the two characters that come right after. You can use the index in the for loop to get the corresponding codon using slicing. Modify your function so that it prints each codon.

Now all that remains is to put each codon on a list that you can return from the function. You can define a list before your for-loop so you have a list to add codons to.

Translating an open reading frame

Now you can use the two functions `split_codons` and `translate_codon` to write a function that translates an ORF into a protein sequence.

Write a function, `translate_orf`, that takes one argument:

1. A string, which is a DNA sequence.

The function must return

- A string, which is the protein sequence translated from the ORF sequence argument.

Example usage:

```
translate_orf('ATGGAGCTTANCAAATAG')
```

should return

```
'MEL?K*'
```

Project: Identifying the subtype of an HIV sequence

This chapter is a programming project where you will put your new programming skills to use analyzing an HIV DNA sequences.

You have now been introduced to all the programming rules you will see in this course. You now know all the building blocks required to write *any* program -- literally *any*. The reason why computer geeks are good at what they do is not that they know some incomprehensible secrets. It is because they practiced, a lot. With practice, the simple rules you know now will let you write anything from first-person shooter games over jumbo jet autopilots to scripts for simple problems in bioinformatics. In the last three chapters, we will train your ability to solve bioinformatics problems by putting together all the things you have learned.

The programming project in this chapter deals with DNA sequences from HIV viruses. There are two types of HIV: HIV-1, which is by far the most common, and HIV-2, which is mostly found in West Africa. HIV-1 vira are divided into groups M, N, O, and P. The most important group M (for major) is one primarily responsible for the global epidemic. Group M is further divided into subtypes A, B, C, D, F, G, J, K, and CRFs. In this project we will look at sequences from the subtypes A, B, C, and D. You have multiple database sequences for each of these four subtypes and you have one *unknown* sequence from a patient that you need to assign to either subtype A, B, C or D. To do this you will have to write a program that *predicts* the subtype of the unknown sequence. How cool is that?

Under "Assignments" on the Blackboard course page, you will find an assignment with the same name as this chapter. There you can download the data files you need for this project:

- `unknown_type.txt` contains an HIV sequence of unknown subtype
- `subtypeA.txt` contains a database of HIV sequences of subtype A
- `subtypeB.txt` contains a database of HIV sequences of subtype B
- `subtypeC.txt` contains a database of HIV sequences of subtype C
- `subtypeD.txt` contains a database of HIV sequences of subtype D

You also need to download the two project files:

- `hivproject.py` is an empty file where you must write your code.

- `test_hivproject.py` is the test program that lets you test the code you write in `hivproject.py`.

Put all seven files in a folder dedicated to this project. On most computers you can right-click on the link and choose "Save file as..." or "Download linked file".

Now open each file in *SublimeText* and have a look at what is in the data files. (Do *not* change them in any way and do not save them after viewing. If SublimeText asks you if you want to save it before closing, say *no*.) How many sequences are there in each file?

The project is divided into the following parts:

- Compute the similarity of two sequences
- Read the HIV sequences into your program.
- Assess the similarity of your unknown HIV sequence to each of the HIV sequences with known subtype.
- Find the maximum similarity of your unknown sequence to sequences from each subtype.
- Identify the HIV subtype of your sequence as the subtype of that sequence that your sequence is most similar to.

Make sure you read the entire exercise and understand what you are supposed to do before you begin!

Compute the similarity of two sequences

We need to compare our unknown HIV sequence to all the HIV sequences of known subtypes. That way we can identify the sequence of a known subtype that is most similar to your unknown sequence. We will then assume that our unknown sequence has the same subtype as this sequence. To accomplish this we first need to write some code that compares two sequences so we can compare our HIV sequence to each of the other HIV sequences.

Compare two sequences

Write a function `sequence_similarity` that takes two arguments:

1. A string which is a DNA sequence.
2. A string of the same length as argument one, which is also a DNA sequence.

The function must return:

- A float, which is the proportion of bases that are the same in two DNA sequences.

Example usage:

```
sequence_similarity('AGTC' 'AGTT')
```

should return `0.75`.

Start out defining your function like this:

```
def sequence_similarity(seq1, seq2):
    # your code here...
```

Remember that `range(len(seq1))` generates the numbers you can use to index the string `seq1`. You can use those numbers as indexes to look up positions in both strings. You will need a for-loop in your function and a variable that keeps track of how many similarities you have seen as you iterate through the sequences.

Compare aligned sequences

All sequences, including the unknown sequence, are from the same multiple alignment. This ensures that sequence positions match up across all sequences but also means that a lot of gap characters (`'-'`) are inserted. To compute similarities between such sequences you need to make function much like `seqeuence_similarity` that does not consider sequence positions where both bases are a gap (`'-'`) characters. In other words, you must not only count the number of characters that are the same, you also need to count how many alignment columns that are `"-"` for both sequences. E.g. the following mini alignment has five such columns and four columns where the bases are the same. So in the following alignment, the similarity is 0.8 (4/5):

```
A-CT-A
A-CTTA
```

Write a function `alignment_similarity` that takes two arguments:

1. A string which is a DNA sequence with gap characters.
2. A string of the same length as argument one, which is also a DNA sequence with gap characters.

The function must return:

- A float, which is the proportion of bases that are the same in two DNA sequences.

```
alignment_similarity('A-CT-A', 'A-CTTA')
```

should return `0.8`.

Hint: Use an if-statement to test if the two characters at some index are equal to `'-'` in both sequences. You can use an expression like this:

```
seq1[i] == '-' and seq2[i] == '-'
```

Once your function has computed both the number of identical bases and the number of alignment columns that are not both `'-'`, you can have it return the similarity as the ratio of the two.

Read the HIV sequences into your program

To use your `alignment_similarity` function to assess similarity between your unknown sequence and the sequences of known subtype, you need to read the sequences into your program. Here is a function that will read the sequences from one of the files you downloaded into a list:

```
def read_data(file_name):
    f = open(file_name)
    sequence_list = list()
    for line in f:
        seq = line.strip()
        sequence_list.append(seq)
    f.close()
    return sequence_list
```

You can use that function to read the unknown sequence into your program:

```
unknown_list = read_data('unknown_type.txt')
```

In this case, the list only contains the one unknown HIV sequence in unknown_type.txt.

You also need to load the typed HIV sequences into your program. Here is a function that returns a dictionary in which the keys are subtypes (`'A'`, `'B'`, `'C'` and `'D'`) and each value is a lists of sequences with that subtype:

```
def load_typed_sequences():
    return {'A': read_data('subtypeA.txt'),
            'B': read_data('subtypeB.txt'),
            'C': read_data('subtypeC.txt'),
            'D': read_data('subtypeD.txt') }
```

If you use the function like this:

```
typed_data = load_typed_sequences()
```

then you can access the list of sequences of subtype A like this: `typed_data['A']`.

Compare your HIV sequence to HIV sequences of known subtype

To type you HIV sequence you must compare your sequence to all the database sequences to see which group has the best matching sequence.

Write a function `get_similarities` that takes two arguments:

1. A string, which is your unknown HIV sequence.

2. A list of strings, each of which is an HIV sequence of known type.

The function must return:

- A list of floats, which should be the similarities between the unknown sequence given as the first argument and the list of sequences given as the second argument.

Example usage:

```
get_similarities(unknown_list[0], typed_data['A'])
```

should return:

```
[0.8553288474061211, 0.8721742704480066,
 0.854924397221087, 0.8481709291032696,
 0.8498330281159108]
```

The function should use the function `alignment_similarity` to compare your unknown sequence (`unknown_list[0]`) to each of the sequences of some subtype. Start out like this:

```
def get_similarities(unknown, typed_sequences):
    # Your code here...
```

In your function you need to define a list that you can *append* the similarities you compute to:

```
similarities = []
```

This is the list of results that your function must return. To compute the similarity between you unknown sequence and each of the sequences of known subtype, you can use your `alignment_similarity` function inside a for-loop.

Compute maximum similarity to each subtype

To predict the subtype of the unknown HIV sequence you need to compare the unknown sequence to all the sequences of each of the different subtypes. The subtype of the sequence with the highest similarity to your unknown sequence is then our predicted subtype (or our best guess).

Write a function `get_max_similarities` that takes two arguments:

1. A string, which is your unknown HIV sequence.
2. A dictionary, like the one returned by `load_typed_sequences`.

The function must return:

- A dictionary, in which keys are strings representing each subtype (`'A'`, `'B'`, `'C'`, and `'D'`) and values are floats representing the maximum similarity between the

unknown sequence and the sequences of a subtype. The dictionary *could* look like this (it does not, you need to compute the similarities yourself.):

```
{'A': 0.89, 'B': 0.95, 'C': 0.82, 'D': 0.99}
```

To get the highest number in a list of numbers, you can use the `max` function in Python. It works like this:

```
numbers = [3, 8, 53, 12, 7]
print(max(numbers))  # prints 53
```

For example, to get the highest similarity between the unknown sequence and sequences in `typed_data['A']`:

```
subtypeA_similarities = get_similarities(unknown_list[0], typed_data['A'])
subtypeA_max = max(subtypeA_similarities)
```

Identify the HIV subtype

Now for the grand finale! You ultimately want to be able to write code like this:

```
unknown_list = read_data('unknown_type.txt')
typed_data = load_typed_sequences()
subtype = predict_subtype(unknown_list[0], typed_data)
print("Patient HIV is subtype {}".format(subtype))
```

So all you need now is the `predict_subtype` function.

Write a function `predict_subtype` that takes two arguments:

1. A string, which is your unknown HIV sequence.
2. A dictionary, like the one returned by `load_typed_sequences`.

The function must return:

- A string of length one (either `'A'`, `'B'`, `'C'`, or `'D'`) representing the predicted subtype of your unknown HIV sequence.

The function should use `get_max_similarities` to compute the dictionary of max similarities and then extract from that dictionary the key with the highest value (similarity). So the function must return `'A'` if the unknown sequence is most similar to a sequence of subtype A, `'B'` if the unknown sequence is most similar to a sequence of subtype B and so on.

