nclab

Learning for Industry 4.0

Introduction to
Python
Programming

Introduction to Python Programming

Introduction to Python Programming

Dr. Pavel Solin

Revision August 25, 2024

About the Author

Dr. Pavel Solin is the founder of NCLab and Professor of Applied and Computational Mathematics at the University of Nevada, Reno. He is a seasoned educator, expert in scientific computing and software development, leader of open source projects, organizer of international scientific congresses, and the author of research monographs, textbooks, and research articles in international journals.

Acknowledgment

Our sincere thanks go to many educators and students for providing valuable feedback on this textbook as well as on the self-paced Python courses in NCLab.

Copyright:

Preface

Python is a modern high-level dynamic programming language which is widely used in business, science, and engineering applications. According to Github, as of 2024, Python is the second most popular programming language after Javascript:
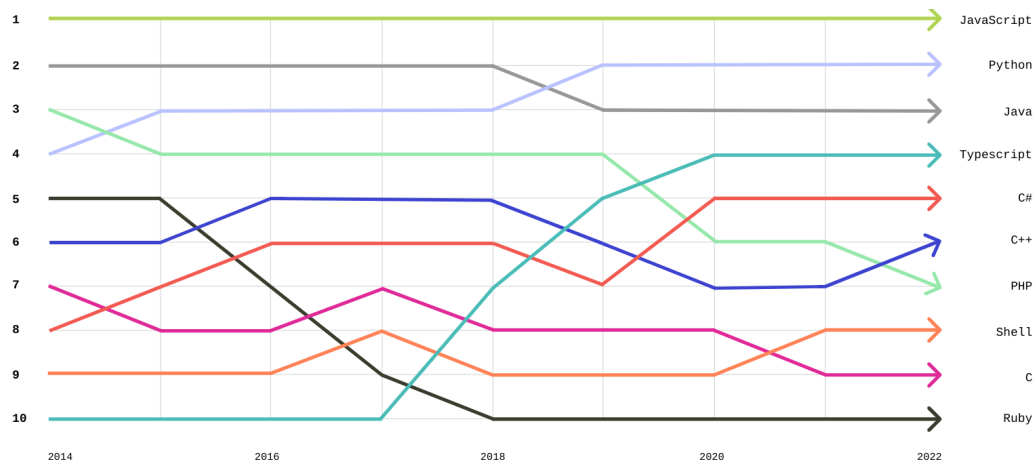

Fig. 1. Popularity of programming languages (source: Github).

Python's remarkable popularity has two main reasons: Firstly, it is known as a beginner's language because of its simplicity. Secondly, it helps developers to be more productive from development to deployment and maintenance. Python's syntax is very simple and high level when compared to Java, C and C++. Therefore, applications can be built with less code. Lastly, Python has a large collection of powerful libraries.

Python libraries

Thanks to its powerful free libraries, Python is replacing traditional commercial software products in many areas. In data analysis and statistics, Python has become more popular than SAS and even than R. In scientific computing and visualization, the collection of the libraries Scipy, Numpy, Sympy, Matplotlib and Mayavi has become more popular than MATLAB. Python also has powerful libraries for machine learning (Tensorflow, Keras), for graph theory (NetworkX), and many others.

How to read this textbook

In contrast to other languages, Python can be used by non-programmers. Therefore, in Sections 1 - 3 we will begin by showing you how to use Python as a powerful command-line scientific calculator and data visualization tool.

Starting with Section 4, we will explore Python as a programming language. We will show you how to work with text strings, variables, tuples, lists, dictionaries, functions, loops, conditions, files, exceptions, recursion, object-oriented programming, recursion, decorators, and more.

In every Section, we will begin with simple examples and gradually progress to more advanced applications. Therefore, you don't need to read every Section until the end. It's perfectly fine to read the first part of a Section, move on, and return to the advanced topics later.

Enjoy the textbook, and contact me at `pavel@nclab.com` if you have any questions, find typos, or have suggestions for other topics you would like to see here. Many thanks!

Pavel

Table of Contents

1 Introduction

1.1 Objectives

In this Section we will learn:

- About compiled and interpreted programming languages.
- The history and basic facts about Python.
- How to access the NCLab virtual desktop and open the Python app.
- How to use the Python app to write and run your Python programs.

1.2 Why learn Python?

Python is one of the most attractive programming languages that one can learn today. It is an undisputed leader in areas of the future such as big data, machine learning, and artificial intelligence. Python is easy to learn and incredibly versatile. It can be even used by non-programmers as a scientific calculator and powerful data analytics and visualization tool. It is almost unbelievable how Python is starting to dominate fields that traditionally belonged to specialized (and expensive) software products. As an example, let us take the area of data analytics which was traditionally dominated by SAS. We are going to show you two graphs. The first one, presented in Fig. 2, was taken from the webpage of Analytics Vidhya (`https://www.analyticsvidhya.com`). It shows that the number of job postings asking for R already surpassed the number of job postings asking for SAS:
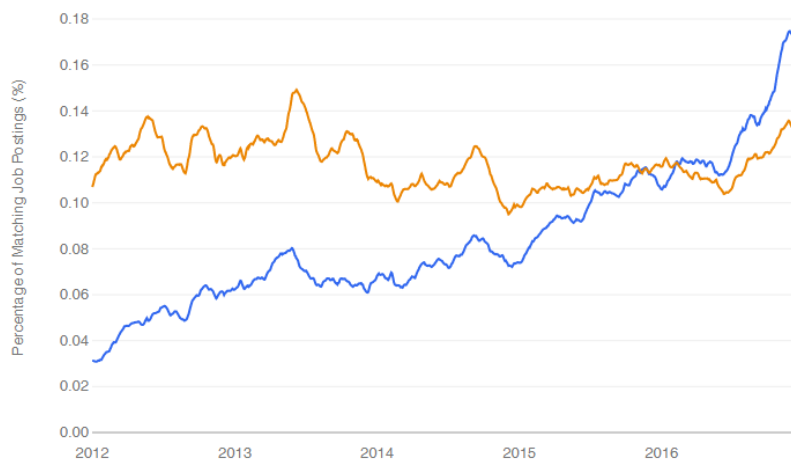
Fig. 2. Number of job postings asking for R (blue) vs. SAS (orange).

The second graph presented in Fig. 3 shows that the number of data analytics job postings asking for Python is much higher than the number of job postings asking for R.
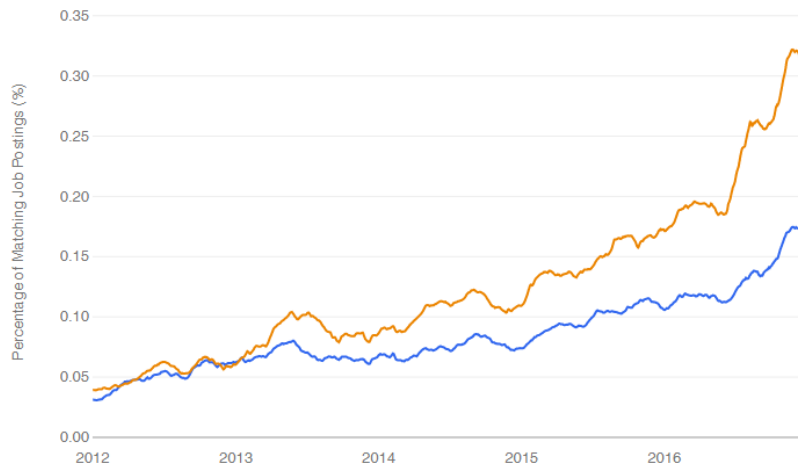
Fig. 3. Number of job postings asking for Python (orange) vs. R (blue).

Another example is general computing which for decades was dominated by MAT-LAB. Nowadays, powerful free Python libraries such as Numpy and Scipy, accompanied by advanced data visualization libraries such as Matplotlib, Mayavi and Seaborn, are taking over this area as well.

In this textbook you will learn the most important aspects of the language and you will be able to use Python to solve a large variety of practical problems. Depending on your objectives, Python may be the only programming language you will ever need. We hope that you will like Python as much as we do, and that after finishing this textbook you will want to learn more – and there always is much more to learn.

1.3   Compiled and interpreted programming languages

Python is an interpreted (as opposed to compiled) programming language. Let's briefly explain what these words mean.

Compilation is a process where human-readable *source code (text)* is translated by means of a *compiler* and a *linker* into a *binary (executable) file* which then can be run on the concrete computer. The same source code, compiled on different hardware architectures, yields different binaries. Compiled programming languages include Pascal, C, C++, Fortran, Java and many others.

2

Interpreted (scripting) programming languages are more recent than the compiled ones. A program that is written using an interpreted language is "read" (parsed) at runtime – it is not compiled and there are no binaries in the traditional sense. Programs written using compiled languages are usually more efficient than programs written using the interpreted ones because the former take better advantage of the underlying hardware. On the other hand, interpreted languages are usually more universal and easier to use. Examples of interpreted languages are Python, Lua, Perl, and Ruby.

1.4 A few more details about Python

Python was conceived in the late 1980s by Guido van Rossum in the Netherlands. Its implementation was started in December 1989. The name of the language comes from the British television series *Monty Python's Flying Circus*. Python 2.0 was released in October 2000 and Python 3.0 in December 2008. The newest version to-date, 3.6.5, was released on March 28, 2018

Python was twice awarded the TIOBE *Programming Language of the Year* award (in 2007 and in 2010). This award is given to the language with the greatest growth in popularity over the course of a year.

Python is a multi-paradigm programming language. Rather than forcing programmers to adopt a particular style of programming, it permits several styles: *structured (procedural) programming* and *object-oriented programming* are fully supported, and there are a number of language features which support *functional programming*.

1.5 Create a user account in NCLab

NCLab (Network Computing Lab) accessible at `http://nclab.com` is a public cloud computing platform that provides many apps related to computer programming, 3D modeling (CAD design), scientific computing and publishing, computer simulations, and other areas. It also provides self-paced and self-graded courses. Take a moment to visit `http://nclab.com` and create a user account now.

We will be programming using the Python app which is available in the Programming section of the Creative Suite as shown in Fig. 4. The Python app allows you to write, run, and debug your programs in the web browser, save them in your NCLab user account, and access them from any web browser, including tablets and other mobile devices. You can also upload data files to your user account, download your programs to a hard disk, and publish your programs to share them with others.
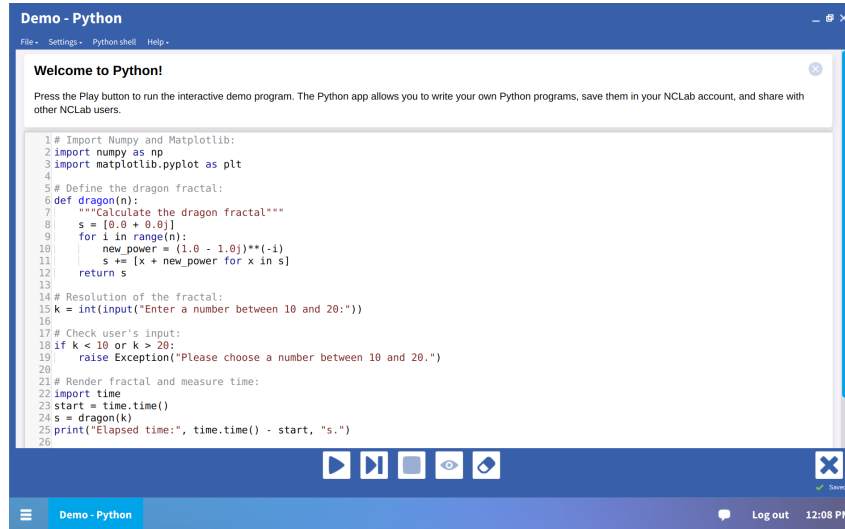
Fig. 4. NCLab Python app with a demo script.

1.6 Working in the cloud setting

In order to use NCLab, your device needs to be connected to the Internet. Initially, the Python app will launch with a demo script - a program which you can just run and see the result. By pressing the Play button on the bottom of the window, the code is sent to a remote cloud server, evaluated, and the results are then sent back to your web browser. This process usually takes less than a second, unless your Python program needs a longer time to run. After you do not need the demo script anymore, you can turn it off in Settings. The Python app can also be launched directly by double-clicking on a Python program in the File Manager.

1.7 Recommended devices

It does not matter if you are using a desktop computer, laptop, chromebook, tablet, or any other hardware platform. Your device is not used for computing - only when you work with graphics, then your graphics card may be used for image rendering. For practical reasons we do not recommend using NCLab on your phone - the phone's display is simply too small to create an enjoyable experience.

2 Using Python as a Scientific Calculator

2.1 Objectives

In this Section we will learn:

– How to use Python for elementary and advanced math calculations.
– How to use the floor division and modulo operations.
– How to import the Numpy library and work with mathematical functions.
– How to work with fractions, random numbers and complex numbers.
– About finite computer arithmetic and rounding.

2.2 Why use Python as a calculator?

Python can be used as an advanced scientific calculator, no need to own a TI-89 ($140 value). Moreover, a TI-89 hardly can compete with the processing power of a desktop computer. In this section we will show you how to use this computing power - no programming needed. Let us begin with the simplest math operations.

2.3 Addition and subtraction

Launch the Python app on NCLab's virtual desktop, erase the demo script, type `3 + 6` and press the Play button. You should see what's shown in Fig. 5:
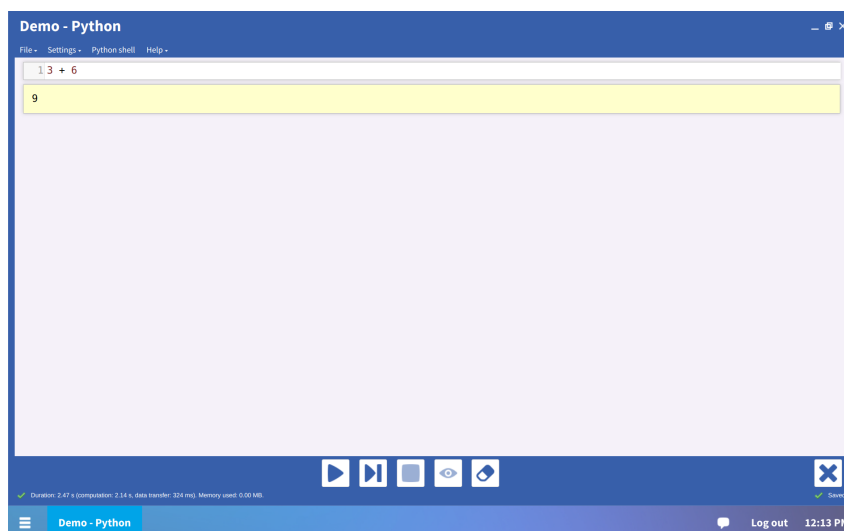
Fig. 5. Evaluationg the first math expression.

Of course you can add real numbers too:

```
3.2 + 6.31
9.51
```

Two numbers can be subtracted using the minus sign:

```
7.5 - 2.1
5.4
```

2.4 Multiplication

Multiplication is done using the asterisk '*' symbol as in

```
3 * 12
36
```

Indeed, real numbers can be multiplied as well:

```
3.7 * 12.17
45.029
```

2.5 Division

The forward slash '/' symbol is used for division:

```
33 / 5
6.6
```

2.6 Floor division

The floor division `a // b` returns the closest integer less than or equal to `a / b`:

```
6 // 4
1
```

It can be used with real numbers as well - in this case it returns a real number with no decimal part:

```
1.7 // 0.5
```
```
3.0
```

Technically, the result of the former example was *integer (int)* and the result of the latter was a *real number (float)*. This has no influence on the accuracy of the calculation.

When one of the numbers a, b is negative, then the floor division a // b again returns the closest integer less than or equal to a / b:

```
-6 // 4
```
```
-2
```

People who interpret floor division incorrectly as "forgetting the decimal part of the result" are sometimes confused by this behavior.

2.7 Modulo

Modulo is the remainder after floor division, and often it is used together with the floor division. In Python, modulo is represented via the percent symbol '%':

```
6 % 4
```
```
2
```

Modulo can be applied to real numbers as well:

```
12.5 % 2.0
```
```
0.5
```

2.8 Powers

For exponents, such as in 2^4, Python has a double-star symbol '**':

```
2**4
```
```
16
```

Both the base and the exponent can be real numbers:

```
3.2**2.5
```
```
18.31786887167828
```

But one has to be careful with negative numbers:

```
(-3.2)**2.5
(5.608229870225436e-15+18.317868871167828j)
```

Calculating

$$(-3.2)^{2.5} = \sqrt{(-3.2)^5} = \sqrt{-335.54432}$$

leads to the square root of a negative number, which is a complex number. We will talk about complex numbers in Subsection 2.17.

2.9   Priority of operations

Python follows the standard priority of math operations:

- Round brackets '(...)' have the highest priority,
- then exponentiation '**',
- then multiplication '*', division '/', floor division '//', and modulo '%',
- the lowest priority have addition '+' and subtraction '-',
- operations with the same priority are evaluated from left to right – for example the result of 20 / 10 * 2 is 4.

Note that no other brackets such as { } and [ ] are admissible in mathematical expressions. The reason is that they have a different meaning in the programming language.

