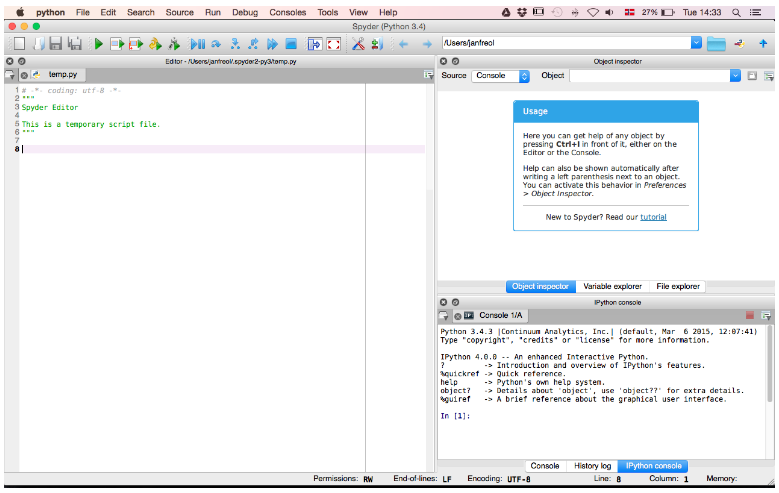
Appendix D

A crash course in Python

D.1 Introduction

How to install Python 3 and write your first program

Go to the webpage `https://www.spyder—ide.org` and download and install the latest version of Spyder. Once installed, launch this program. On a Mac (a few versions ago, at least) this would open the following window.



python File Edit Search Source Run Debug Consoles Tools View Help
27% Tue 14:33
Spyder (Python 3.4)
/Users/janfreol
Editor - /Users/janfreol/.spyder2-py3/temp.py
temp.py
Object inspector
Source Console Object
Object inspector Variable explorer File explorer
IPython console
Console 1/A
Console History log IPython console
Permissions: RW End-of-lines: LF Encoding: UTF-8 Line: 8 Column: 1 Memory:

```
# -*- coding: utf-8 -*-
"""
Spyder Editor

This is a temporary script file.
"""

```

Usage

Here you can get help of any object by pressing **Ctrl+I** in front of it, either on the Editor or the Console.

Help can also be shown automatically after writing a left parenthesis next to an object. You can activate this behavior in *Preferences > Object Inspector*.

New to Spyder? Read our tutorial

```
Python 3.4.3 |Continuum Analytics, Inc.| (default, Mar  6 2015, 12:07:41)
Type "copyright", "credits" or "license" for more information.

IPython 4.0.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.
%guiref   -> A brief reference about the graphical user interface.

In [1]:
```

Fig. 1. The graphical user interface of the Spyder editor for Python.

Exercise D.1 Type `print("Hello world")` in the editor window and press the green play button. What output do you get? Also try `print("2+2 = ", 2+2)`.

Remark: Notice how Python treats `"2+2"` *as text to be printed, and* `2+2` *as something to be computed, and that a comma is used to separate different type of input.*

A first look at basic syntax for arithmetic in Python

We now consider how to do basic arithmetic in Python. Here is a first example.

Example D.2 Enter the following code into the Spyder editor.

```python
1  a = 2
2  b = 4
3  c = a + b
4  print("a + b = ", c)
```

When you press the green play button you get the output `a + b = 6`.

This seems reasonable. Here, a, b and c are what is called variables. Technically speaking, a variable is a space in the memory of the computer which can store one piece of information, such as a number. In the above example, they work pretty much like what we would expect of a mathematical variable. But the next example shows that this is not always the case:

Example D.3 Enter the following code into the Spyder editor.

```python
a = 2
b = 4
c = a + b
a = 10
print("a + b = ", c)
```

When you press the green play button you still get the output `a + b = 6`.

Wait, what? Does Python really mean that $10 + 4 = 6$? Well, no. The program is doing exactly what it is told. The thing is that we, ourselves, do not really understand what we just asked Python to do. So let us try to figure it out. As an example. here is a line by line explanation of what happens in Example D.2:

Line 1: Python creates a *variable* `a` and assigns it the value 2.

Line 2: Python creates a *variable* `b` and assigns it the value 4.

Line 3: Python creates a *variable* `c`, computes the value $a + b$, and assigns this value to `c` .

Line 4: Python prints the value of `c` on the screen (together with the string of text `"a+b"`).

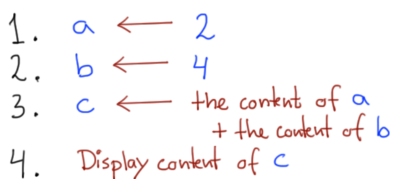


Fig. 2. To explain the code, we replace the equal signs by arrows.

The above example illustrates several peculiarities of arithmetic in Python and how a code is run:

1. The code is executed line by line.

2. Expressions such that `a = b` should be read from the right to the left. That is, `a` is assigned whatever value `b` has, but not vice versa (if `b` does not have a value, the program will (probably) crash). For this reason, an arrow makes more sense than an equal sign.

3. If a variable gets assigned some value, it has no memory of how that happened. That is, in the second example, `c` gets assigned the value `a+b`. But when `a` is changed in the next line, this does not affect the value stored in `c`.

Exercise D.4 Explore the various ways to do arithmetic operations in Python by running the following commands for suitable values of `a` and `b`:

(a) `a+b` (b) `a−b` (c) `a*b` (d) `a**b` (e) `a/b` (f) `a//b` (g) `a%b`

We now turn to a discussion of the basic syntax in Python. That is, what are the basic rules for how we are allowed to write a code? First, let us discuss the types of names you are allowed to give a variable:

Example D.5 You can give variables much more interesting names than just, say, `a`, `b`, `x` or `y`. Here is an example of a perfectly well-functioning program:

```
ponies = 2
cookies = 4
pony_Cookie32 = ponies + cookies
print(pony_Cookie32)
```

We make the following remarks:

4. Python is case sensitive. This means that `n` and `N` are as different as `n` and `m`.

5. By "tradition", the name of a variable should never start with an upper case letter.

6. A variable cannot be given a numerical name such as `2` or `34` (in particular, this means that the code `2 = a` will crash, since Python is trying to create a variable called `2` and assign it whatever value is stored in the variable `a`). However, numbers can be part of the name of a variable, as long as the name starts with a letter or an underscore '_'. In particular, `pony_Cookie32` is perfectly fine. Note that certain special symbols, such as `$`, `#` and `%` can never be used in the name of a variable.

7. The name of a variable cannot contain a "space". That is, you cannot give a variable the name `pony Cookie32`. Instead, you will have to use something like `pony_Cookie32` or just `ponyCookie32`.

8. You should not give a variable a name that already means something different. For instance, do not give a variable the name `print` (however, `Print` is fine).

Here is another example, followed by more comments:

Example D.6 In Python, we need to be careful with how we place indentations. For instance both of the following programs will crash.

```
a = 2
b = 4
  c = a + b
print(c)
```

```
        a = 2
        b = 4
        c = a + b
        print(c)
```

9. In Python we need to be careful with how we place indentations. If Python does not understand why we make some indentation, it will panic and crash.

10. Later in this chapter, we will see situations where indentations become a natural part of the code. This is when we use, for instance, `for`-loops, `while`-loops, the `if-else` structure, and when we define functions.

Here is one last example, followed by even more comments:

Example D.7 In contrast to the situation with indentations, Python is much less sensitive with respect to whether or not we skip a line. In particular, the following code will run just fine:

```
a = 2; b = 4     # Two commands on the same line.
c = (a + b)*3    # Here we use soft parentheses.

print(c)
```

11. Python does not care if you skip a line, or twenty.

12. You can place multiple commands on the same line, as long as they are separated by a semi-colon '`;`'.

13. Everything you write on a line following a hashtag `#` is ignored by Python. This allows programmers to write comments throughout a code to explain it (usually to themselves).

14. You can use "soft" parentheses '`(`' and '`)`' just as you would in a normal computation. However, these cannot be replaced by hard parentheses '`[`', '`]`' or curly parentheses '`{`', '`}`' as these all mean something different to Python.

15. It is a good thing for a program to crash. It is a way for Python of letting you know that whatever result the program would have given you would probably be false (since it was written in a bad way). What is much more dangerous is if a program is wrong because of, say, some mathematical mistake in some formula that still makes sense to Python (say, if a plus is mistakenly replaced by a minus). Then the code will run, and you will not be warned that something is wrong :-(

Exercise D.8 Will the following codes run? If not, why? If yes, what is the output?