Congratulations, you are now a bioinformatician.

Project: Codon usage in *Streptocuccus* bacteria

Codon usage bias refers to differences in the frequency of occurrence of synonymous codons in coding DNA. There are 64 different codons (61 codons encoding for amino acids plus 3 stop codons) but only 20 different translated amino acids. The overabundance in the number of codons allows many amino acids to be encoded by more than one codon. Because of such redundancy, it is said that the genetic code is degenerate. Different organisms often show particular preferences for one of the several codons that encode the same amino acid, that is, a greater frequency of one will be found than expected by chance. How such preferences arise is a much-debated area of molecular evolution.

Given an open reading frame (ORF), you must compute the codon usage. Your goal is to create a data structure where you can look up an amino acid and the frequencies of codons in that encode that amino acid the ORF. Here is a made-up example:

```
{'A': {'GCA': 0.0, 'GCC': 0.0, 'GCT': 1.0, 'GCG': 0.0},
 'C': {'TGC': 0.0, 'TGT': 1.0},
 'E': {'GAG': 0.2, 'GAA': 0.8},
 'D': {'GAT': 1.0, 'GAC': 0.0},
 'G': {'GGT': 0.3, 'GGG': 0.0, 'GGA': 0.7, 'GGC': 0.0},
 'F': {'TTC': 0.0, 'TTT': 1.0},
 'I': {'ATT': 1.0, 'ATC': 0.0, 'ATA': 0.0},
 'H': {'CAC': 0.0, 'CAT': 1.0},
 'K': {'AAG': 0.2, 'AAA': 0.8},
 '*': {'TAG': 0.0, 'TGA': 1.0, 'TAA': 0.0}, 'M': {'ATG': 1.0},
 'L': {'CTT': 0.0, 'CTG': 0.6, 'CTA': 0.0, 'CTC':0.0, 'TTA': 0.4, 'TTG': 0.0},
 'N': {'AAT': 0.5, 'AAC': 0.5},
 'Q': {'CAA': 0.6, 'CAG': 0.4},
 'P': {'CCT': 0.5, 'CCG': 0.0, 'CCA': 0.5, 'CCC': 0.0},
 'S': {'TCT': 0.0, 'AGC': 0.0, 'TCG': 0.0, 'AGT': 0.5, 'TCC': 0.0, 'TCA': 0.5},
 'R': {'CGA': 0.5, 'CGC': 0.0, 'AGA': 0.5, 'AGG': 0.0, 'CGG': 0.0, 'CGT': 0.0},
 'T': {'ACC': 0.0, 'ACA': 0.0, 'ACG': 0.0, 'ACT': 1.0},
 'W': {'TGG': 1.0},
 'V': {'GTA': 0.6, 'GTC': 0.0, 'GTT': 0.2, 'GTG': 0.2},
 'Y': {'TAT': 1.0, 'TAC': 0.0}}
```

That data structure is a dictionary with keys corresponding to amino acids (i.e. single letter strings designating an amino acid such as `'R'` for arginine). The value associated with *each* amino acid key is also a dictionary, and the keys of *this* dictionary should be the different codons that encode the amino acid. The value associated with each codon key should be a number, representing the frequency with which that codon is used to encode that amino acid. The final data structure should only include amino acids that are found in the ORF.

Under "Assignments" on the Blackboard course page, you will find an assignment with the same name as this chapter. There you can download the data file you need for this project:

- `sample_orfs.txt` contains open reading frame sequences you can work on.

You also need to download the two project files:

- `codonbiasproject.py` is an empty file where you must write your code.
- `test_codonbiasproject.py` is the test program that lets you test the code you write in `codonbiasproject.py`.

Put all three files in a folder dedicated to this project. On most computers you can right-click on the link and choose "Save file as..." or "Download linked file".

Now open each file in *SublimeText* and have a look at what is in `sample_orfs.txt`. (Do *not* change it in any way and do not save it after viewing. If SublimeText asks you if you want to save it before closing, say *no*.) How many sequences are there in each file?

As in the translation project, you will need a data structure that pairs each codon to the amino acid it encodes. This is an obvious use of a dictionary and at the top of `codonbiasproject.py` I have defined such a dictionary you can use. Defining it outside the functions means that it is visible inside all your functions (unless you define *another* variable called `codon_map` inside a function). Defining variables globally like this sometimes make sense if some value can be considered a *constant* in your program and is *never* changed.

> **Heads up:** It is normally very bad programming style to access variables outside functions in this way because it may have all kinds of unexpected side effects across function calls. So make it a rule for yourself that code *inside* a function should never to access variables *outside* the function. The reason we define `codon_map` globally in this project is to help you understand that when Python cannot find a variable inside a function, it looks outside the function to find it. In this project, functions will find `codon_map` in this way. However, as I already said, you should *never* do this yourself. The chance that you make an unexpected mistake is overwhelming.

The project is split into four parts:

1. Read an open reading frame (ORF) into your script and count the codons.
2. Group the codon counts by the amino acids the codons encode.
3. Convert counts to frequencies.
4. Build the data structure representing the codon bias information.

Read an open reading frame and count its codons

Read ORFs from a file

You can use this code to read the ORFs into your script:

```
f = open('sample_orfs.txt', 'r')
orf_list = list()
for line in f:
    seq = line.strip()
    orf_list.append(seq)
f.close()
```

Try to print the list to see it. Then pick out the first ORF in the list so you can use that to test your code:

```
test_orf = orf_list[0]
```

Split the ORF into codons

You need a function that splits the ORF into codons. This one you have already implemented in the exercise about translating DNA -- and if, not here it is in my version to get you started.

```
def split_codons(orf):
    codon_list = []
    for i in range(0, len(orf)-2, 3):
        codon_list.append(orf[i:i+3])
    return codon_list
```

Before you go on, make sure you understand/remember how this function works and what it returns.

Count codons in an ORF

Now you need to count the number of times each codon occurs in the ORF.

Write a function, `count_codons`, that take one argument:

1. A string, which represents the open reading frame.

The function must return:

- A dictionary where keys are strings representing codons and associated values are integers representing the number of times each codon occurs in the ORF given as argument.

Example usage:

```
count_codons("ATGTCATCATGA")
```

should return:

```
{'CTT': 0, 'ATG': 1, 'ACA': 0, 'ACG': 0, 'ATC': 0, 'AAC': 0,
 'ATA': 0, 'AGG': 0, 'CCT': 0, 'ACT': 0, 'AGC': 0, 'AAG': 0,
 'AGA': 0, 'CAT': 0, 'AAT': 0, 'ATT': 0, 'CTG': 0, 'CTA': 0,
 'CTC': 0, 'CAC': 0, 'AAA': 0, 'CCG': 0, 'AGT': 0, 'CCA': 0,
 'CAA': 0, 'CCC': 0, 'TAT': 0, 'GGT': 0, 'TGT': 0, 'CGA': 0,
 'CAG': 0, 'TCT': 0, 'GAT': 0, 'CGG': 0, 'TTT': 0, 'TGC': 0,
 'GGG': 0, 'TAG': 0, 'GGA': 0, 'TGG': 0, 'GGC': 0, 'TAC': 0,
 'TTC': 0, 'TCG': 0, 'TTA': 0, 'TTG': 0, 'TCC': 0, 'ACC': 0,
 'TAA': 0, 'GCA': 0, 'GTA': 0, 'GCC': 0, 'GTC': 0, 'GCG': 0,
 'GTG': 0, 'GAG': 0, 'GTT': 0, 'GCT': 0, 'TGA': 1, 'GAC': 0,
 'CGT': 0, 'GAA': 0, 'TCA': 2, 'CGC': 0}
```

-- though not necessarily with key-value pairs in that order.

In the function, you should use the `split_codons` function to split the ORF into a list of codons. Then create an empty dictionary that you can populate with counts. You want *all* the possible codons to be in your dictionary. That way, the codons you do *not* find in your ORF will have a count of `0`. In this case, such absence is also valuable information. To achieve this you must start by filling the dictionary with a key for each codon and give each a count of `0`. You can do that by iterating over the keys in the dictionary that maps codons to amino acids. Then you must iterate over all the codons in the list of codons produced by the `split_codons` function and add counts to the dictionary as you go.

Group codon counts by amino acid

Having counted how many times each codon appears in the ORF, you need to group the counted codons by the amino acid they encode.

Write a function, `group_counts_by_amino_acid`, which takes one argument:

1. A dictionary, as that returned by `count_codons`.

The function must return:

- A dictionary of dictionaries, which pairs each amino acid with a dictionary with counts of how many times each codon is used to encode that amino acid in the ORF.

Example usage: Assuming `counts` is the dictionary returned by `count_codons` as in the previous example.

```
grouped_counts = group_counts_by_amino_acid(counts)
```

then `group_counts_by_amino_acid` should return:

```
{'A': {'GCA': 0, 'GCC': 0, 'GCT': 0, 'GCG': 0},
 'C': {'TGC': 0, 'TGT': 0},
 'E': {'GAG': 0, 'GAA': 0},
 'D': {'GAT': 0, 'GAC': 0},
 'G': {'GGT': 0, 'GGG': 0, 'GGA': 0, 'GGC': 0},
 'F': {'TTC': 0, 'TTT': 0},
 'I': {'ATT': 0, 'ATC': 0, 'ATA': 0},
 'H': {'CAC': 0, 'CAT': 0},
 'K': {'AAG': 0, 'AAA': 0},
 '*': {'TAG': 0, 'TGA': 1, 'TAA': 0},
 'M': {'ATG': 1},
 'L': {'CTT': 0, 'CTG': 0, 'CTA': 0, 'CTC': 0, 'TTA': 0, 'TTG': 0},
 'N': {'AAT': 0, 'AAC': 0},
 'Q': {'CAA': 0, 'CAG': 0},
 'P': {'CCT': 0, 'CCG': 0, 'CCA': 0, 'CCC': 0},
 'S': {'TCT': 0, 'AGC': 0, 'TCG': 0, 'AGT': 0, 'TCC': 0, 'TCA': 2},
 'R': {'AGG': 0, 'CGC': 0, 'CGG': 0, 'CGA': 0, 'AGA': 0, 'CGT': 0},
 'T': {'ACC': 0, 'ACA': 0, 'ACG': 0, 'ACT': 0},
 'W': {'TGG': 0},
 'V': {'GTA': 0, 'GTC': 0, 'GTT': 0, 'GTG': 0},
 'Y': {'TAT': 0, 'TAC': 0}}
```

-- though not necessarily with key-value pairs in that order.

