Informática para Ciências e Engenharias - B

Lecture Notes

Chapter 1

Introduction

Course objectives. The advantages of programming: code as recipe, toolset and documentation. Components of a computational system.

1.1 Objectives

The goal of this course is to help you to understand, and to start using, computer science methods to solve scientific and engineering problems. This is an introductory course that will cover basic aspects of programming, give a brief introduction to databases and computer networks and show some practical applications for simulation and data processing. The main programming language for this course will be Python. A convenient way of setting up Python 3.6 and all the libraries we need for this course is to install the Anaconda distribution, which you can download freely from `https://www.anaconda.com/download`.

For details on classes and evaluation, please see the lecture slides.

1.2 Informatics

The term *informatics* refers the broad field of automated information processing. It includes areas such as human-machine interaction, information theory, formal logical reasoning as well as computer science and engineering. Given that processing information is a crucial problem in all fields of science and engineering, informatics contributes to solving problems in many areas. In this course, we will focus mainly on programming, a small but important part of a this large subject.

1.3 Computer systems

In order to use computer programs, we need computers to run them. The term *computer* dates back at least to the 17^{th} century, originally referring to a person who performed calculations. From the 19^{th} century onwards it has been used to refer to computing machines and, nowadays, it means "a device that can be instructed to carry out arbitrary sequences of arithmetic or logical operations automatically"[1].

[1]Source: Wikipedia

A computer system is the hardware that makes up the computer and the software necessary for the computer to work. The hardware is the set of physical devices that make up the computer. In a typical personal computer, we can find:

Central Processing Unit (CPU) Circuitry that carries out the computations. It operates over binary values (0 or 1), determined by the voltage in the circuit elements. It performs very basic operations, but very quickly, on the order of thousands of millions per second.

Random Access Memory (RAM) Circuitry that stores data and executable code. This is volatile storage, and the data is lost when the computer is turned off. However, it is very fast, and so it is used for storing data being used and code that must be ready for execution. A typical personal computer has a few gigabytes of RAM memory (a few thousand million bytes).

Motherboard A combination of processors and circuitry that houses the CPU, the RAM and manages all communication between then and with the remaining hardware.

Storage Because RAM is volatile (and expensive), computers need additional devices that can store information permanently. Typically, these are Hard Disk Drives (HDD) or, increasingly common, the more expensive but faster Solid State Drives (SSD). These storage devices are where we store our files in our computers. Despite being much slower than RAM, they are not volatile, retaining all information even after we shut down the computer, and they also have a larger capacity, typically hundreds or thousands of megabytes.

Other peripherals Computers also need other devices that add specific functionalities, such as the mouse and keyboard, monitors, loudspeakers or printers. These are connected to the motherboard or to specialized cards that connect to the motherboard, such as graphics cards that handle graphical displays.

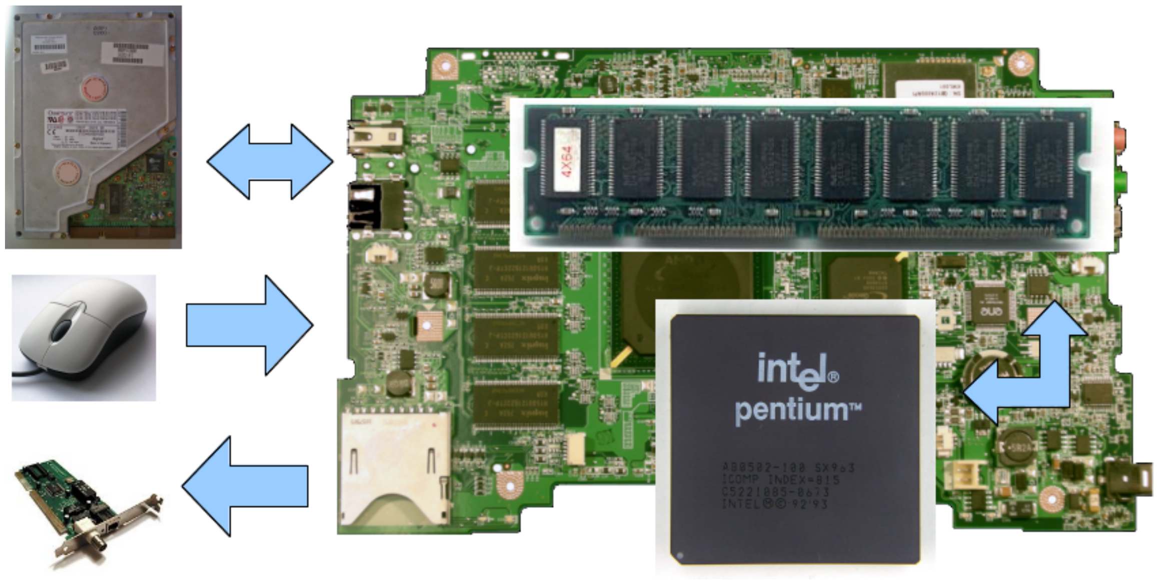


Figure 1.1: Schematic representation of the hardware in a computer system. Images source: Wikipedia.

Figure 4.1 shows the some common hardware components of a modern computer and, schematically, the way they communicate.

Computer hardware is able to process information but will do nothing unless we give it information to process. All information in digital computers is stored in binary values, usually represented as a 0 or 1 for each bit, and conceptually grouped in bytes of 8 bits each. This is mostly for historical reasons, because modern computers generally handle this binary information larger groups (called *words*). For example, in groups of 64 bits, or 8 bytes. Even though all this information is stored in sequences of bits, we can classify it in two kinds, depending on how it is used by the CPU: code and data. Code is interpreted as instructions that the CPU will execute, such as copying values, adding, subtracting, comparing values and so forth. Data is the set of values the instructions operate on. Both are needed for the computer to do anything. Since the CPU only executes very basic instructions, computer programs are generally composed of many thousands or millions of instructions. With all the data needed for images, sounds and other information, the total size can be substantial. The installation image of the Ubuntu operating system, for example, is currently 1.6 gigabytes.

To execute code and operate on data, the CPU needs to communicate with the RAM. In the RAM, each value is stored in an address, with each address working like a drawer: the address is always the same, but its contents — the value stored — can change. The CPU can ask the RAM for the value in an address or place a new value in a specified address. Figure 1.2 illustrates an CPU operation where the CPU is executing code in address 12 of the RAM that instructs it to add the values in addresses 106 and 107 and place the result in address 110. Note that all values stored in RAM are sequences of bits. The only difference between data and code is in what the CPU does with it, with code being interpreted as instructions and data being the operated upon following those instructions.

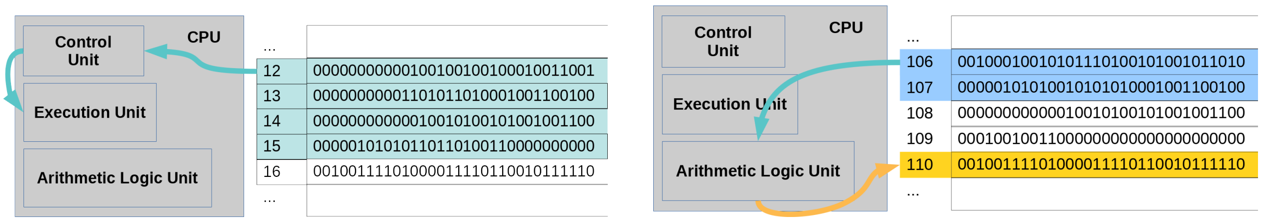


Figure 1.2: Schematic representation of an addition operation performed by the CPU (depicted in gray). In the left panel, the CPU is executing code at memory address 12. The Control Unit controls access to the RAM addresses where the instructions are located and the Execution Unit interprets a sequence of values composing the instruction to add the values in addresses 106 and 107 and store the result in address 110. Then, on the right panel, the values are obtained from their memory addresses, the Arithmetic Logical Unit computes the result and the resulting value is placed in the RAM address 110.

Like the hardware, the software operating on your computer is also composed of interacting modules. The first software to be executed is the Basic Input/Output System (BIOS) or, in more modern computers, the Unified Extensible Firmware Interface (UEFI). This software is stored in special components of the motherboard and includes all necessary instructions for checking that the hardware is working properly, identifying storage devices and starting the operating system.

If you have an operating system installed — and the computer not very useful without one — the BIOS or UEFI loads the boot sector of the storage device where your operating system is installed. In this sector, the operating system has the necessary code to load and initialize the kernel of the operating system. This is the core set of programs that will handle memory allocation, disk access and loading of drivers, which communicate with hardware like graphics and network cards, and applications. In modern operating systems, many programs will be loaded and executed make our computers operational, from the graphical interface to network connections and printer drivers. Together, this software and the hardware it controls make up the computer system we use to work with applications such as text processors or spreadsheets or play games. And, if we are fortunate enough to know some programming, it also allows us to execute our own code.

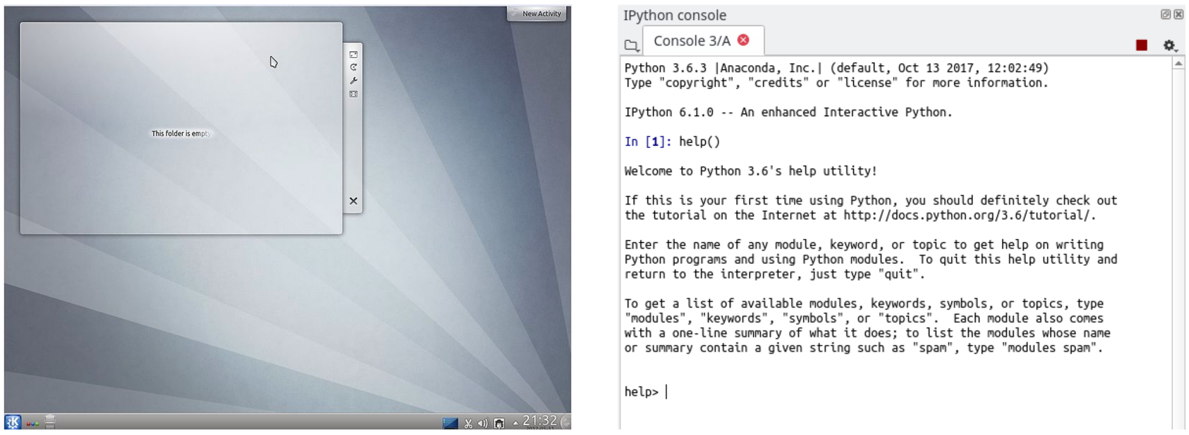
1.4 Why learn programming?

When we process data manually or with spreadsheets, the main result we obtain is the processed data itself. Even if some computation steps are automated, there is generally need for human intervention, raising the possibility of random errors that are difficult to replicate, being different every time we process the data. Programming focuses on the recipe for processing instead of on executing the task of processing itself, since execution is left to the computer.

There are several advantages to writing code and automating information processing, aside from the obvious one of making the computer do the work for us. The first is versatility. If we know how to program the computer, we can make it perform many more tasks than we can if we are restricted to using available software. Programming also leads to more reproducible results than doing things ourselves, since computers execute the same instructions in the same way, unlike humans. This makes programs more reliable, because as long as we make sure the program works correctly. Another advantage is that the source code documents the process in detail, specifying each step in a rigorous and unambiguous fashion. This is a major advantage over manual processing, especially in science and engineering where complex tasks can be crucially dependent on details that can be easily overlooked. And, finally, code can be reused and adapted, which can save a lot of effort when solving similar tasks.

1.5 How can we program the computer?

In order for the computer to do what we want we need some way to give it instructions. One common solution is to use a Graphical User Interface (GUI), which is available in all modern operating systems, mobile phones, tablets and so on. This is an interface with visual elements like buttons and windows that we can use to interact with the computer. Another useful interface is the console, where we can write text commands using the keyboard. Figure 1.3 illustrates these interfaces. The advantage of graphical interfaces is that it is easy to learn to use them. However, for complex commands, using them becomes cumbersome. Using a console and typing commands may be faster, once we know which commands to use. But even better is to organize commands into programs and execute them.



```
Python 3.6.3 |Anaconda, Inc.| (default, Oct 13 2017, 12:02:49)
Type "copyright", "credits" or "license" for more information.

IPython 6.1.0 -- An enhanced Interactive Python.

In [1]: help()

Welcome to Python 3.6's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/3.6/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics".  Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".


help> |
```

Figure 1.3: Examples of two interfaces: a graphical interface (Kubuntu, left panel) and a console interface (iPython console in Spyder, right panel).

In order for the CPU to execute a program, it must be given instructions that belong to the instruction set the CPU recognizes. These are very elementary instructions, like copying values or performing logical or algebraic operations. Programming directly in the language of the CPU would not be practical. This is why we have high-level programming languages, with instructions that a re closer to how a human thinks. Programs written in these languages are then translated into the language of the CPU in one of two ways:

Compilation A compiler translates the complete program into machine language and creates an executable file that the operating system can load into memory and have the CPU execute. This is how languages such as C, C++ or Pascal work. The advantage of a compiled language is that the resulting program is generally more efficient, both in speed and size, but the disadvantage is that a compiled program only works on the CPU and operating system it was compiled for. So you cannot use a compiled Windows program in Linux or your tablet without some additional software, such as emulators.

Interpretation An interpreter translates the program as it is executed. It may create a special representation of the program that is somewhat similar to compilation, but the program is not fed directly into the CPU by the operating system, always requiring the interpreter software to be executed. This is the main disadvantage of interpreted programs, which not only makes them dependent on the interpreter but also generally less efficient. However, an interpreted language has the advantages of not requiring a compilation step and making easier to use the same program indifferent operating systems and computers. Examples of interpreted languages are Matlab and Python.

In this course we will write our programs in Python, an interpreted language. For this, we will use the Python interpreter through the iPython console and the Spyder Integrated Development Environment (IDE). Spyder makes it easy to manage the source code files and test our programs. The *source code* is the code of our programs, which we will write and save to files with the extension `.py`, which is the convention for Python source code files.

Figure 1.4 illustrates the execution of Python code during this development stage, passing from the Spyder IDE to the interpreter, the operating system and finally, after translation, run on the CPU.

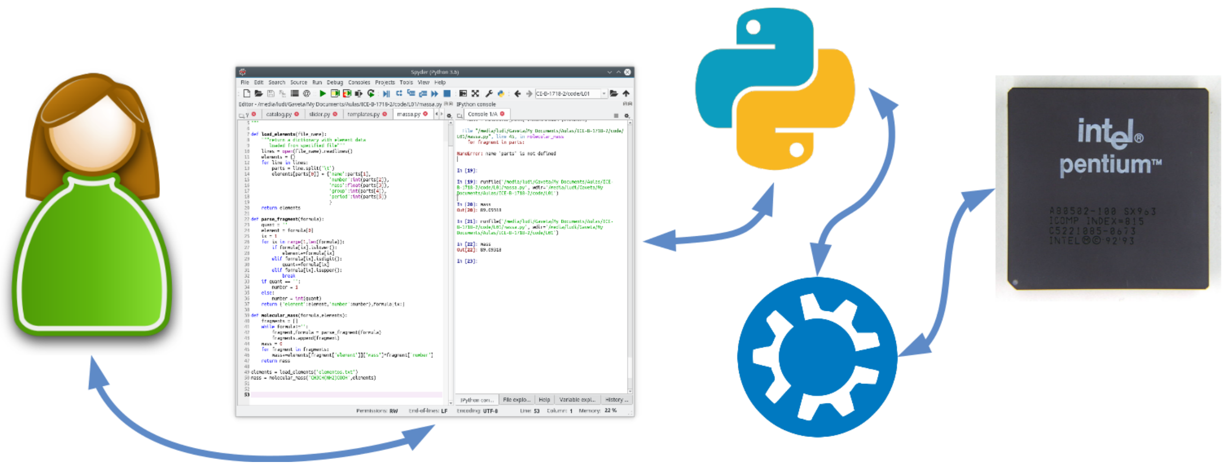


Figure 1.4: Programming Python with the Spyder IDE. The user interacts with the IDE, which includes the editor for the source code and the iPython console. The IDE interacts with the Python interpreter which, in turn, using the operating system, will run the code in the CPU.

1.6 Extra material

Throughout this course you will find some sections marked with this image. This means the subject matter covered in those sections will not directly count for your evaluation in this course. But these additional concepts, Python constructs or examples can help you better understand the core subjects or provide you with practical solutions to problems that can be harder to solve without this knowledge.

1.7 Exercise: Spyder tutorial

This is a simple exercise you can do at home just to start working with the Spyder tutorial. Install Anaconda from the web page `https://www.anaconda.com/download`, and then look for Spyder in your installed programs. Most operating systems have a search box you can use; simply type "spyder".

Once Spyder is running, you can see the source code editor on the left side. Start by selecting "Save as" from the "File" menu and save your source code file in a folder of your choice. Name it, for example, `tutorial.py`. The initial source code already has some text typed in. We will just leave it in and add the two `print` instructions as shown below:

```python
# -*- coding: utf-8 -*-
"""
Created on ...

@author: ...
"""

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

To execute your code, click on the iPython console window on the lower left corner and press the `F5`. You will see something like this appear on the console:

```
In : runfile('/user/student/tutorial.py', wdir='/user/student')
2
4
```

The `runfile` function is a utility function supplied by Spyder that runs all the code in your file in the folder you choose. The two numbers `2` and `4` are the result of the two `print` instructions in the source code.

You can also type instructions directly on the console. For exameple, if you type `2+2` you should see something like this:

```
In : 2+2
Out: 4
```

The iPython console automatically echoes any value that is returned. Note that this does not happen with the source code you run. In that case, if you want to see some value displayed you need to use the `print` function.

You can run part of your file using the F9 key. This will execute the line where the cursor is placed in your source code or the block you selected.

1.8 Further reading

Chapters 1 and 2 of the textbook [1].

Chapter 2

Python variables

First contact with the Python programming: objects and variables; numbers and strings. String methods.

2.1 Objects

Python is an object-oriented programming and most of the elements of a Python program are objects. The term *object* is deliberately vague in this context because an object can stand for nearly anything. In the context of an object-oriented programming, an object is a construct that contains data and functions that operate on the data. Formally, in object-oriented programming we say that data is stored in *fields* and the functions belonging to the object are its *methods*. Thus, can think of an object as a mini-program that can do things we tell it to do.

One of the simplest type of objects in Python are the numeric constants. They are constants because they always have the same value. For example, `2` always has the value of 2. We will see later on that we can also have variables, which can take different values. You can see all the fields and methods of an object using the `dir` function. For example, if you type `dir(2)` in the console, you can see all the fields and methods of the constant object `2`:

```
In : dir(2)
Out:
['__abs__',
 '__add__',
 '__and__',
  [some others omitted]
 'bit_length',
 'conjugate',
 'denominator',
 'from_bytes',
 'imag',
 'numerator',
 'real',
 'to_bytes']
```

In Python, fields and methods that begin and end with two underscore characters, such as `__add__`, have special roles. The most used methods are generally those without the underscores, such as `real`, for example, which returns the real part of a complex number.

Numbers in python can be integers, floating point (meaning they have a decimal point) or complex numbers (with a real and an imaginary parts, denoted with the `j` character). Here are some examples of operations with numbers:

```
In : 4 + 5    # addition
Out: 9

In : 3 * 6    # multiplication
Out: 18

In : 2 ** 5   # exponentiation
Out: 32

In : 5 / 2    # division
Out: 2.5

In : 5 // 2   # integer (floored) division
Out: 2

In : 7 % 2    # modulus (remainder)
Out: 1

In : 23.6 * 0.71  # floating point numbers
Out: 16.756

In : 12+3j + 2j  # complex numbers
Out: (12+5j)
```

Note the comments following the hash sign `#`. In Python, everything following `#` in a line is ignored by the interpreter, so we can use this character to precede any comments we may want to add to our code.

Another type of object is the string. A string constant is a series of characters delimited by either single quotes, `' '`, or double quotes, `" "`. The quotes are necessary to tell the interpreter that the text within is not to be interpreted but to be taken literally. Here is an example

```
In [1]: print('hello')
hello

In [2]: print(hello)
Traceback (most recent call last):

  File "<ipython-input-13-43a14fcd4265>", line 1, in <module>
    print(hello)

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

If we type `print('hello')` the interpreter understands that the `print` word refers the printing function and that `'hello'` is a sequence of characters to print. However, if we type `print(hello)` the interpreter tries to figure out what `hello` means and stops execution with an error when it doesn't find any reference to an object named hello.