To illustrate the priority of operations, let's evaluate the following expression:

```
3**4 / 27 * 5 + 3 * 5
30
```

If you are not sure, it never hurts to use round brackets:

```
(3**4) / 27 * 5 + 3 * 5
30
```

And as a last example, let's calculate

$$\frac{20}{4 \cdot 5}$$

8

The simplest way to do it is:

```
20 / 4 / 5
1
```

If you are not sure about the expression `20 / 4 / 5`, then note that 20 is first divided by 4 and the result (which is 5) is then again divided by 5. Therefore typing `20 / 4 / 5` is the same as typing `20 / (4 * 5)`.

2.10 Using empty spaces makes your code more readable

Your code will be much easier to read if you use empty spaces on either side of arithmetic symbols, as well as after commas. In other words, you should never write congested code such as

```
sin(x+y)+f(x,y,z)*5-2.4
```

The same can be written much more elegantly as

```
sin(x + y) + f(x, y, z) * 5 - 2.4
```

2.11 Importing the Numpy library

In order to calculate square roots, exponentials, sines, cosines, tangents, and many other math functions, the best way is to import Numpy. Numpy is a powerful Python library for numerical computations. To import it, just include the following line at the beginning of your code:

```
import numpy as np
```

Here `numpy` is the name of the library, and `np` is its standard abbreviation (you could use a different one if you wanted). After Numpy is imported, one can calculate, for example, e^2:

```
np.exp(2)
7.389056098930650
```

As you could see, one needs to add a prefix '`np.`' to the `exp` function to make it clear where the `exp()` function is coming from.

9

2.12 Finite computer arithmetic

The reader surely knows that the cosine of 90 degrees (which is $90\pi/180$ radians) is 0. However, when typing

```
np.cos(90 * np.pi / 180)
```

one obtains

```
6.123233995736766e-17
```

The reason is that:

> The computer cannot represent π exactly because it has infinitely many decimal digits. The cosine function also cannot be calculated exactly because it involves the summation of an infinite (Taylor) series.

It is important to get used to the fact that arithmetic calculations sometimes may yield "strange" values like this. In reality they are not strange though - on the contrary - they are natural. Because this is how computer arithmetic works. The reason why one usually does not see such numbers when using standard pocket calculators is that they automatically round the results. However, this is not a choice that should be made by the device - whether round the result or not, and to how many decimal places - should be the user's choice. And that's exactly how Python does it.

2.13 Rounding numbers

Python has a built-in function `round(x, n)` to round a number x to n decimal digits. For example:

```
round(np.pi, 2)
3.14
```

Back to our example of $\cos(90\pi/180)$:

```
round(np.cos(90 * np.pi / 180), 15)
0.0
```

And last - the number of decimal digits `n` can be left out. Then the default value `n = 0` is used:

```
round(np.pi)
```
```
3
```

2.14 List of math functions and constants

Elementary functions (and constants) that one can import from Numpy are listed below. We also show their arguments for clarity, but the functions are imported without them. Remember, the Numpy library is imported via `import numpy as np`.

pi	np.pi	π
e	np.e	e
arccos(x)	np.arccos	inverse cosine of x
arccosh(x)	np.arccosh	inverse hyperbolic cosine of x
arcsin(x)	np.arcsin	inverse sine of x
arcsinh(x)	np.arcsinh	inverse hyperbolic sine of x
arctan(x)	np.arctan	inverse tangent of x
arctanh(x)	np.arctanh	inverse hyperbolic tangent of x
arctan2(x_1, x_2)	np.arctan2	arc tangent of x_1/x_2 choosing the quadrant correctly
cos(x)	np.cos	cosine of x
cosh(x)	np.cosh	hyperbolic tangent of x
exp(x)	np.exp	e^x
log(x)	np.log	natural logarithm of x
pow(a, b)	np.pow	a^b (same as "a**b")
sin(x)	np.sin	sine of x
sinh(x)	np.sinh	hyperbolic sine of x
sqrt(x)	np.sqrt	square root of x
tan(x)	np.tan	tangent of x
tanh(x)	np.tanh	hyperbolic tangent of x

In summary:

Python provides many readily available mathematical functions via the Numpy library. To use them, import Numpy via the command `import numpy as np`.

For additional functions visit `https://docs.scipy.org/doc/numpy-1.14.0/reference/routines.math.html`.

2.15 Using the Fractions library

Let's import the Fractions library as `fr`:

```
import Fractions as fr
```

Then a fraction such as 4/7 can be defined simply as `fr.Fraction(4, 7)`. Fractions can then be added, subtracted, multiplied and divided as you would expect. The result of such an operation always is a `fr.Fraction` again. For example:

```
fr.Fraction(2, 6) + fr.Fraction(2, 3)
fr.Fraction(1, 1)
```

Another example:

```
fr.Fraction(2, 6) / fr.Fraction(2, 3)
fr.Fraction(1, 2)
```

The Fractions library also provides a useful function `gcd()` to calculate the Greatest Common Divisor (GCD) of two integers. For illustration let us calculate the GCD of 867 and 629:

```
fr.gcd(867, 629)
17
```

For the full documentation on the Fractions library visit `https://docs.python.org/3.1/library/fractions.html`.

2.16 Working with random numbers

Python provides a random number generator via the `random()` function that can be imported from the Random library. First let's import the library as `rn`:

```
import random as rn
```

The `random()` function returns a random real number between 0 and 1. Calling

```
rn.random()
```

five times yields

```
0.8562655947624614
0.016865272219651395
0.7551082761266997
0.5599940035041735
0.6110871965133025
```

Of course, you will get different values. When we need a random number in a different interval, for example between 10 and 15, then we multiply the result of `random()` by the length of the interval (here 5) and add the left end point (here 10):

```
10 + 5*rn.random()
```

Again let's evaluate the above code 5 times:

```
12.40747211768802
11.076556529787744
14.092310626104025
14.426494859235547
13.768444261074983
```

Sometimes we need to generate random integers rather than real numbers. This can be done via the `randint()` function. For illustration, a random integer between 1 and 3 can be generated via the code

```
rn.randint(1, 3)
```

Evaluating the code five times, one obtains a sequence of random numbers:

```
1
3
2
1
3
```

The Random library has a lot of additional functionality. For the full documentation visit `https://docs.python.org/3/library/random.html`.

2.17 Complex numbers

Appending 'j' or 'J' to a real number makes it imaginary. Let's calculate i^2 which in Python is represented as $-1 + 0j$:

```
1j * 1j
(-1+0j)
```

Complex numbers are always represented as two floating point numbers, the real part and the imaginary part:

```
1 + 3j
(1+3j)
```

One can also define them using the built-in function `complex()`:

```
complex(1, 3)
(1+3j)
```

All arithmetic operations that are used for real numbers can be used for complex numbers as well, for example:

```
(1 + 2j) / (1 + 1j)
(1.5+0.5j)
```

To extract the real and imaginary parts of a complex number `z`, use `z.real` and `z.imag`. Use `abs()` to get the absolute value:

```
a = 3 + 4j
a.real
a.imag
abs(a)
3
4
5
```

For additional functions related to complex numbers visit `https://docs.python.org/3/library/cmath.html`.

3   Drawing, Plotting, and Data Visualization with Matplotlib

3.1   Objectives

In this Section we will learn how to:

- Work with colors, RGB codes, and RGB color palettes.
- Plot lines, polylines, polygons, simple shapes, and graphs of functions.
- Fill objects with color and make holes into objects.
- Create parametric curves in the 2D plane and 3D space.
- Plot circles, ellipses, and spirals.
- Create wireframe plots, surface plots, contour plots, and use color maps.
- Use Matplotlib's patches and artists.
- Create basic and advanced pie charts, donut charts, and bar charts.

3.2   Why learn about data visualization with Python?

Python provides powerful data visualization libraries including Matplotlib, Seaborn, Mayavi and others. They are free but offer functionality comparable with expensive commercial software – that's why they are increasingly popular not only with individual users but also at companies. In short, these are great job skills!

3.3   RGB colors

Every color can be obtained by mixing the shades of red (R), green (G), and blue (B). These are called *additive primary colors* or just *primary colors*. The resulting color depends on the proportions of the primary colors in the mix. It is customary to define these proportions by an integer number between 0 and 255 for each primary color. So, the resulting color is a triplet of integers between 0 and 255. Examples of primary colors are red = [255, 0, 0], green = [0, 255, 0], blue = [0, 0, 255]. These colors are shown in the following Fig. 6 (left). When the R, G, and B values are the same, the result is a shade of grey. With [0, 0, 0] one obtains black, with [255, 255, 255] white. Other examples are shown in Fig. 6 (right).
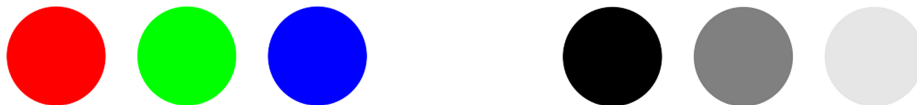
Fig. 6. Left: primary colors (red, green, blue). Right: shades of grey.

3.4 Secondary colors

You already know that by *primary colors* we mean colors where only one of the R, G, B components is nonzero. Analogously, *secondary colors* are colors where only two of the R, G, B components are nonzero. Fig. 7 shows the three most well-known secondary colors cyan, pink and yellow along with their RGB codes.
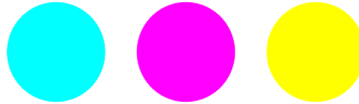
Fig. 7. Left: brightest cyan color [0, 255, 255]. Middle: brightest pink color [255, 0, 255]. Right: brightest yellow color [255, 255, 0].

Keep in mind that the two nonzero components do not have to be the same. For instance, the colors [0, 214, 138], [50, 0, 100] and [145, 189, 0] are also secondary.

3.5 Figuring out RGB codes

The easiest way to find the RGB code for a particular color you need is to google it. For example, to obtain the RGB color of violet color, type "violet RGB code" in your web browser. The result of the query is shown in Fig. 8.

Purple color codes chart

HTML / CSS Color Name	Hex Code #RRGGBB	Decimal Code (R,G,B)
violet	#EE82EE	rgb(238,130,238)
orchid	#DA70D6	rgb(218,112,214)
fuchsia	#FF00FF	rgb(255,0,255)
magenta	#FF00FF	rgb(255,0,255)

11 more rows

Purple color codes - RGB purple colors - RapidTables
https://www.rapidtables.com/web/color/purple-color.html

About this result   Feedback

Fig. 8. Searching the web for the RGB code of the violet color.

16

3.6 RGB color palettes

Sometimes one needs to find the right color for a specific application. In this case, it may be useful to browse some RGB palettes. In order to find them, search for "RGB color palette". There will be many results similar to each other, as shown in Fig. 9.
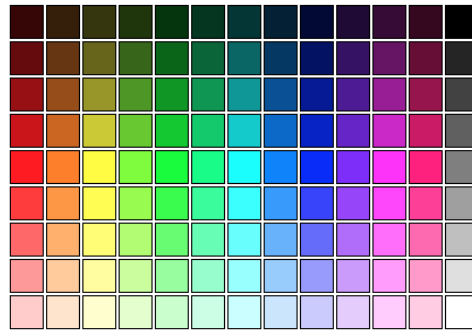
Fig. 9. Sample RGB color palette.

Hovering your mouse over a color, or clicking on it, will reveal its RGB code.

3.7 Starting with Matplotlib

Matplotlib is a powerful Python visualization library originally written by John D. Hunter, an American neurobiologist. The library is fully compatible with Numpy and other Python libraries. It is customary to import it by typing `import matplotlib.pyplot as plt`. As a Hello World project, let's plot a polyline!

The `plot()` function takes two arrays: x-coordinates and y-coordinates of points in the xy plane. Between the points, the curve is interpolated linearly. Let us illustrate this on a simple example with just five points [0, 0], [1, 2], [2, 0.5], [3, 2.5] and [4, 0]:

```python
import matplotlib.pyplot as plt
x = [0.0, 1.0, 2.0, 3.0, 4.0]
y = [0.0, 2.0, 0.5, 2.5, 0.0]
plt.clf()
plt.plot(x, y)
plt.show()
```

The commands `plt.clf()`, `plt.plot()` and `plt.show()` do clear the canvas, plot the graph, and display the image, respectively. The output is shown in Fig. 10.

17
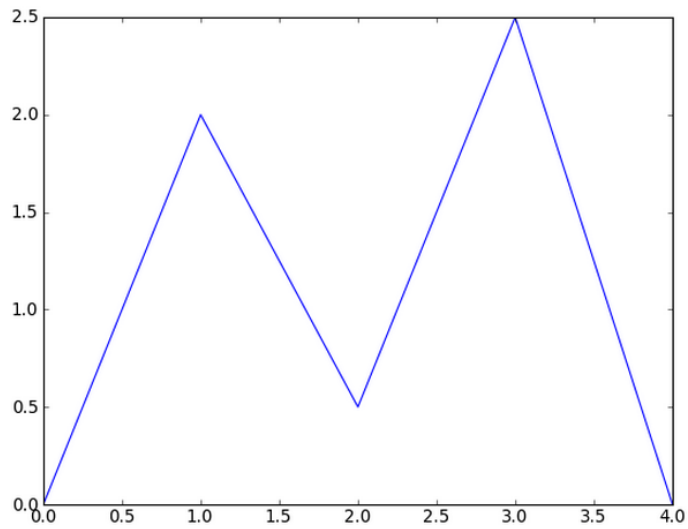
Fig. 10. Polyline with five points.

If the first and last points were the same, then the polyline would form a closed loop and one would obtain a polygon.

3.8 Two ways to specify color

The default blue color can be changed by passing an optional argument `color` to the `plt.plot()` function. With `color = 'pink'` we obtain a pink polyline.
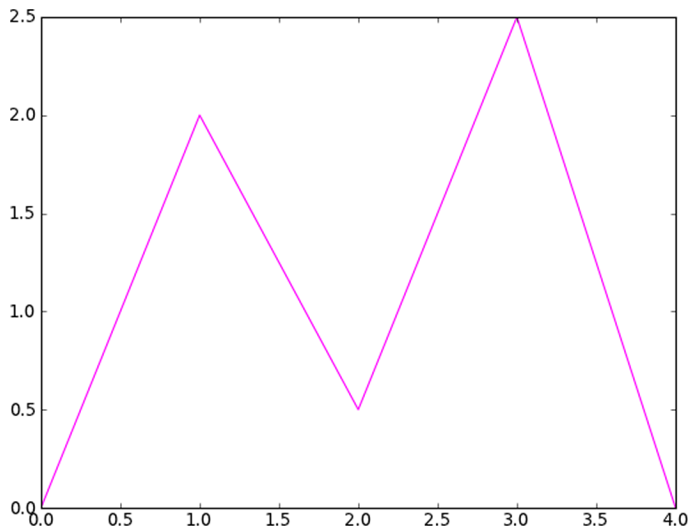
Fig. 11. Setting pink color via `plt.plot(x, y, color = 'pink')`.

18

Matplotlib contains many predefined colors, so it's always worth checking whether your color is available before resorting to RGB codes.

But specialty colors of course must be defined using RGB codes. For example, Kelly-Moore Oakwood color has the RGB code [186, 164, 138]. Before passing it into the `plt.plot()` function, one has to normalize the values (divide them by 255) to lie between 0 and 1. For the Kelly-Moore Oakwood the normalized RGB code is [186/255, 164/255, 138/255]:

```python
import matplotlib.pyplot as plt
x = [0.0, 1.0, 2.0, 3.0, 4.0]
y = [0.0, 2.0, 0.5, 2.5, 0.0]
plt.clf()
plt.plot(x, y, color = [186/255, 164/255, 138/255])
plt.show()
```

The corresponding output:
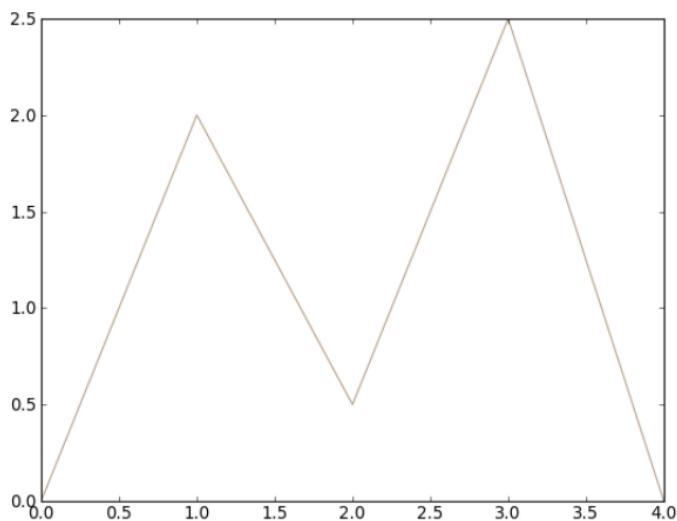
Fig. 12. Kelly-Moore Oakwood color [186, 164, 138].