(a)
```
a = 2
b = 4
6 = a + b
print(a+b)
```

(b)
```
a = 2
b = 4
a = a + b
print(a)
```

(c)
```
a = 2
b = 4
a = b
print(b)
```

(d)
```
a = 2
  b = 4
  c = a + b
print(c)
```

(e)
```
a = 2; b=4; a = a + b


print(b)
```

(f)
```
Ponies = 2
Cookies = 4
Rainbows = ponies + cookies
print(rainbows)
```

Exercise D.9 Consider the following codes. Can you express mathematically what that they compute?

(a)
```
a = 1
a = a + 1/2
a = a + 1/4
a = a + 1/8
a = a + 1/16

print(a)
```

(b)
```
a = 1
a = 1/(1+a)
a = 1/(1+a)
a = 1/(1+a)
a = 1/(1+a)

print(a)
```

Hint: You have already seen these objects on page 87.

A first look at how to efficiently repeat commands in Python

In exercise D.9 above, you were asked to consider a code where an operation was repeated multiple times. When programming, we sometimes want to ask the computer to repeat some operation perhaps a million or more number of times. Do we really need a million or more lines of code to do this. Well, of course not. We now explain one way this can be done using a so-called `for`-loop.

Example D.10 (For-loop) The following codes do exactly the same thing when run:

```python
a=0
print(a)
a=1
print(a)
a=2
print(a)
a=3
print(a)
print("the end!")
```

```python
1  for n in range(0,4):
2      a = n
3      print(a)
4  print("the end!")
```

We explain the right-most code:

Line 1: Here, the command `for n in range(0,4)` means that something is to be repeated once for each n between 0 and 3 (but not 4). What is to be repeated? Well, the code on every line following this one, and which has an indentation[1].

Lines 2 and 3: These lines are indented. This tells Python that they are to be run by the `for`-loop (it does not matter how much they are indented, as long as they have the same indentation). Specifically, lines 2 and 3 will first be run with `n = 0`, next with `n = 1`, and so on until `n = 3`. After this, the `for`-loop ends and Python moves on to the first unindented line following it.

Line 4: Python prints "the end!" as output, and the program ends.

Exercise D.11 Will these codes run? If so, what are their outputs?

(a)

```python
for n in range(0,4):
    print(n)
    print("mississippi")
print("hello")
```

(b)

```python
for n in range(0,4):
    print(n)
print("mississippi")
    print("hello")
```

Exercise D.12 Rewrite the code in exercise D.9 using `for`-loops.

[1]Note that it does not matter how large this indentation is, however, the indentation must be the same for all lines in the loop.

Some words on datatypes in Python

Python stores variables in different ways depending on whether they contain text, numbers or, say, lists. We say that Python uses different *datatypes*.

For instance, Python will treat a number in differently depending on whether or not it is an integer. Indeed, writing `a=2` stores the number 2 as the datatype `integer`, while writing $a = 2/3$ stores the fraction 2/3 as the datatype **`float`** (called *floating point numbers)*. The advantage of doing this is that it is much easier for a computer to deal with numbers that do not have long decimal expansions, which allow for programs to run quicker and use less memory. Moreover, as we will see in Section D.5, a computer cannot actually store an infinite number of decimals, which inevitably leads to round-off errors and lack of precision when doing computations with floating point numbers.

To check the datatype of a number we can use the built-in function **`type`**.

Example D.13

YouTube

```python
a = 2            # Here, we store the value 2 as an integer
print(type(a))   # Here, we check the type and tell Python to print
                 # the result on screen (it will print "int").
```

Exercise D.14 **(a)** Check the type of the variable defined by `b = 3/2`.
(b) Check the type of the variable defined by `c = 4/2`.

In addition to integers and floating point numbers, we will encounter the following datatypes in this chapter:

- **String:** A variable that contains a string of text is called a string. For instance, `a = "Hej"` will create a variable containing the string of text "Hej".
- **List:** A variable that contains a list of other variables is called... well, a list. For instance, `a = [2, 3/2, "Hej"]` creates a variable that contains an integer, a floating point number and a string. A list can even be made up of other lists (more on this on the following pages).
- **Numpy array:** A numpy array is a special type of list that can only contain either integers or floating point numbers (but not both). Since it is more specialised than the datatype list, this also means it can have more "advanced" features (more on this on page D-25).

Remark D.15 As opposed to many other programming languages, Python is rather forgiving when it comes to datatypes. For instance, when creating a new variable containing some number, Python will itself make a decision of whether or not this is to be an integer or a float.

How to store and manipulate sequences using Python lists

One way of storing sequences in Python is by using *lists*. Here are some examples of how to create and manipulate lists:

Example D.16

```
 1  a = [0,1,2,3,4,5]          # list of integers
 2  b = ["cheese", 88]         # list of a string and an icecream
 3
 4  print(a)         # prints the entire list
 5  print(a[0])      # prints 1st entry
 6  print(a[3])      # prints 4th entry
 7  print(a[-1])     # prints last entry
 8  print(a[-2])     # prints second to last entry
 9
10  len(a)           # computes the length of the list
11  sum(a)           # sums the terms of the list
12  max(a)           # returns the largest entry of the list
13  min(a)           # returns the smallest entry of the list
14
15  a.append("oi")   # adds entry "oi" to the end of the list
16  a.pop(3)         # deletes the 4th entry in the list
17  c = a+b          # creates the list [0,1,2,3,4,5,"cheese",88]
18  d = b*2          # creates the list ["cheese",88,"cheese",88]
19
20  e = a[2:5]       # creates the list [2,3,4] (called a "slice")
21  f = a[2:]        # creates the list [2,3,4,5]
22  g = a[:5]        # creates the list [0,1,2,3,4]
23  h = a[1:4:2]     # creates the list [1,3] (every second term)
```

The code is more or less explained by the comments, but we note the following:

Lines 1, 2: It is important that we use the square brackets '`[`' and '`]`' when creating a list and that we separate each entry of the list by a comma '`,`'.

Lines 5, 6, 7, 8: Here we show how to display various entries of a list. We also note that by writing, say, `a[3] = 10`, we change an entry in the list.

Lines 11, 12, 13: These only work if the list only contains numbers.

Lines 15, 16, 17, 18: There are other ways of representing sequences in Python, and these may have many of the above features. However, what makes lists stand out is how easy they are to modify.

Line 20, 21, 22, 23: This is called the *slice* notation for lists.

D.2 How to compute and visualize sequences and sums

Some ways to compute sequences in Python

Suppose we want to compute a lot of entries of the sequence

$$\left(\frac{1}{2^k}\right)_{k=0}^{\infty} = \left(1, \frac{1}{2}, \frac{1}{4}, \cdots\right).$$

First, we point out that it is useful to use **for**-loops (this example should be compared to Example D.10).

Example D.17 (Computing sequences using a for-loop) The following two codes do exactly the same thing when run:

```
print(1/2**0)
print(1/2**1)
print(1/2**2)
```

```
for k in range(0,3):
    print(1/2**k)
```

Another way of computing lists is to use a list commands called a *list comprehension*. It has the benefit of both computing and storing a sequence of numbers in just a single line of code.

Example D.18 (Computing sequences using a list) The following two codes do exactly the same thing when run:

YouTube

```
a = [1/2**0,1/2**1,1/2**2]
print(a)
```

```
a = [1/2**k for k in range(0,3)]
print(a)
```

Exercise D.19 Run and compare the output of the codes in the above examples.

Exercise D.20 A difference between the right-most codes in examples D.17 and D.18 is that in the first, we do *not* store the values of the sequence anywhere. Here are two suggestions for how to fix this:

```
a = 0
for k in range(0,3):
    a = 1/2**k
```

```
a = []
for k in range(0,3):
    a.append(1/2**k)
```

Explain what is going in each code. What information is stored at the end of each program? Also, why do these not give any output? Can you fix this?

Some ways to compute partial sums of infinite series

On the previous page, we discussed how to compute and store values of a sequence. Let us now consider how to compute the sum of the first, say, million terms of the infinite series

$$\sum_{k=0}^{\infty} \frac{1}{2^k} = 1 + \frac{1}{2} + \frac{1}{4} + \cdots .$$

As above, we are going to discuss several ways that this can be done.

This first example should be compared to part (a) of exercise D.9.