String constants are also objects, and also have fields and methods as you can see with the `dir` function:

```
In: dir('hello')
Out:
['__add__',
 '__class__',
 '__contains__',
 '__delattr__',
[many others omitted]
 'startswith',
 'strip',
 'swapcase',
 'title',
 'translate',
 'upper',
 'zfill']
```

There are some operations we can do with strings too, such as concatenation (adding two strings) or repeating strings, multiplying by an integer number.

```
In : 'hello' + 'world'
Out: 'helloworld'

In : 2*'hello'
Out: 'hellohello'
```

Strings also include useful methods that perform operations often used with text. For example, converting to title case, to upper case, replacing parts and so on. Here are some examples.

```
In : 'hello world'.title()
Out: 'Hello World'

In : 'HELLO'.lower()
Out: 'hello'

In : 'hello'.upper()
Out: 'HELLO'

In : 'hello'.replace('l','x')
Out: 'hexxo'

In : 'Hello'.swapcase()
Out: 'hELLO'
```

Note that when we want to execute a function, such as these methods of string objects, we must follow the name of the function with parentheses. In the case of the `replace` method, inside the parentheses we include the strings to replace. In this example, we replace all `l` characters with `x` characters. These are the *arguments* of the method. But even when the method has no arguments, such as the `upper` or `lower` methods, we still have to include the parentheses, even though they are empty. This is because the parentheses tell the interpreter that we want to execute the function and not merely refer to it. Note what happens when we do this:

```
In : 'hello'.upper
Out: <function str.upper>
```

This tells us that the `upper` method is a function that belongs to the string objects, but this does not execute it. One common case where we want to refer the function instead of executing it is when asking for the documentation on the function. This can be done using the `help` function:

```
In : help('hello'.replace)
Help on built-in function replace:

replace(...) method of builtins.str instance
    S.replace(old, new[, count]) -> str

    Return a copy of S with all occurrences of substring
    old replaced by new.  If the optional argument count is
    given, only the first count occurrences are replaced.
```

In this example, we use the `help` function to see what the `replace` method of this string object can do. So we provide the `help` function, within parentheses, with a reference to the `replace` method of a string object (in this case, `'hello'`). It's important to note two syntactic details we already met previously. First, all arguments we pass to any function, whether or not it is the method of an object, is passed inside parentheses. The other is that we refer any method or field of an object with the `.` character between the object and the name of the field or method. Here are some examples of useful Python functions. Their purpose should be obvious but remember that you can always use the `help` function to learn more about any function, as illustrated below.

```
help(len)
Help on built-in function len in module builtins:

len(obj, /)
    Return the number of items in a container.

In : len('hello')
Out: 5

In : len('bye')
Out: 3

In : round(348.2)
Out: 348

In : int('234')
Out: 234

In : str(124)
Out: '124'
```

2.2 Polymorphism

EXTRA

In object oriented programming, polymorphism is the use of a single interface across different object types to allow operations to be applied to the different object For example, the addition operation in Python uses the `__add__` method of the objects involved . If you type `'hello' + ' world'`, the intepreter will use the `__add__` of the constant object `'hello'` to add the constant object `' world'`. Multiplication uses the `__mul__` method of the object. Note that these are special methods, as indicated by the double underscores in the beginning and end of their names. The example below illustrates this:

```
In : 'hello'.__mul__(2)
Out: 'hellohello'

In : 'hello' * 2
Out: 'hellohello'

In : 'hello'.__add__(' world')
Out: 'hello world'

In : 'hello' + ' world'
Out: 'hello world'
```

In practice, we will not generally use these special methods, since it's much simpler (and readable) to write `'hello' + ' world'` than to write `'hello'.__add__(' world')`. However, it is useful to understand that these operations are just a simplified notation for using the methods implemented in the objects. The advantage of using these interface is to make it possible to use these operators with any object that implements the necessary methods. This is why the addition operation can be used between numbers and between strings and why we can multiply a string by an integer. Later, we will see other useful objects for storing numerical vectors and matrices that share this property, simplifying common algebraic operations.

2.3 Variables

In order to process data we need to have some place to store it in, and not everything in our program will have constant values. This is why we need variables. We can think of a variable as the name of a place — a metaphorical drawer, for example — where we store an object. To create a variable we only need to assign an object to a name using the = operator. When we create a variable, the interpreter asks the operating system for a block of memory addresses, creates the object there and stores the correspondence between the variable name and that memory location. From this point onwards, when we use that name the interpreter knows which object we are referring to. We will see later on that the interpreter stores these names in different tables depending on where the names are created, in order to avoid collisions between names in different contexts, but for now this idea of creating the object in memory and remembering its name suffices. Thus, to create variables we only need to think of some name where we want to store a value. For example:

```
# Create variables and choose names
nothing = None                          # no value
minimum_wage = 505                      # integer
pi_squared = 9.8696                     # float
first_name = 'Ludwig'                   # single line string
# multiple lines string:
text = '''this string
has several
lines'''
```

Since all data in memory is stored as a sequence of bits, of 0 and 1, the interpreter must also know how to interpret each sequence. This depends on the type of object the variable is referring to. The first line in the code above creates a variable with no value. The value `None` acts as a placeholder for nothing, and can be useful if we want the interpreter to know that variable exists but do not have a value to assign to it yet.

The second and third lines create an integer variable and a floating point variable. This name may seem confusing, since it stems from the way the number is represented, but what it means is a number that has a fractional part. The fourth type is a string, which is an ordered sequence of characters enclosed by single or double quotation marks. If we want strings to span more than one line, we can define them starting with three quotation marks and terminating with three quotation marks of the same type (single or double).

Variable names can be any sequence of letters, digits or the underscore character _ that does not start with a digit. Common practice in Python is to use descriptive names for variables written in lower case letters and using the underscore character to separate words and make them easier to read. Once a variable is created by assigning a value to its name, we can use the name in place of the value in any expression. For example:

```
In : minimum_wage * pi_squared
Out: 4984.148
```

Note that the = operator always assigns the value on the right to the variable on the left. This is why the first example below is valid but the second one is not, since `32` is a constant value and not a valid variable name:

```
In : thirty_two = 32
In : 32 = thirty_two
File "<ipython-input-16-7ac92e052411>", line 1
    32 = thirty_two
                  ^
SyntaxError: can’t assign to literal
```

Since variables refer to objects, we can use the variable name as a placeholder to access any method or field belonging to the object. For example:

```
In : name = ’Ludwig’

In : name.upper()
Out: ’LUDWIG’

In : name.replace(’d’,’---’)
Out: ’Lu---wig’

In : name.find(’w’)
Out: 3
```

Note that the console echoes any value returned that is not assigned to a variable. This is what happens if we write `name.upper()`, for example. The `upper` method returns the string in `name` converted to uppercase and since this is not placed in any variable, the console echoes the result and we see `Out: 'LUDWIG'`. However, if we store the value in a variable, this does not happen. For example:

```
In : upper_name = name.upper()
In : upper_name
Out: 'LUDWIG'
```

In this case there is no echo from the first instruction because the result was stored in `upper_name`. As an aside, since variable names cannot include the space character, it is usual to use the underscore character _ to separate words when creating variable names with more than one word, sucn as `upper_name`.

Python looks for a name by looking first in a dictionary of locally created names, then in globally created names (we will see more about this when we look at functions) and, finally, in built-in names. This means that names you declare locally have precedence over other names, such as those of built-in functions. This means that if you create a variable named `help`, for example, you are no longer able to use the `help` function. If this happens accidentally, you can "throw away" a variable using the `del` function. This example illustrates the problem and this solution.

```
In : help = 31

In : help(len)
Traceback (most recent call last):

  File "<ipython-input-39-d29297b41f1b>", line 1, in <module>
    help(len)

TypeError: 'int' object is not callable

In : del(help)

In : help(len)
Help on built-in function len in module builtins:

len(obj, /)
    Return the number of items in a container.
```

If you want to see the names of variables declared in the current scope, you can use the `dir` function without arguments: `dir()`. Note, however, that the iPython console creates variables to store the results of operations in the console, so you will likely see a lot more variables in the list than the ones you created.

2.4 Example: computing concentration

To illustrate using variables and operators, we will compute the concentration of two grams of table salt in 125ml of water. We'll start by creating variables for the molecular mass of NaCl (58.4g) and the volume (in dm^3), then compute the amount of NaCl in 2g, in mol, and finally the concentration in mol dm^{-3}. Try out this program by typing it into a new file, save it as a Python source code file (*e.g.* concentration.py) and then run it.

```
# -*- coding: utf-8 -*-
```

```
"""
Compute concentration of 2g of NaCl
"""

mmNaCl = 58.4
volume = 0.125
amount = 2 / mmNaCl
concentration = amount / volume
print('Concentration is ',concentration,'mol/dm^3')
```

2.5 Exercises, demonstration

This set of exercises will be demonstrated in the tutorial preparation session, but feel free to try them out for yourself too.

Using Spyder

Open the IDE Spyder, create a new file if necessary and save it in your working folder with a `.py` extension. This file will be recognized as a source code file by the interpreter and you can edit it on the window on the left in Spyder. On the bottom right corner of the Spyder window is the iPython console, which you can use interactively to execute Python instructions.

Whenever you wish to run all the code in the current source file (called a `module` in Python) you can press F5 and Spyder will run the code on the selected console. A window may pop up asking you to specify which console to run your code in; the simplest option is to run on the current console.

1. Write on the editor the code to create a variable called `number` and assign it a numerical value. Run the source code (F5) and check the value of the variable in the console.
2. Write on the editor the code to increment the variable by 1. Run only this line (F9) and check the value of the variable again.
3. Run this line several times and see the effect on the variable.
4. Run everything again (F5), check the value on the variable and explain what happened.

Sequences of code

For this demonstration, do the NaCl concentration exercise described above. Experiment running the full program (F5) or parts of the program (F9) with different values and try to understand the result. For example, if we change the mass of NaCl used do we have to run all the code again? Why, or why not?

Change the variable names so the program is more generic and not specific for NaCl. Also break it down further and reorganize it so that the parameters the user may want to change are more easily accessible, assigned to specific variables instead of being hidden inside formulas.

2.6 Exercises (tutorial 1)

This set of exercises will cover the basics of variable creation and expressions. It is also meant as an introduction to the Spyder IDE. You can do each exercise by writing your instructions directly in the iPython console window, on the lower right corner of Spyder, but the best approach is to create a source code file (*e.g.* `tutorial1.py`) and write all answer in the editor window. To execute each line individually, you can place the cursor at the desired line and press F9.

Simple expressions

1. Compute 23% of 5726
2. Compute the square root of 62. Note that the square root of x is $x^{\frac{1}{2}}$.
3. Compute the cubic root of 138.
4. Create a string that is a sequence of 82 occurrences of `ACGT`. *I.e.* `ACGTACGTACGT...` (82 times). Why do you need quotes around `ACGT`?
5. Compute the number of digits in the cube of 8963. Hint: use the `str` and `len` functions.

Variables

6. Store the cubic root of 138 in a variable and then verify that the value in the variable is approximately correct (two lines of code).
7. Create a variable called `pi` with the value of 3.1416 and another variable `r` with a value of 3. Write the expressions for computing the perimeter ($2\pi r$) and the area (πr^2) of a circle of radius r.
8. Change the value in variable `3` to 4, 5, 6 and 7 and, for each value, reuse the expressions for the previous question to compute the respective perimeters and areas. Remember that you can execute a line of code from the editor by placing the cursor on that line and pressing F9.
9. The Fibonacci series is a series of numbers starting from 0 and 1 and continuing by adding the two previous numbers: 0, 1, 1, 2, 3, 5, 8, 13, ...

   Each Fibonacci number F_n, starting from $F_0 = 0$, can be computed directly from:

   $$\phi = \frac{1+\sqrt{5}}{2} \qquad \psi = \frac{1-\sqrt{5}}{2} \qquad F_n = \frac{\phi^n - \psi^n}{\sqrt{5}}$$

   a) Compute F_8
   b) Create a variable `n` with the value 9. Then write the expression for computing F_n to compute F_9

10. You can also compute the Fibonacci numbers recursively, from the definition of the series. Start with two variables, `f_minus_two = 0` and `f_minus_one = 1`. These represent, respectively, the values of F_0 and F_1.

    Now use the relation $F_n = F_{n-2} + F_{n-1}$ to put in variable `fn` the value of F_2

11. Write a sequence of three commands that will change `f_minus_two`, then `f_minus_one` and then `fn` so that `fn` takes the value of F_3 using the same relation $F_n = F_{n-2} + F_{n-1}$.
12. Repeat the previous sequence of instructions so that, after each pass, `fn` takes the value of F_4, then F_5, F_6 and so on.

2.7 Further reading

Chapter 3, Chapter 4 up to page 99 (the section on Numbers) and Chapter 5 of the textbook [1].

Chapter 3

Structured programming

How to solve complex problems. Scripts and functions. Structured programming. Scope of names.

3.1 Solving a problem

Let us consider the dissociation equilibrium of benzoic acid:

$$C_6H_5COOH \longleftrightarrow H^+ + C_6H_5COO^-$$

The equilibrium constant in water is $K_a = 6.5 \times 10^{-5}$. To compute the pH of a solution of concentration C_i of benzoic acid, and disregarding the auto-dissociation of water, we can assume that $[H^+] = C_6H_5COO^-$, which we will call x, and solve the quadratic equation:

$$K_a = \frac{[H^+][C_6H_5COO^-]}{[C_6H_5COOH]} = \frac{x^2}{C_i - x} \Leftrightarrow x^2 + K_a x - C_i K_a = 0$$

For which we can use the quadratic formula $x = \frac{-b \pm \sqrt{b - 4ac}}{2a}$. After we compute x, we can compute the pH knowing that:

$$pH = -log_{10}([H^+]) = -log_{10}(x) \approx -\frac{(1000(x^{1/1000} - 1)}{2.305}$$

The approximation for the logarithm comes from the equation $ln(x) = \lim_{x \to \infty} n(x^{1/n} - 1)$, and 2.305 is a conversion factor to obtain the base 10 logarithm. With $n = 1000$ we get a good approximation to the logarithm of x.

Now that we know how to do this computation, we can implement a script to compute the pH of a 0.01M solution of benzoic acid:

```
# -*- coding: utf-8 -*-
"""
Compute pH
"""

Ci =  0.01
```

```python
Ka = 6.5e-5
x=(-Ka + (Ka**2 + 4*Ka*Ci)**0.5)/2
pH = - (1000 * (( x**(1/1000)) -1) ) / 2.305
print(pH)
```

Running this script should print out the pH value of approximately 3.1.

3.2 Namespaces and importing modules

There are several problems with this script. An obvious one is with the computation of the logarithm. This is a very common operation and it makes little sense to have to compute an approximation with a cumbersome formula whenever we need a logarithm. Fortunately, Python makes it easy to import code fragments from libraries and other source code files and the numpy library has a logarithm function.

In Python, each name belongs to a *namespace*. Basic python statements and keywords belong to a global namespace and are always accessible. Some names are defined within the namespace of an object, as we saw before with several string methods. Each Python source code file (the `.py` files), called a *module*, has its own namespace. The `import` command allows us to use in one module names that belong to the namespace of another module, either by importing the module or a specific name from that module. There are different ways in which the `import` command can be used generally used. Here are two examples, both for using the `log10` function from the `numpy` module, which computes the base 10 logarithm of a number.

```python
 1 # -*- coding: utf-8 -*-
 2 """
 3 Compute pH
 4 """
 5 import numpy
 6 
 7 Ci =  0.01
 8 Ka = 6.5e-5
 9 x=(-Ka + (Ka**2 + 4*Ka*Ci)**0.5)/2
10 pH = - numpy.log10(x)
11 print(pH)
```

```python
# -*- coding: utf-8 -*-
"""
Compute pH
"""
from numpy import log10

Ci =  0.01
Ka = 6.5e-5
x=(-Ka + (Ka**2 + 4*Ka*Ci)**0.5)/2
pH = - log10(x)
print(pH)
```

In the example of the left, we use `import numpy` to import the name of the `numpy` module into the namespace of our module. In Python, modules also are objects, and since the `numpy` module has the `log10` function, we can use that function to compute the logarithm of x by writing `numpy.log10(x)`, in line 10 of our program. On the right, we use `from numpy import log10` to import the `log10` function name directly into our namespace. This way, in line 10 we only need to write `log10(x)` to use the function.

This is one way in which we can structure our programs in Python. Since each module can import and use code from any other module, we can split a complex program in several smaller files in any way that makes sense to us and makes it easier to implement the program and reuse our code. We can also use code from pre-programmed modules, and the Anaconda distribution of Python includes many useful modules, some of which (such as `numpy`) we will be using regularly.

3.3 Structured Programming

The first version of our script to compute the pH value was not very convenient. We improved it somewhat by importing the logarithm function from `numpy`, delegating part of our computation to code that was already available. This is an important notion in *structured programming*, a programming paradigm that aims to solve complex problems by breaking them down into smaller problems. The foundation of structured programming is the Böhm-Jacopini theorem, which states that any computational problem can be solved by combining subprograms in just three different ways: sequentially, in conditional branches and in conditional repetitions. We will see how to create branches and repetitions in our code later in the course. For now, we will focus on the ideia of combining subprograms in sequence.

We can see this in our pH script. We start by creating the variables for the initial concentration and equilibrium constant, then we compute x, and then we call a subprogram, the `numpy.log10` function, to compute the logarithm. This gives us the correct result but, as it is, our script cannot be used as a subprogram to solve some larger problem. For example, if we know the mass of benzoic acid dissolved in some volume of water, we can easily compute the concentration. However, then we cannot use our script because our script only works for a concentration of 0.01M. To be able to reuse our code and structure our programs better we need to be able to create subprograms that can take in arguments and return the appropriate values. In other words, we need to create functions like the `log10` in `numpy`

3.4 Creating Functions

Like most things in Python, functions are objects. They are special objects because they can be executed, but they are still objects that we must create and assign some name before we can use. To create a function object in Python we use the `def` keyword, followed by the name of the function and, in parentheses, the function *parameters*, which are variable names where the function arguments will be stored. Here is a script with function that adds two numbers:

```
1 def add(a,b):
2     result = a + b
3     return result
4
5 total =  add(3,5)
```

The first line tells the intepreter to create a function object called `add` and specifies that, when executed, this function will receive two values that will be stored in variables `a` and `b`. This line ends in a colon, `:`, which is the Python convention for the start of a code block. After the colon, all lines that have an indentation relative to the line with the colon belong to that block. We can see this in lines 2 and 3, which are indented to the right, and thus are part of the function.