So if we pretend the resulting dictionary of dictionaries is called `d`, then `d['S']` will be a dictionary where keys are codons encoding the 'S' amino acid and the values are the counts of those codons. That means that you can access a particular count like this:

```
d['S']['TCA']
```

In the above example, this count is `2`. So the task is really just to distribute counts given as the first argument into groups for each amino acid. Depending on what you call your variables it could look something like this:

```
grouped_counts[acid][codon] = codon_counts[codon]
```

Your function should begin by defining an empty dictionary to add to. Use `count_codons` to get a dictionary of all codon counts. Then use a for-loop to run through all codon/amino-acid pairs and populate your dictionary of dictionaries.

Turn counts into frequencies

Now you know how many times each codon represents a certain amino acid, but we would like to know with which *frequency* a certain codon represents an amino acid. So you need to normalize the counts so they become frequencies. You do that by dividing each codon count by the total number of codons encoding the same amino acid. We split the solution to this problem in two. We first write a helper function that turns codon counts for *one* amino acid into frequencies.

Write a function, `normalize_counts`, which takes one argument:

1. A dictionary, where keys are strings representing codons and values are integers representing the counts of these codons.

The function must return:

- A dictionary, where keys are strings representing codons and values are floats representing the frequency at which each codon appear. That is, the count of that codon divided by the total of all codon counts in the dictionary. The frequencies for codons that encode the same amino acid must of sum to one. That means that in cases where the total count is zero, the function must return `None`.

Example usage:

```
normalize_counts({'ATT': 8, 'ATC': 10, 'ATA': 2})
```

should return:

```
{'ATC': 0.5, 'ATA': 0.1, 'ATT': 0.4}
```

-- though not necessarily with key-value pairs in that order.

Now you have solved part of the task, what remains is to now use that function to normalise the codon counts of *each* amino acid in your grouped counts:

Write a function, `normalize_grouped_counts`, which takes one argument:

1. A dictionary of dictionaries, as that returned by `group_counts_by_amino_acid`.

The function must return:

- A new dictionary of dictionaries where the values of the inner dictionaries are frequencies instead of counts as in the example in the introduction. You should not include amino acids for which there are no counted codons.

Example usage: Assuming `gr_counts` is the dictionary of dictionaries returned by `grouped_group_counts_by_amino_acid` in the example above.

```
normalize_grouped_counts(gr_counts)
```

should return:

```
{'*': {'TAA': 0.0, 'TGA': 1.0, 'TAG': 0.0},
'M': {'ATG': 1.0},
'S': {'AGC': 0.0, 'TCG': 0.0, 'TCC': 0.0, 'TCT': 0.0, 'AGT': 0.0, 'TCA': 1.0}}
```

-- though not necessarily with key-value pairs in that order.

Here is some help to get you started:

```
def normalize_grouped_counts(grouped_counts):
    grouped_freqs = {}
    for aa in grouped_counts:
        counts = grouped_counts[aa]
```

Amino acids with no codon counts should *not* be part of the data structure. Remember that in this case `normalize_counts` returns `None`, so you can simply test if the return value from `normalize_counts` is `None`

Compute the codon usage

Now all that remains is to tie together the functions you have written in a final function that generates your big data structure from an ORF:

Write a function, `codon_usage`, which takes one argument:

1. A string, which is an ORF.

The function must return:

- A dictionary of dictionaries, same as that returned by `normalize_grouped_counts`.

Project: Pairwise global alignment

This chapter is about global pairwise alignment and you will implement your own Needleman-Wunch algorithm.

Your task is to find an optimal alignment of two sequences. If two such sequences are both roughly 140 bases long, there are as many different ways to align them as there are atoms in the visible universe, literally. Finding an optimal alignment among those 10^{80} possibilities is obviously a hard problem, but implementing the Needleman-Wunch algorithm will let you do it.

This project is meant to train your coding abilities and consolidate your understanding of the Needleman-Wunch algorithm. The better you understand the algorithm before you begin, the easier and more rewarding the project will be. So re-read the book chapter about pairwise alignment and browse through the lecture slides.

Under "Assignments" on the Blackboard course page, you will find an assignment with the same name as this chapter. There you can download the data file you need for this project. Put these files in a folder dedicated to this project:

- `alignmentproject.py` is the file where you must write your code. It already contains a function I wrote for you.
- `test_alignmentproject.py` is the test program that lets you test the code you write in `alignmentproject.py`.

The project is split into two parts:

1. Filling in a dynamic programming matrix.
2. Reconstructing the optimal alignment by doing traceback through the dynamic programming matrix.

To help you along, the `alignmentproject.py` file already contains a function I wrote for you so you can print your dynamic programming (DP) matrix as you gradually fill it in. You are not expected to understand how this function works. I tried to make it as condensed as possible so it does not take up so much space in your file.

The function is named `print_dp_matrix` and takes two arguments:

1. A string, which represents a DNA sequence of some length `m`.
2. A string, which represents a DNA sequence of some length `n`.

3. A lists of lists of integers, representing a dynamic programming matrix.

The function returns:

- None, but it *prints* a nice representation of the matrix lined with the bases of the two sequences.

Filling in the dynamic programming matrix

Make a matrix

We start out by making a list of lists (a matrix) that has the right shape but only holds `None` values. We use the `None` values as place-holders, which you can later replace with scores. You can think of it as an empty matrix that you can fill scores into, just as we did at the lectures. If you want to align two sequences like `AT` and `GAT` you want a matrix that has 3 rows and 4 columns. Note that the matrix must have one more row than the number of bases in sequence one, and one more column than the number of bases in sequence two.

Write a function, `empty_matrix`, that takes two arguments

1. An integer (which represents the length of sequence one + 1).
2. An integer (which represents the length of sequence two + 1).

The function must return:

- A list of lists. The number of sub-lists be must equal to the first integer argument. Each sublist must contain a number of `None` values equal to the second integer argument.

Example usage:

```
empty_matrix(3, 4)
```

returns a list with *three* lists each of length *four*:

```
[[None, None, None, None], [None, None, None, None], [None, None, None, None]]
```

Even though this is a list of lists we can *think of it* as a three by four matrix:

```
[[None, None, None, None],
 [None, None, None, None],
 [None, None, None, None]]
```

If you want to print the matrix in a way that looks like the slides I showed you at the lecture, you can use the `print_dp_matrix` function (again `None` represents empty cells):

```
          G    A    T
  None None None None
A None None None None
```

```
T None None None None
```

Important: You can implement `empty_matrix` in a way superficially looks ok but will cause you all kinds of grief when you start filling it in. When you create the list of lists (e.g. three lists as above) you need to generate and add three *separate* lists. If you add the *same* list three times you do not have three separate rows in your matrix. Instead you have *three* references to the *same* row. You can test if you did it right this way -- by changing the value of one cell to see what happens:

```
empty = empty_matrix(3, 4)
empty[0][0] = 'Mogens'
print(empty)
```

If this only changed *one* value like below, you are ok:

```
[['Mogens', None, None, None], [None, None, None, None], [None, None, None, None]]
```

If it changed the first value in *all* the lists, it means that all your lists are the same (which is not what you want).

```
[['Mogens', None, None, None], ['Mogens', None, None, None], ['Mogens', None, None, None]]
```

Fill top row and left column

Now that you can make a matrix with the correct dimensions, you need to write a function that fills in the top row and the left column in accordance with what the gap score is. E.g. if the gap score is `-2` you want the matrix to look something like this when you print it with `print_dp_matrix`:

```
      G    A    T
   0  -2   -4   -6
A  -2 None None None
T  -4 None None None
```

Write a function, `prepare_matrix`, which takes three arguments:

1. An integer (which represents the length of sequence one plus one)
2. An integer (which represents the length of sequence two plus one)
3. An integer, which represents the gap_score used for alignment.

The function must return:

- A list of lists. The number of sub-lists be must equal to the first integer argument. The values in the first sub-list must be multiples of the gap score given as the third argument. The first elements of remaining sub-lists must be multiples of the gap score. All remaining elements of sub-lists must be `None`.

Example usage:

```
prepare_matrix(3, 4, -2)
```

must return:

```
[[0, -2, -4, -6], [-2, None, None, None], [-4, None, None, None]]
```

Hint: You should call `empty_matrix` inside `prepare_matrix` to get a matrix filled with `None`.

Now all you need to do is replace the right `None` values with *multiples* of the gap score. E.g. the third element in the first sub-list is `matrix[2][0]`, which you would need to assign the value: `2` times the gap score. In the same way `matrix[3][0]` should be `3` times the gap score. So you need to figure out which elements you should replace and which pairs of indexes you need to access those elements. Then use `range` to generate those indexes and for-loops to loop over them.

Fill the entire matrix

Now that we are able to fill the top row and left column we can start thinking about how to fill the whole matrix.

For that we need a score matrix of match scores. In Python that is most easily represented as a dictionary of dictionaries like this:

```
match_scores = {'A': {'A': 2, 'T': 0, 'G': 0, 'C': 0},
                'T': {'A': 0, 'T': 2, 'G': 0, 'C': 0},
                'G': {'A': 0, 'T': 0, 'G': 2, 'C': 0},
                'C': {'A': 0, 'T': 0, 'G': 0, 'C': 2}}
```

That lets you get the score for matching an `A` with a `T` like this: `match_scores['A']['T']`. Note that the match scores are only for uppercase letters (`A`, `T`, `G`, `C`).