Example D.21 (Computing sums using a for-loop) The following two codes both compute the sum

$$1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8}.$$

```python
a = 0
a = a + 1/2**0
a = a + 1/2**1
a = a + 1/2**2
```

```python
a = 0
for k in range(0,3):
    a = a + 1/2**k
```

To understand the code in the above example, recall Figure 2 on page D-2, and read the explanation following Example D.10.

Here is how to compute the same sum using lists and list comprehensions.

Example D.22 (Computing a sum using lists) The following two codes do exactly the same thing when run:

```python
a = [1/2**0,1/2**1,1/2**2]
b = sum(a)
```

```python
a = [1/2**k for k in range(0,3)]
b = sum(a)
```

Note that using the technique of this example, we could actually compute the sum in just one line (!):

```python
b = sum([1/2**k for k in range(0,3)])
```

Exercise D.23 (a) Run the codes from the examples on this page. Why do they not give any output? Fix this.

(b) Modify (some of) the code on this page so that you can compute the sum of the first million terms or so. What result do you get?

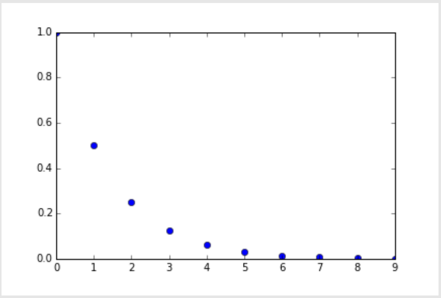
How to visualise sequences and partial sums using for-loops

We now consider how to visualise the data we computed on the previous pages. This is the first point where we will see that for-loops are more flexible, and gives code that is easier to read, than list comprehensions[2].

Example D.24 The following code is perhaps the simplest way to visualise a sequence in Python. Here, we visualise the sequence

$$\left(\frac{1}{2^k}\right)_{k=0}^{9}.$$

```
1  import matplotlib.pyplot as plt
2
3  for k in range(0,10):
4      plt.plot(k,1/2**k,"bo")
5
6  plt.show()
```



To the right, we see the output.

Here is how this code works:

Line 1: We import the "package" `matplotlib.pyplot`. At this point, we do not really have to know what this means. But, in short, this package provides Python with additional commands that help us visualise stuff. Adding `as plt` allows us to refer to this package by the shorter name `plt`.

Lines 3, 4: Here we run a **`for`**-loop. At each iteration, it asks Python to plot a blue filled circle (this is what the `"bo"` means) at the coordinate $(k,1/2^k)$.

Line 6: The command `plt.show()` tells Python that we are done with our figure, and that it is time to display it. Any subsequent use of `plt.plot` will create a new figure to be displayed separately.

Let us now immediately modify the above code, so that we can plot partial sums of the infinite series

$$\sum_{k=0}^{\infty} \frac{1}{2^k}.$$

That is, we want to plot the first few we get when we compute

$$\sum_{k=0}^{0} \frac{1}{2^k} = 1, \qquad \sum_{k=0}^{1} \frac{1}{2^k} = 1 + \frac{1}{2}, \qquad \sum_{k=0}^{2} \frac{1}{2^k} = 1 + \frac{1}{2} + \frac{1}{4},$$

[2]We show how to visualise data stored as lists on page D-13.

and so on. The nice thing is that we can achieve this by a relatively minor modification of the above code. Notice how this code combines what we did in examples D.21 and D.24.

Example D.25 The following code computes and visualises exactly the partial sums expressed above.

```python
import matplotlib.pyplot as plt

a = 0

for k in range(0,3):
    a = a + 1/2**k
    plt.plot(k,a,"bo")

plt.show()
```

Exercise D.26 Run this code, and verify by computing the first three partial sums by hand that the plot is correct. What happens when you increase `range(0,3)` to, say, `range(0,100)`?

Exercise D.27 Several commands can be used to change how the above figure looks. Try inserting the following into the code. This can be done anywhere after the `import` and before `plt.plot()`. What happens?

- **(a)** Replace `"bo"` by `"rx"`.
- **(b)** `plt.xlim(−3,15)`
- **(c)** `plt.ylim(−5,2)`
- **(d)** `plt.xlabel("There was")`
- **(e)** `plt.ylabel("a graph")`
- **(f)** `plt.title("that had a title")`
- **(g)** `plt.xticks([−2,0,3,4,6.5,10])`
- **(h)** `plt.yticks([0,1,1.5])`
- **(i)** `plt.grid(True)`
- **(j)** `plt.figure(figsize=(4,3))`
- **(k)** `plt.savefig("myfigure.jpg")`

Remark: Note that by choosing the extension ".jpg" in (k), you actually tell Python to save your figure as a jpg-file. Note that only a limited number of file formats are supported. For more on how to plot in Python, check out the official tutorial `https://matplotlib.org/users/pyplot_tutorial.html` *which has a ton of information and examples.*

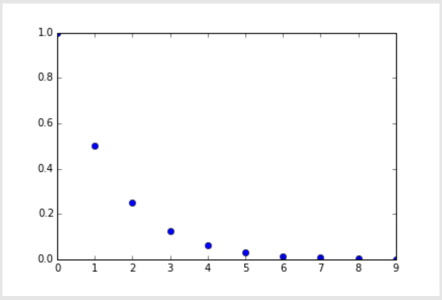
Visualising sequences and partial sums stored as lists

We now give an example where we visualize a sequence stored as a list.

Example D.28 We now use lists to create and plot the sequence

$$\left(\frac{1}{2^n}\right)_{n=0}^{9}.$$

```
1  import matplotlib.pyplot as plt
2
3  nValues = [n for n in range(0,10)]
4  a = [2**(−n) for n in nValues]
5
6  plt.plot(nValues,a,"bo")
7  plt.show()
```



To the right, we see the output. Note that it matches exactly that of Example D.24.

While this example is quite similar to Example D.24, let us explain what happens in lines 3, 4 and 6 a bit more carefully:

Lines 3, 4: Here we create two lists. The first is `nValues = [0, 1, 2, ..., 9]` which will give us the n-values to be used in the plot (these will be placed on the x-axis). The second list, `a`, contains the sequence we are trying to plot. These will play the part of the y-values in our plot. Note that in the `for` command, we can replace the `range` command by any other list. This is a particular, and rather elegant, feature of Python.

Line 6: Here, we are creating the plot itself. In the command `plt.plot(nValues,a,"bo")`, the we use the two lists created in lines 3 and 4. The first will be interpreted as the x-coordinates to be used, and the second as their corresponding y-coordinates. Note that it is therefore crucial that these two lists are of the same length (if not, Python will crash), and explains why it is a good idea to let the `for`-loop in line 4 be defined in terms of `n` running through `nValues`, as this will guarantee that the lists `nValues` and `a` have the same length.

We now give an example of how to visualise partial sums using lists.

Example D.29 As in Example D.25, we consider the infinite series

$$\sum_{k=0}^{\infty} \frac{1}{2^k}.$$

The following code should be compared to that of Example D.28, above. It will compute the 19 (!) first partial sums of the series. Here, we use the name "indices" instead of "nValues", and vary the letter used for the index from line to line to emphasise that the choice of letter really does not matter.

```
1  import matplotlib.pyplot as plt
2
3  indices = [i for i in range(0,20)]
4  a = [1/2**k for k in indices]
5
6  S = [sum(a[0:n+1]) for n in indices]
7
8  plt.plot(indices,S,"bo")
9  plt.show()
```

To understand this code, you should first read the explanation for Example D.28. The difference is what happens in line 6:

> **Line 6:** Here, we use the sum command to sum slices (see Example D.16) of the list. Here, you should keep in mind that the slice, say, `a[0:19]` gives you the entries `a[0]`, `a[1]`, ..., `a[18]`. In particular, there is no point in computing `a[0:0]` as this slice contains no terms. More explicitly:

$$\begin{aligned}
n = 0 &\implies \texttt{sum}(a[0:0+1]) = \texttt{sum}(a[0:1]) = \sum_{k=0}^{0} \frac{1}{2^k},\\
n = 1 &\implies \texttt{sum}(a[0:1+1]) = \texttt{sum}(a[0:2]) = \sum_{k=0}^{1} \frac{1}{2^k},\\
&\vdots\\
n = 19 &\implies \texttt{sum}(a[0:19+1]) = \texttt{sum}(a[0:20]) = \sum_{k=0}^{19} \frac{1}{2^k}.
\end{aligned}$$

Exercise D.30 Implement the code in Example D.29. Verify that it gives the same output as Example D.25 when the range in the latter example is suitably adjusted.