Line 2 computes the sum of the values in variables `a` and `b`. Note that, when we are writing the function, we do not know what the values of these variables will be. But, since we specified that they are parameters of this function, we know that they must be supplied by whoever calls the function when the function is executed. So we just write the "recipe" for the function in a way that will use whatever values are supplied.

The `return` keyword causes the interpreter to leave the function immediately and return references to the objects named after the `return` keyword. In this case, line 2 created an object called `result` with the sum of texttta and `b` and so, in line 4, the variable `total` will refer the same object. Thus, after we run our script, we can do this in the console:

```
In : total = add(3,5)
In : total
Out: 8

In : add
Out: <function __main__.add>

In : add(total,4)
Out: 12

In : add(10,-24)
Out: -14
```

Variable `total`, created in our script outside the function (note the indentation, at the same level as the `def` statement), has the value 8, which is the sum of 3 and 5 computed in line 5 of our script. Furthermore, the name `(`add`)` is defined, referring to a function in our main module (the main module is the script we are executing). This means we can call this function by writing the parentheses and the arguments, which are the values to put into the parameters. In this way, we can execute the function to add 1 and 3, or 10 and -24. In each execution, these numbers can be referred to by the variables `a` and `b` inside the function.

Functions have local namespaces

The variables that are created inside the function, such as `a`, `b` and texttttresult in this example, only exist while the function is being executed and are only visible inside the function. Once the interpreter reaches the `return` statement or the end of the function, these local variables are eliminated and they are isolated from any code outside the function. That is why this happens:

```
In : a
  File "<ipython-input-64-60b725f10c9c>", line 1, in <module>
    a

NameError: name ’a’ is not defined

In : result
  File "<ipython-input-65-a5b1e83cd027>", line 1, in <module>
    result

NameError: name ’result’ is not defined
```

Furthermore, the variables `a`, `b` and texttttresult are created when the function is executed and exist in a different namespace from that of variables outside the function. Note this example:

```
In : result = 0

add(5,5)
Out: 10

In : result
Out: 0
```

We start by creating a variable named `result`, with a value of 0. Then we call the function to add 5 with 5, and the function creates a variable named `result` with a value of 10. But this `result` variable is local to the function and independent of other variables outside the function, even if the name is the same. That is why the `result` we declared previously still has the same value of 0.

Variable isolation is an important feature of functions. If you want to remember only one thing from this section, this is the one to keep. In order to be able to decompose complex problems into subprograms, we need to be able to call these subprograms without them interfering with each other. It would not be practical to keep track of all variable names just to guarantee one function would not inadvertently change the value of a variable another function needs. To prevent this, each function has its own namespace where its own variables (local variables) are created. So we must remember that anything we need to access outside the function must be returned by the function in the `return` statement.

3.5 Algorithms and problem decomposition

Now that we know how to write functions — the subprograms we need for structured programming — we can start thinking about how to solve complex problems. The main idea is that we can make any minimally complex problem easier to solve if we break it down in smaller, simpler, subproblems. And one crucial concept in this is the *algorithm*.

In the 12th century, the persian mathematician Muhammad ibn Mūsā al-Khwārizmī proposed methods for solving algebraic equations using what were then called the “Indian numerals”. This was a number system that the arabs adapted from India and which replaced the roman numerals in Europe. The Portuguese word “algarismo” comes from the name of al-Khwārizmī, for his introduction in the west of this numbering system we still use today. Another word that comes from his name is “algorithm”, originally referring to his methods for solving equations. In computer science, an *algorithm* is a process defined by a set of steps that solves some computational problem. Taking some liberty with the term, we can illustrate some basic ideas with the “algorithm” for toasting bread. Here is an outline of the procedure:

```
1. Pick up bread
2. If the bread knife is on the table:
         Pick up the bread knife
   Else:
         Get the bread knife from the drawer
3. Cut a slice of bread
4. Put slice on toaster
5. While bread is not toasted:
         Wait
```

This simple procedure illustrates the three elements that the Böhm-Jacopini theorem proves sufficient for any computation: sequential execution, conditional branching (the “If ... Else” in line 2) and conditional repetition (the “While” in line 5). We will postpone conditions and loops for later, and focus now on decomposing one problem in subproblems.

This scheme gives us the algorithm for making toast. It’s a set of instructions that, if followed, result in toasted bread. Once we have an idea of how to solve the problem, we can break it down into smaller problems that we can solve independently. For example, suppose that we already solved the problem of finding the bread knife and can reuse that procedure. In that case, we can simplify our algorithm to:

```
1. Pick up bread
2. Find bread knife
3. Cut a slice of bread
4. Put slice on toaster
5. While bread is not toasted:
       Wait
```

The “Find knife” procedure in line 2 is responsible for doing whatever is necessary for getting the bread-knife, either from the table top or the drawer. This reutilization of a previous solution is what we did with the logarithm function in the pH problem. But note that we can reason in this way even if we do not yet have the solution. Even if we did not have a function to find the bread knife, it would still be useful to break down this problem into the two different ones: making the toast assuming we can find the bread knife, and finding the bread knife, regardless of what we want to use it for. This reasoning is what allows us to solve complex problems and is a fundamental part of computational thinking. Here is a generic roadmap for this approach:

Requirements The first step to solving a problem is understanding in detail what you need to obtain. Think about the data you will start from and what is your desired result. For example, you have bread and want some toast.

Algorithm Think about how to solve the problem. List the steps and operations required to go from the starting data to the desired results and make sure you understand what needs to be done in each step. If the problem is complex, you can do this in larger steps, with less detail.

Decomposition Once you have an idea of what needs to be done, break the process down into more manageable subproblems. It is not useful to try to solve everything at once. For each subproblem, you can repeat all these steps but in increasingly greater detail. Do this iteratively until you have problems that are simple enough that you understand completely how to solve them.

Generalization Remember that subprograms that help solve one particular problem may be useful for solving other problems. The `log10` function we used in the pH calculation is one example. So, when you are breaking down your problem into subproblems, think about ways of making the subprograms you write more general and useful in different contexts, abstracting more generic solutions from the details of the current problem.

Implementation This is the stage where you actually write the code. You can start by writing the name and arguments of each function you need to implement and decide what values each function returns.

This is a general framework for programming. One important detail to note is that you should only start writing code after you know exactly what you want to implement. Of course, things will not always run smoothly from the problem to the final solution. You will generally need to test your code, fix errors, rethink some parts that may turn out differently than originally intended and so on. In the next chapter we will see this in more detail.

3.6 Exercise

In the next chapter we will implement our pH calculation using functions, generalizing the solutions and writing everything in a more structured manner. But, before you read the next chapter, try to apply this framework to the problem of computing the pH of a solution of a weak acid. Think about the information that you need, what the result should be, what are the steps to solving the problem, how to break the problem down into simpler subproblems and solve each one.

3.7 Further reading

Chapter 16 up to page 479 (Calls) and Chapter 17 up to page 491 (The Built-in Scope) of the textbook [1].

Chapter 4

Implementation

Writing our first structured program. Unit tests and the life cycle of a program. Errors and numerical precision.

4.1 Once more, with structure

Let us consider again our pH problem. In the previous chapters we considered the specific case of benzoic acid, with $K_a = 6.5 \times 10^{-5}$:

$$C_6H_5COOH \longleftrightarrow H^+ + C_6H_5COO^-$$

But we can generalize this problem to a generic acid AH where we have a dissociation equilibrium and can disregard the auto-dissociation of water. Abstracting from the specific details, what we need is a program that receives as input an initial concentration C_i and a K_a value and returns the pH value.

Knowing what we want, we can think about the algorithm. We can do this by solving the equation below for x, which corresponds to $[H^+]$, and then computing symmetric of the base 10 logarithm of x.

$$K_a = \frac{[H^+][A^-]}{[AH]} = \frac{x^2}{C_i - x} \Leftrightarrow x^2 + K_a x - C_i K_a = 0$$

This equation can be solved using the quadratic formula $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$.

Now we can break down this procedure and look more carefully at the parts. Let's start by outlying the different functions we will need. We can write one function to give us the positive root of the quadratic equation. We want the positive root because only a positive concentration makes sense. The other function can be the one who computes the pH from the C_i and a K_a values using the positive root function. Here is the skeleton of the program, with the two functions, in a file we can save as `phcalc.py`.

```python
# -*- coding: utf-8 -*-
"""
Computes pH of an acid from concentration
"""

def positive_root(a, b, c):
    """
    return positive root of quadratic equation
    """

def compute_pH(Ka, Ci):
    """
    return pH value for weak acid from
    dissociation constant and initial concentration
    """
```

This code does not do anything yet, because there are nothing inside the functions except the documentation. The documentation of a function is a string that follows immediately after the signature of the function (the `def ... :` line). This allows us to form a more detailed idea of how our program will look like, making sure we have everything we will need, before we start solving each subproblem.

If we run our `phcalc.py` file, apparently nothing will happen but these functions will be defined. So, after running (press F5) we have the two function objects `positive_root` and `compute_pH`. We can see the usefulness of the documentation strings with the `help` function, which prints the documentation of the function we provide:

```
In : help(positive_root)
Help on function positive_root in module __main__:

positive_root(a, b, c)
    return positive root of quadratic equation


In : help(compute_pH)
Help on function compute_pH in module __main__:

compute_pH(Ka, Ci)
    return pH value for weak acid from
    dissociation constant and initial concentration
```

This includes the documentation of the module itself. After running the file with the F5 key, Spyder changes our working directory to the folder where the file has been saved. This allows the Python interpreter to find the file and we can import the module `phcalc` and ask for its documentation:

```
In : import phcalc
In : help(phcalc)
Help on module phcalc:

NAME
    phcalc - Computes pH of an acid from concentration

FUNCTIONS
    compute_pH(Ka, Ci)
        return pH value for weak acid from
        dissociation constant and initial concentration

    positive_root(a, b, c)
        return positive root of quadratic equation
```

This shows the usefulness of adding docstrings at the beginning of your modules. It makes it easier to understand what each module does if you then want to use them in more complex programs.

Now we can work on the implementation of each function. Since this is a simple example, we do not need to subdivide our problems even more, but note that with complex programs we may have to repeat this process several times until we find parts that are simple enough to implement. In this case, since `compute_pH` needs to use the texttt positive_root function, we need to start by implementing the `positive_root` function. The `positive_root` function simply uses the quadratic formula, and we must not forget to tell the interpreter what to return when exiting the function. If we omit the `return` instruction the function will return a `None` value.

```
def positive_root(a, b, c):
    """
    return positive root of quadratic equation
    """
    root = (-b + (b**2 - 4*a*c)**0.5)/(2*a)
    return root
```

This, in essence, is the procedure for creating a program. After understanding the objectives and the algorithm, we outline the subprograms we need to write — the functions, in this case — and then implement them in the most convenient order. But before we continue our implementation, we need to check if the `positive_root` function is working.

4.2 Unit testing

Before we assemble our functions into the final program, we should test each one individually. The reason for this is that it makes it much easier to identify and fix any mistakes (the technical term is *bugs*). In this example, if we had a bug in the `positive_root` function that resulted in an incorrect value being returned, we would get a wrong pH value at the end. But a wrong pH value would not tell us which function had the error.

The best practice is to write tests for each individual function in a test script so that you can easily test each of your functions whenever you need. This is important because programs generally have to be maintained, changed or improved during their lifetime, and tests should be run after each change. However, for now we'll just do the tests manually. We'll see how to set up a tests script later.

To test our function and make sure it is working, we have to run the script again (F5 in Spyder) so the changes to the file are saved and the function objects are created again and then we can test the function in the console, for example, on equations $x^2 - 1 = 0$ and $x^2 - 2x = 0$, for which the roots are easy to check. The results should be, respectively, 1 and 2:

```
In : positive_root(1,0,-1)
Out: 1.0

In : positive_root(1,-2,0)
Out: 2.0
```

Since this function seems to be working, we implement the `compute_pH` function. For this function we will need the `log10` function from the `numpy` library. It is good practice to put all `import` statements in the beginning of each module, just after the documentation for the module, which is a docstring similar to that used for the functions. So this is how our `phconc.py` module should look like:

```
# -*- coding: utf-8 -*-
"""
Computes pH of an acid from concentration
"""
from numpy import log10

def positive_root(a, b, c):
    """
    return positive root of quadratic equation
    """
    root = (-b + (b**2 - 4*a*c)**0.5)/(2*a)
    return root

def compute_pH(Ka, Ci):
    """
    return pH value for weak acid from
    dissociation constant and initial concentration
    """
    H = positive_root(1, Ka, -Ka*Ci)
    return -log10(H)
```

Note that, in this case, we did not create a `result` variable with the final pH value. Rather, we simply return the symmetric (with the minus sign) of the logarithm of the positive root `H`, corresponding to $[H^+]$. This works as well because the `log10` function returns the value of the logarithm, the minus sign changes it from negative to positive and it is this final value that is returned from the function.

To save changes and recreate the function objects we run the script again (by pressing F5 in Spyder) and then we can do the unit tests for the `compute_pH` function. We can use the values for the benzoic acid example, since we know the result should be a pH of around 3.1:

```
In : compute_pH(6.5e-5,0.01)
Out: 3.11104555393015
```

It may be useful to automate testing, because when we find errors we may need to test again until everything works and sometimes we need to change something in our implementation and it is a good idea to always test after any changes. One way to do this is to add a function to the end of your module that you can use to run all tests:

```
def tests():
    """
    run unit tests on all functions
    """
    print("positive_root, 1.0:", positive_root(1,0,-1))
    print("positive_root, 2.0:",positive_root(1,-2,0))
    print("compute_pH, 3.111:",compute_pH(6.5e-5,0.01))
```

Each test prints out the correct result and the result obtained from the tested function.

Important: note that all functions in your program are objects created by the interpreter when it reads and executes your `def` declarations. That is when the interpreter "understands" the recipes in your function. Because of this, whenever you change any function in your program you must run it again (or at least that function declaration) in order to recreate the function object with the changes in effect. The simplest way to do this in Spyder is just to press F5.

Also note that the `def` declaration only tells the interpreter to create the function. It learns the "recipe", so to speak, but does not execute it. The interpreter will only execute the code in the function when you call your function using its name and parentheses, such as with `tests()`, for example.

Because of this, if your script only has function declarations, it only creates the function objects but does nothing with them. You can have your script automatically run the functions by adding the function calls outside the body of any function. For example, this code below would run all tests automatically whenever you press F5. Please note that this may be useful during development but may not be a good idea for the final version of the script, because it will always run all tests whenever anyone uses the script.

```
def tests():
    """
    run unit tests on all functions
    """
    print("positive_root, 1.0:", positive_root(1,0,-1))
    print("positive_root, 2.0:",positive_root(1,-2,0))
    print("compute_pH, 3.111:",compute_pH(6.5e-5,0.01))

tests()
```

An alternative way of automating your tests is to create a test module. For example, `phtests.ph`, which imports all functions you need to test and runs the tests. Note that, since this module is only used for testing, we do not need to put the tests in a function. This way they are always executed whenever the module is executed.

```
# -*- coding: utf-8 -*-
"""
run unit tests on all functions in phcalc
"""
from phcalc import positive_root, compute_pH, compute_pH_mass

print("positive_root, 1.0:", positive_root(1,0,-1))
print("positive_root, 2.0:",positive_root(1,-2,0))
print("compute_pH, 3.111:",compute_pH(6.5e-5,0.01))
```

4.3 Errors

Making mistakes is an inevitable part of programming. Fortunately, the interpreter gives us useful information about our mistakes and if we make sure to carefully test each part of our program, errors will generally be easy to solve.

The easiest error to fix is the *syntax error*. This is any error that prevents the interpreter from parsing the source code. The reason these are the simplest to fix is because the interpret identifies the error even before running the program shows exactly where the error occurs. Here are two examples. Note the error messages the interpreter gives us.

```
In : 6 * * 2

    6 * * 2
        ^
```

```
SyntaxError: invalid syntax

In : 8 + 5.3.2

    8 + 5.3.2
            ^
SyntaxError: invalid syntax
```

Another kind of error is the *exception*. This error occurs when the interpreter understands the commands we give it but is unable to execute them for some reason. These errors may not be trivial to fix because the mistake may actually occur somewhere else. Here is an

```
In : y = z*2
Traceback (most recent call last):

    y = z*2

NameError: name 'z' is not defined

In : y = 0
In : z = 1 / y
Traceback (most recent call last):

    z = 1 / y

ZeroDivisionError: division by zero
```

In the first example, the interpreter raised a `NameError` exception because the name `z` is not defined. Note that this may be because of some mistake in that instruction or it may be due to some error in another part of our code where we did not create the `z` variable. The second example also illustrates this problem. We set variable `y` to zero and then try to place `1/y` in variable `z`. The error here may be either in the value assigned to `y` or in the expression for `z`. When a program causes an exception to be raised, we generally have to think about where the error may really be occurring. This makes such errors harder to correct.

Perhaps the worst errors are the *logical errors*. These are the hardest to diagnose and fix. A logical error occurs when the program works without the interpreter raising any exception or syntax error but the result is not the result that we wanted.

Suppose we have this program to compute the pH of a weak acid solution:

```python
# -*- coding: utf-8 -*-
"""
Computes pH of an acid from concentration
"""
from numpy import log10

def positive_root(a, b, c):
    """
    return positive root of quadratic equation
    """
    root = (-b + (b**2 - 4*a*c)**0.5)/(2*a)
    return root

```

```
14 def compute_pH(Ka, Ci):
15     """
16     return pH value for weak acid from
17     dissociation constant and initial concentration
18     """
19     H = positive_root(1, Ka, -Ka*Ci)
20     return log10(H)
```

We test our `compute_pH` function and get this result:

```
compute_pH(6.5e-5,0.01)
Out[12]: -3.11104555393015
```

It is evident that something went wrong because we should not get a negative pH value. The error is in line 20, where we forgot the minus sign before the call to the `log10` function. This is one reason why careful unit testing is so important. We need to test or functions thoroughly to make sure there are no logical errors, and we should test each function individually to make it easier to figure out where the error is occurring when something goes wrong.

Finally, there are numerical errors. These are a special type of logical errors that arise due to the numbers being represented with finite precision. The representation of *floating point* numbers in Python uses a total of 64 bits, and is split in three parts, according to the IEEE-754 standard. Figure **??** illustrates this. The first bit specifies the sign of the number (positive or negative), followed by 11 bits specifying an exponent (of base 2) and 52 bits for the significand. The numeric value is thus represented as:

$$value = significand \times 2^{exponent}$$

The term *floating point* comes from the fact that the radix point (the decimal point in decimal numbers) can "float" anywhere depending on the value of the exponent.

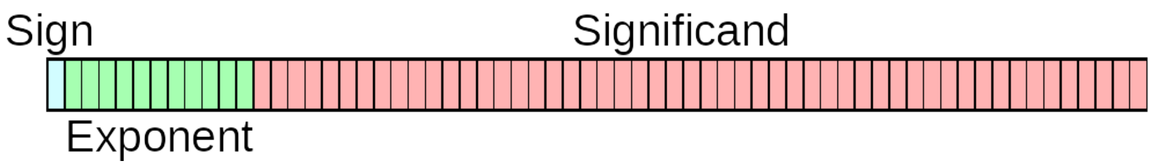


Figure 4.1: Floating point representation of numbers in Python. Image source: Wikipedia.