Write a function, `fill_matrix`, which takes four arguments:

1. A string, which represents the first sequence.
2. A string, which represents the second sequence.
3. A dictionary of dictionaries like the one shown above, which represents match scores.
4. An integer, which represents the gap score.

The function must return:

- A list of lists of integers, which represents a correctly filled dynamic programming matrix given the two sequences, the match scores, and the gap score.

Example usage: If `match_scores` is defined as above then

```
fill_matrix('AT', 'GAT', match_scores, -2)
```

must return:

```
[[0, -2, -4, -6], [-2, 0, 0, -2], [-4, -2, 0, 2]]
```

If you print that matrix using `print_dp_matrix` it should look like this:

```
      G  A  T
   0 -2 -4 -6
A -2  0  0 -2
T -4 -2  0  2
```

Hint: You should call `prepare_matrix` inside `fill_matrix` to get a matrix with the top row and left column filled. Assuming `seq1` is sequence one and `seq2` is sequence two then you can do it like this:

```
matrix = prepare_matrix(len(seq1)+1, len(seq2)+1, gap_score)
```

Now you only need to fill out the rest. To produce the indexes of the elements in the list of lists that you need to assign values to, you need two nested for-loops.

```
for i in range(1, len(seq1)+1):
    for j in range(1, len(seq2)+1):
        print(i, j) # just to see what i and j are
```

Examine this code and make sure you understand why we give those arguments to `range`. Each combination of `i` and `j` let you access an element `matrix[i][j]` in `matrix` (list of lists) that you can to assign a value to. The value to assign to `matrix[i][j]` (green cell on the slides) is the maximum of three values (the yellow cells on the slide):

1. The value of the cell to the left (`matrix[i][j-1]`) plus the gap score.
2. The cell above (`matrix[i-1][j]`) plus the gap score.
3. The diagonal cell (`matrix[i-1][j-1]`) plus the match score for base number `i` (index `i-1`) of sequence one and base number `j` (index `j-1`) of sequence two.

Reconstructing the optimal alignment

This is the most difficult part, so I will hold your hand here. Below is first a function that identifies which of three cells (the yellow cells on the slides) some cell (green cell on the slides) is derived from. On the slides, this is the cell pointed to by the red arrow.

```
def get_traceback_arrow(matrix, row, col, match_score, gap_score):

    # yellow cells:
    score_diagonal = matrix[row-1][col-1]
    score_left = matrix[row][col-1]
    score_up = matrix[row-1][col]
```

```
    # gree cell:
    score_current = matrix[row][col]

    if score_current == score_diagonal + match_score:
        return 'diagonal'
    elif score_current == score_left + gap_score:
        return 'left'
    elif score_current == score_up + gap_score:
        return 'up'
```

Write (no copy paste) this into your file and make sure that it works and that you understand exactly how it works before you go on.

Here is a function that uses `get_traceback_arrow` to do the traceback. It reconstructs the alignment starting from the last column adding columns in front as the traceback proceeds. It is big, so it breaks across three pages.

```
def trace_back(seq1, seq2, matrix, score_matrix, gap_score):

    # Strings to store the growing alignment strings:
    aligned1 = ''
    aligned2 = ''
# continues...
```

```
    # The row and col index of the bottom right cell:
    row = len(seq1)
    col = len(seq2)

    # Keep stepping backwards through the matrix untill
    # we get to the top row or the left col:
    while row > 0 and col > 0:

        # The two bases we available to match:
        base1 = seq1[row-1]
        base2 = seq2[col-1]

        # The score for mathing those two bases:
        match_score = score_matrix[base1][base2]

        # Find out which cell the score in the current cell was derived from:
        traceback_arrow = get_traceback_arrow(matrix, row, col, match_score, gap_score)

        if traceback_arrow == 'diagonal':
                # last column of the sub alignment is base1 over base2:
```

```
            aligned1 = base1 + aligned1
            aligned2 = base2 + aligned2
            # next cell is the diagonal cell:
            row -= 1
            col -= 1
        elif traceback_arrow == 'up':
                # last column in the sub alignment is base1 over a gap:
            aligned1 = base1 + aligned1
            aligned2 = '-' + aligned2
            # next cell is the cell above:
            row -= 1
        elif traceback_arrow == 'left':
                # last column in the sub alignment is a gap over base2:
            aligned1 = '-' + aligned1
            aligned2 = base2 + aligned2
            # next cell is the cell to the left:
            col -= 1
# continues...
```

```
    # If row is not zero, step along the top row to the top left cell:
    while row > 0:
        base1 = seq1[row-1]
        aligned1 = base1 + aligned1
        aligned2 = '-' + aligned2
        row -= 1

    # If col is not zero, step upwards in the left col to the top left cell:
    while col > 0:
        base2 = seq2[col-1]
        aligned1 = '-' + aligned1
        aligned2 = base2 + aligned2
        col -= 1

    return [aligned1, aligned2]
```

Once you have *written* it into your file, make sure you understand the correspondence to the sequences of events on the lecture slides.

Now you can write a function you can call to do perform the alignment. You get to do that yourself. It just calls `fill_matrix` and then `trace_back` to get the optimal alignment

Write a function, `align`, that takes four arguments:

1. A string, which represents sequence one.
2. A string, which represents sequence two.

3. A dictionary of dictionaries, which represents the match scores (as described above).
4. An integer, which represents the gap score.

The function must return:

- A list of length two. The first element of that list must be a string, representing the aligned sequence one. The second element must be a string, representing the aligned sequence two.

Example usage:

```
align('ATAT', 'GATGAT', score_matrix, -2)
```

must return:

```
['-AT-AT', 'GATGAT']
```

Once you have written that function you can print your alignment like this:

```
alignment = align('ATAT', 'GATGAT', score_matrix, -2)
for s in alignment:
    print(s)
```

Project: Clustering sequences based on distance

This chapter is about clustering sequence based on the evolutionary distance between them.

Measuring sequence distance

Clustering is based on the distances between all pairs of sequences. So before you can build your tree you must compute those distances and fill them into a table like that in the book. Here we break that task into three parts:

1. Compare two sequences
2. Make the Jukes-Cantor correction
3. Generate a (lower triangular) distance matrix

Compare two sequences

The first function you must write is one that finds the proportion of different bases between two sequences:

Write a function, `sequence_difference`, which takes two arguments:

1. A string, which represents a DNA sequence.
2. A string, which represents a DNA sequence of the same length as argument one.

The function must return:

- A float, which represents the proportion of different bases between the two sequences.

Example usage:

```
sequence_difference('AAATTAAA', 'AAAAAAAA')
```

should return

```
0.25
```

Make the Jukes-Cantor correction

To take into account that some substitutions may fall on top of others you must do the Jukes-Cantor correction you read about in the book. The formula is like this:

$$K = -\frac{3}{4}\ln(1 - \frac{4}{3} * D)$$

Where D is the proportion of differences as returned by `seqeunce_diff` and K is the Jukes-Cantor corrected distance. In the top of `seqdistproject.py` it already says:

```
from math import log
```

That line makes the `log` (logarithm) builtin function from the `math` python library available to your programme. Try to find its Python documentation to see how you use it.

Write a function, `jukes_cantor`, which takes one argument:

1. A float, which represents a proportion of different bases between two sequences.

The function must return:

- A float, which represents the Jukes-Cantor corrected distance corresponding the proportion of differences given as argument.

Example usage:

```
jukes_cantor(0.1)
```

should return

```
0.10732563273050497
```

Lower triangular distance matrices

This project is all about distances between pairs (of sequences), and what would be more natural than to put all the distances in a matrix so you can look up the distance between the sequences with indexes `i` and `j` as the matrix element in row `i` and column `j`. You already know how matrices can be represented by lists of lists. E.g. a matrix like this:

can be expressed as a list of lists like this:

```
[[0.0, 0.1, 0.3, 0.3, 0.1, 0.2],
 [0.1, 0.0, 0.2, 0.4, 0.1, 0.1],
 [0.3, 0.2, 0.0, 0.4, 0.2, 0.3],
 [0.3, 0.4, 0.4, 0.0, 0.2, 0.1],
 [0.1, 0.1, 0.2, 0.2, 0.0, 0.1],
 [0.2, 0.1, 0.3, 0.1, 0.1, 0.0]]
```

Notice how the diagonal is all zeros because these distances represent the distance of a sequence to itself. Also, notice that the part above the diagonal is a mirror the part below the diagonal (in bold). This is all a bit redundant, especially in this project where you will have to reduce the matrix as you group (or cluster) sequences together. We want something nice and lean where we only have the numbers we need -- and that is the *lower triangular matrix*:

```
0.1
0.3 0.2
0.3 0.4 0.4
0.1 0.1 0.2 0.2
0.2 0.1 0.3 0.1 0.1
```

In Python this is still just a list of lists, only, each sublist now has the same length as its index in the big list (E.g. `[0.3, 0.2]` has index `2` in the list and has length `2`):

```
matrix = [[],
              [0.1],
              [0.3, 0.2],
              [0.3, 0.4, 0.4],
              [0.1, 0.1, 0.2, 0.2],
              [0.2, 0.1, 0.3, 0.1, 0.1]]
```

Here I am just writing it nicely. If you where to print that list of lists it would look like this:

```
[[], [0.1], [0.3, 0.2], [0.3, 0.4, 0.4], [0.1, 0.1, 0.2, 0.2], [0.2, 0.1, 0.3, 0.1, 0.1]]
```

Say your sequences had names: A, B, C, D, E, and F, then the above data structure represents distances between each pair like this:

```
A
B 0.1
C 0.3 0.2
D 0.3 0.4 0.4
E 0.1 0.1 0.2 0.2
F 0.2 0.1 0.3 0.1 0.1
   A   B   C   D   E   F
```

There is only one drawback with this reduced representation of the full square matrix: In the full matrix you can get the distance between the sequences with indexes `i` and `j` as *both* `matrix[i][j]` and `matrix[j][i]` because the part above and below the diagonal are the same. Using the lower triangular matrix, you must always use the *largest* index first. Using the smaller one first will give you an `IndexError`. So if you want the distance between sequences with index `2` and `4`, you must use the bigger index first (as the row index): `matrix[4][2]`.