3.9 Making axes equally scaled

Sometimes one does not mind that axes are not equally scaled - such as in the previous example. Then Matplotlib tries to make the best use of the entire viewing area. But it might become an issue when one wants to display a geometrical object with accurate proportions. For example, displaying a square with differently-scaled axes does not really work, as shown in Fig. 13.
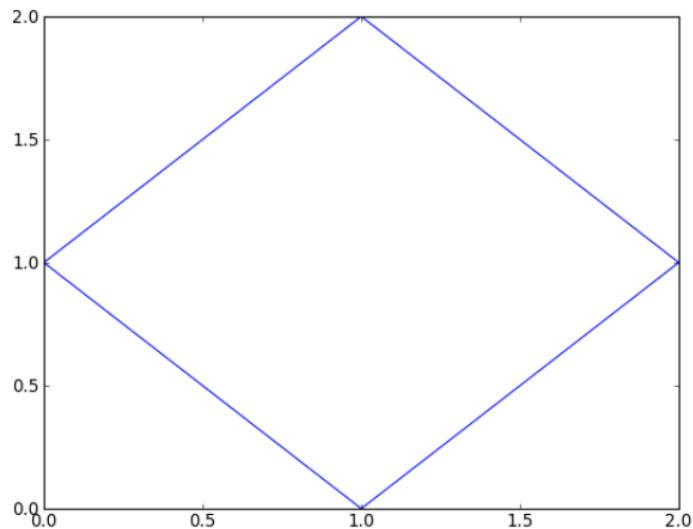
Fig. 13. Plotting a square with differently-scaled axes.

To correct this, it is enough to add the command `plt.axis('equal')`:

```python
import matplotlib.pyplot as plt
x = [1, 2, 1, 0, 1]
y = [0, 1, 2, 1, 0]
plt.clf()
plt.axis('equal')
plt.plot(x, y)
plt.show()
```

The result is shown in Fig. 14.
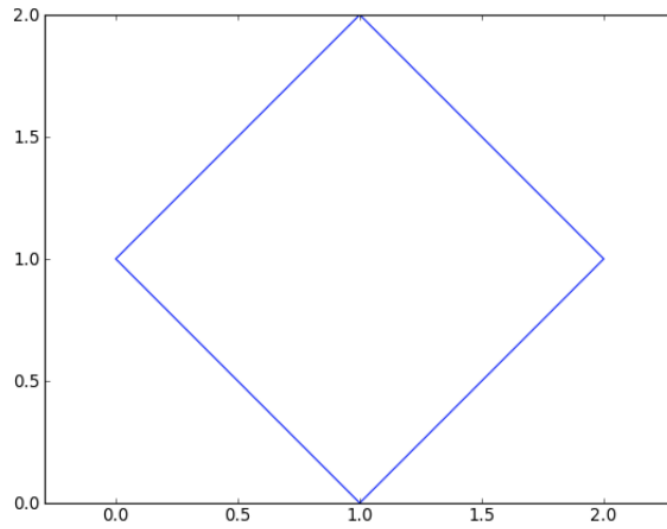
Fig. 14. Same square as above, but now with equally-scaled axes.

3.10 Filling shapes with color

A shape can be filled with color by using `plt.fill()` instead of `plt.plot()`, as shown in Fig. 15.
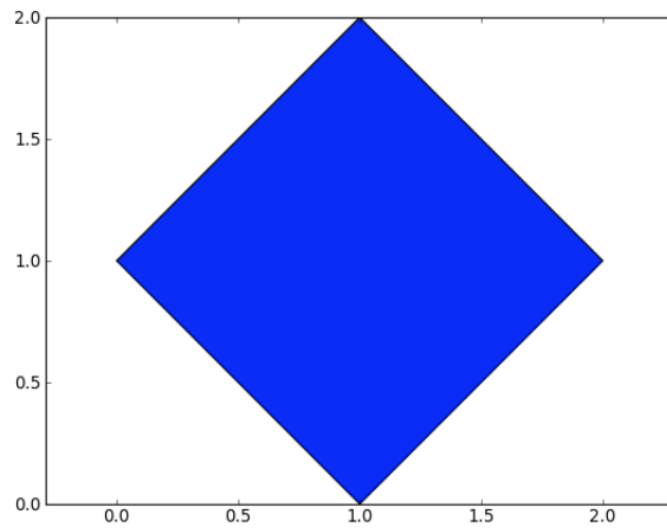
Fig. 15. Replacing `plt.plot()` with `plt.fill()`.

3.11 Borders

Have you noticed a thin black border in Fig. 15? One can make it thicker by including an optional parameter `linewidth` (or just `lw`) in `plt.fill()`:
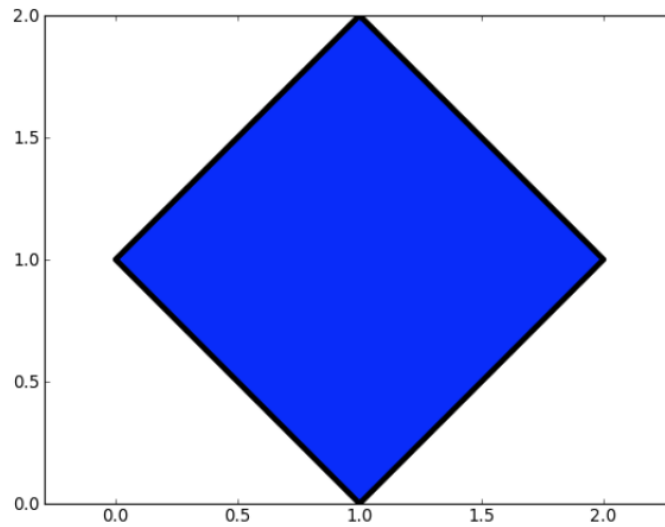
Fig. 16. Making the border thicker via `plt.fill(x, y, lw = 4)`.

The same optional parameter can be used to change the line width when plotting poly-lines. To eliminate the border, one can set `lw = 0`. This is shown in Fig. 17.
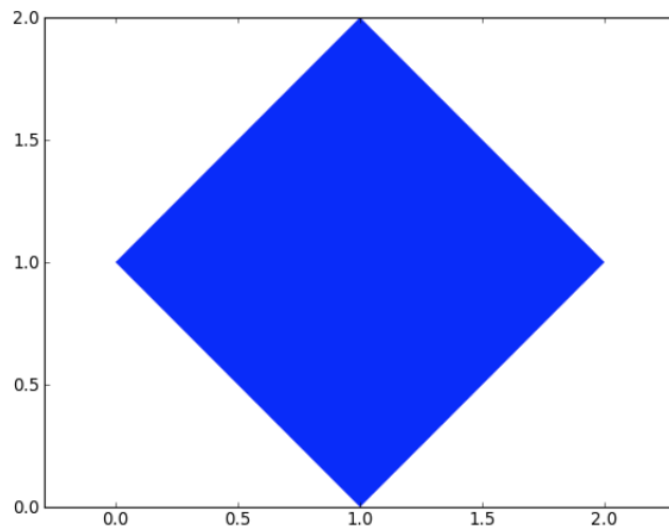
Fig. 17. Eliminating the border with `lw = 0`.

3.12 Interrupting lines

Lines can be interrupted with the keyword `None`. For example, the following code will plot three separate lines $[0, 0] \rightarrow [1, 0]$, $[1, 1] \rightarrow [2, 1]$ and $[2, 2] \rightarrow [3, 2]$:

```python
import matplotlib.pyplot as plt
x = [0, 1, None, 1, 2, None, 2, 3]
y = [0, 0, None, 1, 1, None, 2, 2]
plt.clf()
plt.axis('equal')
plt.plot(x, y, lw = 2)
plt.show()
```

This is the corresponding output:
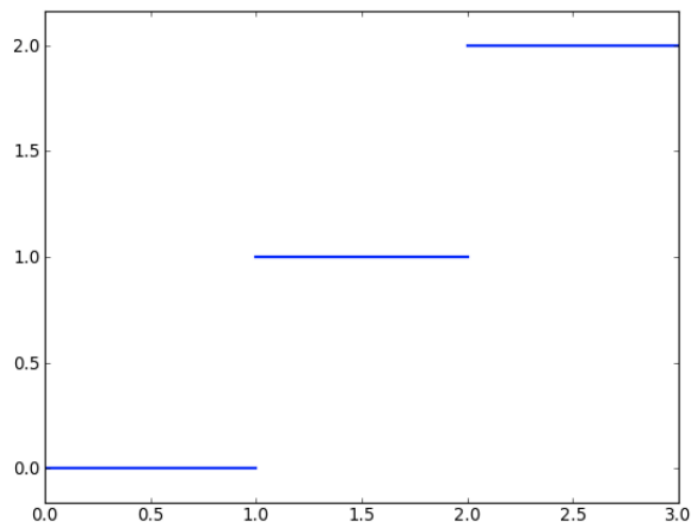
Fig. 18. Interrupting lines with the keyword `None`.

3.13 Making holes

To make a hole into a shape, the boundary of the shape and the boundary of the hole must have opposite orientations. What do we mean by orientation? Look at these two arrays which correspond to the square from Fig. 17:

```python
x = [1, 2, 1, 0, 1]
y = [0, 1, 2, 1, 0]
```

This polyline is connecting the points [1, 0] → [2, 1] → [1, 2] → [0, 1] → [1, 0] in the counter-clockwise direction. Let's say that we want to make a square hole of dimensions (0.75, 1.25) × (0.75, 1.25). This will be a polyline connecting the points [0.75, 0.75] → [1.25, 0.75] → [1.25, 1.25] → [0.75, 1.25] → [0.75, 0.75]. Except, this orientation also is counter-clockwise so it would not work. We need to flip the orientation of the hole to clockwise: [0.75, 0.75] → [0.75, 1.25] → [1.25, 1.25] → [1.25, 0.75] → [0.75, 0.75]. Then it will work:
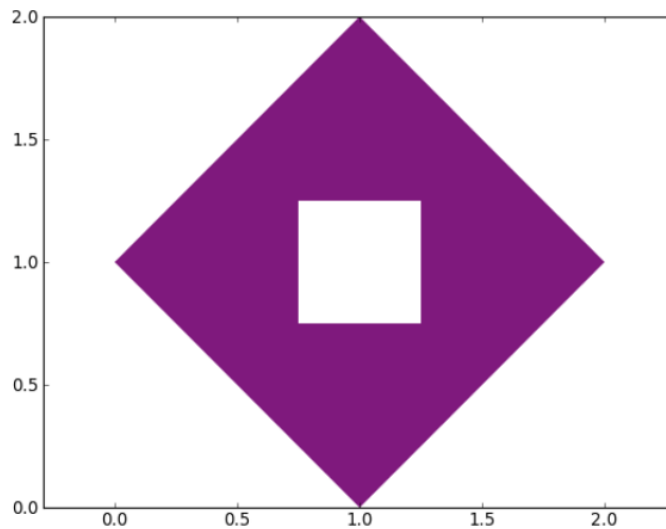
Fig. 19. Holes are made using polylines of opposite orientation.

Here is the corresponding code:

```python
import matplotlib.pyplot as plt
x = [1, 2, 1, 0, 1, None, 0.75, 0.75, 1.25, 1.25, 0.75]
y = [0, 1, 2, 1, 0, None, 0.75, 1.25, 1.25, 0.75, 0.75]
plt.clf()
plt.axis('equal')
plt.fill(x, y, color = 'purple')
plt.show()
```

3.14 Plotting functions of one variable

Let's say that we want to plot the function $f(x) = \sin(x)$ in the interval $(0, 2\pi)$. The array of x-coordinates of equidistant points between 0 and π can be created easily using

24

Numpy's function `linspace ():`

```python
import numpy as np
x = np.linspace(0, 2*np.pi, 101)
```

Here 101 means the number of points in the division, including endpoints. Hence, with 101 points one subdivides the interval into 100 equally long subintervals. Increasing this number will improve the resolution and vice versa. Next, the array of y-coordinates of the points is obtained via

```python
y = np.sin(x)
```

The last part you already know:

```python
import matplotlib.pyplot as plt
plt.clf()
plt.plot(x, y)
plt.show()
```

The output is shown in Fig. 20.
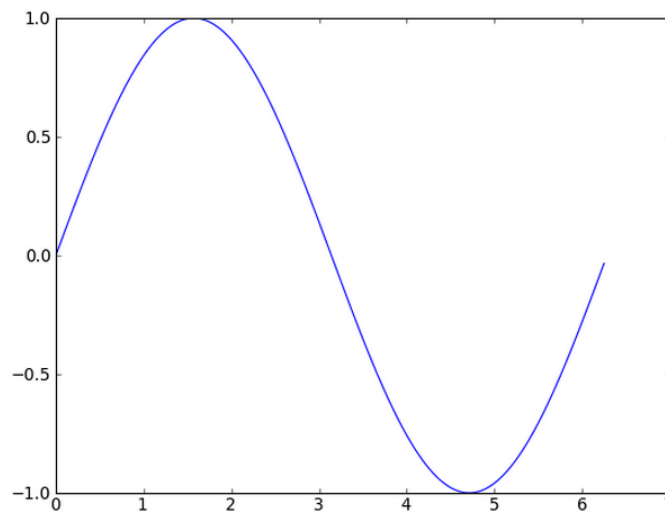
Fig. 20. Plotting $\sin(x)$ in interval $(0, 2\pi)$.

3.15 Line style, label, and legend

The plot can be made nicer by adding a label. When adding a label, one needs to call the function `plt.legend()` to display the legend. Also the line color and line style can be changed. Let us start with adding a label:

```
plt.plot(x, y, 'b-', label = 'Solid blue line')
plt.legend()
plt.show()
```

The output is shown in Fig. 21.
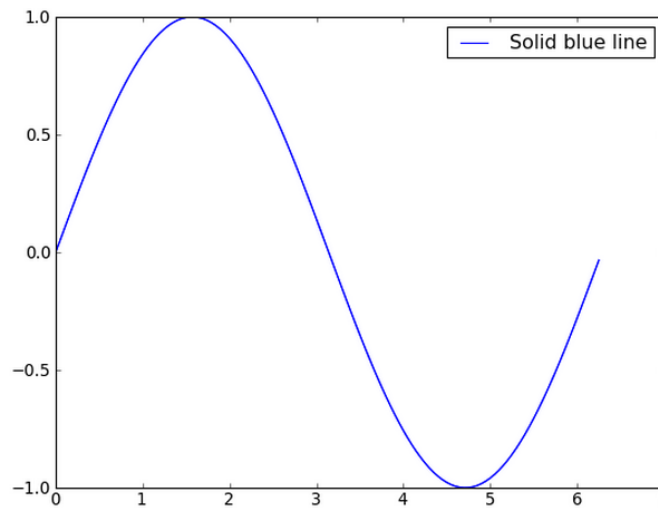
Fig. 21. Adding a label.

Next let us change the color to red and line style to dashed:

```
plt.plot(x, y, 'r--', label = 'Dashed red line')
plt.legend()
plt.show()
```

The output is shown in Fig. 22.
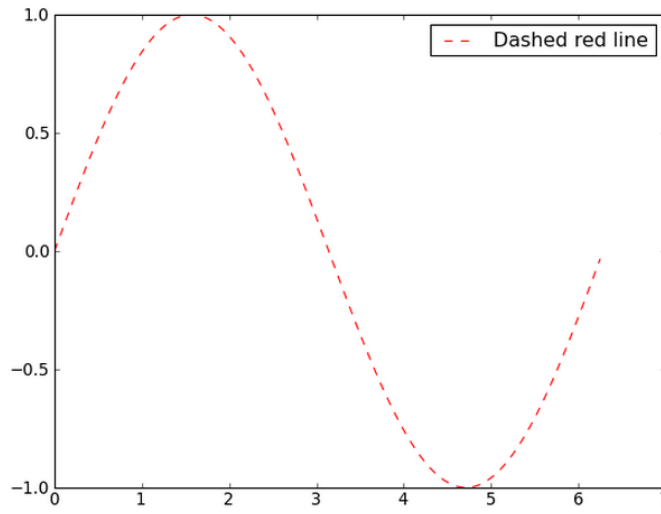
Fig. 22. Using dashed red line.

The graph can be plotted using green color and small dots rather than a solid or dashed line:

```
plt.plot(x, y, 'g.', label = 'Dotted green line')
plt.legend()
plt.show()
```

The output is shown in Fig. 23.
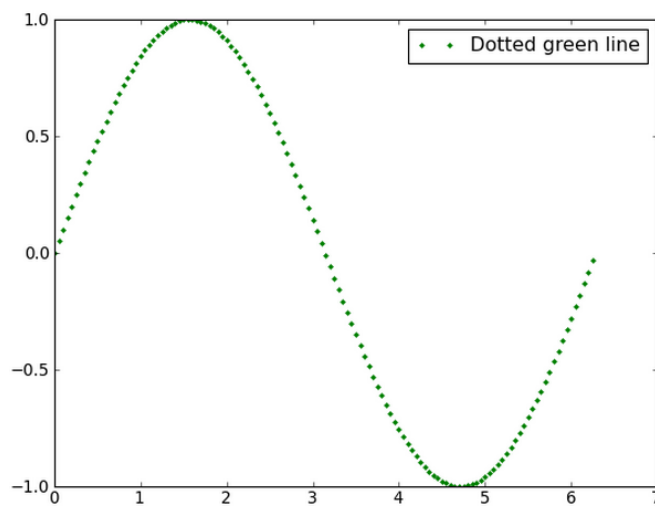
Fig. 23. Using dotted green line.