Because the significand has only 52 bits, there are rounding errors in the representations, which can only have a maximum of about 17 decimal significant digits. This can give rise to results such as

```
In : (2**0.5)**2
Out: 2.0000000000000004
```

The square root of two must be rounded to the limited precision of 17 significant digits, and so when we raise it to two the result will not be exactly 2. Numerical errors can be difficult to solve but, fortunately, for most applications the precision limit of these floating point numbers is sufficient for these errors not to be relevant. The `numpy` library provides information on the limits of different numerical representations, with the `finfo` function. Here are some values for the standard Python floating point numbers.

```
In : import numpy
In : info = numpy.finfo(float)
In : info.bits
Out: 64
In : info.eps
Out: 2.2204460492503131e-16
In : info.tiny
Out: 2.2250738585072014e-308
In : info.min
Out: -1.7976931348623157e+308
In : info.max
Out: 1.7976931348623157e+308
```

The `bits` value is the number of bits used for this representation. The value `eps` is the smallest value that we can add to 1 and get a representation different from 1; `tiny` is the smallest positive value that can be represented with full precision, with smaller values decreasing the number of significant digits; `min` and `max` are, respectively, the smallest (most negative) and largest values we can represent with a floating point number.

4.4 The life cycle of a computer program

After we understand the problem, think of the algorithm and decompose it as needed in sub-problems, we start implementation by writing our source code file or files. This begins the life cycle of our program. We edit the source code, then we have the interpreter translate our source code as we can see from the problem of testing and fixing errors, this is not the end of the process.

4.5 Reusing code

One important advantage of structured programming is making it easier to reuse subprograms to solve new problems. To illustrate this, we will consider the problem of computing the pH of a solution of a weak acid without knowing the concentration. We will start from the mass of the solute, its molar mass, and the volume of the solution, in addition to the dissociation constant. Instead of rewriting everything to solve this task, we will simply add to our original script a new function that computes the concentration and then uses the `compute_pH` function to get the result:

```
def compute_pH_mass(mass, mol_mass, volume, Ka):
    """
    return pH value of a weak acid solution from
    mass of solute and volume of solution
    """
    Ci = mass / mol_mass / volume
    return compute_pH(Ka,Ci)
```

And now we test it. Knowing that the molar mass of benzoic acid is 122.1g/mol, we check if we get the same pH value for the same concentration of our previous tests, and can try other values too, such as half a gram of benzoic acid in 250ml of solution.

```
In : help(compute_pH_mass)
Help on function compute_pH_mass in module __main__:
```

```
compute_pH_mass(mass, mol_mass, volume, Ka)
    return pH value of a weak acid solution from
    mass of solute and volume of solution

compute_pH_mass(1.221, 122.1, 1, 6.5e-5)
Out: 3.11104555393015

In : compute_pH_mass(0.5, 122.1, 0.25, 6.5e-5)
Out: 3.000062878669473
```

We can include these tests in our test function:

```
def tests():
    """
    run unit tests on all functions
    """
    print("positive_root, 1.0:", positive_root(1,0,-1))
    print("positive_root, 2.0:",positive_root(1,-2,0))
    print("compute_pH, 3.111:",compute_pH(6.5e-5,0.01))
    print("compute_pH_mass, 3.1111",compute_pH_mass(1.221, 122.1, 1, 6.5e-5))
    print("compute_pH_mass, 3.000",compute_pH_mass(0.5, 122.1, 0.25, 6.5e-5))
```

4.6 Coding Style

Source code is not meant only for the interpreter to run. It must also be readable by humans. Poorly written source code makes it more likely to make mistakes, harder to correct and adapt and harder to understand. We should think of source code as also as a detailed description of the algorithms it implements, and a description should be easy to understand.

In order to write readable code, take care to choose names carefully. Variable names should make it clear what each variable is. This means choosing descriptive names or names that agree with some previous convention. Function names should also be descriptive and make it clear what each function does.

Finally, each line of code should be simple and easy to understand. Avoid writing complex expressions in a single line. It's safer to break them down in smaller steps.

4.7 Exercises for tutorial 2

These are the exercises for the second tutorial class. Download to your working folder the files `tutorial2.py` and `t2_tests.py`, open them in Spyder and run the file `t2_tests.py`. This is a script with test functions that checks some results of the functions you will implement. Note that this testing script uses some Python functions and constructs that we have not covered yet, but don't worry. All you need to do is run the script and check the console for the test results. Since the `tutorial2.py` only has an incomplete function, this is what you should see on the console when you run `t2_tests.py`

```
In: runfile('/.../t2_tests.py', wdir='/...')
fib_direct(0,)  Incorrect result: None
fib_direct(1,)  Incorrect result: None
fib_direct(2,)  Incorrect result: None
fib_direct(3,)  Incorrect result: None
fib_direct(8,)  Incorrect result: None
cel2fahr not found
fahr2cel not found
```

Note that you need to implement your functions with the correct names for these tests to work. Also note that the `fib_direct` function is not yet returning any values, and this is why the `None` appears on the tests. To help you start the implementation, the `tutorial2.py` already has a “(”skeleton) of the functions you need to implement.

1. Recall the formula for computing each Fibonacci number F_n in Chapter 2:

$$\phi = \frac{1+\sqrt{5}}{2} \qquad \psi = \frac{1-\sqrt{5}}{2} \qquad F_n = \frac{\phi^n - \psi^n}{\sqrt{5}}$$

   Implement the function `fib_direct(n)` that uses this formula to return the value of F_n as a function of n. Run the tests script to test your function. Note that all tests for this function are OK except for $n = 8$. Run your function with $n = 8$ and explain why the value your function returns is not exactly 21.

   Improve your function using the `round` function. Use the `(`help`)` function to see how the `round` function works. Afterwards, test your function again.

2. Implement a function named `cel2fahr` with a single parameter, which is the temperature in Celsius degree, and returns the temperature in Fahrenheit, using the conversion $F = 1.8C + 32$. Note that you need to implement your function as specified, receiving a single argument, for the tests to run correctly.

3. Implement and test the function for converting Fahrenheit to Celsius (`fahr2cel`).

Problems

Recall the steps to solving problems with programming: understand the problem and what you want; formulate the algorithm; break down complex tasks into subproblems, repeating the process for each subproblem; think about the subprograms — the functions — you need to implement and how they can be generalized; after you have outlined the program, implement each function. Use this approach for each of the problems below. For each of these programs, create a new source code file.

4. Create a program to compute the pH value of a weak acid starting from the concentration of the acid. You can follow the example in this chapter and the previous one, but try to do as much as possible for yourself. Remember to import the `log10` function from the numpy library. Also, create a function `tests()` that runs the tests shown in this chapter and prints out the results. This way you can test your functions by executing `tests()` function in this module, and keep adding new tests to this function as you implement the others. Remember to import the `log10` function from `numpy`, since you will be needing it to compute the base 10 logarithm.

   Expand your program to include the computation starting from the mass of the solute, the volume and the molar mass, as shown before.

5. Given the coordinates of 3 points — (x_1, y_1), (x_2, y_2) and (x_3, y_3) — we can compute the area of the triangle from the formula:

$$A = \sqrt{s(s-a)(s-b)(s-c)} \qquad S = \frac{a+b+c}{2}$$

where a, b and c are the distances between points 1 and 2, 2 and 3 and 3 and 1 (the sides of the triangle). These distances can be computed from the formula:

$$D_{i,j} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

Remember to think on the decomposition of this problem before implementing your functions. Also think about generalizing your functions. You need to compute three distances between atom pairs. Do you need three functions for this or will one function be sufficient?

6. Zircon crystals ($ZrSiO_4$) readily incorporate uranium atoms but not lead atoms. So one reasonable approximation is to assume that all lead atoms in zircon come from the radioactive decay of uranium atoms that were previously incorporated in the crystal.

   The isotope ^{238}U decays into ^{206}Pb with a half-life of 4470 million years. The half-life is the time it takes for half the isotopes in the original population to decay. The isotope ^{235}U decays into ^{207}Pb with a half-life of 704 million years. Assuming that all the lead atoms in a zircon crystal come from the decay of uranium atoms, we can compute the age of the crystal from each of these nuclear reactions using the formula:

$$\lambda = \frac{ln(2)}{t_{1/2}} \qquad P = R(e^{\lambda t} - 1) \Leftrightarrow t = \frac{ln(\frac{P}{R} + 1)}{\lambda}$$

   where λ is the decay constant, computed from the half-life time; $t_{1/2}$ is the half-life time; R is the present quantity of the original reactant isotope (either ^{238}U or ^{235}U); P is the present quantity of the product isotope (either ^{206}Pb or ^{207}Pb) and t is the age of the sample.

   Since lead may leach from the crystal, there may be some discrepancy between the ages computed using different isotopes. The relative discrepancy can be computed as the absolute value of the difference between the ages divided by the average value of the ages. Write a program to compute the ages and the relative discrepancy given the concentrations of all four isotopes.

   Remember to try to understand the problem in detail first, then outline the algorithm, break it down in smaller tasks, and generalize them. For example, do you need two functions to compute the age of a sample?

   Note that you need to compute the natural (base e) logarithm for this program. To do this, you can import the `log` function from the `numpy` library. You can also use the `abs` function to compute the absolute value of a number.

   After implementing your program, compute the age from each reaction and the discrepancy from a zircon crystal with the following isotope concentrations:

   [^{235}U] = 0.9ppm, [^{207}Pb] = 0.6 ppm

   [^{238}U] = 2.4ppm, [^{206}Pb] = 0.2 ppm

   Your result should be a discordance of about 0.0051 and ages of 518.82 and 516.18 million years.

7. Two types of seismic waves travel at different speeds. Primary waves are longitudinal compression waves that travel faster than Secondary waves, which are transverse waves. The propagation speed depends on the material the waves travel through, but, using adequate parameters, we can approximate the calculus of the distance between a detector and the epicenter as:

$$D = \Delta_t \frac{V_P V_S}{V_P - V_S}$$

   where D is the distance to the epicenter, Δ_t is the time difference between the arrival of the P and S waves, and V_P and V_S are the velocities of primary and secondary waves.

   From the measured amplitude of the waves and the distance to the epicenter, we can compute the magnitude of the seismic event using Lillie's empirical formula:

$$M = log_{10}A + 1.6log_{10}D - 0.15$$

   where M is the magnitude in the Richter scale, A the measured amplitude in mm and D the distance to the epicenter in km.

   Write a program that computes the magnitude of a seismic event from the time difference and measured amplitude of the P and S waves. Assume that $V_P = 5.5km/s$ and $V_S = 3.5km/s$

   Test your program with the standard Richter event, which is a magnitude 0 event with a wave amplitude of 0.001mm and distance of 100km. Note that, to do this, you need to compute the arrival time difference of the P and V waves after traveling 100km. Note that you should get a magnitude of approximately 0 and not exactly 0.

   Compute the magnitude and distance of a seismic event if the delay between P and S waves is 12 seconds and the measured amplitude is 8mm. (The answer should be 4.05 and 115.5 km)

4.8 Further reading

The same as for the previous chapter: Chapter 16 up to page 479 (Calls) and Chapter 17 up to page 491 (The Built-in Scope) of the textbook [1].

Chapter 5

Arrays

Lists and tuples. Creating and using lists. Indexing and slicing of lists and strings

5.1 Lists, tuples and strings

Lists and tuples are objects that store an ordered array of other objects. The main difference between lists and tuples is that lists are *mutable*, meaning that we can remove, add or replace objects in the array, whereas tuples are *immutable*, because after we create them we cannot change them. Both of these arrays can be created by extension with their constituent objects separated by commas, between square brackets for lists and between parentheses for tuples. Here are two examples:

```
years = [1999,2000,2001]              # list
coordinates = (23.4, 12.6, 13.5)      # tuple
```

Though lists can contain heterogeneous data, with values of different types, they are generally used to store ordered sets of homogeneous values, all with the same meaning. In this example, an ordered sequence of years. The syntax for creating a list by extension is to write the values separated by commas and enclosing everything in square brackets.

Tuples are often used when the position of each value in the tuple has a different meaning. In this example, the values correspond to the X, Y and Z coordinates, and so it matters where each value is in the tuple. This, however, is mostly conventional. It is also possible to use lists for these purposes.

Another array object we have met previously are the strings. Strings are ordered arrays of characters and are immutable like tuples, meaning you cannot change part of the string.

5.2 Indexing and slicing arrays

For all these arrays, individual elements can be accessed by their indexes in the array, counting the first element as having index 0. Here are some examples

```
In : years = [1999,2000,2001]
In : coordinates = (23.4, 12.6, 13.5)
In : text = 'once upon a time'

In : text[0]
```

```
Out: 'o'

In : text[1]
Out: 'n'

In : years[2]
Out: 2001

In : coordinates[1]
Out: 12.6

In : text[-2]
Out: 'm'
```

It is also possible to index arrays counting from the end, using negative indexes, with -1 referring the last element. The left panel of the scheme belos illustrates the indexes for a string of characters, but the principle is the same for lists, tuples and other arrays:

We can also slice parts of arrays. A slice is an expression of the form `start:end:step`, where the start, end and step are integers defining the slice. Slice cutting points are indexed in a similar way to indexes but we should think of the slice points as placed between the elements:

```
 +---+---+---+---+---+---+        +---+---+---+---+---+---+
 | P | y | t | h | o | n |        | P | y | t | h | o | n |
 +---+---+---+---+---+---+        +---+---+---+---+---+---+
   0   1   2   3   4   5          0   1   2   3   4   5   6
  -6  -5  -4  -3  -2  -1        -6  -5  -4  -3  -2  -1
```

Here are a few examples of slicing a list of numbers. Note that the beginning, end or step of the slice can be omitted. If the beginning is omitted, it is assumed to be zero. If the end is omitted, it is assumed to be the end of the array. And if the step is omitted it is assume to be 1. If we omit all, using a slice of `[:]`, then this slices the full array.

```
In : numbers=[0, 1, 2, 3, 4, 5, 6]
In : numbers[1:3]
Out: [1, 2]

In : numbers[2:]
Out: [2, 3, 4, 5, 6]

In : numbers[::3]
Out: [0, 3, 6]

In : numbers[-2:]
Out: [5, 6]

In : numbers[1:3] = [-2,-4]
In : numbers
Out: [0, -2, -4, 3, 4, 5, 6]
```

The first example shows the result of asking for a [1:3] *slice* of the list. Referring to the scheme above, we see that the cuts lie between the elements. Thus, slicing from position 1 means starting between the second element (index of 1) and the third element (index of 2) and slicing to position 3 means ending between the third and fourth elements. Thus the result is `[1,2]`. If an index is missing from the slice, then the interpreter assumes the missing index is the extreme position of the list. Thus, in the second example, slicing cuts at position two (between the second and third elements) and returns the list from there to the end. An additional colon can be used to specify the step, and in example 5 the slice returned consists of all elements from the beginning to the end of the list (no indexes are provided) but with a step of 3. We can also use negative values to indicate index positions counting from the end. In the case of lists (but not tuples or strings) we can use slicing or indexing to assign new values to elements in the list.

5.3 Unpacking

Python allows *unpacking* values from arrays into the same number of objects. This was illustrated in the last example with slicing above, but can be generalized to different cases. The code below shows some examples.

```
In : numbers=[0, 1, 2, 3, 4, 5, 6]
In : numbers[1:3] = 'xx'
In : numbers
Out: [0, 'x', 'x', 3, 4, 5, 6]

In : a, b = numbers[3:5]
In : a
Out: 3

In : b
Out: 4

In : a, b, c = (1,'xxx',[1,3,4])
In : a
Out: 1

In : b
Out: 'xxx'

In : c
Out: [1, 3, 4]
```

When we assign the string `'xx'` to two elements of the `numbers`, each character is assigned to one of the elements of the list. We can also assign elements of an array to the same number of variables.

5.4 Unpacking arguments

EXTRA

Python has an unpacking operator, $*$, that can be used in the function arguments to unpack an array into different arguments. This allows us to use an array with the same number of elements as the arguments it will substitute instead of those arguments. The example below illustrates this. If we use the tuple `(6,9)` as an argument for the `add` function we get an error because the tuple is only one argument and the function requires two arguments, since it has two parameters (`a` and `b`).

But preceding the tuple with the ∗ operator will indicate that we want the array to be unpacked into the different arguments.

```
def add(a,b):
    result = a+b
    return result
```

```
In : add( (6,9) )

    add( (6,9) )
TypeError: add() missing 1 required positional argument: 'b'

In : add( *(6,9) )
Out: 15
```

5.5 Array methods

Another useful feature of lists are the methods associated with list objects, which we can use to add elements to the list, find elements in the list, remove elements and so on.. Below are some examples of useful list methods. Note that these list methods change the list, unlike the string methods we saw before which do not change the string but return a new string.

```
In : numbers.append(20)
In : numbers
Out: [0, 1, 2, 3, 4, 5, 6, 20]

In : numbers.extend([1,2,3])
In : numbers
Out: [0, 1, 2, 3, 4, 5, 6, 20, 1, 2, 3]

In : numbers.index(20)
In : Out: 7

In : numbers.remove(4)
In : numbers.remove(5)
In : numbers
Out: [0, 1, 2, 3, 6, 20, 1, 2, 3]

In : numbers.reverse()
In : numbers
Out: [3, 2, 1, 20, 6, 3, 2, 1, 0]

In : del(numbers[3:6])
In : numbers
Out: [3, 2, 1, 2, 1, 0]
```

The first example shows how to append an element to the list and the second example shows how to extend the list with another list of elements, also at the end. The `index` method returns the index of the value given in the argument and the `remove` method removes the first instance of the given value. The `reverse` method reverses the order of the elements on the list and `del` can be used to delete one or more elements from the list. Note that these methods change the actual contents of the list, which is a mutable object in python.

This contrasts with string objects. Strings are ordered arrays of characters and can be sliced like lists. However, strings in Python are immutable, meaning that their contents cannot be changed. Note that you can assign a different value to the variable, but this will simply make the variable refer to that value and not change the original. Here are some examples.

```
In : name = 'Python'
In : name[2:5]
Out: 'tho'

In : name.index('h')
Out: 3

In : name[3]='x'

TypeError: 'str' object does not support item assignment

In : name.upper()
Out: 'PYTHON'

In : name.split('t')
Out: ['Py', 'hon']
```

The first examples show slicing and finding the index of an element of a string. But trying to assign a value to a character in the string results in an error, because strings cannot be altered. So string methods return values instead of changing the string. For example, when calling `name.upper()`, the methor returns the upper-case characters of the string in `name` but does not change the string. The same goes for the `split` method. This method returns a list of substrings resulting from the split of the main string wherever the substring used for splitting was found.

Tuples are similar to lists and strings. Like lists, tuples can be used to store heterogeneous data. However, like strings, tuples are immutable. We cannot assign values to elements of a tuple except when creating the tuple. The advantage of this is that strings and tuples, like integer and float numbers, can be used as keys in dictionaries (though it is not wise to use float numbers as keys due to rounding errors). We shall see some examples of these data structures and applications later in the course.

5.6 Mutable objects and function arguments

When we create a variable with a new object, Python creates the object in memory and makes the variable name refer to that object. But if we create a variable from a pre-existing variable, then both variables will refer the same object. If the object is immutable, such as a number or a string, this will not cause any problem. There is no problem in having only one object representing the number `2` or the string `'abc'` because these objects cannot be change. But when we create mutable objects we need to take care with this. Consider the example below:

```
In : a = [1,2,3]
In : b = a
In : b.append(4)
In : b
Out: [1, 2, 3, 4]
In : a
Out: [1, 2, 3, 4]
```

The first line creates a new list in memory, containing the elements 1, 2 and 3, and makes the name a refer to that list. In other words, we create a variable with the new object. The second line creates variable b and assigns it to the same object as a. This list already exists in memory, so Python does not create a new one. It merely copies the reference of variable a so that b will refer the same object. When we append the number 4 to the list in variable b, since the list referred to by variable b is the same as that referred to by a, this will also change the contents of variable a.