Generate a distance matrix

Write a function, `lower_trian_matrix`, which takes one argument:

1. A list of strings. All strings have equal length and represent DNA sequences.

The function must return:

- A list of lists of floats, which represent the lower triangular matrix of Jukes-Cantor distances between DNA sequences given as argument.

Example usage:

```
sequences = ['TAAAAAAAAAA',
             'TTAAAAAAAAA',
             'AAAAAAAAAGG',
             'AAAAAAAAGGGG']
lower_trian_matrix(sequences)
```

here `lower_trian_matrix` should return:

```
[[],
 [0.08833727674228764],
 [0.30409883108112323, 0.4408399986765892],
 [0.6081976621622466, 0.8239592165010822, 0.1884858212106795 3]]
```

You should use `sequence_difference` to compute the proportion of differences between each pair of sequences and `jukes_cantor` to produce the corrected distance to fill into the matrix.

Start by figuring out what pairs of indexes you need and then figure out how you can make two nested for-loops generate them. Remember that the length of each sublist is equal to its index in the big list.

Clustering

Now that you have the distance matrix you are ready for the actual clustering. There are three steps to that:

1. Find the pair you want to join
2. Compute the distances between the joined pair and all other elements (linkage)
3. Keep going until you only have one left

Depending on how you choose which pair to join and how you compute the new distances for the joined pair determines what kind of clustering you do. Here we will try a centroid-like linkage called WPGMA. It does not work as well as UPGMA but is a bit easier to implement (you can look up WPGMA on wikipedia).

Find the smallest cell

So you need to be able to find the pair with the smallest distance. To do that we identify the cell in the matrix with the smallest value:

Write a function, `find_lowest_cell`, which takes one argument:

1. A list of lists, which represents a lower triangular distance matrix as returned by `lower_trian_matrix`.

The function must return:

- A list of two integers, which represent the row and column index of the cell with the smallest value in the matrix.

Remember that the row index is always smaller than the column index. The two indexes tell you which two elements to join next.

Example usage: Assuming that `matrix` is the lower triangular matrix returned by `lower_trian_matrix` in the previous example, then

```
find_lowest_cell(matrix)
```

Should return

```
[1, 0]
```

Decide on a linkage method

You also need a function that computes a new distance from two original ones using the the centroid-like linkage we have decided to use.

Write a function, `link`, that takes two arguments:

1. A float, which represents a matrix element.
2. A float, which represents another matrix element.

The function must return:

- A float, which is the average of the two arguments

Example usage:

```
link(0.4, 0.2)
```

Should return:

```
0.3
```

Perform the clustering

The three functions that do the actual clustering are complicated but you should be able to follow what they do. The first one updates the table to reflect that you join a pair. The second updates the list of sequence names (labels) to reflect that you joined a pair. The last one uses the two other functions to cluster pair until there is only one cluster left.

Your task is to faithfully copy the code for each function and to read and understand how they work.

Updating the matrix

The function `update_table` takes three arguments:

1. A list of lists, which represents a lower triangular distance matrix.
2. An integer representing the index of one of the elements to join.
3. An integer representing the index of the other element to join.

The way this function is implemented, it is assumed that the second argument is always larger than the third argument. I.e. the second argument is a row index and the third argument is a column index.

The function does not return anything, but it updates the matrix to reflect that a pair has been joined. If your matrix looks like this *before* you call the function:

```
m = [[], [0.1], [0.3, 0.4], [0.6, 0.8, 0.2]]
```

Then *after* you call the function like this `update_table(m, 1, 0)`, the matrix will look like this:

```
[[], [0.35], [0.7, 0.2]]
```

Here is the function:

```
def update_table(table, a, b):

    # For the lower index, reconstruct the entire row (ORANGE)
    for i in range(0, b):
        table[b][i] = link(table[b][i], table[a][i])

    # Link cells to update the column above the min cell (BLUE)
    for i in range(b+1, a):
        table[i][b] = link(table[i][b], table[a][i])

    # Link cells to update the column below the min cell (RED)
    for i in range(a+1, len(table)):
        table[i][b] = link(table[i][b], table[i][a])
```

```
    # Delete cells we no longer need (lighter colors)
    for i in range(a+1, len(table)):
        # Remove the (now redundant) first index column entry
        del table[i][a]
    # Remove the (now redundant) first index row
    del table[a]
```

The colors refer to cell colors on the slide you I showed you at the lecture.

Updating labels

The function `update_labels` takes three arguments:

1. A list of strings representing sequence names.
2. An integer representing the index of a sequence name.
3. An integer representing the index of another sequence name.

The function does not return anything, but it updates the list of names to reflect that you joined a pair. If your list looks like this *before* you call the function:

```
labels = ['A', 'B', 'C', 'D']
```

Then *after* you call the function like this `update_labels(labels, 1, 0)`, the list will look like this:

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

Here is the function:

```
def update_labels(labels, i, j):

    # turn the label at first index into a combination of both labels
    labels[j] = "({},{})".format(labels[j], labels[i])

    # Remove the (now redundant) label in the first index
    del labels[i]
```

Do the clustering

Now onto the real task, the actual clustering. The function `cluster` takes two arguments:

1. A list of strings representing DNA sequences of equal length.
2. A list of strings representing sequence names.

The function returns:

- A string representing the generated clustering.

Here is the function:

```
def cluster(sequences, names):

    table = lower_trian_matrix(sequences)
    labels = names[:]

    # Until all labels have been joined...
    while len(labels) > 1:
        # Locate lowest cell in the table
        i, j = find_lowest_cell(table)

        # Join the table on the cell co-ordinates
        update_table(table, i, j)

        # Update the labels accordingly
        update_labels(labels, i, j)

    # Return the final label
    return labels[0]
```

Here is a simple example of how you can use your new clustering code:

```
names = ['A', 'B', 'C', 'D']
sequences = ['TAAAAAAAAAA',
             'TTAAAAAAAAA',
             'AAAAAAAAAGG',
             'AAAAAAAGGGG']

tree = cluster(sequences, names)
print(tree)
```

On your own

From here on you are on your own. If you find a FASTA file with aligned (ungapped) homologous sequences, you can use the function below to read it into your program and try your code out on real-world sequences. I will leave it to you to figure out how it works.

```
def read_fasta(filename):
    f = open(filename, 'r')
    record_list = []
    header = ""
    sequence = ""
    for line in f:
        line = line.strip() ## get rid of whitespace and newline
```

```
    if line.startswith(">"):
        if header != "": ## if it is the first header
            record_list.append([header, sequence])
            sequence = ""
        header = line[1:]
    else:
        sequence += line
record_list.append([header, sequence])
return record_list
```

Project: Genome assembly

This chapter is a programming project where you will assemble a small genomic sequence from a set of short sequencing reads.

In genome assembly, many short sequences (reads) from a sequencing machine are assembled into long sequences -- ultimately chromosomes. This is done by ordering overlapping reads so that they together represent genomic sequences. For example given these three reads: `AGGTCGTAG`, `CGTAGAGCTGGGAG`, `GGGAGGTTGAAA` ordering them based on their overlap like this

```
AGGTCGTAG
    CGTAGAGCTGGGAG
              GGGAGGTTGAAA
```

produces the following genomic sequence:

```
AGGTCGTAGAGCTGGGAGGTTGAAA
```

Real genome assembly is, of course, more sophisticated than what we do here, but the idea is the same. To limit the complexity of the problem we make two simplifying assumptions:

1. There are no sequencing errors, implying that true overlaps between reads can be identified as perfectly matching overlaps.
2. No reads are nested in other reads. I.e. They are never a smaller part of another read.

The second assumption implies that overlaps are always of this type:

```
    CGTAGAGCTGGGAG
              GGGAGGTTGAAA
```

and never of this type:

```
    CGTAGAGCTGGGAG
       AGAGCTG
```

In this project, you will be asked to write functions that together solve the problem of assembling a genomic sequence. Each function each solve a small problem, and you may

need to call these functions inside other functions, to put together solutions to larger subproblems.

Under "Assignments" on the Blackboard course page, you will find an assignment with the same name as this chapter. There you can download the data file you need for this project:

- `sequencing_reads.txt` contains the sequencing reads.

You also need to download the two project files:

- `assemblyproject.py` is an empty file where you must write your code.
- `test_assemblyproject.py` is the test program that lets you test the code you write in `assemblyproject.py`.

Put all three files in a folder dedicated to this project. On most computers you can right-click on the link and choose "Save file as..." or "Download linked file".

Now open each file in *SublimeText* and have a look at what is in `sequencing_reads.txt`. (Do *not* change it in any way and do not save it after viewing. If SublimeText asks you if you want to save it before closing, say *no*.) How many sequences are there in each file?

The project is split into four parts:

1. Read and analyze the sequencing reads.
2. Compute the overlaps between reads.
3. Find the right order of reads.
4. Reconstruct the genomic sequence.

Here is an overview of the functions you will write in each part of the project and of which functions that are used by other functions.

Make sure you read the entire exercise and understand what you are supposed to do before you begin!

Read and analyze the sequencing reads

The first task is to read and parse the input data. The sequence reads for the mini-assembly are in the file `sequencing_reads.txt`. The first two lines of the file look like this:

```
Read1 GGCTCCCCACGGGGTACCCATAACTTGACAGTAGATCTCGTCCAGACCCCTAGC
Read2 CTTTACCCGGAAGAGCGGGACGCTGCCCTGCGCGATTCCAGGCTCCCCACGGG
```

Each line represents a read. The first field on each line is the name of the read and the second field is the read sequence itself. So for the first line `Read1` is the name and `ATGCG...` is the sequence.