27

Last, let us stay with green color but make the dots bigger:

```
plt.plot(x, y, 'go', label = 'Bigger green dots')
plt.legend()
plt.show()
```

The output is shown in Fig. 24.
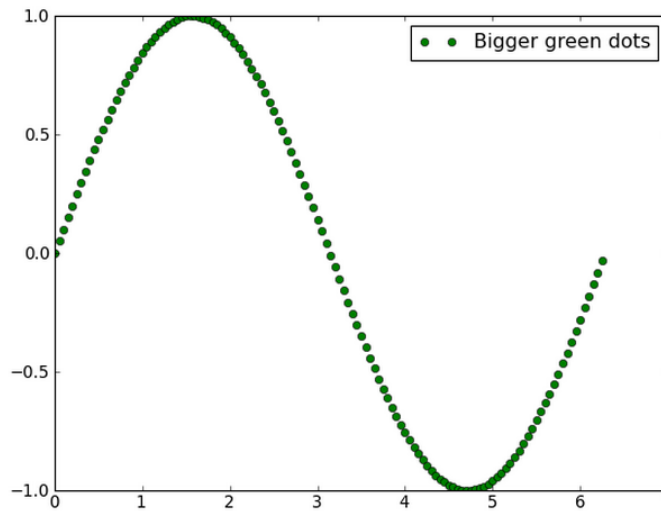
Fig. 24. Same graph using bigger green dots.

Line style can also be set separately using the optional parameter `linestyle` (or just `ls`) of the function `plt.plot()`. This parameter can have the values `'solid'`, `'dashed'`,`'dashdot'`,`'dotted'`,`'-'`,`'--'`,`'-.'`, and `':'`.

3.16   Showing the grid

When plotting functions, displaying the grid makes reading the values off the graph easier. The grid can be displayed using `plt.grid()`:

```
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(0, 2*np.pi, 101)
y = np.sin(x)
plt.clf()
plt.plot(x, y, label='sin(x)')
plt.axis('equal')
plt.grid()
plt.legend()
plt.show()
```

The output is shown in Fig. 25.
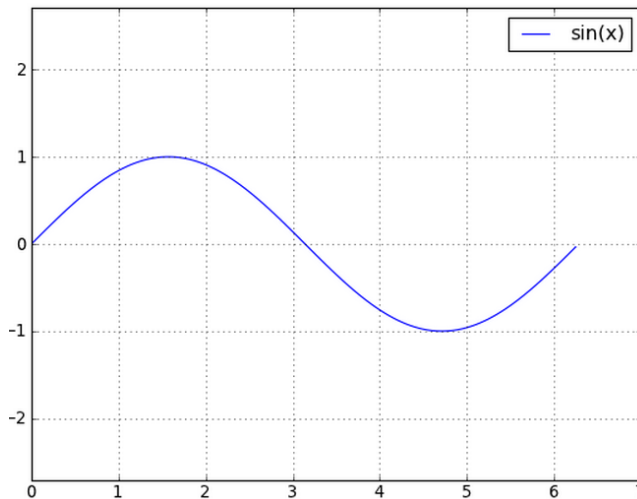
Fig. 25. Scaling axes equally and showing grid.

The scaling of axes can be returned to the automatic fit option via `plt.axis('auto')` if needed.

3.17   Adjusting plot limits

Plot limits can be set using the `plt.xlim()` and `plt.ylim()` functions after `plt.plot()`. For example, we can stretch the sine function from Fig. 25 to span the entire width of the canvas as follows:

```
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(0, 2*np.pi, 101)
y = np.sin(x)
plt.clf()
plt.plot(x, y, label='sin(x)')
plt.xlim(0, 2*np.pi)
plt.axis('equal')
plt.grid()
plt.legend()
plt.show()
```

The output is shown in Fig. 26.
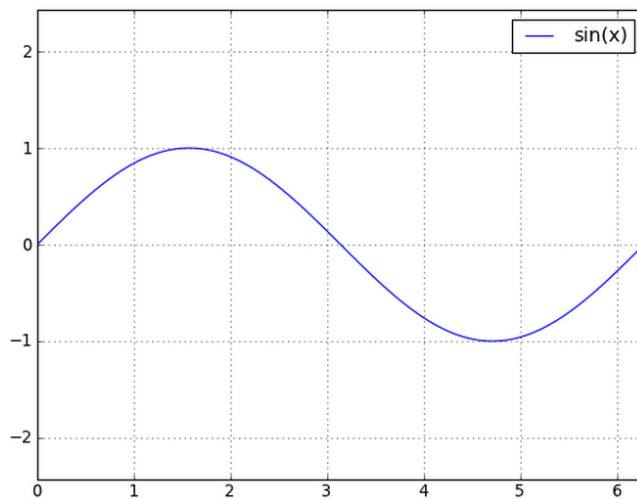
Fig. 26. Setting plot linits on the horizontal axis to 0 and 2π.

3.18   Plotting multiple functions at once

Multiple graphs can be displayed in one plot by just leaving out the `plt.clf()` command between them. For illustration, let us display the graphs of three different functions together:

```
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(0.5, 5, 101)
y1 = 1 / x
y2 = 1 / (1 + x**2)
y3 = np.exp(-x)
plt.axis('equal')
plt.clf()
plt.plot(x, y1, label='y1')
plt.plot(x, y2, label='y2')
plt.plot(x, y3, label='y3')
plt.legend()
plt.show()
```

The output is shown in Fig. 27.
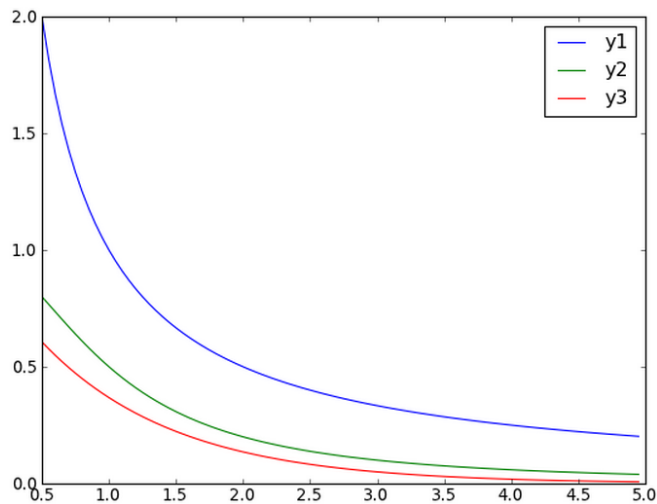
Fig. 27. Plotting graphs of functions $1/x$, $1/(1+x^2)$ and e^{-x} in interval $(0.5, 5)$.

The `plt.plot()` command in Matplotlib has many options. For a complete list visit the reference page `https://matplotlib.org/api/_as_gen/matplotlib.pyplot.plot.html#matplotlib.pyplot.plot`.

31

3.19 Plotting circles

The concept of plotting based on two arrays of x and y coordinates allows us to plot parametric curves such as circles, ellipses, and spirals. Let's begin with a circle of radius R centered at $[c_x, c_y]$ whose parametric equation is

$$x(t) = c_x + R\cos(t), \quad y(t) = c_y + R\sin(t).$$

The parameter t is defined in the interval $(0, 2\pi)$. The code:

```python
import numpy as np
import matplotlib.pyplot as plt
t = np.linspace(0, 2*np.pi, 101)
x = cx + R*np.cos(t)
y = cy + R*np.sin(t)
plt.clf()
plt.axis('equal')
plt.plot(x, y)
plt.show()
```

The output for $R = 1$ and $c_x = c_y = 0$ is shown in Fig. 28.
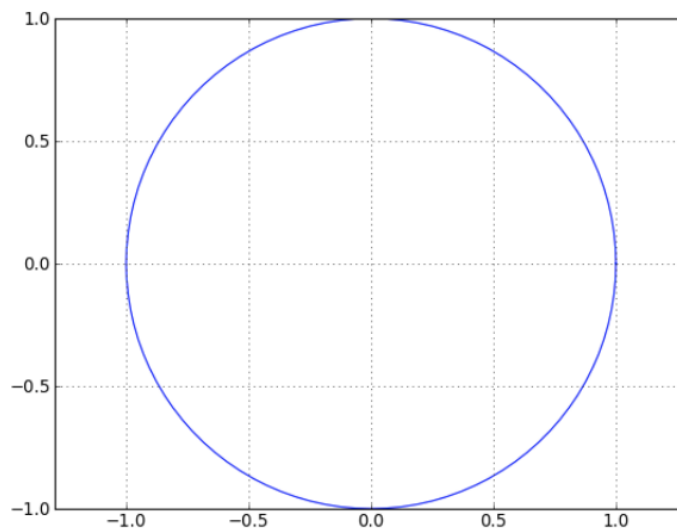
Fig. 28. Plotting a circle of radius $R = 1$ centered at $[0, 0]$.

32

3.20 Making a circular hole

Let's briefly return to Fig. 19 where we made a hole into a square using a polyline of opposite orientation:
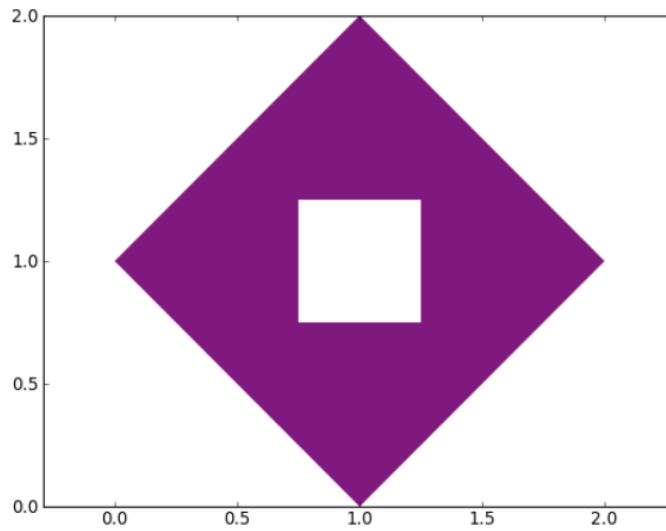
Fig. 29. Shape with a 0.5×0.5 square hole from Fig. 19.

Now we want to create a circular hole with radius $R = 0.25$ and centered at $[1, 1]$:
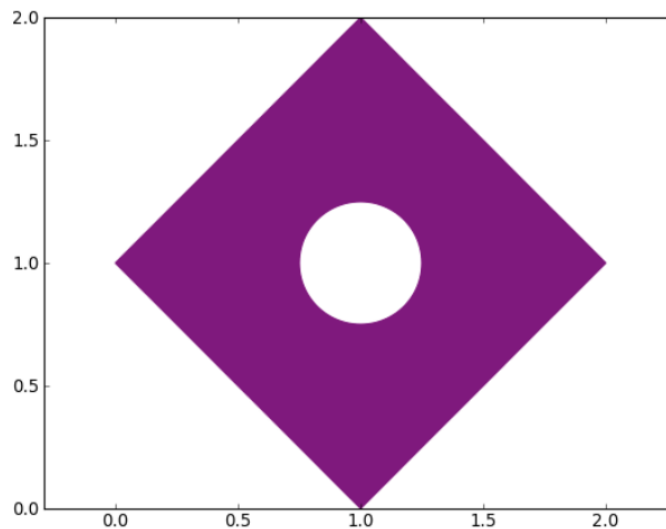
Fig. 30. Circular hole of radius $R = 0.25$ centered at $[1, 1]$.

33

Since the outer shape's boundary is oriented counter-clockwise and so is the circle, this would not work. But it is easy to change the orientation of the circle to clockwise:

$$x(t) = c_x - R\cos(t), \quad y(t) = c_y + R\sin(t).$$

Here is the corresponding code:

```python
import numpy as np
import matplotlib.pyplot as plt
xs = [1, 2, 1, 0, 1, None]
ys = [0, 1, 2, 1, 0, None]
t = np.linspace(0, 2*np.pi, 101)
xc = 1 - 0.25*np.cos(t)
yc = 1 + 0.25*np.sin(t)
x = xs + list(xc)
y = ys + list(yc)
plt.clf()
plt.axis('equal')
plt.fill(x, y, color = 'purple')
plt.show()
```

Notice that we used `list(xc)` and `list(yc)` to cast Numpy arrays `xc` and `yc` to lists. This was necessary in order to add them to the lists `xs` and `ys`. We will explain operations with lists in more detail later.

3.21   Plotting ellipses

If you know how to plot a circle, you also know how to plot an ellipse. All you need to do is replace the radius R with two different values in the X and Y directions. To illustrate this, let's replace the radius 0.25 in the previous example with 0.5 in the X direction and 0.3 in the Y direction. Here is the new version of the corresponding two lines:

```python
xc = 1 - 0.5*np.cos(t)
yc = 1 + 0.3*np.sin(t)
```

Otherwise the code remains unchanged. The output is shown in Fig. 31.
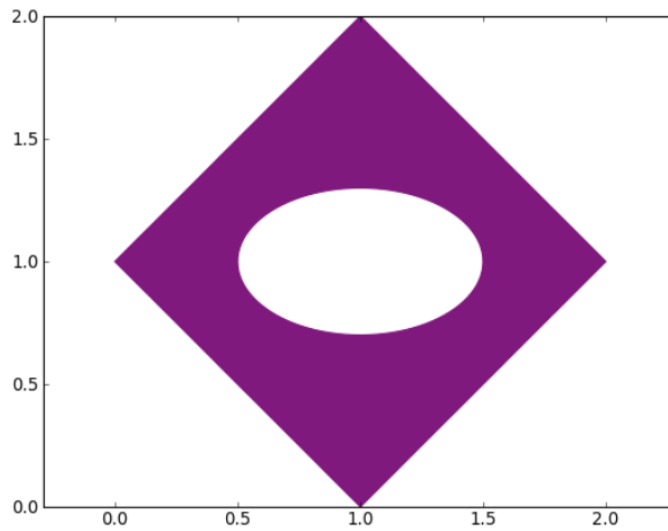
Fig. 31. Converting a circle of radius 0.25 into an ellipse with radii 0.5 and 0.3.

3.22   Plotting spirals

If you know how to plot a circle, you also know how to plot a spiral. All you need to do is make the radius increase with angle t. Let us illustrate this on a spiral that is parameterized by $x(t) = t\cos(t)$ and $y(t) = t\sin(t)$ in the interval $(0, 10)$ for t:
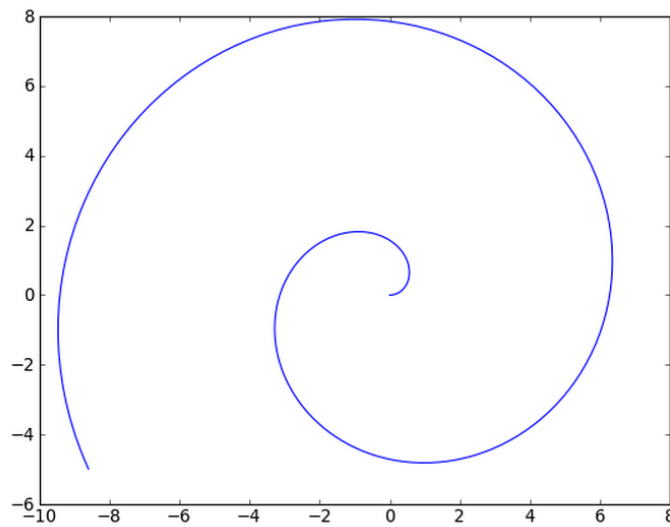
Fig. 32. Plotting a spiral.

35

The complete code for the spiral is

```python
import numpy as np
import matplotlib.pyplot as plt
t = np.linspace(0, 10, 101)
x = t*np.cos(t)
y = t*np.sin(t)
plt.clf()
plt.plot(x, y)
plt.show()
```

3.23   Parametric curves in the 3D space

For 3D plots it is practical to use Matplotlib's `mplot3d` toolkit. Parametric 3D curves are defined analogously to planar curves that we plotted in the previous paragraphs. 3D curves are sequences of linearly interpolated 3D points represented via three arrays of X, Y and Z coordinates. As an example, let's plot the curve

$$x(t) = (1 + t^2)\sin(2\pi t), \quad y(t) = (1 + t^2)\cos(2\pi t), \quad z(t) = t.$$

Here the parameter t lies in the interval $(-2, 2)$.

```python
# Import Numpy and Matplotlib:
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import axes3d
# Setup the 3D plot:
plt.gca(projection='3d')
# Define the division and the curve:
t = np.linspace(-2, 2, 101)
x = (1 + t**2) * np.sin(2 * np.pi * t)
y = (1 + t**2) * np.cos(2 * np.pi * t)
z = t
# Plot the curve and show the plot:
plt.plot(x, y, z, label='Parametric 3D curve')
plt.legend()
plt.show()
```

Compared to plotting planar curves, the only new thing is setting up the 3D plot via `plt.gca(projection='3d')`. Here, `gca` means "get current axes". It is a technicality that you don't have to worry about. The output is shown in Fig. 33.
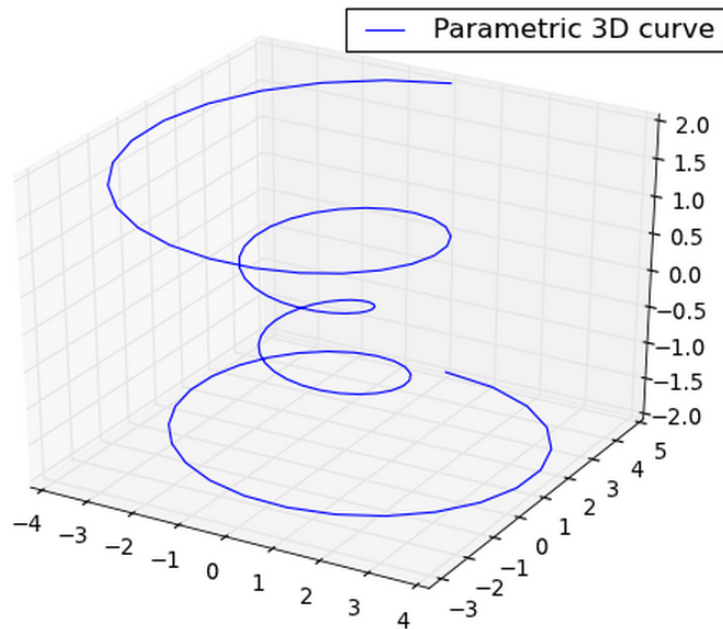
Fig. 33. Plotting a parametric 3D curve.