This is also important for functions. Function arguments are passed to functions by object reference. This means that the parameter in the function will refer the same object in memory, just like when we copy the reference of a variable. But it is still an independent reference. Thus, if the function receives an object, such as a list, and changes the contents of the object — for example, by appending an element — the object will change and the change will be visible outside the function. However, assigning new values to the variable will not change anything outside the function because this is a different, independent, variable. The code below illustrates this. Note that the append instruction in line 2 changes the elements list, because the a_list variable refers the same object. However, assigning a new object to a_list, such as an empty string, has no impact on the object elements refers to.

```
1 def add_to_list(a_list, a_number):
2     a_list.append(a_number)
3     a_list = []
```

```
In : elements = [1,2,3]
In : add_to_list(elements, 22)
In : elements
Out: [1, 2, 3, 22]
```

5.7 Example: restriction enzymes

Bsp1407I is a type II restriction enzyme that cleaves DNA on the TGTACA sites, cleaving between the first T and the first G, on both strands. We will see how we can use Python to find the Bsp1407I cleavage sites and the fragment lengths given the $5' \to 3'$ DNA sequence. We will start with our sequence in the sequence variable and the restriction site in the site variable. Now we can use the split method in the sequence variable to break the sequence into a list of strings, which we will store in the fragments variable. Note that the sequence is broken in two lines. The
character at the end of the line tells the interpreter to consider the next line as a continuation of the current line.

```
sequence = 'GATCCTCCATATACAACGGTATCTGTACATCCACCTCAGGTTT'+\
           'AGATCTCAACAATGTACACGGAACCATTGCCGACATG'
recon= 'TGTACA'
fragments = sequence.split(recon)
print(fragments)
```

If we run this code, we will see the output of the print command on the console:

```
['GATCCTCCATATACAACGGTATC',
 'TCCACCTCAGGTTTAGATCTCAACAA',
 'CGGAACCATTGCCGACATG']
```

This is close to what we want, but it is not quite the right answer yet. The `split` method separates a string into a list of strings wherever the separator pattern is found, but it also discards the separator. This means that we are losing the recognition sequence, with all instances of `TGTACA` being discarded. We can solve this problem using the `replace` method to replace the recognition sequence with a sequence including a special character, `-`, identifying the cleavage point.

```
sequence = 'GATCCTCCATATACAACGGTATCTGTACATCCACCTCAGGTTT'+\
           'AGATCTCAACAATGTACACGGAACCATTGCCGACATG'
recon =  'TGTACA'
cut = 'T-GTACA'
marked = sequence.replace(recon,cut)
fragments = marked.split('-')
print(fragments)
```

This way, we can cut the sequence only on the correct cutting points, which are marked by inserting the `'-'` character in the cutting point wherever the recognition sequence is found. Running this program, we can see the correct fragments:

```
['GATCCTCCATATACAACGGTATCT',
'GTACATCCACCTCAGGTTTAGATCTCAACAAT',
'GTACACGGAACCATTGCCGACATG']
```

Now that we experimented with the algorithm and understand how to implement it, we abstract from the details and can write a more general function to give us the fragments obtained by digestion with an enzyme. The function receives the sequence, a string with the recognition sites and a string with the same sequence as the recognition site but with the cutting point marked with a `-` character.

```
# -*- coding: utf-8 -*-
"""
Simulates DNA digestion by enzymes
"""

def digest_sequence(sequence,recon,cut):
    """Return a list of fragments from cutting the sequence"""
    marked = sequence.replace(recon,cut)
    frags = marked.split('-')
    return frags

recon = 'TGTACA'
cut = 'T-GTACA'
sequence = 'GATCCTCCATATACAACGGTATCTGTACATCCACCTCAGGTTT'+\
           'AGATCTCAACAATGTACACGGAACCATTGCCGACATG'

fragments = digest_sequence(sequence, recon, cut)
```

Now we can obtain the length of each fragment by asking for the length of each element of the `fragments` variable. This is done by indexing each element of the list, in which the first element has the index of 0. To index an array in python we use square brackets, and the `len` function gives us the length of an array or a string. Thus:

```
In : len(fragments[0])
Out: 24

In : len(fragments[1])
Out: 32

In : len(fragments[2])
Out: 24
```

This works, but is not very practical since we need to select each fragment manually to obtain its length. It would be useful to repeat the `len()` instruction for each of the fragments. This we can do with a `for` loop, which we will cover in the next chapter.

5.8 Further reading

Chapter 7, Chapter 8 up to page 250 (Dictionaries) and Chapter 9 up to page 282 (files) of the textbook [1].

Chapter 6

Iteration

The for loop. Iterating over lists. Creating lists with loops: `append` *and* `extend`*. Processing strings and lists of strings.*

6.1 The `for` loop.

In some cases, we may want to repeat some set of instructions a number of times. In Python, we can iterate through an array of values by using the `for var in array:` expression, where `var` is any variable and `array` any iterable array, one that can provide one element at a time, such as a list, tuple or or string. Instructions inside the `for` loop will be executed once for each element in `array` and, for that execution, the `var` variable will hold that element. For example, this code will add in variable `total` all the values in the array `vals`:

```
total = 0
vals = [3,5,7,1,2]
for v in vals:
    total = total + v
print("Sum:",total)
```

If we run this program, it will output `Sum: 18` on the console. Note that the line `total = total + v` is indented relative to the line `total = total + v`. This tells the interpreter that this line belongs inside the `for` loop and will be repeated for each iteration of the loop. The line texttprint("Sum:",total) is indented at the same level as the line initiating the for loop, which tells the interpreter that this line does not belong to the loop. So the only line executed for each step of the loop is the line adding the value to the total.

6.2 Iterables and Iterators

EXTRA

The `for` instruction can be used on any iterable object. An iterable object in Python is an object that implements an `__iter__` method that returns an iterator. And an iterator is an object that implements a `__next__` method that returns one element at a time until no elements are left, after which it raises an `StopIteration`. Here is an example, using a list.

```
In : things = ['larch',42, 3.14153]
In : iterator = things.__iter__()
In : iterator.__next__()
Out: 'larch'

In : iterator.__next__()
Out: 42

In : iterator.__next__()
Out: 3.14153

iterator.__next__()
    iterator.__next__()
StopIteration
```

We can think of a `for` loop as first asking the iterable object for its iterator. It then proceeds by asking the iterator for the next element and executing all the code inside the loop, and keeps doing this until the `StopIteration` exception is raised. In each pass, the element returned by the `__next__` method of the iterator is stored in the variable we provide in the `for` instruction. Here is an example with the same list:

```
In : for one_thing in things:
         print(one_thing)

larch
42
3.14153
```

Note the `:` at the end of the `for` line and the indentation of the block inside the loop.

6.3 Restriction enzymes, with a loop

In the previous chapter we created this program to simulate the action of a restriction enzyme:

```
# -*- coding: utf-8 -*-
"""
Simulates DNA digestion by enzymes
"""

def digest_sequence(sequence,recon,cut):
    """Return a list of fragments from cutting the sequence"""
    marked = sequence.replace(recon,cut)
    frags = marked.split('-')
    return frags

recon = 'TGTACA'
cut = 'T-GTACA'
sequence = 'GATCCTCCATATACAACGGTATCTGTACATCCACCTCAGGTTT'+\
           'AGATCTCAACAATGTACACGGAACCATTGCCGACATG'

fragments = digest_sequence(sequence, recon, cut)
```

To print out the length of each fragment, we had to index each fragment individually. But now that we now how to write a `for` loop we can do this:

```
In : for frag in fragments:
  ...    print(len(frag))
  ...
24
32
24
```

Note that when we write a block of code with multiple lines in the console, as this `for` loop, the console will automatically include the indentation and input all code lines until we enter an empty line. Then it executes the whole block.

6.4 The `range` function

Suppose we wanted to indicate the index of each fragment as well as the length. Something like this:

```
0 24
1 32
2 24
```

meaning that the fragment of index 0 has a length of 24 nucleotides, the fragment of index 1 a length of 32 nucleotides, and so on. This is a common problem, having to know the index of the element we are working with, and writing the loop as we did before does not allow us to do this. This is where we can use the `range` function[1]. This function accepts at least one argument indicating the number of elements to return and then returns an iterable object that provides the numbers 0, 1, 2, and so on, as many as indicated in the range function. Here is an example:

```
In : for ix in range(5):
         print(ix)

0
1
2
3
4
```

```
In : for ix in range(6,18,3):
         print(ix)
6
9
12
15
```

Calling `range(5)` returns an iterable object that will output 5 numbers, starting from 0 (and, thus, ending in 4). This is useful for iterating through the indexes of any array. We can also use `range` with two arguments specifying the starting value and the upper limit. With a third argument we can also specify a step for the values. Note that `range` always stops before reaching the upper limit. To iterate over the indexes of an array all we need is to give the `range` function the length of the array. Here is a solution to the problem of writing the index and length of each fragment:

```
In : for ix in range(len(fragments)):
         print(ix,len(fragments[ix]))

0 24
1 32
2 24
```

[1]Strictly speaking, `range` is not a function but a class for objects of type `range`. However, for our purposes, we can think of it as just another function

Note that the `ix` variable will take values from 0 through 2 and then we use this variable to index the `fragments` array. Also note that the argument for the `range` function is the length of the `fragments` array but in the `print` function we use the length of each of the elements of the `fragments` array (*i.e.* each individual fragment).

6.5 The `enumerate` function

EXTRA

Although not strictly necessary, the `enumerate` function can be convenient for cases like this. This function receives an iterable object as an argument returns an iterator that output pairs of (index, element). Here is an example:

```
In : things = ['larch',42, 3.14153]
In : for x in enumerate(things):
         print(x)

(0, 'larch')
(1, 42)
(2, 3.14153)
```

This is especially useful if we unpack each tuple returned by the `enumerate` iterator:

```
In : things = ['larch',42, 3.14153]
In : for ix,thing in enumerate(things):
         print('Thing', ix, 'is', thing)

Thing 0 is larch
Thing 1 is 42
Thing 2 is 3.14153
```

6.6 Example: average and geometric means

To help understand the mechanics of the `for` loop, let us start with the simple example of computing the average of an array of numeric values:

```
1 def mean(values):
2     """
3     return the average of an array of values
4     """
5     total = 0
6     for val in values:
7         total = total + val
8     return total/len(values)
```

This function starts by setting the `total` variable at zero, because we will be using this variable to keep track of the sum of the values in the array. This is done in line 7, inside the loop, adding the current total to the value of the current element and then storing the result once more in the same variable. Line 8, outside the body of the `for` loop, is executed only after the loop ends. Note the indentation. The value returned is the sum of the elements divided by the number of the elements (the length of the `values` array). As always, we should test the function:

```
In : average([1,2,3,4,5])
Out: 3.0
```

Here is another example, the geometric mean. The geometric mean is the n root of the product of the n values:

$$(\prod_{i=1}^{n} x_i)^{\frac{1}{n}}$$

The function is similar to the one above:

```
def geometric_mean(values):
    """
    return the geometric mean of an array of values
    """
    product = 1
    for val in values:
        product = product * val
    return product**(1/len(values))
```

Note that, in this case, the variable `product` starts at 1 and, in each iteration, we keep track of the current product multiplying `product` by the current value. The value returned, after the loop terminates (note the indentation) is the product raised to 1 over the number of elements in the array. Again, we should test our function:

```
In : geometric_mean([1,2,3,4,5])
Out: 2.605171084697352
```

6.7 Example: basic statistics

Suppose we have an array of contamination values from a set of samples and want to describe the distribution of values with two statistics, the mean and standard deviation. The standard deviation of a sample of values is:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \mu)^2}{n}}$$

where μ is the mean of the sample. Since we already have a function to compute the mean, we can use that function. Decomposing the problem further, we can create a function to compute the standard deviation from the array of values and the precomputed mean, and then bring the two functions together in a third function to compute the statistics. The skeleton of our program is thus:

```
def mean(values):
    """
    return the average of an array of values
    """
    total = 0
    for val in values:
        total = total + val
    return total/len(values)

def standard_dev(values,mean):
    """
    return the standard deviation from values and mean
    """

def statistics(values):
    """
    return the mean and standard deviation of a set of values
    """
```

Note that the `standard_dev` function has a parameter named `mean`, which is the same name as the `mean` function. This is not a problem because `standard_dev` will not use the `mean` function and in Python local names, such as that of a local variable, take precedence over global names, outside the function in the same module.

Now that we understand the algorithm and planned our program, we can implement the functions, starting with the standard deviation:

```
def standard_dev(values,mean):
    """
    return the standard deviation from values and mean
    """
    total = 0
    for val in values:
        total = total + (val-mean)**2
    return (total/len(values))**0.5
```

Like always, we need to test each function before working on the next one. Note that in these tests we use the `mean` to compute the mean and provide it as an argument for the `standard_dev` function.

```
In : vals = [4,4,4,4,4]
In : standard_dev(vals,mean(vals))
Out: 0.0

In : vals2= [2,4]
standard_dev(vals2,mean(vals2))
Out: 1.0
```

Finally, we can bring these functions together in the `statistics` function. This function will have to return two values, but this is not a problem. We can put several values after the `return` instruction and the function will return a tuple with all the values.

```
def statistics(values):
    """
    return the mean and standard deviation of a set of values
    """
    sample_mean = mean(values)
    st_dev = standard_dev(values, sample_mean)
    return sample_mean, st_dev
```

Now we can test our function. Note that the function returns a tuple with two values, because we put two values in the `return` instruction. But if we want the mean and standard deviation in different variables, we can unpack the result of the `statistics` function. The example below illustrate this.

```
In : vals = [1,2,3,4,4,5,5]
In : statistics(vals)
Out: (3.4285714285714284, 1.3997084244475304)

In : m, std = statistics(vals)
In : m
Out: 3.4285714285714284
In : std
Out: 1.3997084244475304
```

6.8 Creating a new list

Sometimes, we may need to create a list of values. We can start with an empty list and then use the `append` method to add new elements to the list. Here is a function for returning a list with the square of each element of an array supplied as an argument.

```
def squares(values):
    """
    return a list with the elements of values squared
    """
    result = []
    for val in values:
        result.append(val**2)
    return result
```

```
In : squares([1,2,3,4,5])
Out: [1, 4, 9, 16, 25]
```

To illustrate with another example, let us revisit the Fibonacci series, recursively defined as:

$$F_0 = 0 \qquad F_1 = 1 \qquad F_n = F_{n-1} + F_{n-2}$$

We can compute a list with the first N Fibonacci numbers using a `for` loop:

```
def fibonacci(N):
    "return a list with the first N Fibonacci numbers"
    f_numbers = [0,1]
    for _ in range(N-2):
        f_numbers.append(f_numbers[-1]+f_numbers[-2])
    return f_numbers
```

In this function, we assume that we need to provide at least the first two Fibonacci numbers, so we start by creating the list with the first two numbers. Then we loop through the remaining $N - 2$ numbers, so this is the argument we provide to `range`. This for loop will only be needed to count the right number of elements and compute each of them, and we do not need to use the variable that will hold each number generated by `range(N-2)`. In Python, whenever we need to provide a variable due to syntactic requirements but without needing to use that variable, it is conventional to name it using a single underscore. This indicates a "don't care" variable.

For each of the N-2 iterations, inside the loop, we append to the `f_numbers` variable the sum of its two last elements, which we can access using negative indexes. However, there is a problem with this function. Although it works when N is less than 2, since using `range` with a negative number just skips the loop, the function still returns the first two Fibonacci numbers:

```
In : fibonacci(10)
Out: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]


In : fibonacci(1)
Out: [0, 1]
```

We can fix this by returning a slice of the list of numbers, with only the first N numbers. In most cases, N will be greater than 1 and so this will make no difference. But if N is 1 or 0, our function will return the expected value:

```
def fibonacci(N):
    "return a list with the first N Fibonacci numbers"
    f_numbers = [0,1]
    for _ in range(N-2):
        f_numbers.append(f_numbers[-1]+f_numbers[-2])
    return f_numbers[:N]
```

```
In : fibonacci(1)
Out: [0]

In : fibonacci(0)
Out: []

In : fibonacci(-10)
Out: []
```

Aside from the `append` method, another useful method to add elements to a list is the `extend` method. While the `append` method appends its argument as an element to a list, the `extend` method appends all elements of the argument to the list, individually. Here are some examples:

```
In : a_list = [0]
Out: a_list.append('abc')
In : a_list
Out: [0, 'abc']
In : a_list.extend('abc')
In : a_list
Out: [0, 'abc', 'a', 'b', 'c']
In : a_list.extend([1,2,3])
In : a_list
Out: [0, 'abc', 'a', 'b', 'c', 1, 2, 3]
```

6.9 Demonstration

For a more elaborate example of creating lists, let us suppose we want an array with the lengths of all the words in a text. In general, words are separated by spaces, but it may also happen that words are separated by a comma or period, and this may or may not include spaces. Furthermore, words may be separated by line breaks (the `\n` special character). One useful method for this is the `split` method of strings. Without arguments, or with `None` as an argument, the `split` method of a string returns the parts of the string separated by all whitespace, which includes spaces, tabs and line breaks. For example:

```
In : text = """word     word
...  another another_word    end"""

In : text.split()
Out: ['word', 'word', 'another', 'another_word', 'end']

In : text.split(None)
Out: ['word', 'word', 'another', 'another_word', 'end']
```

However, if the space is missing, we need to split the text with other separators too. We can do this with (split) also by supplying the separator as an argument.

Let us consider this text

```
text =  """
Portugal,oficialmente República Portuguesa,é um país soberano
unitário localizado no sudoeste da Europa,cujo território se situa na zona
ocidental da Península Ibérica e em arquipélagos no Atlântico Norte.O território
 português tem uma área total de 92090km2, sendo delimitado
a norte e leste por Espanha e a sul e oeste pelo oceano Atlântico,
compreendendo uma parte continental e duas regiões autónomas:
os arquipélagos dos Açores e da Madeira. Portugal é a nação mais
a ocidente do continente europeu.O nome do país provém da sua segunda
maior cidade, Porto, cujo nome latino-celta era Portus Cale.
"""
```

If we want to count word lengths, we need to take into account all whitespace but, before that, we also need to break the text in parts using `','` and `'.'` as separators. This would be the outline of the algorithm: we break the text into different parts using one separator, then break each part using another separator, and so on, until finally we break all parts using whitespace, with the `split(None)`, for example.

Now that we know the algorithm we want to implement, we can decompose it into simpler problems implemented as functions that we want to make as general as possible. We need a function that receives a list of strings and a separator and returns a list with all the original strings split using that separator. We also need a function that receives a string and all the separators and, using the previous function, iteratively breaks down all fragments until the final list of words. Then we need a function to count the lengths of all words in a list and return a list with the lengths. Since we want these functions to be as generic as possible and work with any list of separators, we can also implement a final function that solves our problem of counting word lengths in texts like the one above and in which we build in the separators we want.

```
 1 def split_all(string_list, separator):
 2     """splits all strings in a list of strings, returns resulting fragments"""
 3     result = []
 4     for s in string_list:
 5         result.extend(s.split(separator))
 6     return result
```

This is the first function we need, which receives a list of strings and outputs a list with all strings split by the separator. Note that the `split` method returns a list of strings, so we use the `extend` methods of our `result` to add all fragments as new elements.