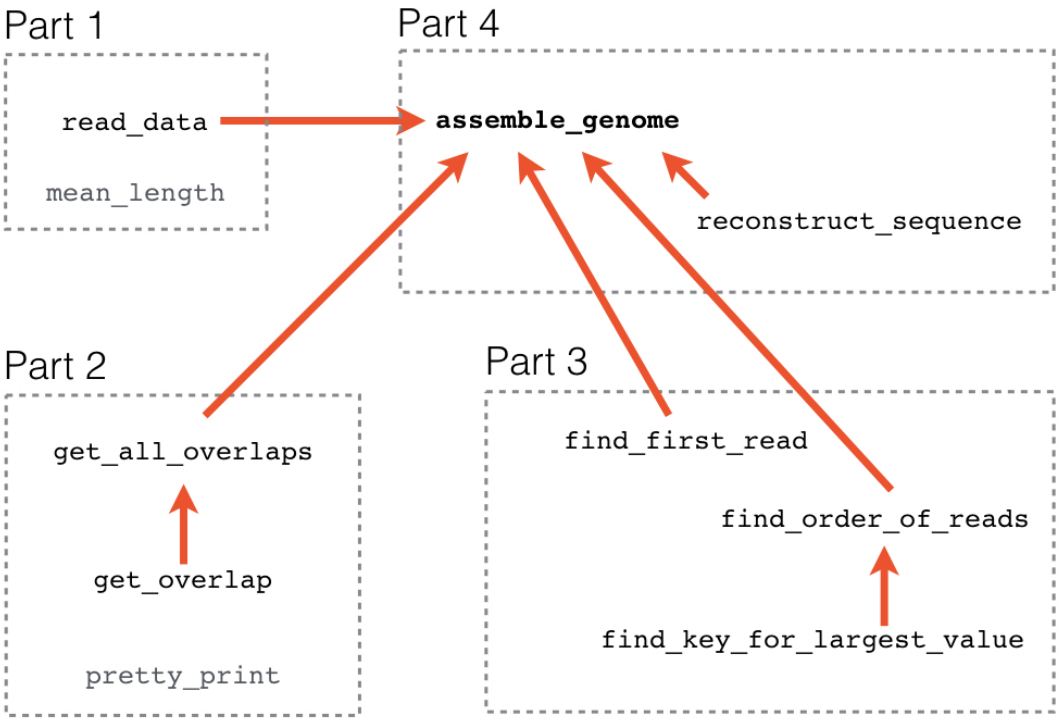


Figure 0.7: Overview

Read the sequencing reads into your program

Write a function, `read_data`, that takes one argument:

1. A string, which is the name of the data file.

The function must return

- A dictionary, where the keys are the *names* of reads and the values are the associated read *sequences*. Both keys and values must be strings.

Example usage:

```
read_data('sequencing_reads.txt')
```

should return a dictionary with the following content (maybe not with key-value pairs in that order)

```
{'Read1': 'GGCTCCCCACGGGGTACCCATAACTTGACAGTAGATCTCGTCCAGACCCCTAGC',
 'Read3': 'GTCTTCAGTAGAAAATTGTTTTTTTCTTCCAAGAGGTCGGAGTCGTGAACACATCAGT',
 'Read2': 'CTTTACCCGGAAGAGCGGGACGCTGCCCTGCGCGATTCCAGGCTCCCCACGGG',
 'Read5': 'CGATTCCAGGCTCCCCACGGGGTACCCATAACTTGACAGTAGATCTC',
 'Read4': 'TGCGAGGGAAGTGAAGTATTTGACCCTTTACCCGGAAGAGCG',
 'Read6': 'TGACAGTAGATCTCGTCCAGACCCCTAGCTGGTACGTCTTCAGTAGAAAATTGTTTTTTTCTTCCAAGAGGTCGGAGT
```

Here is some scaffold code to get you started:

```
def read_data(file_name):
    input_file = open(file_name)
    # ...
    for line in input_file:
        # ...
    # ...

    input_file.close()
```

Compute the mean length of reads

Having written that function we would like to get some idea about how long the reads are. More often than not there are too many reads to look at them manually, so we need to make a function that computes the mean length of the reads.

Write a function, `mean_length`, that takes one argument:

1. A dictionary, in which keys are read names and values are read sequences (this is a dictionary like that returned by `read_data`).

The function must return

- A float, which is the average length of the sequence reads.

Compute overlaps between reads

Next thing is to figure out which reads overlap each other. To do that we need a function that takes two read sequences and computes their overlap. Remember that in the input data none of the reads are completely nested in another read.

Compute the overlap between two reads

We know that there are no sequencing errors, so in the overlap, the sequence match will be perfect. To compute the overlap between the 3′ (right) end of the left read with the 5′ (left) end of the right read, you need to loop over all possible overlaps honoring that one sequence is the left one and the other is the right one. In the for loop, start with the largest possible overlap (`min(len(left), len(right))`) and evaluate smaller and smaller overlaps until you find an exact match.

Write a function, `get_overlap`, that takes two arguments

1. A string, which is the left read sequence.
2. A string, which is the right read sequence.

The function must return

- A string, which is the overlapping sequence. If there is no overlap it should return an empty string.

Example usage:

```
s1 = "CGATTCCAGGCTCCCCACGGGGTACCCATAACTTGACAGTAGATCTC"
s2 = "GGCTCCCCACGGGGTACCCATAACTTGACAGTAGATCTCGTCCAGACCCCTAGC"
get_overlap(s1, s2)
```

should return the string

```
'GGCTCCCCACGGGGTACCCATAACTTGACAGTAGATCTC'
```

and `get_overlap(s2, s1)`

should return the string

```
'C'
```

From these two examples it seems that `s1` and `s2` overlap and that `s1` is the left one and `s2` is the right one. Treating `s2` as the left one and `s1` as the right one only gives an overlap of one base (we expect a few bases of overlap even for unrelated sequences).

Compute all read overlaps

When you have written `get_overlap` you can use it to evaluate the overlap between all pairs of reads in both left-right and right-left orientations.

Write a function, `get_all_overlaps`, that takes one argument:

1. A dictionary with read data as returned by `read_data`.

The function must return

- A dictionary of dictionaries, specifying the number of overlapping bases for a pair of reads in a specific left-right orientation. Computing the overlap of a read to itself is meaningless and must not be included. Assuming the resulting dictionary of dictionaries is called `d`, then `d['Read2']` will be a dictionary where keys are the names of reads that have an overlap with read `'Read2'` when `'Read2'` is put in the left position, and the values for these keys are the number of overlapping bases for those reads.

Example usage: assuming that `reads` is a dictionary returned by `read_data` then:

```
get_all_overlaps(reads)
```

should return the following dictionary of dictionaries (but not necessarily with the same ordering of the key-value pairs):

```
{'Read1': {'Read3': 0, 'Read2': 1, 'Read5': 1, 'Read4': 0, 'Read6': 29},
 'Read3': {'Read1': 0, 'Read2': 0, 'Read5': 0, 'Read4': 1, 'Read6': 1},
 'Read2': {'Read1': 13, 'Read3': 1, 'Read5': 21, 'Read4': 0, 'Read6': 0},
 'Read5': {'Read1': 39, 'Read3': 0, 'Read2': 1, 'Read4': 0, 'Read6': 14},
 'Read4': {'Read1': 1, 'Read3': 1, 'Read2': 17, 'Read5': 2, 'Read6': 0},
 'Read6': {'Read1': 0, 'Read3': 43, 'Read2': 0, 'Read5': 0, 'Read4': 1}}
```

Print overlaps as a nice table

The dictionary returned by `get_all_overlaps` is a little messy to look at. We want to print it in a nice matrix-like format so we can better see which pairs overlap in what orientations.

This `pretty_print` function, should take one argument:

1. A dictionary of dictionaries as returned by `get_all_overlaps`.

The function should *not* return anything but must *print* a matrix exactly as shown in the example below with nicely aligned and right-justified columns. The first column must hold names of reads in left orientation. The top row holds names of reads in right orientation. Remaining cells must each hold the number of overlapping bases for a left-right read pair. The diagonal corresponds to overlaps to the read itself. You must put dashes in these cells.

Example usage: assuming that `overlaps` is a dictionary of dictionaries returned by `get_all_overlaps` then:

```
pretty_print(overlaps)
```

should print exactly

```
       Read1  Read2  Read3  Read4  Read5  Read6
Read1      -      1      0      0      1     29
Read2     13      -      1      0     21      0
Read3      0      0      -      1      0      1
Read4      1     17      1      -      2      0
Read5     39      1      0      0      -     14
Read6      0      0     43      1      0      -
```

This function is hard to get completely right. So to spare you the frustration, this one is on me:

```
def pretty_print(d):
    print('      ', end='')
    for j in sorted(d):
        print("{: >6}".format(j), end='')
    print()
    for i in sorted(d):
        print("{: >6}".format(i), end='')
        for j in sorted(d):
            if i == j:
                s = '     -'
            else:
                s = "{: >6}".format(d[str(i)][str(j)])
            print(s, end='')
        print()
```

Make sure you understand how it works. You can look up in the documentation what `"{: >6}".format(i)` does.

Find the right order of reads

Now that we know how the reads overlap we can chain them together pair by pair from left to right to get the order in which they represent the genomic sequence. To do this we take the first (left-most) read and identify which read has the largest overlap to its right end. Then we take that read and find the read with the largest overlap to the right end of that -- and so on until we reach the rightmost (last) read.

Find the first read

The first thing you need to do is to identify the first (leftmost) read so we know where to start. This read is identified as the one that has no significant (>2) overlaps to its left end (it only has a good overlap when positioned to the left of other reads). In the example output from `pretty_print` above the first read would be read `'Read4'` because the `'Read4'` column has no significant overlaps (no one larger than two).