A slightly more advanced example: the Fibonacci sequence

Above we have considered examples of sequences and partial sums that can be computed both using "pure" **for**-loops or **for**-loops inside of lists (i.e., list comprehensions). Let us now consider a situation where we need to adapt these methods slightly, and use the best of both worlds.

Namely, we consider the sequence

$$(1,1,2,3,5,8,13,\ldots).$$

These are the Fibonacci numbers. In general, the n'th Fibonacci number is given by continuing this list using the rule

$$a_n = a_{n-1} + a_{n-2}.$$

Note that if we intend for the sequence to start at a_0, then this rule cannot be used to compute a_0 or a_1, since they would then depend on a_{-2} and a_{-1}. Therefore, it is more correct to say that the Fibonacci numbers are created from the set of rules:

$$\begin{cases} a_0 = 1, \\ a_1 = 1, \\ a_n = a_{n-1} + a_{n-2} \quad \text{for } n \geq 2. \end{cases}$$

So, how to create a list in Python containing, say, the 20 first Fibonacci numbers? Well, this cannot be done by using a command on the form

```
[??? for n in range(0,21)],
```

since there is no way to let the n'th number depend on the previous numbers in the sequence[3]. What to do? Well, here we use a "pure" **for**-loop, where we store the Fibonacci numbers, as they are computed, in a list so that we can use previous Fibonacci numbers to compute the next numbers.

Example D.31 The following code can be used to compute Fibonacci numbers.

```python
a = [1,1]

for n in range(2,20):
    newterm = a[n-1] + a[n-2]
    a.append(newterm)

for n in range(0,20):
    print("The", n+1,"'th Fibonacci number is", a[n])
```

[3]Well, strictly speaking, there is, but there is no "natural" way to do this in Python.

Exercise D.32 Implement the code from the above example. Use, say, the wiki-page for the Fibonacci numbers to verify that it gives the correct output.

Exercise D.33 Use the method of the above examples to create a list containing a few partial sums of the infinite series $\sum_{k=0}^{\infty} 1/2^k$.

Remark D.34 ("pure" `for`-loops versus list comprehensions) Above, we saw an example of a situation where the flexibility of "pure" `for`-loops made them easier to use than the more rigid list comprehensions. Does this mean list comprehensions are less useful? Well, sort of. In addition, "pure" `for`-loops are available in most programming languages, while comprehensions are really specific for Python. This means that if you want to be able to adapt to other programming languages, you should use list comprehensions sparingly.

Another feature of Python is that you can measure how long it takes for a program to run. In this way, you can time how long it takes Python to run a limited number of iterations of some for-loop, and then you can make an informed guesstimate of how long it will take to run, say, a million iterations. Here is how this is done:

Remark D.35 (how to time a computation)

```python
import time
time_start = time.process_time()
# your code
time_elapsed = (time.process_time() - time_start)
print(time_elapsed)
```

Exercise D.36 (a) What does the code to the right compute? **(b)** Do the same as in (a), except do it by first getting Python to create a suitable sequence `a`, and then compute its sum using the command `sum(a)`. **(c)** Use the code from the above remark to compute which method is the fastest.

```python
S = 0
for n in range(1,100000):
    S = S + 1/n
```

Remark D.37 Python is a useful, but fairly slow, programming language. What you are supposed to observe in the previous exercise is that the built-in commands in Python actually are written using much faster languages, such as C++. The morale is: if speed is important, use the built-in functions as much as possible.

D.3 Some additional control statements in Python

In the previous section, we considered `for`-loops. Here, we consider some additional control statements in Python.

While-loops

The `while`-loop is a close cousin of the `for`-loop. Indeed, in most (if not all) cases, what you can do with one of them, you can also do with the other. The point is that in most situations, one of them will usually be much easier to use than the other.

Example D.38 (partial sums with a while-loop) The following code computes the sum

$$\sum_{n=1}^{10} \frac{1}{n} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{10}.$$

```
1  S = 0; n = 1
2  while n <= 10:
3      S = S + 1/n
4      n = n+1
5  print(S)
```

We explain the code:

Line 1: We initialise the variables S and n. (Recall that code separated by a semi-colon is treated as if it was written on separate lines.)

Line 2: We start the while-loop. By writing `while n <= 10:` we tell Python that the code inside of the while-loop (that is, the indented lines) should be repeated until the variable `n` is no longer *less than or equal to* 10.

Line 3: The first time the while-loop is executed, we have `S=0` and `n=1`. This means that the line `S = S + 1/n` assigns `S` the value `1`.

Line 4: We increase the value of n by 1. (This is something we did not have to do in the for-loop.) Python now takes this, new, value of n, jumps up to line 2, and checks if it is less than or equal to 10. If it is, the while-loop runs again, if not, it jumps down to the first non-indented line (line 5).

Remark D.39 (logical operators) In the above example, we see the symbol <=. This is a logical operator that checks if something is *less than or equal to* something else (note that it is important to remember the order of the symbols since =< means nothing to Python). When doing while-loops, the symbols >=, == and != are also useful, where the latter two checks if two variables are equal or not equal, respectively (note that round-off errors usually makes it impossible for Python to check if two numbers that are *not* integers are equal – more on this in Section D.5).

Exercise D.40 Use a while-loop to check how large n has to be for the partial sums $S_n = \sum_{k=0}^{n} 1/2^k$ is closer to 2 than 1/10000.

If-else statements

Alongside **for**- and **while**-loops, the **if-else** statement is the most important tool in programming.

Here is a basic example:

Example D.41 (if-statements in Python)

```
1 a = 10
2
3 if a >= 3:
4     print("a is a BIG number")
5 else:
6     print("a is a tiny number")
```

Here is an explanation of the code:

Lines 3 to 6: Like the while-loop, an if-statement starts out by checking if some condition is true or not. Here, if the condition is true, line 3 is run. If the condition is false, then line 5 is run.

If you want to add more conditions, you can do this by adding as many **elif** commands as you want (notice how we are allowed to use the word **and** to combine two inequalities in our condition – it is also possible to use the keyword **or**):

Example D.42

```
a = 10

if a >= 3:
    print("a is a BIG number")
elif a > -3 and a < 3:
    print("a is a tiny number")
elif a <= -3:
    print("a is a BIG but negative number")
```

Exercise D.43 Use an **if**-type statement to modify the code from Example D.31 so that 1'th, 2'th and 3'th are replaced by 1'st, 2'nd and 3'rd, respectively.

The break command

Another keyword we can combine with if-statements (and while-loops) is **`break`**. This command tells Python to stop the if-statement (or while-loop) and to continue the program on the first unindented line below it. Since we will not need it, we refer the interested reader to Google for more further information.

Here is an example where we combine the **`if`**, **`break`** and **`for`** keywords to mimic a **`while`**-loop.

Example D.44 The following code does exactly the same as the one in Example D.38

```
1  S = 0
2  for n in range(1,1000):
3      if n > 10:
4          break
5      S = S + 1/n
6  print(S)
```

Let us briefly explain this code:

Line 1: We initialise an integer `S` (that we will use to keep track of partial sums).

Line 2: Here we start out the for-loop. We choose `range(0,1000)` large to ensure that the break command, further down, will have time to kick in.

Line 3: We are now inside of the for-loop. Here, we ask the if-command to check if $n > 10$. If this is not the case, the code will skip the indented lines and continue on line 5. If $n > 10$ is true, then the indented code on line 4 will be run.

Line 4: Here, the break command is activated. It means that the for-loop is stopped. The code will continue on line 8.

Line 5: Here, the value of `S` is updated.

Remark D.45 Sometimes we need to put for-loops inside of for-loops (when computing matrices, this may be the case). When this happens, the break command will only stop the "inner-most" loop.

Exercise D.46 Consider the list `myList = [n/(1+n**2) for n in range(0,10**6)]`.

(a) Write a code that combines a for-loop with the break command to check when the first entry in the list is smaller than 10^{-4}.

(b) Do the same as in (a), but with a while-loop and no break command.

D.4 Functions in Python

In this section, we will mostly limit ourselves to discussing *mathematical* functions in Python.

How to define a function in Python

Here is a basic example showing how to define a function in Python.