We can test our function like this:

```
In : split_all(['aa,bb','cc','d,e,f'],',')
Out: ['aa', 'bb', 'cc', 'd', 'e', 'f']
```

The next function we need splits the original text using all given separators. For this we need to iterate over all separators and cumulatively use the `split_all` function to break all fragments. But because the `split_all` function must receive a list of strings and not just a strung, we need to start with a list containing the original text as the first list to break down. This is what line 10 in the code below does:

```
 8 def split_text(text,separators):
 9     """splits text over all separators"""
10     result = [text]
11     for sep in separators:
12         result = split_all(result,sep)
13     return result
```

Again, we must test our function before proceeding. We will also use `None` as the final separator to split over all whitespace and include tabulators and linebreaks to test this.

```
In : split_text('some,.text.that \t includes \n special whitespace',['.',',',None])
Out: ['some', 'text', 'that', 'includes', 'special', 'whitespace']
```

Now we implement the word counting function. This function simply appends the length of this word to a new list:

```
16 def count_word_lengths(words):
17     """return list with word lengths from list with strings"""
18     lengths = []
19     for word in words:
20         lengths.append(len(word))
21     return lengths
```

Once more, we test our function with a simple example:

```
In : words = ['a','aa','aaa','aaaa','aaaaa']
In : count_word_lengths(words)
Out: [1, 2, 3, 4, 5]
```

Finally, the function to combine all these different elements and use a specific set of separators to split all words. Note that we include other common punctuation marks in our separators so that this function can be used more generally than in this specific example:

```
def text_word_lengths(text):
    """return list with word lengths from text"""
    separators = [',','.','?','!',':',';',None]
    words = split_text(text,separators)
    return count_word_lengths(words)
```

Now we do a simple test and then use our function on the example above.

```
In : text_word_lengths('some,.text.that \t includes \n special whitespace')
Out: [4, 4, 4, 8, 7, 10]
In : text_word_lengths(text)
Out: [8, 12, 9, 10, 1, 2, 4, 8, 8, 10, 2, 8, 2, 6, 4, 10, 2, 5, 2, 4, 9, 2, 9, 7, 1, 2, 12, 2,
```

6.10 Exercises

Numerical errors

1. What are the values of these expressions? Explain the results. Note that the function `sin` and the constant `pi` are available in the `numpy` module.

   a) $(\sqrt{2})^2 - 2$

   b) $sin(\pi)$

2. Test how rounding errors accumulate adding one millionth (0.000001) a million times (1000000). Assign a value of zero to a variable, then use a `for` loop to add a value of 0.000001 to that variable a million times. Finally, subtract 1 from the variable. If there were no rounding errors the result should be 0. Compare the magnitude of the remaining value in this case, due to rounding errors, with the rounding errors in the previous exercises.

Lists and loops

1. The covariance of two non empty arrays $X = (x_1, ..., x_n)$ and $Y = (y_1, ..., y_n)$, can be computed with the following formula:

$$Cov(X,Y) = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{n}$$

   where $\bar{x}$ and $\bar{y}$ are the mean values of X and Y.

a) Create a function that computes the covariance of two lists of values (with the same length). Remember to decompose the problem into simpler sub-problems, including other auxiliary functions.
b) Compute the covariance of these two arrays:

```
v1 = [1,2,5,4,7,6,9,10]
v2 = [3,2,1,2,8,9,12,15]
```

Answer: 12.75

2. To test the effect of a drug for hair growth, researchers measured the length of the hair of volunteers during a year. The `week` and `length` lists record, respectively, the week in which each measurement was taken and the length, in mm, for one of the volunteers:

```
week = [1, 5, 15, 18, 29, 33, 36, 37, 38, 40, 45, 49, 50, 52]
length = [30, 36, 48, 69, 93, 105, 108, 111, 126, 141, 147, 153, 156, 161]
```

a) Create a function that returns a list with the growth between each two consecutive measures (the length difference between each measurement and the one before). For example, between week 15 and week 5, hair growth was 12mm. How many elements does the resulting list have, compared to the number of elements in the original `week` and `length` lists?
b) We will also need a function that returns a list with the number of weeks between consecutive measurements. For example, between the measurement on week 5 and the measurement on week 15 there was a period of 10 weeks. Do you need another function or can you generalize the function for the growth between consecutive measurements.
c) To aggregate sets of measurements, it is convenient to have a weekly growth rate computed for each week. Create a function that returns a list of weekly growth rates estimating the growth for each week as the average growth rate between the last measurement taken before that week and the next measurement. For example, for weeks 2 through 5, including week 5, the net growth was 6mm (30mm of hair length on week 1 and 36mm of hair length on week 5), giving an average growth rate of 6 / (5-1) = 1.5mm for each of those 4 weeks (weeks 2, 3, 4 and 5). Assume that the growth rate for week 1 is 0 since we have no measurements before that week. Using the data above, the first values in this list should be:
`[0, 1.5, 1.5, 1.5, 1.5, 1.2, 1.2, 1.2, 1.2, 1.2, 1.2, 1.2, ...`
Suggestion: first try to solve this problem by hand to understand the algorithm. Remember to use any of the previous functions you may need.
d) What is the average hair growth rate for the whole year?

Processing strings: `replace`

In the demonstration above, we saw how to count the word lengths for all words in some text. To do this, we used several loops to split lists of strings consecutively until we reached the final split into individual words. A simpler alternative would be to replace all separators (comma, period, question mark and so on) with space characters and then just do a split with no arguments to obtain the words. Reimplement the program in this simplified version. Check the documentation on the `replace` method or look for information online as necessary. Then outline the algorithm, plan the decomposition and implementation and then implement the necessary function or functions. Remember to reuse previously implemented functions as convenient.

6.11 Tutorial (revisions)

In this tutorial you will create a small program to process strings, using sequences and restriction enzymes as an example to illustrate the different concepts we covered so far. To begin, you must place in your working folder the two modules provided, `data.py` and `t4_tests.py`. The tests module works like the one we used on the second tutorial. The `data` module stores the sequence and restriction enzyme information in the variables `sequences`, `enzyme_names`, `recon_sites`, `cutting_points`.

To begin this exercise, create a new source-code file named `tutorial4.py`. Note that the name of your module is important because the test module assumes this name. In order to use the sequence and enzyme data, you can import the variables from the `data` module:

```
from data import sequences, enzyme_names, recon_sites, cutting_points
```

in which case you can use the variable names directly in your module. Alternatively, you can import the `data` module name with `import data` and then access its variables as members of the module. For example `data.sequences` .

Measuring GC content

The GC content of RNA or DNA is the percentage of nitrogenous bases guanine and cytosine, and can vary significantly between organisms and within different regions of a genome. Generalizing the problem of computing the GC content of a sequence, we will implement a function called `content` that has as parameters, in order, first the sequence and, second, an ordered set (list, tuple or string) of the substrings to count. To count the number of times a substring is present in a string we can use the `count` method. Here are some examples:

```
In : s = 'blablablaaaa'
In : s.count('bla')
Out: 3
In : s.count('a')
Out: 6
```

The function `content` should return the total percentage of occurrences of all residues specified in the second argument. The percentage should be an integer value between 0 and 100 (you can use the `round` function). Here are some examples:

```
In : content('ACTGGCCTACC', ['G','C'])
Out: 64

In : content('CGCGCGGCG', ['G','C'])
Out: 100

In : content('TTATATATTA', ['G','C'])
Out: 0
```

After implementing your function, run the tests module and check the report. Note that the function must have the name and parameters as specified here.

Generalizing

Can your function work with proteins too? For example, suppose you have this protein sequence:

```
MANEGDVYKCELCGQVVKVLEEGGGTLVCCGEDMVKQ
```

and want to know the percentage of negatively charged amino acid residues in this sequence. These are apartic acid (code `D`) and glutamic acid (code `E`). Can you use your function as implemented to solve this problem and find the answer? (it's 19%)

List content for several strings

Create the function `list_content` that receives as parameters an ordered set of strings (the sequences) and an ordered set of substrings (the residues to count) and returns a list of integer values with the rounded value of the percentage composition of these residues in each string. For example:

```
In : list_content(['CGCGCGGCG','TAGCTAGC','TTATATA'],['G','C'])
Out: [100, 50, 0]
```

Test your function in with your own tests and using the test module.

Use this function to get the GC content of all sequences in the `sequences` list of the `data` module. The answer should be

```
[50, 47, 51, 51, 50, 47, 55, 47, 46, 46, ...  , 49, 51, 46, 51]
```

Simulating digestion

Implement the function `digest_sequence` which receives, as arguments, a sequence, the recognition site for a restriction enzyme and the sequence with the cutting point marked with the `-` character, and returns a list with the fragments resulting from the digestion. Test this with your own tests and the `t4_tests` module.

Number of fragments per enzyme

Implement the function `fragment_count` that receives, as arguments, a list of sequences, a recognition site and the string marking the cutting point and returns a list with the number of fragments obtained by digesting each sequence with an enzyme with those characteristics.

Simulating digestion by multiple enzymes

Implement the function `simulate_digestion` that receives, as arguments, a list of sequences, a list of enzyme names, a list of recognition sites and a list of the strings marking the cutting point of each enzyme. Assume that these three lists with enzyme information all have the same number of elements and in the same order. This function must return list of tuples, each one with the name of an enzyme and the list with the number of fragments obtained by digesting all sequences with that enzyme. Here is an example of the expected results:

```
In : report = simulate_digestion(sequences, enzyme_names, recon_sites, cutting_points)
In : for r in report:
    ...  print(r)

('EcoRI', [2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...])
('AccII', [2, 1, 4, 5, 2, 2, 7, 2, 2, 1, 4, 4, 1, 4, 2, 5, 2, 3, 2, 2, 2, 5, ...])
('BamHI', [1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 2, 1, 2, 1, 1, 1, 1, 1, 1, ...])
('TaqI', [3, 3, 4, 3, 3, 3, 6, 1, 5, 3, 3, 2, 3, 6, 2, 4, 5, 1, 2, 4, 4, 3, ...])
('Sau3AI', [5, 3, 5, 2, 3, 4, 4, 1, 6, 5, 4, 2, 4, 10, 2, 3, 4, 2, 3, 2, 2, ...])
('AluI', [3, 2, 5, 5, 5, 2, 4, 4, 2, 2, 4, 2, 3, 4, 2, 4, 4, 6, 1, 3, 4, 3, ...])
```

Chapter 7

Conditions

Boolean expressions. Conditions and the `if` statement. Examples

Boolean expressions

Boolean expressions, named after the english mathematician George Boole, are expressions that are evaluated either as `True` or `False`. The code below shows some examples of relational operators that return boolean values.

```
In : 1 == 2
Out: False

In : 1 != 2
Out: True

In : 2>=5
Out: False

In : 2<=5
Out: True
```

```
In : 'abc' < 'abd'
Out: True

In : [1,2,3] < [1,2,3,4]
Out: True

In : 1 < 4 < 6
Out: True

In : 1 < 4 > 2
Out: True
```

The comparison operators for equal and different are, respectively, == and `!=`. Note that the equal operator has two equal signs, to distinguish it form the attribution operator. The greater or lesser than operators, with or without including equality, can be applied to any type of objects for which an ordering is defined. With strings, for example, this corresponds to alphabetical order. Note, however, that this depends on how each object class implements these comparisons. We can also chain relational operators, as shown in the right column. When testing `1 < 4 < 6` the interpreter will evaluate the conjunction of the values of `1 < 4` and `4 < 6`. Only if both are true will the result also be `True`. In some cases, different types can be compared automatically. For example, floating point numbers and integer numbers, as shown below. However, in many cases it is not possible to compare different types of values, such as integers and strings. In such cases we must convert them explicitly before comparing.

```
In : 10.0 < 10
Out: False

In : 11.0 == 10
Out: False
```

```
In : 10.0 == 10
Out: True

In : 10 < '10'
Traceback (most recent call last):

TypeError: '<' not supported between instances of 'int' and 'str'

In: 11 < int('12')
Out: True
```

The operators `and`, `or` and `not` can be used to combine boolean expression:

```
In : 2==3 or 1<2
Out: True

In : 2==3 and 1<2
Out: False

In : 2==2 and 1>0
Out: True

In : 2==2 and not(1>0)
Out: False
```

```
In : (1==1 or 1==0) and (2==2)
Out: True

In : 1==1 or 1==0 and 2==2
Out: True

In : 1==1 or 1==0 and 2==1
Out: True

In : (1==1 or 1==0) and (2==1)
Out: False
```

Another useful boolean operator is the `in` operator, which checks if an element is present in a set. This also works with substrings. The `is` operator tells us whether the objects are the same object. This can be useful to distinguish between the case were two variables are referring the same object or if they refer two different objects which may happen to have the same values.

```
In : 12 in [1,2,3]
Out: False

In : 2 in [1,2,3]
Out: True

In : 'ab' in 'xabx'
Out: True
```

```
In : a = [1,2,3]
In : b = [1,2,3]
In : c = b

In : a == b
Out: True

In : a is b
Out: False

In : b is c
Out: True
```

Conditions: `if`

In order to implement any algorithm, we must have a way to control the execution of operations. Of the three three basic elements needed for controlling the flow of the execution, we previously saw the order of the instructions in the source code, which determines the order in which they are executed by the interpreter, and the `for` loop, which allows us to repeat instructions. Now we will see the remaining element we need, the `if` statement, which allows us to execute different blocks of code depending on some condition or conditions. We will use as an example a function to solve quadratic equations.

Unlike the first version we wrote for computing the pH, which only returned the positive root, this time we will be able to check how many solutions the equation has and return the correct values:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

where the expression $b^2 - 4ac$ is called the discriminant. If the discriminant is negative there are no real solutions; if it is zero there is only one solution and if it is greater than zero there are two solutions to the equation.

To execute different instructions depending on the value of the discriminant we will use the `if` statement. The `if` statement always begins with the word `if` followed by a condition and a colon, `:`. Indented below this line is a block of code that will only be executed if the condition is true. This is the simplest version of the `if` statement:

```
In : a = 12
In : if a>5:
     ... print('large')
Out: large

In : a = 4
In : if a>5:
     ... print('large')
Out:
```

In the first example, since the value in variable `a` is greater than 5, the `print` statement in the first `if` is executed. In the second example the condition is false and so the `print` statement is not executed. When the condition is false the interpreter skips the block inside the `if`.

When Python must interpret an object as a boolean expression, then the value 0 and all empty arrays are considered `False`. Any other object is considered `True`. This can happen if you use an object without relational operators inside the `if` condition. Note, however, that this can make the code harder to understand.

```
In : if '':
   ... print('True')
Out:

In : if 'xx':
   ... print('True')
Out: True

In : if 1:
   ... print('True')
Out: True
```

```
In : if 0:
   ... print('True')
Out:

In : if []:
   ... print('True')
Out:

In : if [0]:
   ... print('True')
Out:True
```

The `if` statement can also, optionally, include a single `else:` branch. The block following the textttelse: line is only executed if all other conditions in the `if` statement fail. Note that all branches are mutually exclusive; only one, at most will be executed. The `else` statement guarantees that one branch will be executed.

```
In : x = 10
In : if x>5:
 ...    print('x is large')
 ... else:
 ...    print('x is small')

Out: x is large
```

```
In : x = 2
In : if x>5:
 ...    print('x is large')
 ... else:
 ...    print('x is small')

Out: x is small
```

In its most general form, an `if` statement can include any number of `elif` clauses, optionally followed by the `else` clause. The `elif` branches work in the same way as the `if` line: if the condition is true, the following (indented) block of code is executed. However, any condition is only tested if no conditions so far were met. Only one branch of the `if ... elif ...` construct is executed. Once one condition is found to be true and the corresponding block is executed, the interpreter resumes execution after the end of all branches.

For our example of solving a quadratic equation, we have three possible cases. If the discriminant is less than zero we want to return an empty list indicating that there are no real solutions for this equation. If not, then we need to check if the discriminant is greater than zero, in which case we want both solutions and, finally, if the discriminant is neither less than or greater than zero, we return the single solution. Here is the code for the function.

```
def solve_quadratic(a,b,c):
    """
    Solve a quadratic equation defined by the coefficients
    Return the solutions in a list
    """
    delta = b**2 - 4*a*c
    #check the discriminant
    if delta<0:
        solution = []
    elif delta>0:
        solution = [(-b -delta**0.5)/(2*a),(-b + delta**0.5)/(2*a)]
    else:
        solution = [-b/(2*a)]
    return solution
```

Note the indentation of each block corresponding to each of the branches, and that the line preceding a new block ends with a colon, `:`.

Now we can test our new function for the equation $2x^2 - 3x + 1 = 0$:

```
In : test_solution = solve_quadratic(2.0,-3.0,1.0)
In : test_solution
Out: [0.5, 1.0]
In : 2.0*test_solution[0]**2.0-3.0*test_solution[0]+1.0
Out: 0.0
In : 2.0*test_solution[1]**2.0-3.0*test_solution[1]+1.0
Out: 0.0
```

When dealing with conditions it is important to test all branches because there may be an error in one of the branches. We can do this in a function for testing purposes:

```
def tests():
    print('solve_quadratic(3.0, 2.0, 1.0)',solve_quadratic(3.0, 2.0, 1.0))
    print('solve_quadratic(3.0, 4.0, 1.0)',solve_quadratic(3.0, 4.0, 1.0))
    print('solve_quadratic(1.0, 0.0, 0.0)',solve_quadratic(1.0, 0.0, 0.0))
```

```
tests()
solve_quadratic(3.0, 2.0, 1.0) []
solve_quadratic(3.0, 4.0, 1.0) [-1.0, -0.3333333333333333]
solve_quadratic(1.0, 0.0, 0.0) [-0.0]
```

7.1 Example: ICE grades

For another example of using conditions, and also to review previous subjects like functions and loops, we will write a program to compute the final grades in ICE from the grades of tests and assignments of each student. In the `assessment.py` file we wave the following variable with the grades for the practical assignments and tests, for each student, in a tuple containing the student's number, the grades of the two assignments and of the two tests.

```
# -*- coding: utf-8 -*-
"""
Student test and assignment grades
"""
grades = [[1000 ,7.9 ,18.9 ,7.2 ,10.2 ,],
          [1001 ,8.2 ,7.5 ,6.5 ,19.3 ,],
          [1002 ,19.1 ,3.7 ,13.5 ,4.6 ,],
          [1003 ,17.5 ,14.8 ,14.6 ,9.1 ,],
           ...
 ]
```

The final grade is computed by first computing the lab component as the average of the two assignments, rounded to the first decimal place. The theoretical component is the weighted average of the first test (0.4) and the second test (0.6), rounded to the first decimal place. Then, if both assignments have grades below 9.5, the student does not have frequency and the final grade should be marked as -1. Otherwise, if the theoretical component is less than 8.0, the student is not approved and the final grade is simply the grade of the theoretical component. Otherwise, the final grade is the weighted average of the lab component (0.4) and the theoretical component (0.6), rounded to the closest integer.

There are several problems to solve here. We need a function to compute the values rounded to the first decimal point; we need a function to compute the final grade of one student and, finally, a function to compute the list with a tuple with the student number and the final grade for all students.