We break the problem in two and first write a function that gets all the overlaps to the left end of a read (i.e. when it is in the right position):

Write a function, `get_left_overlaps`, that takes two arguments:

1. A dictionary of dictionaries as returned from `get_all_overlaps`.
2. A string, which represents the name of a read.

The function must return

- A sorted list of integers, which represent the overlaps of other reads to its left end.

Example usage: assuming that `overlaps` is a dictionary of dictionaries returned by `get_all_overlaps` then.

```
get_left_overlaps(overlaps, 'Read1')
```

should return

```
[0, 0, 1, 13, 39]
```

Hint: once you have made a list of left overlaps, you can use the builtin function `sorted` to make a sorted version of the list that you can return from the function.

OK, now that we have a function that can find all the overlaps to the left end of a given read, all we need to do is find the particular read that have no signifiant (>2) overlaps to its left end.

Write a function, `find_first_read`, that takes one argument:

1. A dictionary of dictionaries as returned from `get_all_overlaps`.

The function must return

- A string containing the name of the first read.

Example usage: assuming that `overlaps` is a dictionary of dictionaries returned by `get_all_overlaps` then.

```
find_first_read(overlaps)
```

should return

```
'Read4'
```

Find the order of reads

Now that we have the first read we can find the correct ordering of reads. We want a list of the read names in the right order.

Given the first (left) read, the next read is the one that has the largest overlap to the right end of that read. To figure out which read that is, we use our dictionary of overlaps. If the first read is `'Read4'` then `overlaps['Read4']` is a dictionary of reads with overlap to the right end of read `'Read4'`. So to find the name of the read with the largest overlap you must write a function that finds the key associated with the largest value in a dictionary. We do that first:

Write a function, `find_key_for_largest_value`, that takes one argument:

1. A dictionary.

The function must return the key associated with the largest value in the dictionary argument.

Having written `find_key_for_largest_value` you can use it as a tool in the function the finds the order of reads:

Write a function, `find_order_of_reads`, that takes two arguments:

1. A string, which is the name of the first (left-most) read (that returned by `find_first_read`).
2. A dictionary of dictionaries, of all overlaps (that returned by `get_all_overlaps`).

The function must return

- A list of strings, which are read names in the order in which they represent the genomic sequence.

Example usage: assuming that `overlaps` is a dictionary of dictionaries returned by `get_all_overlaps` then:

```
find_order_of_reads('Read4', overlaps)
```

should return:

```
['Read4', 'Read2', 'Read5', 'Read1', 'Read6', 'Read3']
```

Make sure you understand why this is the right list of read names before you try to implement the function.

Reconstruct the genomic sequence

Now that you have the number of overlapping bases between reads and the correct order of the reads you can reconstruct the genomic sequence.

Reconstruct the genomic sequence from the reads

Write a function, `reconstruct_sequence`, that takes three arguments:

1. A list of strings, which are the names of reads in the order identified by `find_order_of_reads`.
2. A dictionary, with read data as returned from `read_data`.
3. A dictionary of dictionaries with overlaps as returned from `get_all_overlaps`.

The function must return

- A string, which is the genomic sequence.

Example usage: assuming that `order` is the list of strings returned by `find_order_of_reads`, that `reads` is the dictionary returned by `read_data` and that `overlaps` is a dictionary of dictionaries returned by `get_all_overlaps` then:

```
reconstruct_sequence(order, reads, overlaps)
```

should return one long DNA string (had to break it in three to make it fit on the page):

```
TGCGAGGGAAGTGAAGTATTTGACCCTTTACCCGGAAGAGCGGGACGCTGCCCTGCGCGATT
CCAGGCTCCCCACGGGGTACCCATAACTTGACAGTAGATCTCGTCCAGACCCCTAGCTGGTA
CGTCTTCAGTAGAAAATTGTTTTTTTCTTCCAAGAGGTCGGAGTCGTGAACACATCAGT
```

Putting the whole thing together

Now that you have written functions to take care of each step you can write one last function that uses them to do the entire assembly.

Write a function, `assemble_genome`, that takes one argument:

1. A string, which is the name of a file with sequencing reads in the format described in the beginning of this project description.

The function must return

- A string, which is the genome assembled from the sequencing reads

Example usage:

```
assemble_genome('sequencing_reads.txt')
```

should return the assembled genome:

```
TGCGAGGGAAGTGAAGTATTTGACCCTTTACCCGGAAGAGCGGGACGCTGCCCTGCGCGATT
CCAGGCTCCCCACGGGGTACCCATAACTTGACAGTAGATCTCGTCCAGACCCCTAGCTGGTA
CGTCTTCAGTAGAAAATTGTTTTTTTCTTCCAAGAGGTCGGAGTCGTGAACACATCAGT
```

Project: Finding genes in bacteria

This chapter is a programming project where you will find open reading frames in the genome of a particularly virulent strain of *E. coli*.

In this project, you will analyze DNA to identify the open reading frames (ORFs) and the proteins they encode.

Under "Assignments" on the Blackboard course page, you will find an assignment with the same name as this chapter. There you can download the data file you need for this project:

- `e_coli_O157_H157_str_Sakai.fasta` contains the full genome of the *Escherichia coli* O157:H7 Sakai strain (yes the full genome).

You also need to download the two project files:

- `orfproject.py` is an empty file where you must write your code.
- `test_orfproject.py` is the test program that lets you test the code you write in `orfproject.py`.

Put all three files in a folder dedicated to this project. On most computers you can right-click on the link and choose "Save file as..." or "Download linked file".

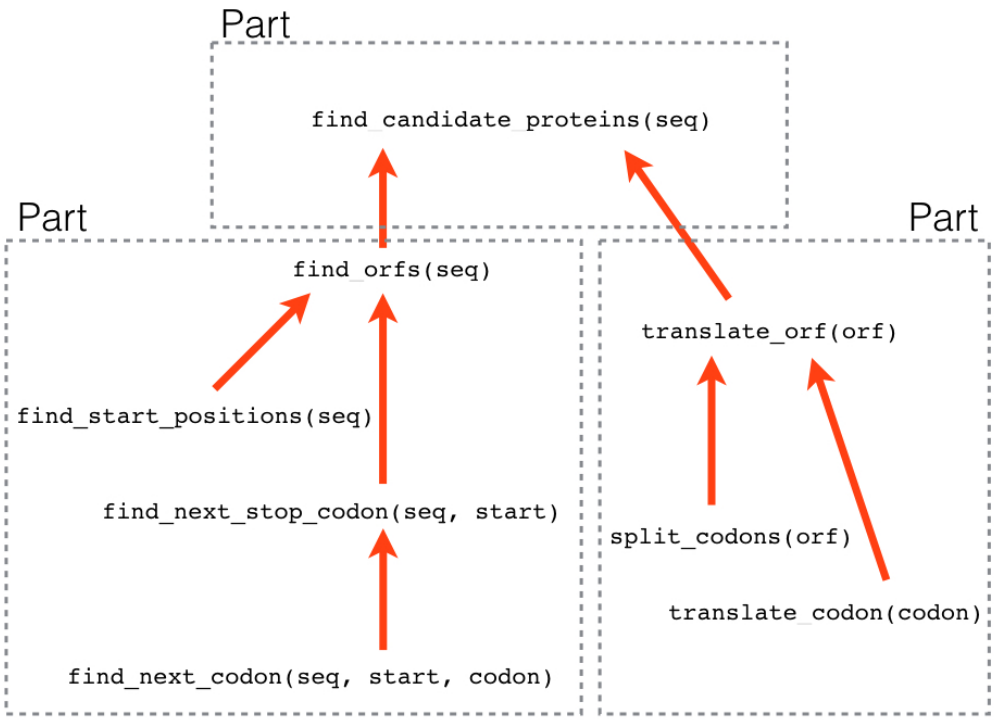
Four your convenience, the file `orfproject.py` already contains three global *constants* (variables that must never be changed by the code). One is a dictionary `codon_map`, which maps codons to letters that represent amino acids. The other two are a string, `start_codon`, and a list, `stop_codons`. You can refer to these three variables in your code, but obviously, never change them.

The project has three parts.

1. In the first part of the project, you will write the code necessary to identify ORFs in a long genomic sequence.
2. In the second part, you will use the code you wrote on the programming project about translating DNA to translate each ORF into a protein sequence.
3. In the third part, you will use the code from parts one and two together to find candidate proteins encoded by ORFs in a genomic sequence.

Start by reading through the exercise before you do anything else. That way you have a good overview of the tasks ahead. Here is a mind map of how we split the larger

problem into smaller bits and how they fit together:



Finding Open Reading Frames

Find the start positions of ORFs in a DNA sequence

The first task is to write a function that finds all the possible positions where an ORF can begin.

Write a function, `find_start_positions`, which takes one argument:

1. A string, which is a DNA sequence.

The function must return:

- A list of integers, which represent the indexes of the first base in start codons in the DNA sequence argument.

Example usage:

```
find_start_positions('TATGCATGATG')
```

should return

```
[1, 5, 8]
```

Your function should contain a for-loop that iterates over all possible positions in a DNA string where a codon can begin. Not surprisingly, these are all the positions except for the last two. So start out with this:

```
def find_start_positions(seq):
    for i in range(len(seq) - 2):
        print(i)
```

Now, instead of just printing `i`, try and make it print the three bases following `i` using the slicing technique:

```
triplet = seq[i:i+3]
```

I.e. if your sequence is `'TATGCATGATG'` it should first print `'TAT'` then `'ATG'` then `'TGC'` and so on.

When you have this working you should change the code so that triplets are only printed if they are start codons. You can use an if-statement that tests if each triplet is equal to `start_codon`.

Then try and make your function print `i` only when `i` is the first base of a start codon.

Finally, modify the function so all the relevant values of `i` are collected in a list using the same technique as in the `split_codons(orf)` function, and then return this list from the function.