Example D.47 (Defining functions in Python) In the following code we define $f(x) = x^2 - 5x + 4$ in Python and compute the value $f(3)$.

```
1  def f(x):
2      y =  x**2-5*x+4
3      return y
4
5  z = f(3)
6  print(z)
```

Here is what happens:

Line 1: We write `def f(x):` to let Python know that we are now about to define a function that has the name `f` and that takes one variable, given the name `x`, as input. It is absolute critical to understand that, at this point, Python will not execute this code. Instead, we are just telling Python that if the function `f` is used at any subsequent point in the code, then this is the code that should be executed.

Line 2: The variable `y` is assigned the value `x**2-5*x+4`.

Line 3: The value of `y` is *returned* as the output of `f`.

Lines 5 and 6: In line 5, we ask Python to apply the function `f` to the value `3`. This means that Python will run the code in lines 1, 2 and 3 with `x = 3`. When line 3 is executed, the *returned* value will be stored in the variable `z`. In line 6, this value is printed on screen.

Exercise D.48 **(a)** Use the `if`-control statement to implement the absolute value function. Use it on a few values to check that it works.

(b) Implement the absolute value function without using the `if`-control statement.

Remark: It is not really necessary to program the absolute value function by hand since it already exists in Python as the function `abs(x)`.

Exercise D.49 Write the code for a function that will take a list of numbers as input, and return their product as output.

Remark: This will be analogue to the function `sum()` *which takes a list of numbers as input and returns their sum as output.*

Warning: The importance of local namespaces

We face some dangers when defining functions in Python. First, we need to be aware that the variables appearing inside the function definition are *local* and cannot be accessed outside of the definition. Since this is a subtle point that leads to a lot of bad code, we are going to spend a little bit of time discussing it.

Here is a first example:

Example D.50 The following code makes no sense and will crash.

```
1  x = 3
2  def f(x):
3      y = x**2-5*x+4
4      return y
5  print(y)
```

Here is what happens:

Line 1: We define a variable `x` and set its value to `3`.

Lines 2, 3, 4: We define the function `f`. However, this code is never run.

Lines 5: Here, Python becomes confused and crashes *since no variable called* `y` was ever created.

While the above explanation may seem to make perfect sense, it is actually sort of misleading. To understand why, let us take a look at a second example.

Example D.51 The following code makes no sense to Python and will crash.

```
1  x = 3
2  def f(x):
3      y = x**2-5*x+4
4      return y
5  f(x)
6  print(y)
```

Take a moment and think about what happens when this code runs. Hopefully, it will confuse you since the error is rather subtle. Here is the explanation:

Lines 1 – 4: This is the same as the code in the previous example. For clarity, we reiterate that the code in lines 2, 3, 4 is not run at this stage.

Lines 5, 6: In line 5, finally, the code in lines 2, 3, 4 is run. Since we put `x = 3` in line 1, it is run with the value `x = 3`. In line 3, `y` gets the value -2, and in line 4, this value is *returned*. The program moves on to line 6, where it tries to print the value of the variable `y`. *But this variable has never been created*, and so Python crashes.

Wait, what? Surely, this makes no sense since the variable `y` was created in line 4 and given the value -2. Well, no: the point is that all variables created in lines 3, 4, 5 are *local* and only exist when these lines are run. After the code is done running line 5, all of these local variables are deleted.

Maybe things become clearer if we rewrite the explanation of Lines 7, 8 as follows:

Lines 7, 8: In line 7, finally, the code in lines 3, 4, 5 is run. Since we put `x = 3` in line 1, the command `f(x)` is read by Python as `f(3)`. Now, the variable `x` in line 3 is not the same as the variable `x` from line 1, instead, we should think of it has having some other name, say, `local_x`. Since the command `f(3)` activated line 3, the first thing that happens is that Python sets `local_x = 3`. Next, in line 4, the variable is not really `y`, it is `local_y`, which gets the value -2. Finally, in line 5, this value is *returned*. This means that `f(x)` in line 7 now represents the value -2. However, since this value is all alone on this line, nothing is done to it, and it is simply forgotten (we could have written, say, `z = f(x)` to store it). The program merrily continues on line 8, where it tries to print the value of the variable `y`. *But this variable has never been created*, and so Python crashes. (Note that as Python jumped from line 5 to line 7, the variables `local_x` and `local_y` are deleted from memory.)

In technical terms, it is said that the code inside of the definition of a function has its own local *namespace* which cannot be called upon in the code outside of the function. This is done to "protect" the program from the code inside of the function. To understand why this is necessary, let us look at the following example.

Example D.52 Thanks to functions having their own namespace, we can be sure that the following code works:

```
a = [1, 2, 3]; b = [4, 5, 6]
c = sum(a)
print(b)
```

The point is that we have no idea how the function `sum(a)` is coded. If the local namespace was not kept separate from the (global) namespace, we could be so unlucky that a variable called `b` is used inside of the code for the function. This would then overwrite the contents of the variable `b` that we created before running `sum(a)`. The point of having a local namespace is to avoid this and to keep our variables safe from harm. Yay!

How to plot functions in Python using lists

Here, we demonstrate the first out of two methods for plotting functions in Python. This method is really similar to what we did when we plotted sequences stored as lists (recall Example D.28).

Example D.53 (plotting with datatype list) The following code plots the graph of $f(x) = x^2 + 2x + 3$ over the interval [0,1].

```
1   import matplotlib.pyplot as plt
2
3   def f(x):
4       y = x**2 + 2*x + 3
5       return y
6
7   X = [n/10 for n in range(0,11)]
8   Y = [f(x) for x in X]
9
10  plt.plot(X,Y)
```

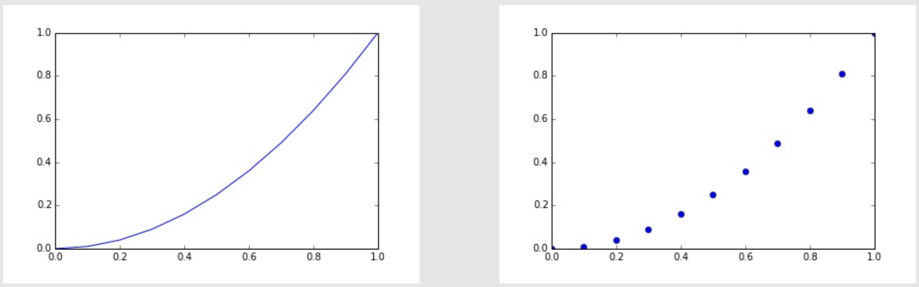


Fig. 3. To the left, we see the output of the above code. To the right, we see the output if we replace the line `plt.plot(X,Y)` with `plt.plot(X,Y,"bo")`.

Here, we explain the code.

Lines 1 – 5: Here, we import the package `matplotlib.pyplot` and define the function `f`.

Line 7: Here, we create the list `[0, 1/10, 2/10, ..., 9/10, 1]`. This list will play the part of the x-axis.

Line 8: Here, we create the list `[f(0), f(1/10), f(2/10), ..., f(9/10), f(1)]`. This list provides the y-values given by $y = f(x)$ for each `x` in the list `X`. Notice how we can replace `range(0,11)` in the `for`-statement with other lists.

Line 10: The command `plt.plot(X,Y)` is very similar to the one appearing when we plotted the sequence in Example D.28. It takes two lists, in this case $X = [x_0, x_1, \ldots]$ and $Y = [y_0, y_1, \ldots]$ as input. It then draws a straight (blue) line from the point (x_0, y_0) to (x_1,y_1). And then from (x_1,y_1) to (x_2,y_2) and so on until it runs out of points. If the lists X and Y are not of equal length, the program gets confused and crashes. If we add the option `"bo"` to the command `plt.plot(X,Y)`, we tell Python not to draw lines, and instead just put blue dots, as we have done before (and as is shown above)!

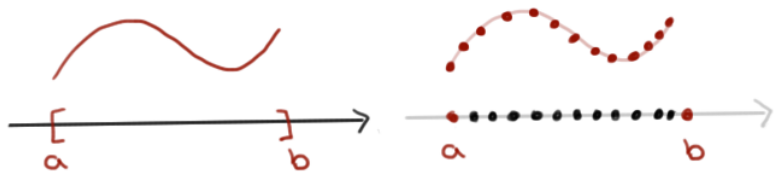


Fig. 4. We are never really plotting the graph of a function f based on all its values on an interval $[a,b]$. In reality, we only check the y-values for certain x-values and then ask Python to connect the dots.