3.24 Wireframe plots of functions of two variables

The Matplotlib's `mplot3d` toolkit provides several ways to plot graphs of functions of two variables. First let's create a *wireframe plot* of the graph of the function

$$f(x, y) = \sin(x)\sin(y)$$

in the square $(-\pi, \pi) \times (-\pi, \pi)$:

```python
# Import Numpy and Matplotlib:
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import axes3d
# Define intervals on the 'x' and 'y' axes and divisions:
x = np.linspace(-3, 3, 31)
y = np.linspace(-3, 3, 31)
```

```
# Create Cartesian grid:
X, Y = np.meshgrid(x, y)
# Calculate values at grid points:
Z = np.sin(X) * np.sin(Y)
# Create 3D axes and plot the data:
axes = plt.axes(projection='3d')
axes.plot_wireframe(X, Y, Z, rstride=1, cstride=1, linewidth=1)
# Show the plot:
plt.show()
```

The parameters `rstride` (row stride) and `cstride` (column stride) can be used to make the plot coarser when they are set to 2, 3, etc. The output is shown in Fig. 34.
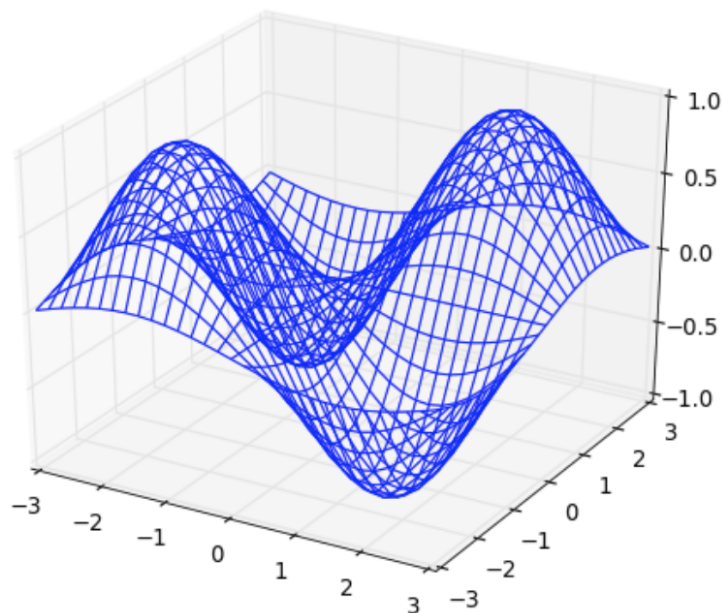
Fig. 34. Wireframe plot of the function $f(x, y) = \sin(x)\sin(y)$.

3.25 Surface plots

Surface plot can be obtained by replacing in the above code

```
axes.plot_wireframe(X, Y, Z, rstride=1, cstride=1, linewidth=1)
```

38

with

```
axes.plot_surface(X, Y, Z, rstride=1, cstride=1, linewidth=1)
```

The output is shown in Fig. 35.
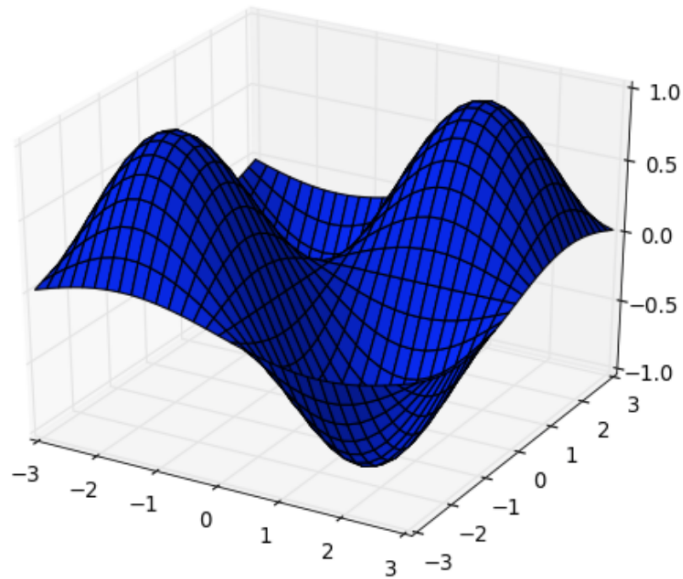
Fig. 35. Surface plot of the function $f(x, y) = \sin(x)\sin(y)$.

3.26 Color maps

Function `plot_surface` accepts an optional parameter `cmap` ("color map"). For example, selecting `cmap='jet'`,

```
axes.plot_surface(X, Y, Z, rstride=1, cstride=1, linewidth=1,
                  cmap='jet')
```

yields
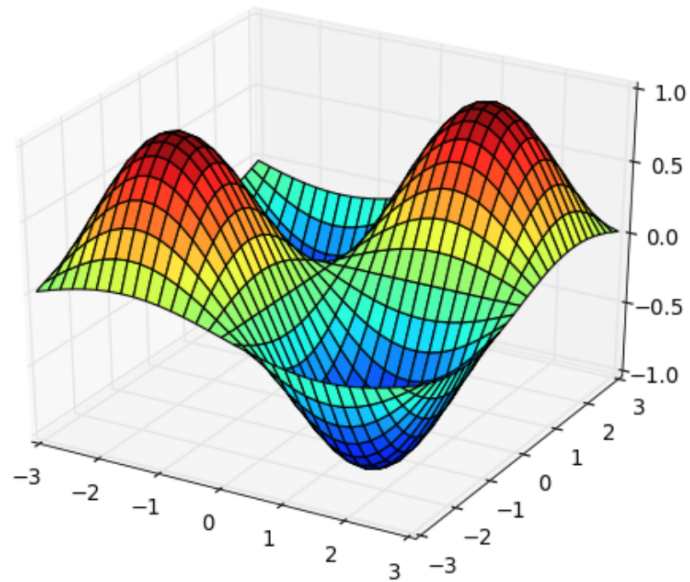
Fig. 36. The "jet" color map.

Another choice `cmap='viridis'` yields
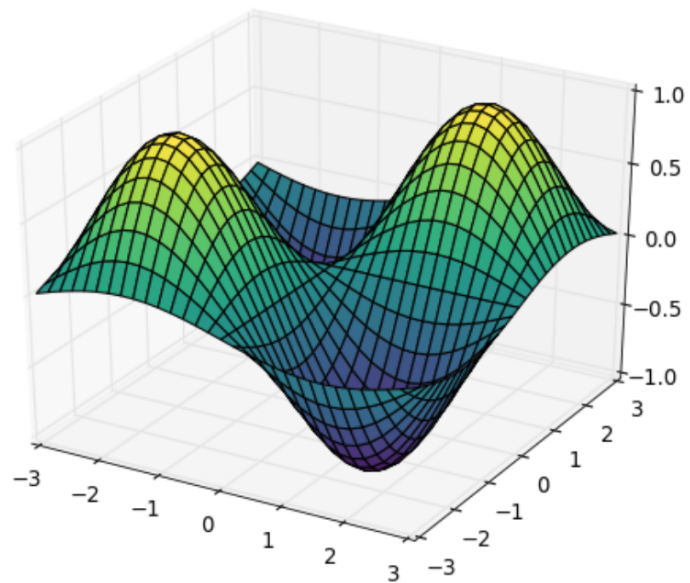
Fig. 37. The "viridis" color map.

In the next example we will use the color map `cmap='ocean'`, finer division `np.linspace(-3, 3, 61)`, and hide the black lines by setting `linewidth=0`.

40
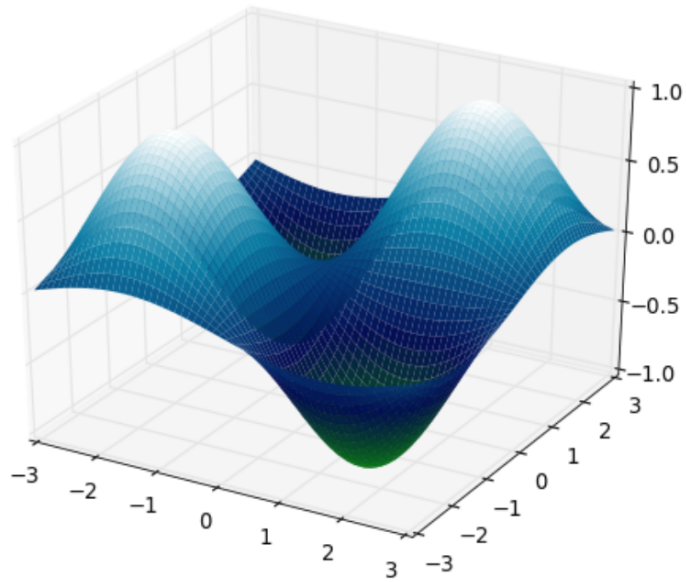
Fig. 38. The "ocean" color map and `linewidth=0`.

Last, let's show the color map `cmap='rainbow'`:
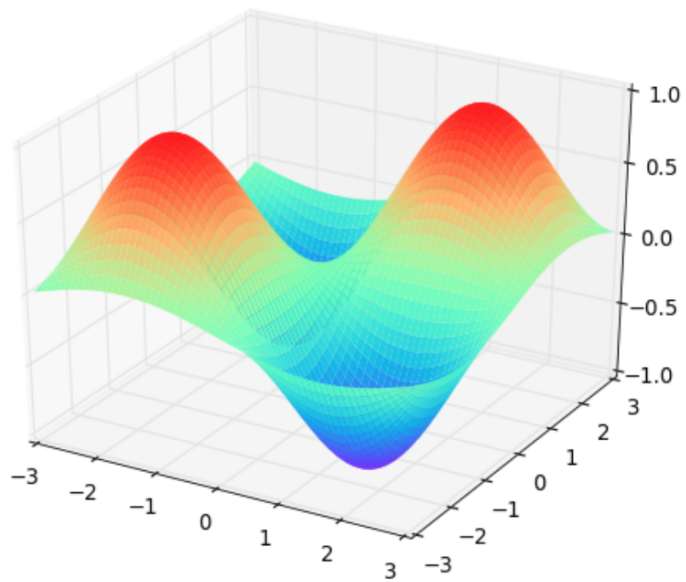
Fig. 39. The "rainbow" color map.

Matplotlib provides many additional color maps, make sure to check them out at `https://matplotlib.org/examples/color/colormaps_reference.html`!

3.27 Contour plots

Contour plot can be obtained by replacing in the above code the line

```
axes.plot_wireframe(X, Y, Z, rstride=1, cstride=1, linewidth=1)
```

with

```
axes.contour(X, Y, Z, num_contours)
```

where `num_contours` is the desired number of contours. If no color map is selected, Matplotlib uses "jet". Sample output with `num_con tours` = 50 is shown in Fig. 40.
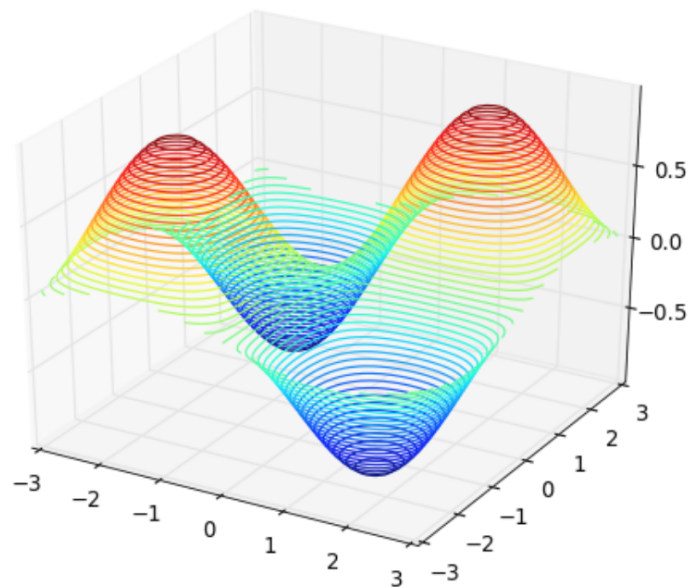
Fig. 40. Contour plot with 50 contours.

The `contour` function supports color maps. For example, with

```
axes.contour(X, Y, Z, num_contours, cmap='terrain')
```

one obtains the following contour plot:

42
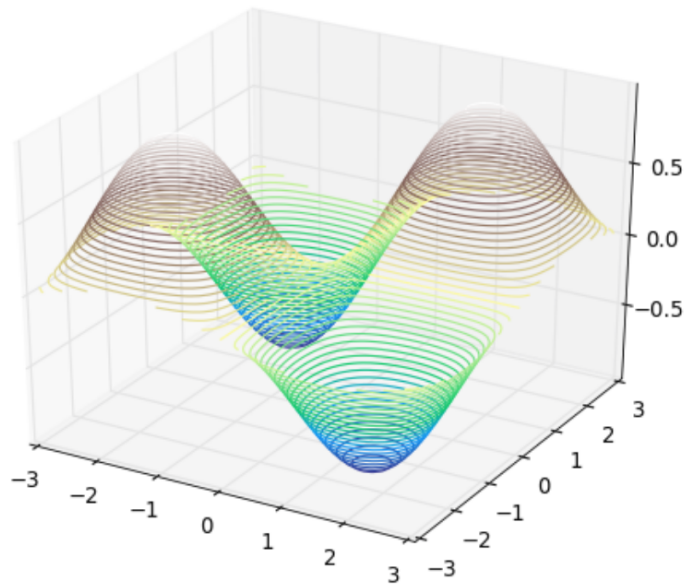
Fig. 41. The color map "terrain".

3.28 Changing view and adding labels

Sometimes the default view is not optimal, in which case one can change the elevation and azimuthal angles via the function `view_init`. By *elevation angle* we mean the angle between the XY plane and the line connecting the camera and the origin [0, 0, 0]. By *azimuthal angle* we mean the angle of rotation about the Z axis. With `axes` defined as in the above code, calling

```
print(axes.elev, axes.azim)
```

will reveal the current values of these two angles (their default values are 30 and -60 degrees BTW). To change them to (say) 70 and 40 degrees, add the following line right after the line `axes = plt.axes(projection='3d')`:

```
axes.view_init(70, 40)
```

The result is shown in Fig. 42. When changing the view, it is a good idea to put labels on the X, Y and Z axes:

```
axes.set_xlabel('X')
axes.set_ylabel('Y')
axes.set_zlabel('Z')
```

43
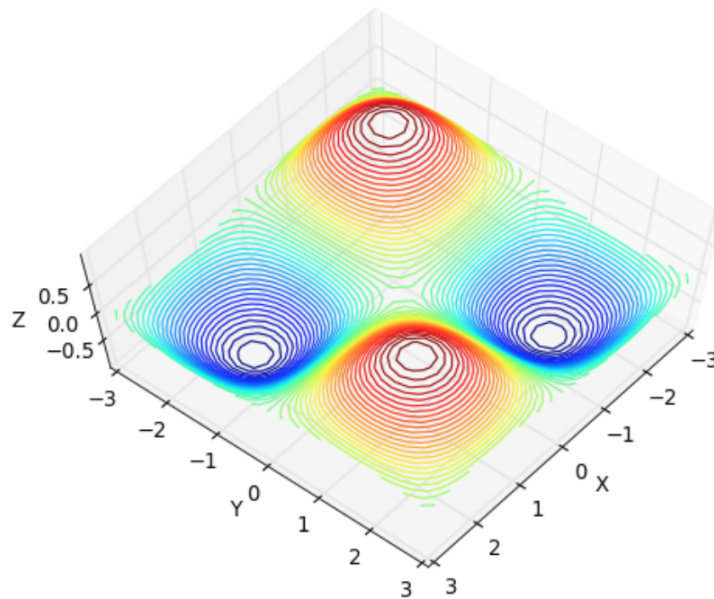
Fig. 42. Changing elevation and azimuthal angles to 70 and 40 degrees, respectively.

We will now leave the `mplot3d` toolkit but we invite you to explore more of it at `http://matplotlib.sourceforge.net/mpl_toolkits/mplot3d`.

3.29 Patches and artists

Now that you know the mechanics of creating various 2D shapes, graphs and curves via polylines, we can shift to a higher gear and explore Matplotlib's *patches* (patch = 2D shape). The patches provide a quicker way to create various 2D shapes including triangles, squares, rectangles, polygons, circles, ellipses, arcs, arrows, rectangles with round corners, and more. For a complete list of these shapes see

$$https://matplotlib.org/api/patches\_api.html$$

On top of what you already can do using polylines, patches can be *transparent* (the function `fill` that we have used so far does not support transparency). Moreover, patches can be *rotated*. We could show you how to do this with our polylines but we would need some advanced linear algebra and Python programming. In short - Matplotlib's patches make all of this very easy. All one needs is to specify a few parameters while defining the 2D shapes.