Here is the code below, including a function for testing.

```
# -*- coding: utf-8 -*-
"""
Student final grades
"""
from assessment import grades

def round_dec(value):
    "round to first decimal place"
    return round(value*10)/10

```

```
def final_grade(grades):
    """
    compute final grade for a single student
    grades are two practical assignments and two tests
    """
    p1,p2,t1,t2 = grades
    comp_tp = round_dec(t1*0.4+t2*0.6)
    comp_l = round_dec((p1+p2)/2)
    if p1<9.5 and p2<9.5:
        final = -1
    elif comp_tp<8.0:
        final = comp_tp
    else:
        final = 0.4*comp_l + comp_tp*0.6
    return round(final)

def final_grades_table(student_grades):
    """
    return table with number and final grades for all students
    """
    table = []
    for student in student_grades:
        final = final_grade(student[1:])
        table.append((student[0],final))
    return table

def tests():
    print('round_dec(1.122)',round_dec(1.122))
    print('final_grade((4,4,10,10))',final_grade((4,4,10,10)))
    print('final_grade((10,10,5,5))',final_grade((10,10,5,5)))
    print('final_grade((10,10,10,10))',final_grade((10,10,10,10)))
    print('final_grades_table(grades)',final_grades_table(grades))
```

The first function returns a value rounded to the first decimal place by multiplying by 10, rounding and then dividing by 10 again. The second function returns the final grade for a single student. It expects a tuple with the grades of the practical assignments and the tests, and starts by unpacking these values into four variables (`p1, p2, t1, t2`). Then it computes the two grade components rounded to the first decimal point and checks the two conditions: frequency with the lab component at least 9.5 and theoretical grade at least 8.0. Note the `if ... elif ... else` construct to account for the three possible cases. Also note the order of the `if` branches. Frequency is tested first because it takes precedence over the other cases. Then the minimum passing grade for the tests and finally the grade combining the two components only if the two previous conditions were not met.

The last function loops through all the student grades and builds a list of tuples with the student number and the final grade. Note that the call to the `final_grade` passes as an argument a slice of the tuple for the student that ommits the first value, which is the student number, since the `final_grade` function only expects the grades and not the number of the student.

7.2 Example: diameter of a protein

Desulforedoxin is a small homodimer from *Desulfovibrio gigas*, a sulphate reducing bacterium. The id code for the structure of this protein in the Protein Data Bank (PDB) is 1DXG.

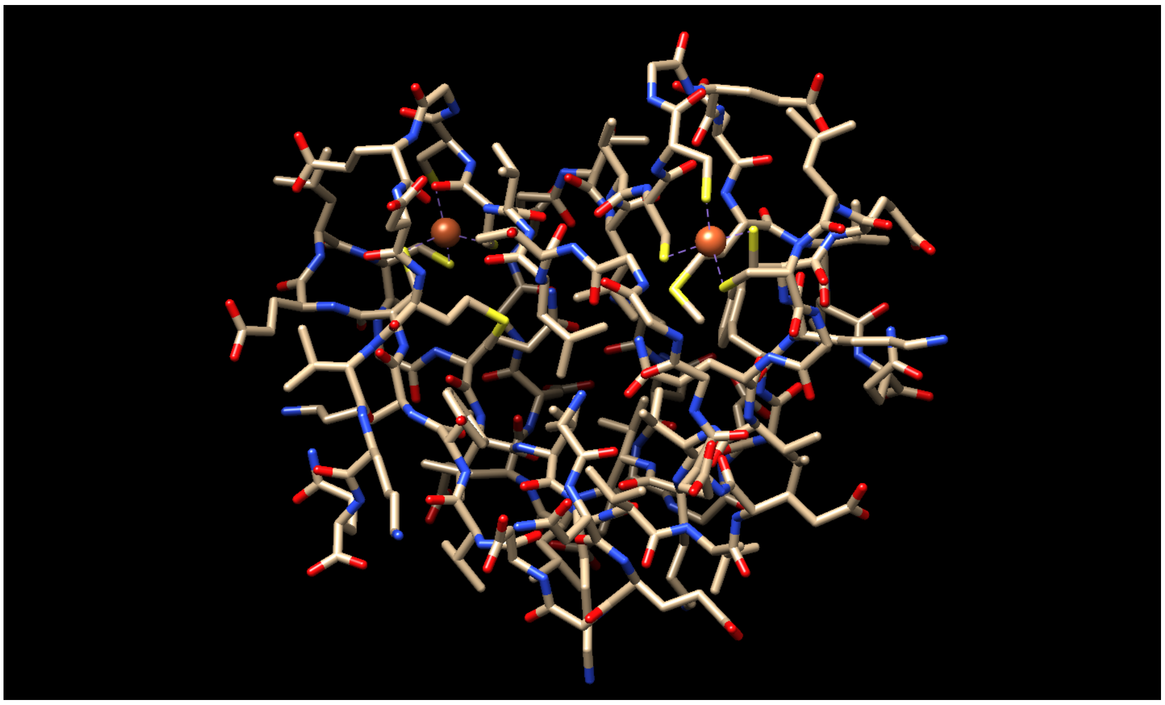


Figure 7.1: Desulforedoxin, PDB structure 1DXG. Image created with USFC Chimera

The file texttprotein.py has a `coords_1dxg` variable with the coordinates of all atoms of this structure, in Ångstrom, as a list of tuples:

```
# -*- coding: utf-8 -*-
"""
Coordinates of desulforedoxin
"""

coords_1dxg=[
 (2.947, 31.497, 15.827),
 (4.159, 30.906, 16.375),
 (5.262, 30.978, 15.333),
 (4.988, 31.483, 14.245),
   ...
 (-5.508, 25.661, 2.712),
 (-4.744, 20.784, -0.064)]
```

In this example we want to find the diameter of this structure, which we will consider to be the longest distance between any two atoms. To decompose this problem of measuring all distances from all atoms to all other atoms, we can think of three tasks, each corresponding to one function.

- Distance between one point and one other point: `dist(p1,p2)`
- Maximum distance between a list of points and one point: `max_distance(coords, reference)`

- Maximum distance between all pairs of points, which is the diameter: `diameter(coords)`

This is a good general approach whenever we have a problem that requires doing the same operation over many values or combinations of values. We can break it down to the simplest problem, which in this case is measuring the distance between two points, and then build up gradually to solving the complete problem.

Here is the code, below, for a first version:

```python
# -*- coding: utf-8 -*-
"""
Compute protein length
"""
from protein import coords_1dxg

def dist(p1,p2):
    "distance between two points in 3D"
    x1,y1,z1 = p1
    x2,y2,z2 = p2
    return ((x1-x2)**2 + (y1-y2)**2 + (z1-z2)**2)**0.5

def max_distance(coords, reference):
    "return longest distance between points and reference"
    maxd = 0
    for c in coords:
        d = dist(c,reference)
        if d>maxd:
            maxd=d
    return maxd

def diameter(coords):
    "return longest distance between all points"
    maxd = 0
    for ref in coords:
        d = max_distance(coords, ref)
        if d>maxd:
            maxd=d
    return maxd

def tests():
    print('dist( (0,0,0), (1,0,0))',dist( (0,0,0), (1,0,0)))
    print('dist( (0,0,0), (1,1,1))',dist( (0,0,0), (1,1,1)))
    test_coords = [(0,0,0),(0,0,1),(1,1,1),(0,0,10)]
    print('max_distance([(0,0,0),(0,0,1),(1,1,1),(0,0,10)],(0,0,0))',max_distance(test_coords,(0,
    print('diameter(coords_1dxg)',diameter(coords_1dxg))
```

The first function computes the euclidean distance between two points in three dimensions. The second function determines the longest distance between a list of coordinates and one reference point. To do this, we need to measure the distance between each point in the list and the reference point, and update the maximum distance, which we start at 0, every time we find a larger distance than found before. This is done with an `if` statement insider the `for` loop.

Finally, we find the maximum of all these maximum distances by looping through all the points and setting each one as the reference. This is done in the last function, which returns the diameter of the protein. We can run the tests to check if everything is working. Note the `import` statement that allows us to use the desulforedoxin coordinates.

```
tests()
dist( (0,0,0), (1,0,0))) 1.0
dist( (0,0,0), (1,1,1))) 1.7320508075688772
max_distance([(0,0,0),(0,0,1),(1,1,1),(0,0,10)],(0,0,0)) 10.0
diameter(coords_1dxg) 29.90105367708637
```

One problem with this implementation is that it does twice the necessary work, since it measures the distance between point i and point j when i is the reference, and then again when j is the reference. We can prevent this if, for each point in the list we take as reference, we measure the distance only to points with a higher index in the list. This is easy to do in the last function by slicing the list and looping through the indexes instead of the tuples directly:

```
def diameter(coords):
    "return longest distance between all points"
    maxd = 0
    for ix in range(len(coords)-1):
        d = max_distance(coords[ix+1:],coords[ix])
        if d>maxd:
            maxd=d
    return maxd
```

Note that the `for` loop now stops one point before the last.

7.3 Further reading

Chapter 6, pages 297 through 305, and Chapter 12 of the textbook [1].

Chapter 8

Matrices

Using Numpy matrices and functions. Broadcasting operations. Boolean indexing with Numpy matrices.

Numpy vectors and matrices

The `numpy` library has a useful class of objects for handling n-dimensional arrays such as vectors and matrices. We can create a numpy array in several different ways. First we need to import the `numpy` module, and since we will be using it for many operations, it is useful to give it a shorter alias. This can be done with `import numpy as np`, which imports the module and renames it `np` in the current namespace.

Now we can use any element of the `numpy` module with the `np` name. For example, the `np.ones` function, which returns a matrix filled with 1 with the dimensions specified in a tuple given as the first argument. We can also create vectors or matrices using the `np.array` function, which receives as argument a list or tuple, which can also contain lists or tuples if the dimension is larger than 1. The code below illustrates this.

```
In : import numpy as np
In : vec = np.ones(5)
Out: array([1., 1., 1., 1., 1.])

In : mat = np.zeros( (5,5) )
Out:
array([[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.]])

mat2 = np.array( [ [1,2,3],[4,5,6] ] )
Out:
array([[1, 2, 3],
       [4, 5, 6]])
```

All these examples produce objects of type `ndarray`, which can store n-dimensional arrays (vectors, matrices, and so on). The type of values in the array is determined automatically by the values used to create the array. Using the `astype` method of a Numpy array we can obtain a copy of the array with the values converted to the specified type. For example, from floating point to integer. The `dtype` attribute stores the type of the values in the array, and the `shape` attribute is a tuple with the length of the array in each dimension.

```
In : vec1 = np.array([1,2,3])
In : vec1.dtype
Out: dtype('int64')

In : vec2 = np.array([1.0,1.5,2.0]
Out: array([1. , 1.5, 2. ])

In : vec2.dtype
Out: dtype('float64')

In : vec2.shape
Out: (3,)
```

```
In : vec2.astype(int)
Out: array([1, 1, 2])

mat = np.array([[1,2],[3,4],[5,6]])
In : mat.shape
Out: (3, 2)

In : mat
Out:
array([[1, 2],
       [3, 4],
       [5, 6]])
```

Numpy arrays can them be used in many operations. Numpy automatically broadcasts the operands even if they have different dimensions. Each dimension is compared in order, from the last dimension to the first, and if the arrays match in that dimension (either having equal lengths in that dimension or having a length of one) the operator is applied[1]:

```
In : mat1=np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])
In : mat1/2
array([[0.5, 1. , 1.5, 2. ],
       [2.5, 3. , 3.5, 4. ],
       [4.5, 5. , 5.5, 6. ]])

In : mat1 + 5
array([[ 6,  7,  8,  9],
       [10, 11, 12, 13],
       [14, 15, 16, 17]])

In : mat1 + [5,10,15,20]
array([[ 6, 12, 18, 24],
       [10, 16, 22, 28],
       [14, 20, 26, 32]])
```

In the first example, each element of array `mat1` is divided by 2. In the second example, 5 is added to each element. The third example shows how the array matches a list of four values to the last dimension of the array (`mat1` has four columns) and then broadcasts the operation over all the rows of the matrix.

The Numpy library also implements useful functions for matrices and vectors. These examples below illustrate some functions for adding all elements, finding the mean and standard deviation, the maximum and the minimum.

```
In : mat1=np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])
Out: array([[ 1,  2,  3,  4],
            [ 5,  6,  7,  8],
            [ 9, 10, 11, 12]])
```

[1] for more in numpy broadcasting rules, see http://docs.scipy.org/doc/numpy/user/basics.broadcasting.html

```
In : np.sum(mat1)
Out: 78

In : np.mean(mat1)
Out: 6.5

In : np.std(mat1)
Out: 3.452052529534663

In : np.max(mat1)
Out: 12

In : np.min(mat1)
Out: 1
```

Default values and passing arguments by name

Function parameters can have default values, specified in the function signature. These are the values the parameters will take if no arguments are provided for these parameters. Here is a simple example.

```
def test_size(number, min_val=1, max_val=10):
    "tests if number is large, medium or small"
    if number > max_val:
        return 'Large!'
    elif number < min_val:
        return 'small...'
    else:
        return 'Medium.'
```

```
In : test_size(5)
Out: 'Medium.'

In : test_size(5,6)
Out: 'small...'

In : test_size(5,1,4)
Out: 'Large!'
```

When we call the `test_size` function without the second and third arguments, the values assumed are the default values of `min_val=1, max_val=10`, specified in the signature of the function. If we provide the corresponding arguments, then the values provided replace the default values.

This is especially useful when passing arguments by the name of the parameter instead of the position, which Python allows as long as no positional arguments are given after an argument given by name. For example:

```
In : test_size(15, max_val=20)
Out: 'Medium.'

In : test_size(15, max_val=200, min_val=100)
Out: 'small...'
```

```
In : test_size(max_val = 10, min_val = 0, number = 20)
Out: 'Large!'
```

Matrix operations in a given axis

Most Numpy functions that operate on matrices allow us to specify an axis for the direction in which the operation will be applied. By default, the axis is set to `None`, which considers the whole matrix, as we saw in the examples for the sum, mean, maximum and standard deviation above. If we specify the axis, then the operation will return an array of the results computed in the direction of the axis. For example:

```
In : mat1=np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])
Out: array([[ 1,  2,  3,  4],
            [ 5,  6,  7,  8],
            [ 9, 10, 11, 12]])

In : np.max(mat1,axis=0)
Out: array([ 9, 10, 11, 12])

In : np.max(mat1,axis=1)
Out: array([ 4,  8, 12])

In : np.sum(mat1,axis=1)
Out: array([10, 26, 42])

In : np.mean(mat1,axis=0)
Out: array([5., 6., 7., 8.])

In : np.mean(mat1,axis=1)
Out: array([ 2.5,  6.5, 10.5])
```

In the first example, we compute the maximum values of `mat1` in the axis of index 0. This is the first dimension of the matrix and corresponds to the rows. Since the maximum is computed by "moving" along the rows, we get a vector of four elements, the number of columns, each with the maximum value for that column, computed over all rows.

If we use an axis of index 1, then the maximum is computed over the columns, and we have a vector of three values, corresponding to the number of rows, each with the maximum along each row.

8.1 Example: protein diameter

Recall the protein diameter example from the previous chapter. We will rewrite the code using Numpy matrices. Originally, we decomposed the problem in three functions: computing distance between two points, between a set of points and a reference point and then the diameter, finding the maximum distance between all pairs of points. With Numpy matrices we do not need to do all these loops and can compute the maximum distance between one point and a set of points easily, so we only need one function:

```
def diameter(coords):
    "compute diameter of structure given by coords"
    c_mat = np.array(coords)
    diam = 0
    for row in range(c_mat.shape[0]-1):
        reference = c_mat[row,:]
        dists = np.sqrt(np.sum( (c_mat[row+1:,:]-reference)**2, axis=1) )
        max_d = np.max(dists)
        if max_d > diam:
            diam = max_d
    return diam
```

First, we convert the list of tuples into a Numpy array so we can use the functionalities of these objects. Then, for each index except for the last, we consider as the reference point all values from the row. Note that we can use the same slicing we saw in strings and lists, but Numpy matrices allow us to do this for each dimension of the matrix. So `c_mat[row,:]` means all the columns of the row of index `row`, which is a vector of length 3.

Then we can compute all the distances to this point in a single line. The difference is broadcast from the `reference` vector, with three elements, to each row of the matrix `c_mat`, which has 3 columns, producing a matrix with 3 columns and the same number of rows. These values are then all squared by the `**2` and the resulting matrix is summed along the columns direction (`axis=1`), resulting in a vector with all the squared distances, which is then converted into distances by the `np.sqrt` function which computes the square root of the array. This way we only need one loop to solve this problem.

8.2 Example: ICE grades

The other example we saw previously was in computing the grades for this course from the assignment and test grades. We can also simplify the code using Numpy matrices:

```
def final_grades(grade_list):
    """
    compute final grade for a single student
    grades are two practical assignments and two tests
    """
    grades = np.array(grade_list)
    comp_tp = np.round(grades[:,3]*0.4+grades[:,4]*0.6,1)
    comp_l = np.round((grades[:,1]+grades[:,2])/2,1)
    freq = np.max(grades[:,1:3],axis=1)
    final = np.zeros((grades.shape[0],2))
    final[:,0]=grades[:,0]
    for ix in range(final.shape[0]):
        if freq[ix]<9.5:
            final[ix,1]=-1
        elif comp_tp[ix]<8.0:
            final[ix,1]= np.round(comp_tp[ix])
        else:
            final[ix,1]= np.round(comp_tp[ix]*0.6+comp_l[ix]*0.4)
    return final
```

First we convert the list of grades into a Numpy array. Then we can use the `np.round` function with the added argument to specify rounding to the first decimal place (if this argument is omitted the function rounds to the closest integer). Since this function can handle Numpy arrays, we can compute all the student grades for both CompTP and CompL components without needing a loop. The frequency grade is the maximum of the assignments. Note how we select the columns we need by specifying all rows with `:` and the index or slice of the desired columns, taking into account that the first column (index 0) has the student numbers.

Then we create an empty matrix (filled with zeros) with two columns and the same number of rows as the `grades` matrix and fill in the first column with the student numbers (lines 10 and 11). Finally, we loop through all the rows of the final grades matrix and, for each student, fill in the final grade by checking the frequency grade, the test grade and the final grade.

One problem with this implementation is that the result is a floating point matrix:

```
In : from assessment import grades
In : final_grades(grades)
Out: array([[ 1.000e+03,  1.100e+01],
            [ 1.001e+03, -1.000e+00],
            [ 1.002e+03,  9.000e+00],
            [ 1.003e+03,  1.300e+01],
            ...
```

But this is easy to solve by simply returning the final grades matrix converted to integer values. We just need to change the last line of our function to `return final.astype(int)`. This gives us the correct result:

```
In : from assessment import grades
In : final_grades(grades)
Out: array([[1000,   11],
            [1001,   -1],
            [1002,    9],
            [1003,   13],
            ...
```

But we can even do away with the `for` loop if we use boolean indexing instead of checking the conditions one student at a time.

Boolean indexing and Numpy matrices

Numpy arrays can also be indexed by using boolean arrays of the same dimension. In this case, `True` values indicate selected elements.

```
In : mat1=np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])
Out: array([[ 1,  2,  3,  4],
            [ 5,  6,  7,  8],
            [ 9, 10, 11, 12]])

In : vec = np.array([1,2,3])
Out: array([1, 2, 3])

In : vec>1
Out: array([False,  True,  True])

In : mat1[vec>1,:]
Out: array([[ 5,  6,  7,  8],
            [ 9, 10, 11, 12]])
```

```
In : mat1[mat1>4]
Out: array([ 5,  6,  7,  8,  9, 10, 11, 12])
```

When we compare the variable `vec` with 1, we obtain an array of booleans with the result of each comparison, since the comparison was broadcast over the vector. We can use such an array of booleans to index any dimension of a matrix or vector provided the array of boolean values has the same length as the matrix or vector in that dimension. This way, for example, we can obtain the rows of `mat1` for which `vec` has a value greater than 1.