Exercise D.54 Plot the functions you implemented in exercise D.48 on $[-2,2]$.

Exercise D.55 Consider the following code.

```python
import matplotlib.pyplot as plt

N = 10

def a(k):
    return 1/2**k

def S(a,n):
    return  sum([a(k) for k in range(0,n)])

nValues = [n for n in range(0,N)]
Y = [S(a,n) for n in nValues]

plt.plot(nValues,Y, "bo")
```

(a) Explain in mathematical term what this code does.

(b) The symbol `a` takes more than one role in the above code. Explain what roles these are.

(c) Make the code easier to read by giving `a` different names where possible.

Plotting functions using numpy arrays

We now give an alternative, and rather elegant, way to plot functions. The price to pay is that we need to introduce a new datatype commonly referred to as `numpy arrays` (strictly speaking, Python calls them `numpy.ndarray`'s, but we will ignore this).

In a sense, numpy arrays are just like lists, but with the restriction that they can consist of numbers (lists can also consist of, say, strings of text). But this means that it makes sense for numpy arrays to have additional features related to numbers.

Example D.56 (features of the datatype numpy array) The following code is meant to illustrate some of the things we can do with numpy arrays.

```
 1  import numpy as np
 2
 3  def f(x):
 4      return x**2
 5
 6  a = [1,2,3,4]; b = [1,2,5,3]
 7
 8  A = np.array(a)  # Here we convert the lists a and b into numpy
 9  B = np.array(b)  # arrays. Usually, arrays are expressed like lists
10                   # but without commas. E.g., now A = [1 2 3 4].
11
12  C = A*B     # This results in C = [1 4 15 12].
13  D = A+B     # This results in D = [2 4 8 7].
14  E = A/B     # This results in... well, it divides the two lists,
15              # entry by entry :-)
16
17  F = np.zeros((3))        # Creates the array [0 0 0]
18  G = np.ones((4))         # Creates the array [1 1 1 1]
19  H = np.arange(4)         # Creates the array [0 1 2 3]
20  I = np.linspace(0,1,5)   # Creates the array [0 0.25 0.5 0.75 1]
21
22  J = I.tolist()  # Converts the numpy array I to a list J.
23  K = f(A)        # this results in K = [1 4 9 16]
```

Let us make the following comments with respect to the above example:

Line 1: Before we can use numpy arrays we have to import the package `numpy`.

Lines 8, 9, 22: We can always convert a list (with only numerical entries) to a numpy array, and vice versa. When creating a numpy array, normally we would just write, say, `A = np.array([1,2,3,4])`.

Line 20: The command `np.linspace(a,b,N)` creates an array with N equally spaced points from the interval $[a,b]$, starting and ending at the left and right endpoints, respectively. This command is very useful when we want to plot functions!

Line 23: When applying a mathematical function `f` to the numpy array `A = [01 2 3]`, what happens is that we obtain the array `J = f(A) = [f(0) f(1) f(2) f(3)]`. This is also very useful when we want to plot functions!

Here is how to plot functions using numpy arrays.

Example D.57 The following code gives exactly the same output as the code in Example D.53.

```
import matplotlib.pyplot as plt
import numpy as np

def f(x):
    y = x**2 + 2*x + 3
    return y

X = np.linspace(0,1,11)
Y = f(X)

plt.plot(X,Y)
```

Exercise D.58 Try to use numpy arrays to plot the functions from exercise D.48. For one of them, the code will not work. Can you imagine why?

Predefined functions in Python

We now describe some of the pre-defined function that comes with Python. Usually, it is a good idea to use these whenever you can since they are optimised to run fast using techniques way outside the scope of these lecture notes.

Remark D.59 (Built in functions in Python) Here is a list of some of the functions that are built into Python.

- `abs(x)` – computes the absolute value of `x`.
- `complex(a,b)` – returns the complex number $a + ib$.
- `float(x)` – converts integer to a float.
- `int(x)` – convertes float to integer (rounds down to nearest integer).
- `round(a,n)` – rounds the floating point number a to its n first digits.
- `type(x)` – returns the datatype of the variable `x`.

For the full list, see, e.g., https://docs.python.org/3/library/functions.html.

Additional functions can be imported from packages. For instance, here is a list of some functions from the `numpy` (numerical Python) package:

Remark D.60 (Functions in the numpy package) To use these functions, you need to start your code with `import numpy as np`. You now have access to the following functions:

- `np.exp(x)` – the exponential function
- `np.log(x)` – the natural logarithm
- `np.log2(x)` – the logarithm with base 2
- `np.log10(x)` – the logarithm with base 10
- `np.sin(x)` – the sine function (radians)
- `np.cos(x)` – the cosine function (radians)
- `np.tan(x)` – the tangent function (radians)
- `np.arcsin(x)` – the arcsine
- `np.arccos(x)` – the arccosine
- `np.arctan(x)` – the arctangent

Here are some other useful functions:

- `np.absolute(x)` – gives the absolute value of x
- `np.deg2rad(x)` – converts degrees into radians
- `np.rad2deg(x)` – converts radians into degrees

- `np.sum(x)` – returns the sum of the elements of a list/array
- `np.prod(x)`– returns the product of the elements of a list/array
- `np.imag(z)` – returns imaginary part of the complex number z.
- `np.real(z)` – returns real part of the complex number z.
- `np.angle(z)` – returns the angle of the complex number z (radians).
- `np.conj(z)` – returns the complex conjugate of the complex number z.

Here are some constants:

- `np.pi` – gives π
- `np.e` – gives e

For more, see https://docs.scipy.org/doc/numpy-1.13.0/reference/routines.math.html

Here is a brief example of how to use the `numpy` package.

Example D.61

```
import numpy as np
y = np.sin(np.pi)     # Computes sin(pi).
print(y)              # Prints the result on screen.
```

There are also other packages with even more functions. We can mention the `math` and `scipy` (scientific python) packages. However, since we are quite happy with what we have mentioned above, we will skip these. (In particular, the `numpy` package essentially makes the `math` package obsolete – especially when working with numpy arrays)

D.5 How numbers are represented in Python

We end this appendix by examining how numbers are represented in Python and some peculiar consequences of this.

Example D.62 At the start of Chapter 3, we considered the infinite series

$$\sum_{k=0}^{\infty} \frac{1}{2^k} = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots .$$

Letting S_n denote the n'th partial sum of this infinite series, the point of exercise D.5 was for you to notice that

$$2 - S_n = \frac{1}{2^n}.$$

In particular, for all $n \in \mathbb{N}$, we have $S_n \neq 2$.

But here is the thing: if if we ask Python to compute, say, S_{100}, then Python will give the output $S_{100} = 2$. But this is wrong! While S_{100} is close to 2, it is not equal to 2. Yikes!

Exercise D.63 Does Python really mean that $S_{100} = 2$, or is there something else going on? One way to double check this is to ask Python to compute $1/(2 - S_{100})$. What happens?

So, what is going on here? Well, the point is that since there is a limit to how much memory Python is willing to use to represent a number, there is also a limit to how precisely Python will represent that number. This inevitably leads to so-called *round-off errors* when working with computers, and it is vital that have some understanding of why they occur.

How integers are represented in Python

The first question we ask is the following, what exactly happens when we run, say:

```python
myNumber = 82
```

If the code is run on an 8 bit computer, the following happens:

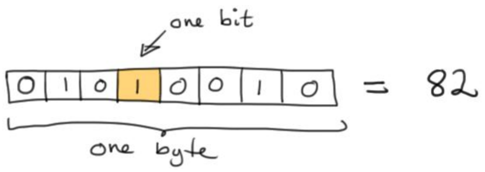


Fig. 5. On the most fundamental level, the memory of the computer is described in terms of bits and bytes. A bit can be either 0 or 1, while a byte is a string of 8 bits. Modern computers normally set aside 64 bits for each integer.

That is, Python sets aside a certain amount of memory (depending on the type of computer you run it on), and use it to store your integer. By giving it a name such as `myNumber`, we know how to access this part of the memory, and by giving it a *datatype*, Python will knows how to interpret the of 0's and 1's located there. Integers are usually stored using variables of the datatype `int` (short for integer).

Remark D.64 Inside the circuits of a computer, the 1's are represented by a short pulse of electricity, while the 0's are represented by the lack of such a pulse. Now, it would make sense to design the circuitry of a computer so that every integer between 0 and 9 was represented. Indeed, one could model the integers by using varying intensities of the pulse. However, a reason for just working with 0's and 1's is that this reduces the chance that some disturbance will make the computer mistake one value for another.