Let's show an example by creating three ellipses `e1`, `e2` and `e3`, centered at [0, 0] and rotated by 0, 60 and 120 degrees, respectively. They will all have diameters 4 in

44

the X direction and 1 in the Y direction, respectively, and 50% transparency (`alpha = 0.5`). Then we will add a red circle `c1` of radius 0.4. Adding each 2D shape requires adding a new "artist". Here is the code:

```python
import matplotlib.pyplot as plt
from matplotlib.patches import Circle, Ellipse
e1 = Ellipse((0, 0), 4, 1, alpha=0.5, angle=0, color='pink')
e2 = Ellipse((0, 0), 4, 1, alpha=0.5, angle=60, color='cyan')
e3 = Ellipse((0, 0), 4, 1, alpha=0.5, angle=120, color='yellow')
c1 = Circle((0, 0), 0.4, color='red')
plt.gcf().gca().add_artist(e1)
plt.gcf().gca().add_artist(e2)
plt.gcf().gca().add_artist(e3)
plt.gcf().gca().add_artist(c1)
plt.axis('equal')
plt.xlim(-3, 3)
plt.ylim(-2, 2)
plt.show()
```

Do not worry about the calls to `gcf` ("get current figure"), `gca` ("get current axes") and `add_artist`. These are technicalities whose sole purpose is to add a new shape to the current figure. The output:
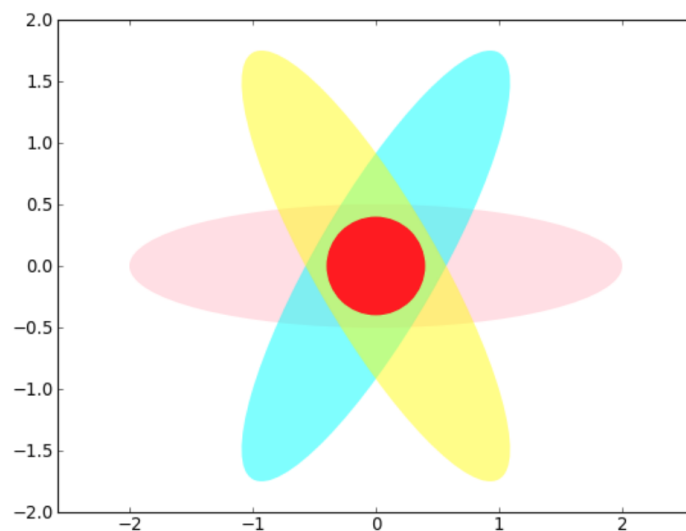
Fig. 43. Three ellipses rotated by 0, 60 and 120 degrees, and a circle.

45

3.30 Pie charts

Matplotlib's pie charts are an elegant way to report the breakdown of an ensemble into parts. It provides the function `pie` for that. The simplest example of a pie chart involves just 7 lines of code:

```python
import matplotlib.pyplot as plt
data = [450, 550, 300]
labels = ['cats', 'dogs', 'mice']
plt.clf()
plt.axis('equal')
plt.pie(data, labels=labels)
plt.show()
```

The code is self-explanatory. Its output is shown in Fig. 44.
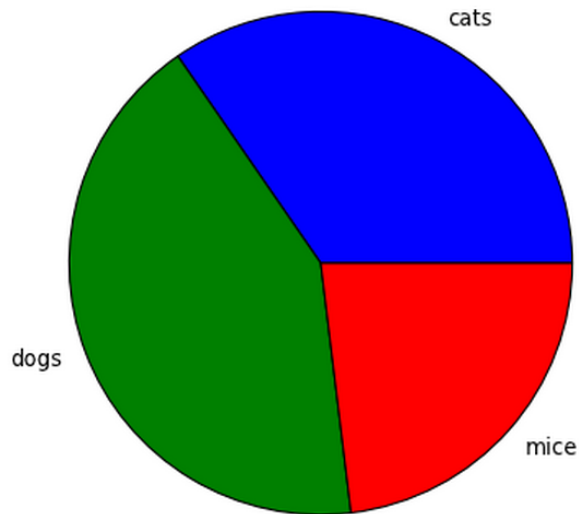
Fig. 44. The most basic pie chart.

Next let's present a fancier pie chart showing the production of an electronic company that sells TVs, computer monitors, cameras, printers and scanners. Make sure to read the comments in the code:

```
# Import Matplotlib:
import matplotlib.pyplot as plt
# This time there are five sections:
labels = ['TVs', 'Monitors', 'Cameras', 'Printers', 'Scanners']
# Fractions in percent:
fractions = [15, 30, 5, 40, 10]
# "Monitors" section will be highlighted by moving
# out of the chart by 0.05:
explode = (0, 0.05, 0, 0, 0)
# Create the pie chart. The "autopct" parameter formats the
# numerical percentages. Shadow can be suppressed by setting
# it to False:
plt.clf()
plt.pie(fractions, explode=explode, labels=labels,
    autopct='%1.1f%%', shadow=True)
# Display the pie chart:
plt.axis('equal')
plt.show()
```

The output is shown in Fig. 45.
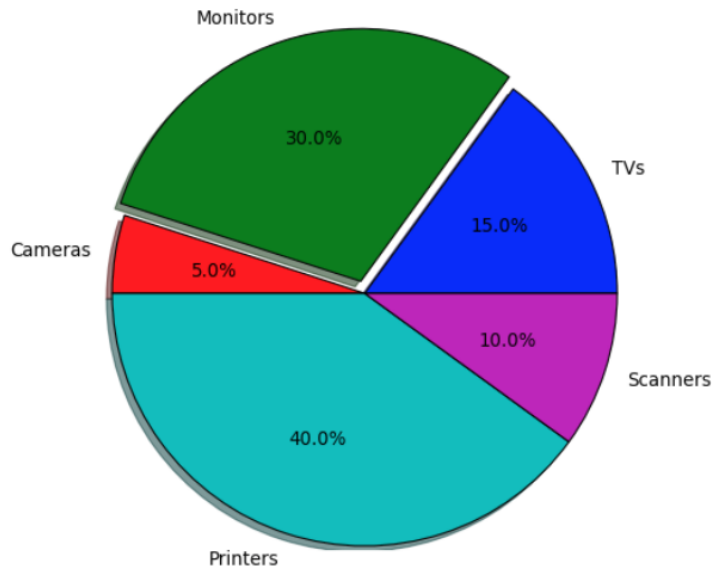
Fig. 45. Fancier pie chart.

The `autopct` option determines the numerical format in which the numbers inside the pie chart will be displayed. The letter `f` stands for a floating-point number. The `1.1` means that one decimal digit will be present after the decimal point (`1.2` would print the percentages with two digits after the decimal point, etc.). The parameter `shadow` indicates whether a shadow will be used. It can be set to `True` or `False`.

3.31 Donut charts

A special case of the pie chart is the "donut chart" which has a white circle inserted at the center. This time we will also show you how to use custom colors and how to rotate the pie chart. Do not worry if you do not understand all the details in the following code. The main things are self-explanatory. Hexadecimal RGB color codes will be discussed in detail later. You should understand the code well enough to be able to take an example like this and tweak it to display your own data.

```python
# Import Matplotlib:
import matplotlib.pyplot as plt
labels = ['TVs', 'Monitors', 'Cameras', 'Printers', 'Scanners']
fractions = [15, 30, 5, 40, 10]
# Custom colors:
colors = ['#ff9999','#66b3ff','#99ff99','#ffcc99','#e6e6fa' ]
explode = (0, 0, 0, 0, 0) # No explode this time.
# Create the pie chart:
plt.clf()
plt.pie(fractions, colors=colors, labels=labels,
        autopct='%1.1f%%', startangle=130,
        pctdistance=0.85, explode=explode)
# Draw a white circle at the center:
center_circle = plt.Circle((0, 0), 0.70, facecolor='white')
plt.gcf().gca().add_artist(center_circle)
# Display the pie chart:
plt.axis('equal')
plt.show()
```

The output is shown in Fig. 46.
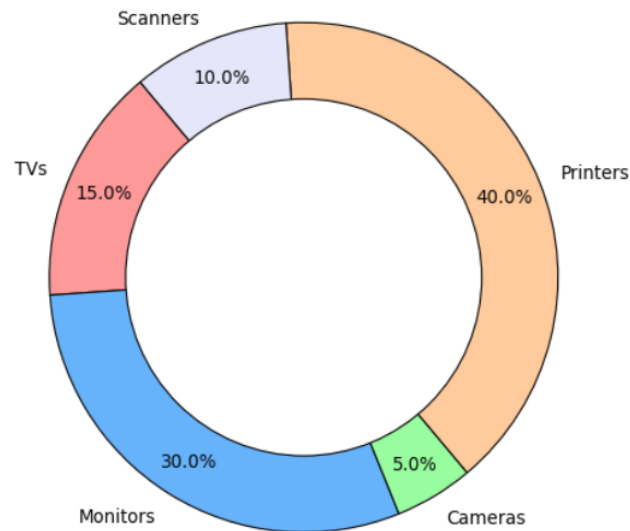
Fig. 46. Donut chart with custom colors and rotation.

Matplotlib provides more options for creating pie charts. It is easy to find many additional examples by typing "matplotlib pie" in your web browser.

3.32 Bar charts

Bar charts are a great way to visualize sequential data such as, for example, the yearly growth of a company's revenues, number of daily visits to a web page, weekly rate of unemployment in the state, etc. The simplest bar chart involves just 6 lines of code:

```python
import matplotlib.pyplot as plt
data = [450, 550, 300]
positions = [1, 2, 3]
plt.clf()
plt.bar(positions, data)
plt.show()
```

The array `[1, 2, 3]` specifies the positions on the horizontal axis where the bars begin. The default width of the bars is 0.8 and it can be adjusted via the parameter `width` of the `bar` function. The output is shown in Fig. 47.
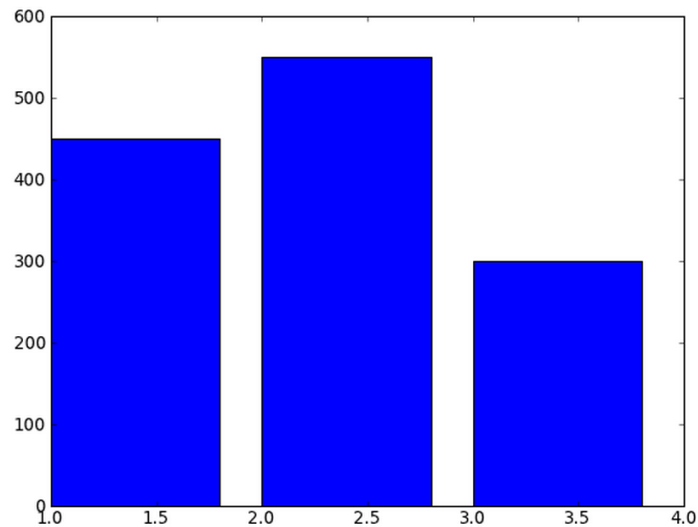
Fig. 47. Basic bar chart.

Next let us present a more advanced bar chart that shows a company's growth both in the total number of employees and the yearly increments. Again, do not worry if you do not understand all details in the code. You need to understand enough to be able to tweak such an example to display your own data - increase or decrease the number of columns, change the values, etc.

```
# Import Matplotlib, and define totals and increments:
import matplotlib.pyplot as plt
totals = (20, 25, 32, 42, 61)
increments = [5, 7, 10, 19, 30]

# Positions and width of the bars:
positions = [0, 1, 2, 3, 4]
width = 0.4

# Create figure and subplot, define bars:
fig = plt.figure()
ax = fig.add_subplot(111)
bars1 = ax.bar(ind, totals, width, color='y')
bars2 = ax.bar(ind + width, increments, width, color='b')
```

50

```
# Add descriptions:
ax.set_ylabel('Numbers')
ax.set_title('Number of employees')
ax.set_xticks(ind + width)
ax.set_xticklabels( ('2008', '2009', '2010', '2011', '2012') )

# Function to add numbers above bars:
def autolabel(rects):
    for rect in rects:
        height = rect.get_height()
        ax.text(rect.get_x() + rect.get_width() / 2,
                height + 0.02 * max(totals), '%d'%int(height),
                ha = 'center', va = 'bottom')

# Create the bars and display the bar chart:
autolabel(bars1)
autolabel(bars2)
plt.show()
```

The output is shown in Fig. 48.
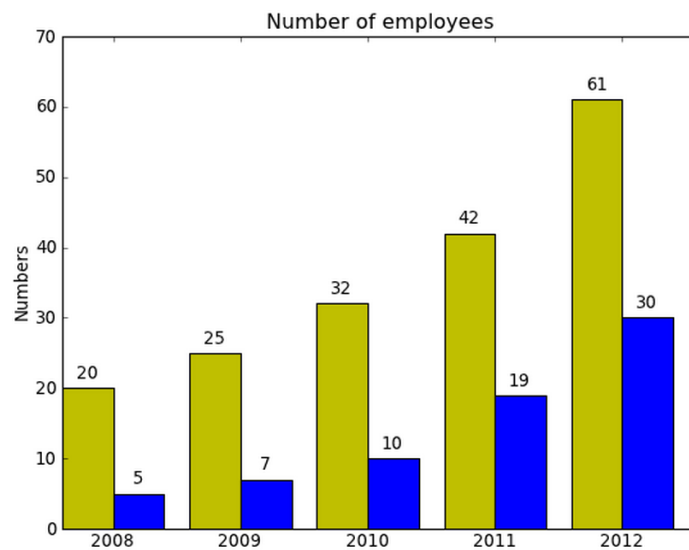
Fig. 48. Bar chart showing a company's growth.

51

The functionality provided by Matplotlib is so extensive that we could spend an almost unlimited amount of time exploring it. A huge number of examples are available online. At this point we need to move on to other aspects of Python, but we invite the reader to learn more about Matplotlib at `http://matplotlib.org`.

3.33 Statistical data visualization with the Seaborn library

Another powerful Python library for data visualization is Seaborn:

$$\text{https://seaborn.pydata.org}$$

This library is based on Matplotlib, and its primary goal is to facilitate the visualization of statistical data such as densities, histograms, kernel density estimations (KDEs), heatmaps, box plots, scatter plots, violin plots, etc. A nice overview of the various types of visualizations that can be done with Seaborn can be found at

$$\text{http://seaborn.pydata.org/examples}$$

Seaborn is free and distributed under the BSD License. Fig. 49 shows a sample visualization of histograms and KDEs.
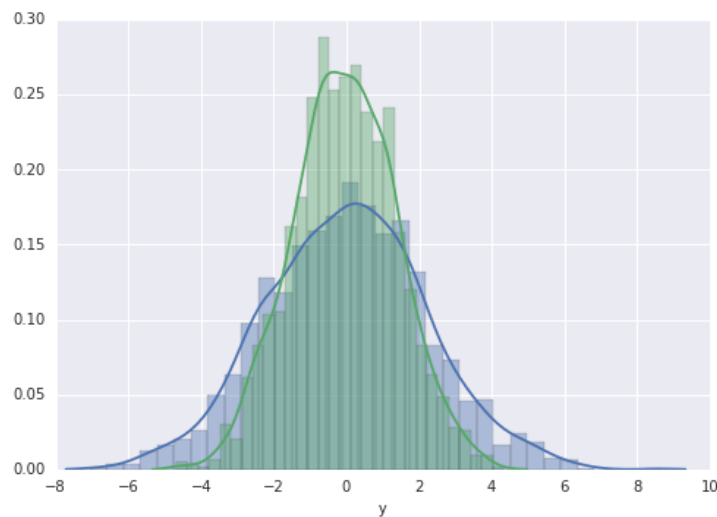
Fig. 49. Sample histograms and kernel density estimations (KDEs).