ICE grades, second version

We can use this to simplify our grades code even further.

```
1  def final_grades(grade_list):
2      """
3      compute final grade for a single student
4      grades are two practical assignments and two tests
5      """
6      grades = np.array(grade_list)
7      comp_tp = np.round(grades[:,3]*0.4+grades[:,4]*0.6,1)
8      comp_l = np.round((grades[:,1]+grades[:,2])/2,1)
9      freq = np.max(grades[:,1:3],axis=1)
10     final = np.zeros((grades.shape[0],2))
11     final[:,0] = grades[:,0]
12     final[:,1] = np.round(0.4*comp_l+0.6*comp_tp)
13     final[comp_tp<8,1] = np.round(comp_tp[comp_tp<8])
14     final[freq<9.5,1] = -1
15     return final.astype(int)
```

Instead of the `for` loop and the `if` checks for every student, we compute the final grades for all students in line 12 by adding the weighted components. This is placed in the second column of the `final` matrix. Then we select all rows of `final` for which the `comp_tp` value is below 8 and set those to the rounded value of those elements of `comp_tp`. Note the boolean indexing in this case. Finally, we set the final grades to -1 for all students with a `freq` value below 9.5.

Note that, this time, we have to invert the order of the computations, since the final value will be the last one written into the matrix.

8.3 Exercises

Contaminants

Lead concentration was measured on soil samples taken from different locations and at different depths. The `datat5.py` file has a variable `lead_conc` with a list of three lists, the first with the identifier of the sample locations (an integer value between 1 and 20); the second with the depth each sample was taken from (in cm) and the third list with the measured Pb concentrations in parts per million.

There is a total of 100 samples from 20 different points. Write a program that allows you to answer questions like the ones below. First write your program without using then Numpy library. Then try to solve these problems with the functionalities of the Numpy arrays.

Note that to index an element from a list inside another list you can chain the indexes. For example, if `samples` is a list of lists with values, `samples[1][20]` gives you element of index 20 from the list with index 1 inside the `samples` variable. If you use Numpy arrays, then you can index all dimensions.

- How many samples were taken from point 1? (answer: 5)
- How many samples were taken at a depth of at least 90cm? (answer: 63)
- What is the mean lead concentration in samples at more than 1m in depth? (answer: 248.28)
- What was the largest measured lead concentration? (answer: 500)
- What was the smallest lead concentration measured between 1m and 2m in depth? (answer: 30)

Hints for using Numpy:

- The `np.sum` function can add an array of boolean values, counting `True` as 1 and `False` as 0.
- To compute the conjunction or disjunction of boolean arrays you can use `np.logical_and` or `np.logical_or`.

Radius of a protein

We can define the radius of a protein as the largest distance between any atom and the centre of the protein, defined as the average of all coordinates. Write a program that computes the radius of a protein from the atomic coordinates given in a list of tuples. Test your program with the `coords_1dxg` variable in the `protein.py` file. You can base your code on the examples given in this and the previous chapter, and use Numpy arrays if you want.

8.4 Further reading

If you want to learn more about Numpy or need help with some operations, the Quickstart tutorial is a good place to start: `https://docs.scipy.org/doc/numpy-dev/user/quickstart.html`.

Chapter 9

Remote resources

Computer networks and the Internet. The World Wide Web. Using remote resources in Python. Example: Nobel prize winners per country, from Wikipedia.

Networks and the Internet

World Wide Web

Using remote resources

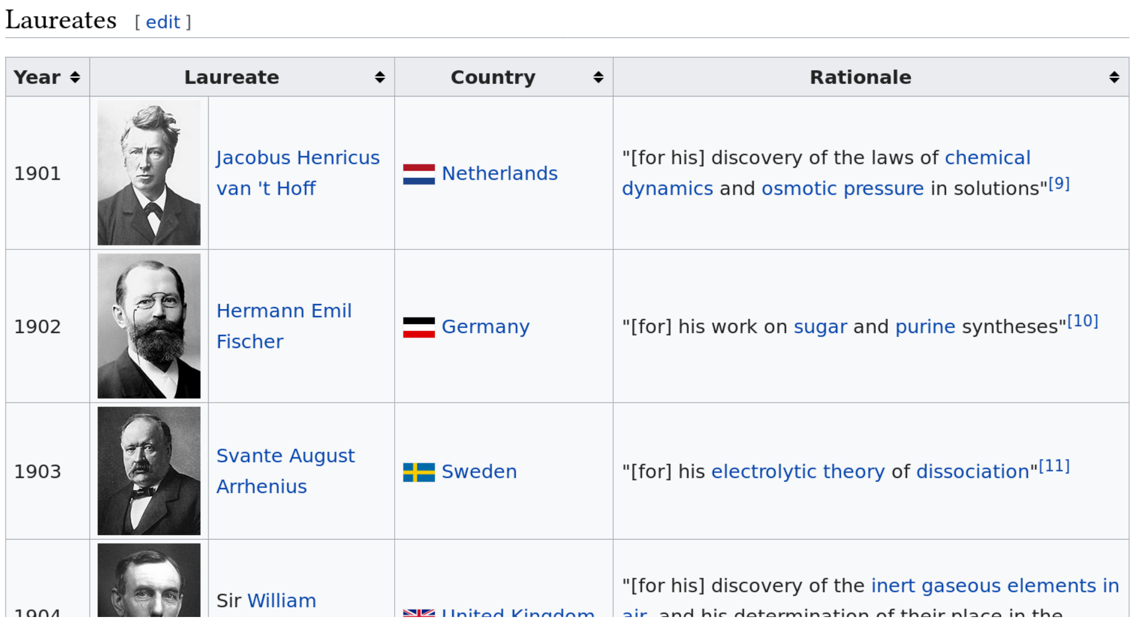
In Python, it is easy to access web pages using *HTTP* or *HTTPS* with the `requests` module. Simply import this module and use the `requests.get` function with the desires URL. This function returns a response object that includes the source of the web page in the `text` field. Here is an example:

```
In : import requests
In : r = requests.get('https://en.wikipedia.org/wiki/List_of_Nobel_laureates_in_Chemistry')
In : r.text
Out: '<!DOCTYPE html>\n<html class="client-nojs" lang="en"
      dir="ltr">\n<head>\n<meta charset="UTF-8"/>\n<title>
      List of Nobel laureates in Chemistry - Wikipedia</title>\n
      <script>document.documentElement.className =
      ...''
In : r.ok
Out: True
In : r.status_code
Out: 200
```

In addition to the `text` field, containing the contents of the accessed page, other useful fields in the response object is the `ok` field, which holds `True` if the page was successfully retrieved, and the `status_code` which returns the `HTTP` code for the request. Successful retrieval returns a status code of 200, with other codes for different errors.

To illustrate using this module, we will create a table (as a string) with the count of chemistry Nobel prize winners for each country. We will obtain the data from Wikipedia, as illustrated above.

The first thing to do is to look at the source code of the page. This is how the table of laureates in the page looks when viewed on the browser:



Laureates [edit]

Year	Laureate		Country	Rationale
1901		Jacobus Henricus van 't Hoff	Netherlands	"[for his] discovery of the laws of chemical dynamics and osmotic pressure in solutions"[9]
1902		Hermann Emil Fischer	Germany	"[for] his work on sugar and purine syntheses"[10]
1903		Svante August Arrhenius	Sweden	"[for] his electrolytic theory of dissociation"[11]
1904		Sir William	United Kingdom	"[for his] discovery of the inert gaseous elements in air, and his determination of their place in the

Figure 9.1: Table of Chemistry Nobel prize winners, from Wikipedia.

If we examine the source code (in the browser, for example, or saving the page and opening with a simple text editor), we can see the different elements, delimited by their tags. The table starts on the `<table>` tag. Each cell in the table is delimited by the `<td>` tag, and we can see that the cells containing the name of the country start with `<span class="flagicon"`, indicating the class for displaying the country flag.

```
...
 <table class="wikitable sortable">
 <tr>
 <th>Year</th>
 <th colspan="2">Laureate</th>
 <th>Country</th>
 <th>Rationale</th>
 </tr>
 <tr>
 <td>1901</td>
 <td><a href="/wiki/File:Vant_Hoff.jpg" class="image">...
 <td><a href="/wiki/Jacobus_Henricus_van_%27t_Hoff" ...
 <td><span class="flagicon"><img alt="" src="...">&#160;</span>
     <a href="/wiki/Netherlands" title="Netherlands">Netherlands</a></td>
 <td>[for his] discovery of the laws of <a href="/wiki/Chemical_dynamics"...
 </tr>
 <tr>
 <td>1902</td>
 ...
```

To process this page, we will have to solve the following sub-problems:

1. Get the source code of the HTML page from the Wikipedia URL

2. Extract all mentioned countries, including repetitions for countries with more than one Nobel prize winner.
3. From the list of all countries, obtain a list with only the different names, without repetitions, and sorted alphabetically
4. Count how many times each different country appears in the list with all country entries in the table
5. Finally, bring everything together and generate a string with the table of laureates by country

Here is the code for the first function and the beginning of our module, including the `import` instruction to import the `requests` module.

```python
#!/usr/bin/env python3
"Nobel prize winners"

import requests

def all_countries(html):
    """Return list of all countries in html table"""
    cells = html.split('<td>')
    countries = []
    for cell in cells:
        if cell.startswith('<span class="flagicon"'):
            countries.append(cell.split('>')[4].split('<')[0])
    return countries
```

First, we split the string with the full HTML source in fragments on the table cell tag `<td>`. Then we iterate over all these fragments to find those cells starting with the string that indicates a country cell, `<span class="flagicon"`. We can see from the HTML source that the name of the country is after the fourth > marker, so we can get it by extracting the fifth fragment resulting from splitting the string on the > markers. Then we split this fragment on the < marker and take the first of the resulting fragments, thus extracting the name of the country from the string where the country is between > and <:

```
<td><span class="flagicon"><img alt="" src="..."">&#160;</span>
 <a href="/wiki/Netherlands" title="Netherlands">Netherlands</a></td>
```

We can test our function and check if the result is what we want:

```
In : r = requests.get('https://...Nobel_laureates_in_Chemistry')
In : countries = all_countries(r.text)
Out: ['Netherlands', 'Germany', 'Sweden', 'United Kingdom', 'Germany', 'France',
      'Germany', 'United Kingdom', 'Germany', 'Germany', 'Poland', 'France',
      'France', 'Switzerland', 'United States', 'Germany', 'Germany', 'Germany',
      'United Kingdom', 'United Kingdom', 'Austria', 'Germany', 'Sweden',
      'Germany', 'Germany', 'United Kingdom', 'Sweden', 'Germany', 'Germany',
      'Germany', 'United States', 'United States', 'France', 'France', ...]
```

Now we need to extract from this list an alphabetically sorted list with only the different country names, without repetitions. This will be the basis of our final table and serve as references to count laureates by country.

```
def unique_countries(countries):
    """Return unique countries in list"""
    unique = []
    for c in countries:
        if c not in unique:
            unique.append(c)
    unique.sort()
    return unique
```

We start with an empty string, then loop through the original list of countries and add a country to the list of unique countries only if that country is not yet in the list. This means each country will be added only once. After creating the list, we use the `sort` method of the list to sort its items alphabetically before returning it. Testing our function we can check the results:

```
In : names = unique_countries(countries)
Out: ['Argentina', 'Australia', 'Austria', 'Belgium', 'Canada', 'Croatia',
      'Czechoslovakia', 'Denmark', 'Finland', 'France', 'Germany', 'Israel',
      'Italy', 'Japan', 'Mexico', 'Netherlands', 'Norway', 'Poland',
      'Soviet Union', 'Sweden', 'Switzerland', 'United Kingdom',
      'United States', 'West Germany', 'Yugoslavia']
```

The next step is to count how many times each country occurs on the initial table. This tells us the number of laureates per country. The algorithm for this consists in looping through the countries on the table, find the position of each country in the list with the different country names and then increase a counter for that country. This means we must first create a list with the correct number of counters, all starting at zero.

```
def count_per_category(values, categories):
    counts = []
    for _ in categories:
        counts.append(0)
    for v in values:
        ix = categories.index(v)
        counts[ix] = counts[ix]+1
    return counts
```

The first `for` loop creates a list of counters, all set at zero. To do this we loop through the categories we want to count (the different countries, in this case) and add one zero for each category. To do this we only need the for loop to give us the right number of iterations but we do not need the value of each item, so the iteration variable is merely an underscore. Note that this loop must finish before starting the next one, because we need to set up all the counters from the start.

The next `for` loop iterates through all the values to count (all the countries in the Wikipedia table), for each value finds the position in the categories list using the `index` method, and then increments the corresponding counter. Testing our function, we can check the result:

```
In : counts = count_per_category(countries,names)
Out: [1, 1, 1, 1, 4, 1, 1, 1, 1, 8, 21, 4, 1, 7, 1, 4, 1, 1, 1, 5, 5, 26, 69, 10, 1]
```

Now we bring everything together in the final function.

```
def laurates_per_country(url):
    """Returns string with laureates per country from wiki page"""
    r = requests.get(url)
    if not r.ok:
        return 'Error getting data'
    countries = all_countries(r.text)
    names = unique_countries(countries)
    counts = count_per_category(countries,names)
    table = ''
    for ix in range(len(names)):
        table = table + names[ix] +'\t'+str(counts[ix])+'\n'
    return table
```

This function attempts to retrieve the page from the URL provided. If the retrieval is not successful, then the function exits immediately with an error function. This we can check in the `ok` field of the response object returned by the `get` function. If everything goes well, then each of the previous functions is invoked in turn to process all necessary steps to obtain the list of different country names and the list of Nobel laureate counts. The last part of the function creates `table` with an empty string and then loop through the indexes of the names of the countries and create new lines on this string with the name and the corresponding count of Nobel laureates. Note that we separate the name and the count with a tab character (
`t`) and then terminate the line with the newline character (
`n`)

```
In : chemistry = laurates_per_country('https://...Nobel_laureates_in_Chemistry')
In : print(chemistry)
 Argentina      1
 Australia      1
 ...
 United States   69
 West Germany    10
 Yugoslavia     1
```

Note that the URL is truncated to fit the page.

Since Wikipedia uses the same table format for other Nobel categories, we can use the same code for other examples:

```
python_console:
  In : physics = laurates_per_country('https://...Nobel_laureates_in_Physics')
  In : print(physics)
   Australia      2
   Austria 3
   Austria-Hungary 1
   Belgium 1
   Canada  4
   ...
   United Kingdom  22
   United States   78
   West Germany    9
  In : medicine = laurates_per_country('https://...Nobel_laureates_in_Medicine')
  In : print(medicine)
   Argentina      2
   Australia      6
   ...
```

```
Norway  2
Portugal        1
...
Venezuela       1
West Germany    4
```

Chapter 10

While

Conditional iteration with the `while` *loop. Example: processing a Protein Data Bank (PDB) file.*

10.1 Conditional iteration

We have previously used the `for` loop to iterate a number of times that can be determined when the loop begins. However, sometimes we need to iterate without knowing in advance how many iterations we will need. In this case, we can use conditional iteration, with a `while` loop that iterates as long as a condition is evaluated as `True`.

Consider, for example, a function that determines how many times a number must be divided by another number until the result is less than one:

```
def count_divisions(number,divisor):
    "number of divisions until less than 1"
    count = 0
    while number>=1:
        number = number/divisor
        count = count +1
    return count
```

When encountering the `while` statement, the interpreter checks if the condition is true and, if it is — in this case, if `number` is at least 1 — then it executes the block of code indented below the `while` statement. After the block is executed, the condition is checked again and the block of code executed again if the condition is true. This is repeated as long as the condition is true when the check is made. Note that the interpreter only checks the condition immediately before executing the block of code inside the `while` loop.

We can test our function with some values:

```
In : count_divisions(2355,2)
Out: 12

In : count_divisions(2355,4)
Out: 6
```

Note that, unlike with the `for` loop, the `while` loop requires some care to avoid a situation where the condition never becomes false. Consider the following example:

```
In : count_divisions(2355,1)

...
```

In this case, since the `divisor` is 1, the number will never fall below 1 and so the loop never ends. If this happens, you can terminate the execution of your program by selecting the console and pressing Ctrl+C.

10.2 Processing a PDB file

10.3 Exercises

The Babylonian square root

A method of computing the square root of a number, attributed to the Babylonians, is based on noting that, $\sqrt{S}$ must fall between x and S/x, because if $x < \sqrt{S}$ then $S/x > \sqrt{S}$ and if $x > \sqrt{S}$ then $S/x < \sqrt{S}$. Thus, we can iteratively approximate $\sqrt{S}$ by computing:

$$x_0 \approx \sqrt{S} \qquad x_{n+1} = \frac{x_n + \frac{S}{x_n}}{2}$$

This way, as n increases, x_n will get closer to $\sqrt{S}$. The initial value x_0 does not need to be a good approximation of $\sqrt{S}$. For example, we can make $x_0 = S/2$.

Implement a function that receives as arguments the number whose square root we want to compute and a precision value. The function should compute x_n iteratively until the absolute value of the difference between x_n and x_{n+1} is no greater than the precision value. You can obtain the absolute of some number with the `abs` function in Python. Here is an example of how this function should work:

```
In : babylon(235,1)
Out: 15.336621701080201
In : babylon(235,0.1)
Out: 15.329711274319488
In : babylon(235,0.0001)
Out: 15.329709716755971
```

Computing π

The Newton/Euler Convergence formula allows us to compute an approximation to π from by adding a finite number of terms:

$$\frac{\pi}{2} = \sum_{k=0}^{\infty} \frac{2^k k!^2}{(2k+1)!}$$

Implement a function that computes pi to a given precision by adding consecutive terms of this series until the value of a term is no larger than the given precision.

Note that you need to compute the factorial of k for this function. First think about how to decompose this function into sub-problems before implementing everything.

Here is an example of running this function with different precision values:

```
In : newton_euler(1)
Out: 2.6666666666666665
In : newton_euler(0.1)
Out: 3.0476190476190474
In : newton_euler(0.01)
Out: 3.132156732156732
In : newton_euler(0.001)
Out: 3.140578169680336
In : newton_euler(0.0001)
Out: 3.1414796489611394
In : newton_euler(0.0000001)
Out: 3.141592479958223
```

Hetero groups in PDB files

In this URL there are some PDB files that you can download replacing `[ID]` with the PDB id code:

`http://iceb.ssdi.di.fct.unl.pt/1718/files/[ID].pdb`

The following PDB structures are available here: `1dxg, 1hrc, 1vxa, 1xer`. For this exercise use this URL and not the Protein Data Bank URL to avoid sending many requests to the PDB servers, which may result in a temporary ban.

In the PDB file, atoms that are part of the structure but are not amino acids in the protein chain are identified by the `HETATM`. The name of the group (residue, prosthetic group or other) each such atom belongs to is found in the slice 17:20 of the lines starting with `HETATM`.

Write a function that receives a list of PDB identifiers and prints on the console the identifier followed by a list with all the different groups containing hetero atoms in that PDB structure. It should work like this:

```
In : pdb_ids = ['1dxg','1hrc','1vxa','1xer']
In : print_heterogroups(pdb_ids)
1dxg [' FE', 'HOH']
1hrc ['ACE', 'HEM', 'HOH']
1vxa ['SO4', 'HEM', 'HOH']
1xer ['MLZ', ' ZN', 'F3S', 'HOH']
```

Note that there are different sub-problems to solve here. Think about how to structure your program first. You can adapt code from lectures 9 and 10 as necessary, but remember not to use the Protein Data Bank URL if you are doing this exercise in class (use the URL provided above instead).

Bibliography

[1] Mark Lutz. *Learning python*. " O'Reilly Media, Inc.", 2013.