This leads to the following mathematical question: how to represent all integers using only 0's and 1's? The thing to realise is that this is not so different from the following question: how to represent all integers using only strings of digits from the list $\{0,1,2,3,\ldots,9\}$?

Example D.65 (Decimal and binary notation) So what do we really mean by the integer 4132? Well, this:

$$\begin{aligned} 4132 &= 4 \cdot 1000 + 1 \cdot 100 + 3 \cdot 10 + 2 \cdot 1 \\ &= 4 \cdot 10^3 + 1 \cdot 10^2 + 3 \cdot 10^1 + 2 \cdot 10^0. \end{aligned}$$

(Here, we included the second line to emphasise how the powers of 10 occur in this expression.) In fact, we are so used to thinking about integers as decimal numbers that it is completely obvious for us that we can represent all numbers in this way.

But how to represent integers just using a string of 0's and 1's? For instance, what number should the string, say, 1101 represent? Well, here is the basic idea:

$$\begin{aligned} 1101 &= 1 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 \\ &= 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0. \end{aligned}$$

When 1101 is interpreted in this way, we call it a binary number. (Again, we include the second line to emphasise how the powers of 2 occur in this expression.)

Let us now consider two questions: 1) why is the basic idea shown in the above example actually quite reasonable, and 2) how to know if the number 1101 is supposed to be interpreted in the above sense (i.e., as a binary number) and not as the decimal number *one-thousand-one-hundred-and-one*?

To answer the second question first: when it is not clear if we are talking about binary or decimal representations of numbers, we can use the notations $(1101)_2$ and $(1101)_{10}$ to indicate that we mean binary or decimal notation, respectively.

But what about the first question? Well, using the idea shown above, here is how counting in binary works:

$$\begin{array}{rcr}
(0)_2 = 0 & (11)_2 = 3 & (110)_2 = 6 \\
(1)_2 = 1 & (100)_2 = 4 & (111)_2 = 7 \\
(10)_2 = 2 & (101)_2 = 5 & (1000)_2 = 8
\end{array}$$

Notice that this is exactly how counting with two digits should work! Every time we run out of digits, we start over by including an extra zero and carrying over a one. Indeed, this is what happens when you count using ten digits and want to count past 9 or, for that matter, past 19. The thing with counting in binary is that this happens a lot!

Exercise D.66 Check that the above list is correct, and continue the list to 20.

Remark D.67 (Binary numbers in 8-bit computers) Here is a slightly simplified explanation of how an 8-bit computer would interpret the string of 0's and 1's in the byte shown in Figure 5 (see Remark D.69 below for a hint of the full story):

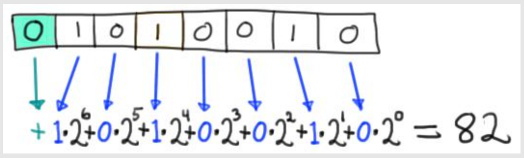

0 1 0 1 0 0 1 0
$+1\cdot 2^6+0\cdot 2^5+1\cdot 2^4+0\cdot 2^3+0\cdot 2^2+1\cdot 2^1+0\cdot 2^0 = 82$

Fig. 6. The first 7 bits (from the right) combine to form the binary representation of the integer. The left-most bit tells us if the binary number is positive or negative.

Exercise D.68 (a) What is the largest integer you can represent as an 8 bit integer? **(b)** Modern computers use 64 bit integers, where 1 bit is used for the sign and 63 for the number itself. What is the largest integer you can represent using a 64 bit integer?

Remark D.69 (Two's complement) Strictly speaking, our explanation for how integers are represented is only correct for positive integers. For negative integers, it would be kind of stupid to do exactly as we describe since then we would have two different ways of representing the integer 0 (indeed, both the bytes 0000 0000 and 1000 0000 would represent 0). Instead, an alternative strategy called *two's complement* is used. We will not explain it here (Wikipedia has a nice page on this), but it allows the computer to represent one extra negative number, meaning that on an 8 bit computer we can represent every integer between −128 and 127. (And, perhaps more importantly, using *two's complement* allows the hardware to speed up integer arithmetic.)

How non-integers are represented in Python

We now turn to how real numbers that are not integers, such as $1/10 = 0.1$ and $\sqrt{2} = 1.4151...$, are represented in a computer. First, we note that since $\sqrt{2}$ has an infinite number of digits, it should be clear that this number cannot be represented exactly. What may come as a surprise is that not even 0.1 can be stored correctly!

So, what is going on? First, we need to know that when storing real numbers that are not integers, your computer uses the datatype `float`.

Remark D.70 (sloppy description of 64 bit floating numbers) A 64 bit computer sets aside 64 bits to represent the real number on the form

$$\alpha \cdot 10^{\beta},$$

where the *fraction* α is (roughly) a 16 digit integer (positive or negative) and the *exponent* β is (roughly) an integer between -340 and $+292$.

Here, we use the words *essentially* and *roughly* since we lose a little bit in the translation from binary to decimal numbers. However, before formulating a more correct description of floating point numbers in binary language, let us try to get some intuition from the sloppy definition. To this end, we consider the following example.

Example D.71 According to the sloppy description, how is the number $\sqrt{2} = 1.4142135623730950488016887242096980785 69...$ stored? Well, *roughly* as

$$1414213562373095 \cdot 10^{-15}.$$

That is, the computer stores up to 16 digits in α (starting with the first non-zero digit from the left), and the position of the decimal point in the β. In particular, since a lot of information is thrown away, this means that we get a round-off error!

Exercise D.72 According to the sloppy definition, **(a)** how far is it between the floating point representation of $\sqrt{2}$ and its closest floating point "neighbour"? **(b)** How far is it between $x = 0$ and the next floating point number?

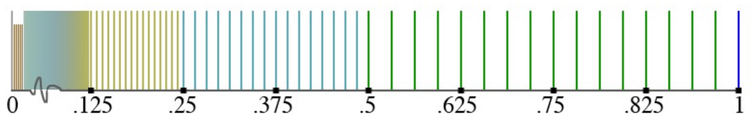


Fig. 7. As indicated by the above exercise, the floating point line is not a continuous line, instead it consists of many points with some short distance between them.

Exercise D.73 Put $a = 325660000$, $b = 0.000032566$ and use Python to compute $100{*}(a{+}b)$, $1000{*}(a{+}b)$ and $10000{*}(a{+}b)$. How are the results of these computations stored? Are there any round-off errors?

Remark: If you want to force Python to show you, say, 20 decimals places of a variable `a`*, you can use the crazy looking command* `print("{:.20f}".format(a))`.

Exercise D.74 Use the sloppy description of float numbers to do the following:

(a) Explain how large N has to be for the computer to think that $1 + 2^{-N} = 1$.

(b) How large does N have to be for the computer to think that $2^{-N} = 0$?

Hint: Recycle your answers from D.72.

Now, notice the following. According to our sloppy description, above, it makes no sense that the number $1/10 = 0.1$ cannot be represented accurately as a floating point number. Indeed, the number $1/10$ ought to have the simple representation

$$1 \cdot 10^{-1}.$$

So what is going on? Well, to explain this, we need a more accurate description of floating point numbers. As a first step, we need to understand how binary notation works for non-integers.

Example D.75 (binary notation for non-integers) The way to represent non-integers in binary notation is essentially completely analogous to how we do this for decimal numbers. Indeed, compare

$$(643.57)_{10} = 6 \cdot 10^2 + 4 \cdot 10^1 + 3 \cdot 10^0 + 5 \cdot \frac{1}{10^1} + 7 \cdot \frac{1}{10^2}$$

and

$$(101.01)_2 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot \frac{1}{2^1} + 1 \cdot \frac{1}{2^2}.$$

Exercise D.76 As a small taste of binary arithmetic, figure out both the decimal and binary representations of the numbers

(a) $(101.01)_2 \cdot 2^2$ **(b)** $(101.01)_2 \cdot 2^1$ **(c)** $(101.01)_2 \cdot 2^{-1}$

We now formulate the more accurate description of floating point numbers.

Remark D.77 (a more accurate description of 64 bit floating point numbers) A 64 bit floating point number is stored on the form

$$(1.\alpha)_2 \cdot 2^{\beta},$$

where the *fraction* α is a string of 52 bits, and the *exponent* β is a 11 bit integer. The remaning bit is used to store the sign of the floating point number.