3.34   Interactive scientific data visualization with Mayavi2

We can't finish this section without mentioning Mayavi:

```
http://docs.enthought.com/mayavi/mayavi
```

Mayavi, and in particular the newer version Mayavi2 is a general purpose, cross-platform tool for 3D scientific visualization of scalar, vector, and tensor data in two and three spatial dimensions. It can be used both in an interactive mode via a GUI, and as a library from withim Python programs. Mayavi2 is free and distributed under the BSD License.
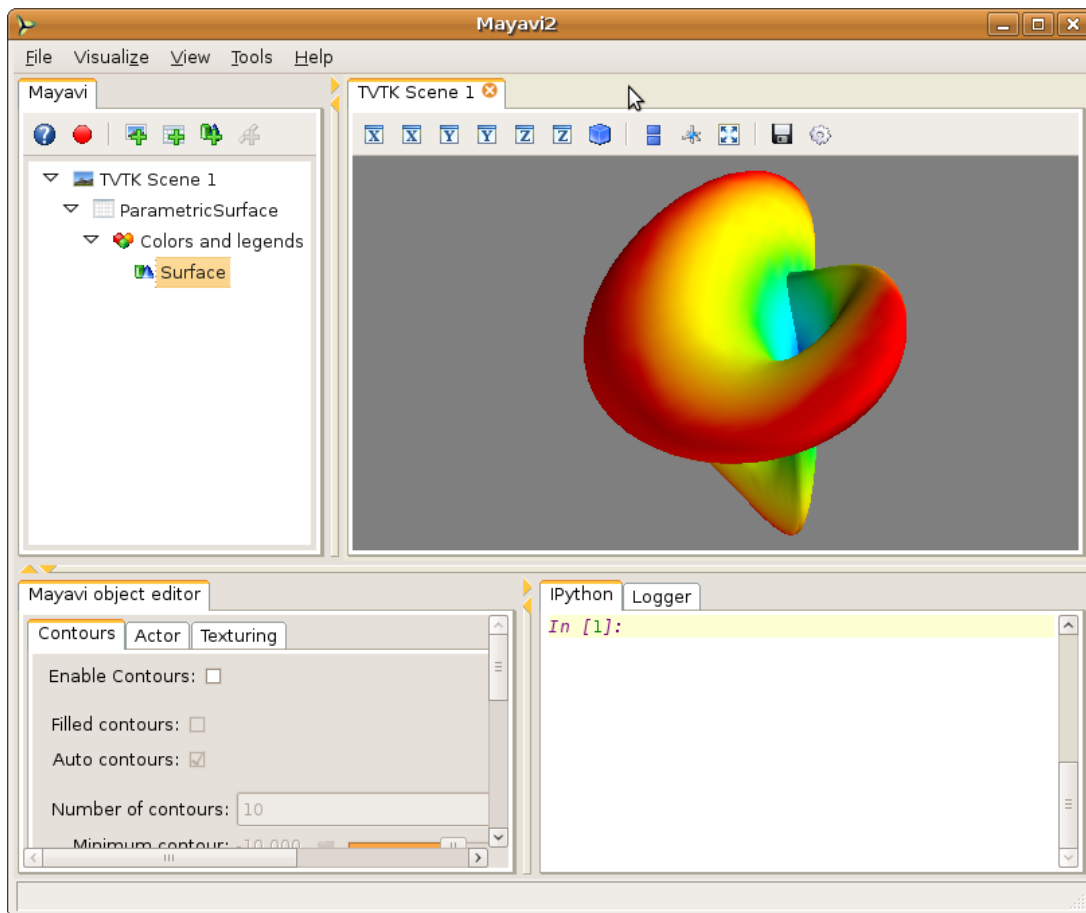
Fig. 50. Graphical user interface (GUI) of Mayavi2.

53

4 Working with Text Strings

4.1 Objectives

In this Section we will learn how to:

- Define text strings and display them with the `print` function.
- Use the `repr` function to identify and remove trailing spaces.
- Measure the length of text strings.
- Use single and double quotes, and the newline character `\n`.
- Concatenate and repeat text strings.
- Access characters by their indices.
- Parse text strings one character at a time,
- Slice, copy, and reverse text strings.
- Check for substrings and count their occurrences.
- Search for and replace substrings.
- Decompose a text string into a list of words.

4.2 Why learn about text strings?

By a *text string* or just *string* we mean a sequence of characters (text). More than 80% of work computers do is processing text. Therefore, becoming fluent in working with text strings is an absolute must for every programmer. Text strings can be used to make outputs of computer programs more informative, but their primary application is to represent data in various databases:

- registry of drivers in the DMV,
- database of customers in a company,
- database of goods in a warehouse,
- to store your tweets on Twitter,
- to store your blog posts,
- in phone books,
- in dictionaries, etc.

Before doing any higher-level programming, one needs to understand text strings very well, and be aware of various things that can be done with them.

4.3 Defining text strings

When defining a text string, it does not matter if the sequence of characters is enclosed in single or double quotes:

```
'Talk is cheap. Show me the code. (Linus Torvalds)'
"Life is not fair. Get used to it. (Bill Gates)"
```

The enclosing quotes are not part of the text string. Sometimes one might want to make them part of a text string, such as when including direct speech. This can be a bit tricky - we are going to talk about it in Subsection 4.14.

Most of the time, text strings in computer programs are shorter, and sometimes then can even be empty. All these are admissible text strings:

```
''
' '
'   '
```

Here only the first text string is empty (=contains no characters). The other two contain empty space characters ' ' which makes them non-empty. They are all different text strings for Python.

4.4 Storing text strings in variables

Just typing a raw text string as above would not be very useful - text strings need to be stored, processed and/or displayed. To store a text string in a variable named (for example) `mytext`, type

```
mytext = 'I want to make a ding in the universe. (Steve Jobs)'
```

When a variable stores a text string, we call it a *text string variable*.

4.5 Measuring the length of text strings

Python has a built-in function `len` to measure the length of text strings. It works on raw text strings,

```
print(len('Talk is cheap. Show me the code.'))
32
```

as well as on text string variables:

```
txt = 'Talk is cheap. Show me the code.'
print(len(txt))
```

4.6   Python is case-sensitive

We will talk about variables in much more detail in Section 5, but for now let's mention that `mytext`, `Mytext`, `MyText` and `MYTEXT` are four different variables. Python is case-sensitive, meaning that it distinguishes between lowercase and uppercase characters. Also, `'Hello!'` and `'hello!'` are different text strings in Python.

4.7   The `print` function

Python can display raw text strings, text string variables, numbers and other things using the function `print`. Here is a simple example displaying just one text string:

```python
print('I am a text string.')
```
```
I am a text string.
```

Notice that the enclosing quotes are not displayed. The next example displays a text string and a text string variable:

```python
name = 'Jennifer'
print('Her name was', name)
```
```
Her name was Jennifer
```

Notice that the displayed text strings are separated with one empty space `' '`. And here is one more example which displays a text string and a number:

```python
name = 'Jennifer'
print('The final answer:', 42)
```
```
The final answer: 42
```

Here the `print` function automatically converted the number `42` into a text string `'42'`. Then an empty space was inserted between the two displayed text strings again.

Separating items with one empty space is the default behavior of the `print` function. In a moment we are going to show you how to change it, but first let's introduce a super-useful function `help`.

4.8 Function `help`

Python has a built-in function `help` which can be used to obtain more information about other built-in functions, operators, classes, and other things. Let's use it to learn more about the function `print`:

```python
help(print)
```

Output:

```
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file: a file-like object (stream); defaults to the current
    sys.stdout.
    sep:  string inserted between values, default a space.
    end:  string appended after the last value, default a
    newline.
```

This explains a number of things! The default separator of items `sep` is one empty space `' '`. Further, the `print` function adds by default the newline character `\n` when it's done with displaying all items. We will talk more about the newline character in Subsection 4.17. Last, `file=sys.stdout` means that by default, items are displayed using standard output. One can change the destination to a file if desired. But this is not needed very often as there are other standard ways to write data to files. We will talk about files and file operations in Section 13.

4.9 Changing the default behavior of the `print` function

Being aware of the optional parameters in the `print` function can be useful sometimes. First let's try to improve the previous example by adding a period to properly end the sentence:

```python
name = 'Jennifer'
print('Her name was', name, '.')
Her name was Jennifer .
```

57

Uh-oh, that does not look very good. So let's change the separator `sep` to an empty text string '' and retry:

```
name = 'Jennifer'
print('Her name was ', name, '.', sep='')
Her name was Jennifer.
```

That looks much better! Notice that we altered the string 'Her name was' to 'Her name was ' in order to compensate for the default ' ' which was eliminated.

Changing the default value of `end` might be useful when we want to display items on the same line. Namely, by default, each call to the `print` function produces a new line:

```
prime1 = 2
prime2 = 3
prime3 = 5
print(prime1)
print(prime2)
print(prime3)
2
3
5
```

So let's change `end` to an empty space ' ':

```
prime1 = 2
prime2 = 3
prime3 = 5
print(prime1, end=' ')
print(prime2, end=' ')
print(prime3)
2 3 5
```

Well, this example is a bit artificial because one could just type

```
print(prime1, prime2, prime3)
```

But imagine that we are calculating and displaying the first 1000 primes, one at a time, using a `for` loop. (The `for` loop will be introduced in Subsection 4.22.)

4.10 Undesired trailing spaces and function `repr`

By *trailing spaces* we mean empty spaces which are present at the end of a text string. They can get there via user input, when decomposing a large text into sentences or words, and in other ways. Most of the time they are "dirt", and they can be the source of nasty bugs which are difficult to find.

For illustration, let's print the value of a variable named `city`:

```
print(city)
Portland
```

Great! Then the text string stored in the variable `city` must be `'Portland'`, right? Hmm - nope. Let's print the value of this variable using the function `repr` which reveals empty spaces and special characters:

```
print(repr(city))
'Portland   '
```

Now if our program was checking the variable `city` against a database of cities which contains `'Portland'`, it would not be found because `'Portland   '` and `'Portland'` are different text strings. That's what we mean by saying that trailing spaces are unwanted. The function `repr` is very useful because it helps us to find them. In the next subsection we will show you how to remove them.

4.11 Cleaning text strings with `strip`, `lstrip`, and `rstrip`

Python can remove trailing spaces and other undesired characters both from the right end and left end of a text string using text string methods `strip`, `lstrip`, and `rstrip`. Usually, in addition to empty spaces `' '`, undesired are also newline characters `\n` and tab characters `\t`. The string method `strip` will remove all of them both from the right end and the left end of the text string:

```
state = '  \t Alaska  \n  '
print(repr(state))
state = state.strip()
print(repr(state))
'  \t Alaska  \n  '
'Alaska'
```

Note that calling just `state.strip()` would not work. The text string `state` would not be changed because the method `strip` returns a new text string. The line `state = state.strip()` actually does two things:

1. Calling `state.strip()` creates a copy of the text string `state`, removes the undesired characters, and returns it.
2. The new cleaned text string `'Alaska'` is then copied back into the variable `state`, which overwrites the original text `'  \t Alaska \n '` that was there before.

The method `rstrip` does the same thing as `strip`, but only at the right end of the text string:

```
state = '   \t Alaska   \n   '
print(repr(state))
state = state.rstrip()
print(repr(state))
```
```
'   \t Alaska   \n   '
'   \t Alaska'
```

And last, the method `lstrip` does the same thing as `strip`, but only at the left end of the text string:

```
state = '   \t Alaska   \n   '
print(repr(state))
state = state.lstrip()
print(repr(state))
```
```
'   \t Alaska   \n   '
'Alaska   \n   '
```

4.12   Wait - what is a "method" exactly?

We owe you an explanation of the word "method" which we used in the previous subsection. In Python, almost everything is an object, including text strings. We will discuss object-oriented programming in great detail in Sections 14 and 15. For now let's just say that every object has *attributes* (data) and *methods* (functions that work with the data). So, every text string that you create automatically has the methods `strip`, `rstrip` and `lstrip` (and many more). They are called by appending a period `'.'` to the name of the text string variable, and typing the name of the method. For example, this is how one calls the method `strip` of the text string `state`:

$$state.strip()$$

4.13   Calling text string methods on raw text strings

As a matter of fact, text string methods can also be called on raw text strings. This is not used very frequently, but it's good to know about it:

```
state = '  \t Alaska  \n  '.strip()
print(repr(state))
```

```
'Alaska'
```

4.14   Using single and double quotes in text strings

Since text strings are enclosed in quotes, it can be expected that there will be some limitations on using quotes inside text strings. Indeed, try this:

```
txt = "I said: "Hello!""
print(txt)
```

Already the syntax highlighting tells you that something is wrong - the first part `"I said: "` is understood as a text string, but the next `Hello!` is not a text string because it is outside the quotes. Then `""` is understood as an empty text string. The error message confirms that Python does not like this:

```
on line 1:
    "I said: "Hello!""
              ^
SyntaxError: Something is missing between "I said: " and Hello.
```

Python probably expected something like `"I said: " + Hello` ... which would make sense if `Hello` was a text string variable, or `"I said: " * Hello` ... which would make sense if `Hello` was an integer. We will talk about adding strings and multiplying them with integers in Subsections 4.20 and 4.21, respectively.

The above example can be fixed by replacing the outer double quotes with single quotes, which removes the ambiguity:

```
txt = 'I said: "Hello!"'
print(txt)
```

```
I said: "Hello!"
```

The other way would work too - replacing the inner double quotes with single quotes removes the ambiguity as well:

```
txt = "I said: 'Hello!'"
print(txt)
```
```
I said: 'Hello!'
```

However, if we needed to combine single and double quotes in the same text string,

```
I said: "It's OK!"
```

this simple solution would not work. Let's show one approach that works always.

4.15   A more robust approach - using characters \' and \"

If we want to include single or double quotes in a text string, the safest way is to use them with a backslash:

```
txt = "I said: \"It\'s OK!\""
print(txt)
```
```
I said: "It's OK!"
```

4.16   Length of text strings containing special characters

It's useful to know that the special characters \' and \", although they consist of two regular characters, have length 1:

```
print(len('\"'))
print(len('\''))
```
```
1
1
```

And one more example:

```
txt = "\"It\'s OK!\""
print(txt)
print(len(txt))
```
```
"It's OK!"
10
```

4.17 Newline character \n

The newline character \n can be inserted anywhere in a text string. It will cause the print function to end the line at that point and continue with a new line:

```python
txt = "Line 1\nLine 2\nLine 3"
print(txt)
```

```
Line 1
Line 2
Line 3
```

The length of the newline character \n is 1:

```python
print(len('\n'))
```

```
1
```

4.18 Writing long code lines over multiple lines with the backslash \

Here is a paragraph from "The Raven" by Edgar Allan Poe:

Once upon a midnight dreary, while I pondered weak and weary,
Over many a quaint and curious volume of forgotten lore,
While I nodded, nearly napping, suddenly there came a tapping,
As of some one gently rapping, rapping at my chamber door.
'Tis some visitor,' I muttered, 'tapping at my chamber door -
Only this, and nothing more.'

Let's say that we want to store this text in a text string variable named edgar. You already know how to use the newline character \n, so it should not be a problem to create the text string using one very long line of code.

However, how can this be done when the Python's PEP8 style guide

```
https://www.python.org/dev/peps/pep-0008/
```

recommends to limit all lines of code to a maximum of 79 characters? The answer is - one can spread a long line of code over two or more lines using the backslash \:

```
edgar = "Once upon a midnight dreary, while I pondered weak \
and weary,\nOver many a quaint and curious volume of forgotten \
lore,\nWhile I nodded, nearly napping, suddenly there came a \
tapping,\nAs of some one gently rapping, rapping at my chamber \
door.\n'Tis some visitor,' I muttered, 'tapping at my chamber \
door -\nOnly this, and nothing more.'"
print(edgar)
```

Here is how the text string looks when displayed:

```
Once upon a midnight dreary, while I pondered weak and weary,
Over many a quaint and curious volume of forgotten lore,
While I nodded, nearly napping, suddenly there came a tapping,
As of some one gently rapping, rapping at my chamber door.
'Tis some visitor,' I muttered, 'tapping at my chamber door -
Only this, and nothing more.'
```

4.19 Multiline strings enclosed in triple quotes

Creating multiline text strings with the newline character \n certainly works for most applications. But sometimes, especially when the text string is very long, it can be cumbersome. Therefore, Python makes it possible to create multiline text strings enclosed in triple quotes. Then one does not have to worry about any special characters at all. Here is the previous code, transformed in this way:

```
edgar = """\
Once upon a midnight dreary, while I pondered weak and weary,
Over many a quaint and curious volume of forgotten lore,
While I nodded, nearly napping, suddenly there came a tapping,
As of some one gently rapping, rapping at my chamber door.
'Tis some visitor,' I muttered, 'tapping at my chamber door -
Only this, and nothing more.'\
"""
print(edgar)
```
```
Once upon a midnight dreary, while I pondered weak and weary,
Over many a quaint and curious volume of forgotten lore,
While I nodded, nearly napping, suddenly there came a tapping,
As of some one gently rapping, rapping at my chamber door.
'Tis some visitor,' I muttered, 'tapping at my chamber door -
Only this, and nothing more.'
```

In case you wonder about the backslashes - they are there to prevent the text string to begin and end with \n (we indeed started and ended the text string with making a new line).

Here is one more example. The code

```
txt = """
This is the first line,
and this is the second line.
"""
print(txt)
```

will display one empty line at the beginning and one at the end:

```
This is the first line,
and this is the second line.
```

Finally, let's show that creating a multiline text string using the newline character \n and using the triple quote approach are fully equivalent:

```
txt1 = "Line 1\nLine 2\nLine 3"
txt2 = """Line 1
Line 2
Line 3"""
print(repr(txt1))
print(repr(txt2))
```
```
'Line 1\nLine 2\nLine 3'
'Line 1\nLine 2\nLine 3'
```

4.20   Adding text strings

Text strings can be concatenated (added together) using the same '+' operator as numbers:

65

```
word1 = 'Edgar'
word2 = 'Allan'
word3 = 'Poe'
name = word1 + word2 + word3
print(name)
```
```
EdgarAllanPoe
```

Oops! Of course we need to add empty spaces if we want them to be part of the resulting text string:

```
word1 = 'Edgar'
word2 = 'Allan'
word3 = 'Poe'
name = word1 + ' ' + word2 + ' ' + word3
print(name)
```
```
Edgar Allan Poe
```

4.21   Multiplying text strings with integers

Text strings can be repeated by multiplying them with positive integers. For example:

```
txt = 'Help me!'
print('I yelled' + 3 * word)
```
```
I yelledHelp me!Help me!Help me!
```

Again, empty spaces matter! So let's try again:

```
txt = ' Help me!'
print('I yelled' + 3 * word)
```
```
I yelled Help me! Help me! Help me!
```

4.22   Parsing text strings with the `for` loop

Python has a keyword `for` which can be used to form a `for` loop, and parse a text string one character at a time:

```
word = 'breakfast'
for c in word:
    print(c, end=' ')
```
```
b r e a k f a s t
```

The mandatory parts of the loop are the keywords `for` and `in`, and the colon `:` at the end of the line. The name of the variable is up to you. For example, with `letter` instead of `c` the above code would look like this:

```
word = 'breakfast'
for letter in word:
    print(letter, end=' ')
```
```
b r e a k f a s t
```

The action or sequence of actions to be repeated for each character is called the *body* of the loop. In this case it is a single line

```
print(letter, end=' ')
```

but often the loop's body is several lines long.