Finding the next occurrence of some codon in an ORF

Now that you can find where the ORFs begin in our sequence you must also be able to identify where each of these end. As you know, an ORF ends at any of three different stop codons in the same reading frame as the start codon. So, starting at the start codon of the ORF, we need to be able to find the next occurrence of some specific codon. I.e. you should look at all codons after the start codon and find the first occurrence of some specified codon. If the function does not find that codon in the string it should return `None`.

Write a function, `find_next_codon`, that takes three arguments:

1. A string, which is the DNA sequence.
2. An integer, which is the index in the sequence where the ORF starts.
3. A string, which is the codon to find the next occurrence of.

The function must return:

- An integer, which is the index of the first base in the next in-frame occurrence of the codon. If the function does not find that codon in the string it should return `None`.

Example usage:

```
find_next_codon('AAAAATTTAATTTAA', 1, 'TTT')
```

should return

```
10
```

Your function should contain a for-loop that iterates over all the relevant starts of codons. Remember that no valid codon can start at the last two positions in the sequence. E.g. if the second argument is 7 and the length of the sequence is 20 then the relevant indexes are 7, 10, 13, 16.

Start by writing a function just with a for-loop that lets you print these indexes produced by `range`. Figure out how to make the `range` function iterate over the appropriate numbers.

```
def find_next_codon(seq, start, codon):
    for idx in range( ?? ):
        print(idx)
```

When you have that working, use the slicing technique to instead print the codons that start at each index.

Finally, add an if-statement that tests if each codon is equal to `codon`. When this is true, the function should return the value of `idx`.

Finding the first stop codon in an ORF

Now that you can find the next occurrence of any codon, you are well set up to write a function that finds the index for the beginning of the next in-frame *stop codon* in an ORF.

Write a function, `find_next_stop_codon`, that takes two arguments:

1. A string, which is the DNA sequence.
2. An integer, which is the index in the sequence where the ORF starts.

The function must return:

- An integer, which is the index of the first base in the next in-frame stop codon. If there is no in-frame stop codon in the sequence the function should return `None`.

Example usage:

```
find_next_stop_codon('AAAAATAGATGAAAA', 2)
```

should return

```
5
```

Here is some inspiration:

1. You should define a list to hold the indexes for the in-frame stop codons we find.
2. Then we loop over the three possible stop codons to find the next in-frame occurrence of each one from the start index. You can use `find_next_codon` for this. Remember that it returns `None` if it does not find any. If it does find a position you can add it to your list.
3. At the end, you should test if you have any indexes in your list.
4. If you do, you should return the smallest index in the list. I.e the ones closest to the start codon.
5. If you did not find any stop codons the function must return `None` to indicate this.

Finding ORFs

Now you can write a function that uses `find_start_positions` and `find_next_stop_codon` to extract the start and end indexes of each ORF in a genomic sequence.

Write a function, `find_orfs`, that takes one argument:

1. A string, which is a DNA sequence.

The function should return:

- A list, which contains lists with two integers. The list returned must contain a list for each ORF in the sequence argument. These lists each contain two integers. The first integer represents the start of the ORF, the second represents the end. The function should handle both uppercase and lowercase sequences.

Example usage:

```
find_orfs("AAAATGGGGTAGAATGAAATGA")
```

should return

```
[[3, 9], [13, 19]]
```

Start by using `find_start_positions` to get a list of all the start positions in sequence:

```
def find_orfs(seq):
    start_positions = find_start_positions(seq)
```

When you have that working, add a for-loop that iterates over the start positions. Inside the for-loop, you can then get the next in-frame stop codon for each start position by calling `find_next_stop_codon`. Try to print the start and end indexes you find to make sure the code does what you think.

Finally, you need to add a `[start, stop]` list for each ORF to the big list that the function returns. To append a list to a list you do write something like this:

```
orf_coordinate_list.append([start, stop])
```

Test your function. Chances are that some of the end positions you get are None. This is because some of the start codons were not followed by an in-frame stop codon. Add an if-statement to your function that controls that only start-stop pairs with a valid stop coordinate are added to the list of results.

Translation of open reading frames

We need to translate the reading frames we find into the proteins they may encode. So why not use the code you already wrote in the programming project where you translated open frames? Copy the content of `translationproject.py` into `orfproject.py`. Now you can use the function `translate_orf` to translate your ORFs.

Put everything together

Read in genomic sequences

The file `e_coli_O157_H157_str_Sakai.fasta` contains the genome that we want to analyze to find open reading frames. This is an especially nasty strain of *Escherichia coli* O157:H7 isolated after a massive outbreak of infection in school children in Sakai City, Japan, associated with consumption of white radish sprouts.

You can use the function below to read the genome sequence into a string.

```
def read_genome(file_name):
    f = open(file_name, 'r')
    lines = f.readlines()
    header = lines.pop(0)
    substrings = []
    for line in lines:
        substrings.append(line.strip())
    genome = ''.join(substrings)
    f.close()
    return genome
```

Now for the grand finale: Using `read_genome`, `find_orfs` and `translate_orf` you can write a function that finds all protein sequences produced by open reading frames in the genome.

Write a function, `find_candidate_proteins`, that takes one argument:

1. A string, which is a genome DNA sequence.

The function must return

- A list of strings, which each represent a possible protein sequence.

Note that this is a full genome so finding all possible proteins will take a while (~5 min.). You can start by working on the first 1000 bases:

Example usage:

```
genome = read_genome('e_coli_O157_H157_str_Sakai.fasta')
first_1000_bases = genome[:1000]
find_candidate_proteins(first_1000_bases)
```

should return

```
['MSLCGLKKESLTAASELVTCRE*', 'MKRISTTITTTITTTITITITTGNGAG*',
 'MQNVFCGLPIFWKAMPGRGRWPPSSLPPPKSPTTWWR*', 'MPGRGRWPPSSLPPPKSPTTWWR*',
 'MLYPISAMPNVFLPNF*', 'MPNVFLPNF*', 'MSCMALVC*', 'MALVC*']
```

The function should call `find_orfs` to get the list of start-end pairs. For each index pair, you must then slice the ORF out of the sequence (remember that the end index represents the first base in the stop codon), translate the ORF to protein, and add it to a list of proteins that the function can return.

Hint: To check your result note that all returned sequences should start with a start codon `'M'`, end with a stop codon `'*'` and contain no stop codons in the middle.

On your own

This is where this project ends, but *you* can continue if you like. Given a long list of candidate proteins of all sizes, what would you do to narrow down your prediction to a smaller set of very likely genes? If you have some ideas, then try them out.

- Maybe you can rank them by length? What is the expected minimum length of proteins?
- Maybe you can look for a Shine-Delgarno motif upstream of the start codon? You know how to do that from the lectures.
- You can also try to BLAST them against the proteins in Genbank. The true ones should have some homologs in other species.

General exercises

Exercise 225

Remind yourself of the different types of Python values you know. E.g. one of them is integer (`int`). Make a list.

Exercise 226

You already know about several types of data values in Python. Two of them are integers called `int`, and decimal numbers (or floating points) called `float`. When you use an operator like + or > it produces a value. No matter what you put on either side of > in it produces a boolean value (`bool`), `True` or `False`. For other operators what type of value that is produced depends on which values the operator work on. Try this and see if you print an integer or a float (`8` or `8.0`):

```
x = 4
y = 2
result = x * y
print(result)
```

Now try to replace `4` with `4.0`. What type is result now?. Try to also replace `2` with `2.0`. What type is result now? Can you extract a rule for what the `*` operator produces depending on the what types the two values have?

Exercise 227

In exercise 226 you investigated what types of values the `*` operator produce. Redo that exercise with the operators: `+`, `-`, `/`, `**`, `//`, and `%`. What are the rules for what is returned if both values are integers, one values is a float, or both values are floats?

Exercise 228

Make a list of all the operators you know so far in order of precedence (without looking in the notes). Then check yourself.

Exercise 229

What does his expression reduce to?

```
3 > 2
```

Exercise 230

What does his expression reduce to? Do all the reduction steps in your head.

```
2 - 4 * 5 - 2 * 9
```

Exercise 231

What does his expression reduce to? Do all the reduction steps in your head.

```
3 > 2 and 2 - 4 * 5 - 2 * 9
```

Exercise 232

What does his expression reduce to? Do all the reduction steps in your head.

```
0 and 1 or 2
```

Exercise 233

What does his expression reduce to? Do all the reduction steps in your head.

```
4 and 1 or 2
```

Exercise 234

What is the value of `results` once the code below has run?

```
x = 7
y = 13
z = x + y
x = 0
result = x + y + z
```

Exercise 235

What is the value of `results` once the code below has run?

```
x = 5
y = x + 1
x = y + 1
y = x + 1
result = x + y
```

Exercise 236

In the code below I have shuffled the statements. Put them in the right order to make the code print `9`. To do that you must think about which values each variable will in each statement depending on the how you order the statements.

```
x = x + 1
y = 5
y = y - 1
print(y)
x = 1
y = y * x
```

Exercise 237

In the code below I have shuffled the statements. Put them in the right order to make the code print “

```
c = b
print(c)
a = b + a
b = 'og'
b = c + a
c = 'M'
a = 'ens'
```

Exercise 238

Make three exercises that requires the knowledge of programming so far. Have your fellow students solve them.

Unleash your functions

Serve to make you understnad exactly how functions work. If you understand recursion, you will undertand functions too.

Recursion

Divide and conquer

Building lists on the fly

List comprehensions

Example from translation project

Your own types of objects

Do you know that type you are? Python objects do, and as you have seen plenty of times already, that

What a type is

Making your own kinds of objects

You will not need this chapter for the exam, but you will need it for the rest of your life as a bioinformatician.

By know I hope you are quite familiar with what objects are.

Bundle data with relevant functionality.

Classes

A point class

- Other relevant classes you could make, orf, etc.

On the shoulders of (other nerd) giants

Python modules

Using code in other files

Standard library modules

BioPython

A birds eye view of Biopython to give you an impression of what you can do if you know just a little bit about modules classes

The Seq class

Reading and writing sequence formats

References