When the exponent β is the smallest possible, then $(1.\alpha)_2$ is replaced by $(0.\alpha)_2$. (This is done to offer additional accuracy close to the origin.)

Here is a visual representation of the memory used for a 64 bit floating point number:

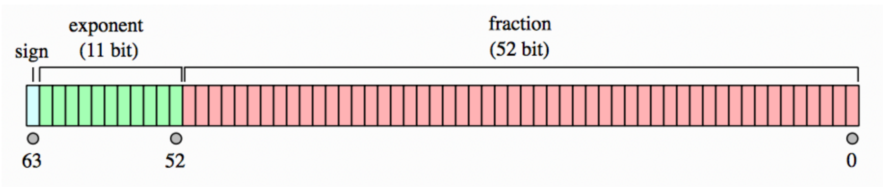


Fig. 8. Keep in mind that out of the 11 bits used for the exponent, one of them is used to denote the sign. In addition, it is not completely accurate to think of the 53 bits used for the fraction as a binary integer (see example below).

Since the notation used in the above description may be a bit confusing, let us consider an example.

Example D.78 How to store the number 1/10 a floating point number? In order not to have to write strings of 53 bits, let us pretend that we are working with 16 bit floating point numbers (so-called half-precision floats).

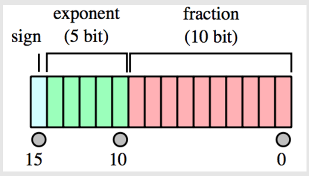


Fig. 9. 16 bit floats are exactly like 64 bit floats, except that less bits are available for the fraction and exponent.

First, expressing 1/10 on binary form (see exercises below for how to do this), we see that

$$\left(\frac{1}{10}\right)_{10} = (0.00011001100110011...)_2,$$

where the pattern keeps repeating. That is, the binary expansion of 1/10 is not finite! This means that to store it as a 16 bit (or 64 bit) floating point number, we are forced into making a round-off error! Indeed, here is the representation of 1/10 as a floating point number:

$$1.\underbrace{100110011}_{=\alpha}\cdot 2^{-(100)_2},$$

and here is exactly how this would look in the memory of the 16 bit computer:

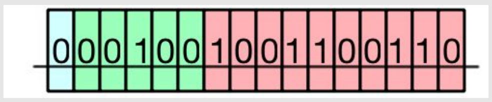


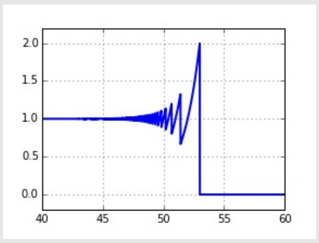
Fig. 10. Note that since the fraction appears on the right-hand side of a "binary comma", its right-most zeroes can be ignored.

Exercise D.79 Translate the "accurate description" of 64 bit floating point numbers to decimal notation to obtain the "sloppy description" at the start of this section. In particular, you need to take into account the added accuracy close to the origin.

Exercise D.80 (Challenging) Explain what are the only fractions that can be represented without round-off error as 64 bit floating point numbers.

Remark D.81 (The effect of round-off errors)

When using Python (or any other computer program) to compute and visualise data, we constantly need to ask ourselves if the results of our computations make sense, or if they are the artificial results of round-off error. For instance, to the right, we see a visualisation of $f(x) = 2^x \cdot \ln(1+2^{-x})$ which is utter nonsense (this function appears in, e.g., Chapter 4).



D.6 Answers to selected exercises

D.4 **(a)** addition, **(b)** subtraction, **(c)** multiplication, **(d)** power, **(e)** division, **(f)** floor division (returns the result of the division rounded down), **(g)** modulo (returns the numerator of the "remainder term" of the division),.

D.8 **(a)**, **(d)** and **(f)** will not run.

D.9 **(a)** $1 + 1/2 + 1/4 + 1/8 + 1/16$, **(b)** $\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+1}}}}$

D.11 The code in **(a)** will run.

D.36 The code computes the partial sum S_{99999} of the harmonic series. The speed of the computations will depend on your processor.

D.40 There are many ways to write this code. Here is one:

```python
sum = 0
k = 0
while abs(sum - 2) >= 1/10000:
    sum = sum + 1/2**k
    k = k +1     # Keep in mind that in a while-loop,
                 # we must update k manually.
print(k-1)
```

D.46 **(a)**

```python
k = 0
for x in myList:
    if x < 10**(-4):
        break
    k = k+1
print(k)
```

(b)

```python
k = 0
while myList[k] > 10**(-4):
    k = k+1
print(k)
```

D.48 Below, notice that we cannot call the absolute value function `abs`, since this keyword is already taken (for Python's own version of the absolute value function). **(a)**

```
def absolute1(x):
    if x>=0:
        return x
    else
        return -x
```

(b)

```
def absolute2(x):
    return (x**2)**(1/2)
```

D.49

```
def product(x):
    temp_prod = 1
    for a in x:
        temp_prod = temp_prod*x
    return temp_prod
```

D.54 Here is how to plot the first:

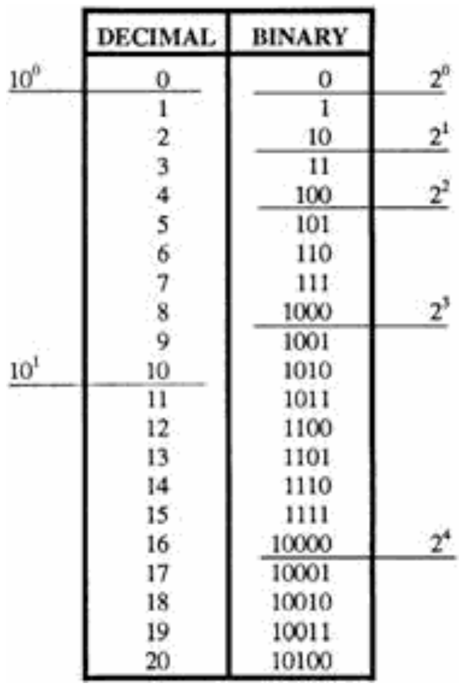
```
import matplotlib.pyplot as plt
# insert code for the definition of absolute1
X = [k/10 for k in range(-20,21)]
Y = [absolute1(x) for x in X]
plt.plot(X,Y)
```

D.58 This will not work for `absolute1` since if-statement in the definition of the function does not make sense if `x` is a list or a numpy array. For `absolute2`, the following code will work:

```
import matplotlib.pyplot as plt
import numpy as np
# insert code for the definition of absolute2
X = np.linspace(-2,2,40)
Y = absolute2(X)
plt.plot(X,Y)
```

D.63 Python crashes and returns the error message: "ZeroDivisionError: division by zero". In other words, Python really believes that $S_{100} = 2$.

D.66 Here are the first twenty numbers in both binary and decimal notation:


10^0
10^1
2^0
2^1
2^2
2^3
2^4

DECIMAL	BINARY
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111
16	10000
17	10001
18	10010
19	10011
20	10100

D.68 **(a)** $\sum_{n=0}^{6} 2^n = 2^7 - 1 = 127$, **(b)** $\sum_{n=0}^{62} 2^n$.

D.72 **(a)** Roughly 10^{-16}, **(b)** 10^{-340}.

D.73 According to the sloppy description `a+b` is stored as $3256600000000325 \cdot 10^{-7}$. When printing $100 * (a+b), 1000 * (a+b)$ and $10000 * (a+b)$ in in Python, you will see that the 17'th digit keeps changing due to the round-off error (Python essentially keeps guessing this digit wrong).

D.74 **(a)** $N \geq 53$, **(b)** $N \geq 1075$ (these answers can be checked by doing the computations in Python).

D.76 **(a)** $(10101)_2 = 21$, **(b)** $(1010.1)_2 = 10.5$, **(c)** $(10.101)_2 = 2.625$.

D.79 β is between -1024 and 1023 (counting two's complement). This means that 2^β is between 10^{-308} and 10^{308}, roughly. Taking into account that the α in our rough notation is a number between 1 and 10^{16}, and that in the accurate description $(1.\alpha)_2$ is replaced by $(0.\alpha)_2$ when $\beta = -1024$, we should get the rough description (roughly).

D.80 Exactly the fractions $a/2^n$, where a is an integer, and n is a natural number. These are exactly the numbers with finite binary expansions (prove this!).