Importantly, note that that the body of the loop is *indented*. The Python style guide `https://www.python.org/dev/peps/pep-0008/` recommends to use 4-indents. The `for` loop will be discussed in more detail in Section 9.

4.23   Reversing text strings with the `for` loop

The `for` loop can be used to reverse text strings. The following program shows how to do it, and moreover for clarity it displays the result-in-progress text string `new` after each cycle of the loop:

```
orig = 'breakfast'
new = ''
for c in orig:
    new = c + new
    print(new)
```
```
b
rb
erb
aerb
kaerb
fkaerb
afkaerb
safkaerb
tsafkaerb
```

Note that this code created a new text string and the original text string stayed unchanged. The reason is that text strings cannot be changed in place – they are *immutable* objects. We will talk in more detail about mutable and immutablle types in Subsection 7.33. And last, in Subsection 4.32 we will show you a quicker way to reverse text strings based on *slicing*.

4.24 Accessing individual characters via their indices

All characters in the text string are enumerated. The first character has index 0, the second one has index 1, the third one has index 2, etc. The character with index `n` can be obtained when appending `[n]` at the end of the text string:

```python
word = 'fast'
print('First character:', word[0])
print('Second character:', word[1])
print('Third character:', word[2])
print('Fourth character:', word[3])
```
```
First character: f
Second character: a
Third character: s
Fourth character: t
```

Make sure that you remember:

Indices in Python start from zero.

4.25 Using negative indices

Sometimes it can be handy to use the index -1 for the last character, -2 for the one-before-last etc:

```python
word = 'fast'
print('Fourth character:', word[-1])
print('Third character:', word[-2])
print('Second character:', word[-3])
print('First character:', word[-4])
```
```
Fourth character: t
Third character: s
Second character: a
First character: f
```

4.26 ASCII table

Every standard text character has its own ASCII code which is used to represent it in the computer memory. ASCII stands for *American Standard Code for Information Interchange*. The table dates back to the 1960s and it contains 256 codes. The first 128 codes are summarized in Fig. 51.

0	NUL	16	DLE	32	SPC	48	0	64	@	80	P	96	`	112	p
1	SOH	17	DC1	33	!	49	1	65	A	81	Q	97	a	113	q
2	STX	18	DC2	34	"	50	2	66	B	82	R	98	b	114	r
3	ETX	19	DC3	35	#	51	3	67	C	83	S	99	c	115	s
4	EOT	20	DC4	36	$	52	4	68	D	84	T	100	d	116	t
5	ENQ	21	NAK	37	%	53	5	69	E	85	U	101	e	117	u
6	ACK	22	SYN	38	&	54	6	70	F	86	V	102	f	118	v
7	BEL	23	ETB	39	'	55	7	71	G	87	W	103	g	119	w
8	BS	24	CAN	40	(	56	8	72	H	88	X	104	h	120	x
9	HT	25	EM	41	)	57	9	73	I	89	Y	105	i	121	y
10	LF	26	SUB	42	*	58	:	74	J	90	Z	106	j	122	z
11	VT	27	ESC	43	+	59	;	75	K	91	[	107	k	123	{
12	FF	28	FS	44	,	60	<	76	L	92	\	108	l	124		
13	CR	29	GS	45	-	61	=	77	M	93	]	109	m	125	}
14	SO	30	RS	46	.	62	>	78	N	94	^	110	n	126	~
15	SI	31	US	47	/	63	?	79	O	95	_	111	o	127	DEL

Fig. 51. ASCII table, codes 0 - 127.

The first 32 codes 0 - 31 represent various non-printable characters, of which some are hardly used today. But some are still used, such as code 8 which means backspace \b, code 9 which means horizontal tab \t, and code 10 which means new line \n. The upper ASCII table (codes 128 - 255) represents various special characters which you can easily find online if you need them.

4.27 Finding the ASCII code of a given character

Python has a built-in function `ord` which can be used to access ASCII codes of text characters:

```python
print(ord('a'))
```

69

```
97
```

The text character representing the first digit `'0'` has ASCII code 48:

```
print(ord('0'))
48
```

Since you already know from Subsection 4.22 how to parse text strings with the `for` loop, it is a simple exercise to convert a text string into a sequence of ASCII codes automatically:

```
txt = 'breakfast'
for c in txt:
    print(ord(c), end=' ')
98 114 101 97 107 102 97 115 116
```

4.28   Finding the character for a given ASCII code

This is the inverse task to what we did in the previous subsection. Python has a function `chr` which accepts an ASCII code and returns the corresponding character:

```
print(chr(97))
a
```

It is a simple exercise to convert a list of ASCII codes into a text string:

```
L = [98, 114, 101, 97, 107, 102, 97, 115, 116]
txt = ''
for n in L:
    txt += chr(n)
print(txt)
breakfast
```

Lists will be discussed in more detail in Section 7.

4.29   Slicing text strings

Python makes it easy to extract substrings of text strings using indices – this is called *slicing*:

```
w1 = 'bicycle'
w2 = w1[1:4]
print(w2)
```
```
icy
```

Note that second index (in this case 4) minus the first one (in this case 1) yields the length of the resulting text string. This also means that the character with the second index (in this case `w1[4]` which is `'c'`) is not part of the resulting slice.

Omitting the first index in the slice defaults to zero:

```
w3 = w1[:2]
print(w3)
```
```
bi
```

Omitting the second index defaults to the length of the string:

```
w4 = w1[2:]
print(w4)
```
```
cycle
```

4.30 Third index in the slice

The slice notation allows for a third index which stands for *stride*. By default it is 1. Setting it to 2 will extract every other character:

```
orig = 'circumstances'
new1 = orig[::2]
print(new1)
```
```
crusacs
```

Here we omitted the first and second indices in the slice, which means that the entire text string was used.

4.31 Creating copies of text strings

The easiest way to create a new copy of a text string is as follows:

```
orig = 'circumstances'
new2 = orig
print(new2)
```
```
circumstances
```

Equivalently, one can use slicing. In the case of text strings these two approaches are equivalent because they are immutable objects:

```
orig = 'circumstances'
new3 = orig[:]
print(new3)
```
```
circumstances
```

Mutability and immutability of objects in Python will be discussed in more detail in Subsection 7.33.

4.32   Reversing text strings using slicing

When the stride is set to -1, the text string will be reversed. This is the fastest and easiest way to reverse text strings:

```
orig = 'circumstances'
new4 = orig[::-1]
print(new4)
```
```
secnatsmucric
```

Note that typing just `orig[::-1]` will not change the text string `orig` because text strings are immutable.

4.33   Retrieving current date and time

Extracting system date and time, and transforming it to various other formats is a nice way to practice slicing. To begin with, let's import the `time` library and call the function `ctime`:

```
import time
txt = time.ctime()
print(txt)
print(len(txt))
```

```
Mon May 11 18:23:03 2018
24
```

The function `ctime` returns a 24-character text string. Here is its structure:

```python
print(txt[:3])     # day (three characters)
print(txt[4:7])    # month (three characters)
print(txt[8:10])   # date (two characters)
print(txt[11:19])  # hh:mm:ss (eight characters)
print(txt[20:])    # year (four characters)
```
```
Mon
May
11
18:23:03
2018
```

For example, here is a code to display the date in a different format `May 11, 2018`:

```python
import time
txt = time.ctime()
newdate = txt[4:7] + ' ' + txt[8:10] + ', ' + txt[20:]
print(newdate)
```
```
May 11, 2018
```

4.34   Making text strings lowercase

This method returns a copy of the text string where all characters are converted to lowercase. It has an important application in search where it is used to make the search case-insensitive. We will talk about this in more detail in Subsection 4.36. Meanwhile, the following example illustrates how the method `lower` works:

```python
txt = 'She lives in New Orleans.'
new = txt.lower()
print(new)
```
```
she lives in new orleans.
```

4.35   Checking for substrings in a text string

Python has a keyword `in` which you already know from Subsection 4.22. There it was used as part of the `for` loop to parse text strings. The same keyword can be used to

check for occurrences of substrings in text strings. The expression

$$\texttt{substr in txt}$$

returns `True` if substring `substr` is present in text string `txt`, and `False` otherwise. Here `True` and `False` are Boolean values which will be discussed in more detail in Section 6. In the meantime, here is an example that illustrates the search for substrings:

```
txt = 'Adam, Michelle, Anita, Zoe, David, Peter'
name = 'Zoe'
result = name in txt
print(result)
```
```
True
```

Usually, the search for a substring is combined with an `if-else` statement. Conditions will be discussed in more detail in Section 10, but let's show a simple example:

```
txt = 'Adam, Michelle, Anita, Zoe, David, Peter'
name = 'Hunter'
if name in txt:
    print('The name ' + name + ' is in the text.')
else:
    print('The name ' + name + ' was not found.')
```
```
The name Hunter was not found.
```

Here, the expression `name in txt` returned `False`. Therefore the condition was not satisfied, and the `else` branch was executed.

4.36 Making the search case-insensitive

As you know, Python is case-sensitive. However, when searching in text strings, one often prefers to make the search case-insensitive to avoid missing some occurrences. For example, searching the following text string `txt` for the word `'planet'` fails for this reason:

```
txt = 'Planet Earth is part of the solar system.'
substr = 'planet'
result = substr in txt
print(result)
```

```
False
```

The solution is to lowercase both text strings, which makes the search case-insensitive:

```
txt = 'Planet Earth is part of the solar system.'
substr = 'planet'
result = substr.lower() in txt.lower()
print(result)
True
```

4.37 Making text strings uppercase

This method works analogously to `lower` except it returns an uppercased copy of the text string:

```
txt = 'don't yell at me like that.'
new = txt.upper()
print(new)
DON'T YELL AT ME LIKE THAT.
```

4.38 Finding and replacing substrings

Method `replace(old, new[, count])` returns a copy of the original text string where all occurrences of text string `old` are replaced by text string `new`:

```
txt = 'First day, second day, and third day.'
substr1 = 'day'
substr2 = 'week'
new = txt.replace(substr1, substr2)
print(new)
First week, second week, and third week.
```

If the optional argument `count` is given, only the first `count` occurrences are replaced:

```
txt = 'First day, second day, and third day.'
substr1 = 'day'
substr2 = 'week'
new = txt.replace(substr1, substr2, 2)
print(new)
First week, second week, and third day.
```

4.39 Counting occurrences of substrings

Method `count(sub[, start[, end]])` returns the number of occurrences of substring `sub` in the slice `[start:end]` of the original text string. If the optional parameters `start` and `end` are left out, the whole text string is used:

```
txt = 'John Stevenson, John Lennon, and John Wayne.'
print(txt.count('John'))
3
```

What we said in Subsection 4.36 about the disadvantages of case-sensitive search applies here as well. If there is a chance that some occurrences will differ by the case of characters, it is necessary to make the search case-insensitive by lowercasing both text strings:

```
txt = 'Planet Earth, planet Venus, planet Uranus, planet Mars.'
substr = 'planet'
print(txt.lower().count(substr.lower()))
4
```

4.40 Locating substrings in text strings

Method `find(sub[, start[, end]])` returns the lowest index in the text string where substring `sub` is found, such that `sub` is contained in the range `[start, end]`. If the optional arguments `start` and `end` are omitted, the whole text string is used. The method returns −1 if the text string `sub` is not found:

```
txt = 'They arrived at the end of summer.'
substr = 'end'
print(txt.find(substr))
20
```

Case-insensitive version:

```
txt = 'They arrived at the end of summer.'
substr = 'end'
print(txt.lower().find(substr.lower()))
20
```

Python also has a method `index(sub[, start[, end]])` which works like `find`, but raises `ValueError` instead of returning `-1` when the substring is not found.

4.41 Splitting a text string into a list of words

Any text string can be split into a list of words using the string method `split`. (Lists will be discussed in more detail in Section 7.) If no extra arguments are given, the words will be separated by arbitrary strings of whitespace characters (space, tab, newline, return, formfeed, ...):

```
txt = 'This is, indeed, a basic use of the split method!'
L = txt.split()
print(L)
```
```
['This', 'is,', 'indeed,', 'a', 'basic', 'use', 'of', 'the',
'split', 'method!']
```

It is possible to use another separator by passing it to the method as an optional argument:

```
txt = 'This is, indeed, a basic use of the split method!'
L = txt.split(',')
print(L)
```
```
['This is', ' indeed', ' a basic use of the split method!']
```

4.42 Splitting a text string while removing punctuation

Notice that in the previous subsection, the commas `','` and the exclamation mark `'!'` remained in the words `'is,'`, `'indeed,'` and `'method!'`. If we want to remove them, redefining the delimiter does not help.

In this case it is best to do it manually – first replace any undesired characters with empty space `' '` and then use the standard `split` method without any extra arguments. If we know that we only want to remove `','` and `'!'`, we can do it one by one as follows:

```
txt = 'This is, indeed, a basic use of the split method!'
txt2 = txt.replace(',', ' ')
txt2 = txt2.replace('!', ' ')
L = txt2.split()
print(L)
```

```
['This', 'is', 'indeed', 'a', 'basic', 'use', 'of', 'the',
'split', 'method']
```

When we deal with a larger text and want to remove all punctuation characters, we can take advantage of the text string `punctuation` which is present in the `string` library:

```python
import string
txt = 'This is, indeed, a basic use of the split method!'
txt2 = txt[:]
for c in string.punctuation:
    txt2 = txt2.replace(c, ' ')
L = txt2.split()
print(L)
```

```
['This', 'is', 'indeed', 'a', 'basic', 'use', 'of', 'the',
'split', 'method']
```

4.43 Joining a list of words into a text string

Method `join(L)` returns a text string which is the concatenation of the strings in the list `L`. In fact, `L` can also be a tuple or any sequence. The base string is used as separator:

```python
txt = '...'
str1 = 'This'
str2 = 'is'
str3 = 'the'
str4 = 'movie.'
print(txt.join([str1, str2, str3, str4]))
```

```
This...is...the...movie.
```

4.44 Method `isalnum`

This method returns `True` if all characters in the string are alphanumeric (letters or numbers, and if there is at least one character. Otherwise it returns `False`:

```
str1 = 'Hello'
if str1.isalnum():
    print(str1, 'is alphanumeric.')
else:
    print(str1, 'is not alphanumeric.')
str2 = 'Hello!'
if str2.isalnum():
    print(str2, 'is alphanumeric.')
else:
    print(str2, 'is not alphanumeric.')
```
```
Hello is alphanumeric.
Hello! is not alphanumeric.
```

4.45 Method `isalpha`

This method returns `True` if all characters in the string are alphabetic and there is at least one character, and `False` otherwise. By 'alphabetic' we mean letters only, not numbers and not even empty spaces:

```
str1 = 'John'
if str1.isalpha():
    print(str1, 'is alphabetic.')
else:
    print(str1, 'is not alphabetic.')
```
```
John is alphabetic.
```

And here is another example:

```
str2 = 'My name is John'
if str2.isalpha():
    print(str2, 'is alphabetic.')
else:
    print(str2, 'is not alphabetic.')
```
```
My name is John is not alphabetic.
```

4.46 Method `isdigit`

This method returns `True` if all characters in the string are digits and there is at least one character, and `False` otherwise:

```
str1 = '2012'
if str1.isdigit():
    print(str1, 'is a number.')
else:
    print(str1, 'is not a number.')
str2 = 'Year 2012'
if str2.isdigit():
    print(str2, 'is a number.')
else:
    print(str2, 'is not a number.')
```
```
2012 is a number.
Year 2012 is not a number.
```

4.47 Method `capitalize`

This method returns a copy of the text string with only its first character capitalized. All remaining characters will be lowercased:

```
txt = 'SENTENCES SHOULD START WITH A CAPITAL LETTER.'
print(txt.capitalize())
```
```
Sentences should start with a capital letter.
```

4.48 Method `title`

This method returns a titlecased version of the string where all words begin with uppercase characters, and all remaining cased characters are lowercase:

```
txt = 'this is the title of my new book'
print(txt.title())
```
```
This Is The Title Of My New Book
```

4.49 C-style string formatting

Python supports C-style string formatting to facilitate creating formatted strings. For example, `%s` can be used to insert a text string:

```
name = 'Prague'
print('The city of %s was the favorite destination.' % name)
```
```
The city of Prague was the favorite destination.
```

Formatting string `%d` can be used to insert an integer:

```python
num = 30
print('At least %d applicants passed the test.' % num)
At least 30 applicants passed the test.
```

Formatting string `%.Nf` can be used to insert a real number rounded to N decimal digits:

```python
import numpy as np
print('%.3f is Pi rounded to three decimal digits.' % np.pi)
3.142 is Pi rounded to three decimal digits.
```

And last, it is possible to insert multiple strings / integers / real numbers as follows:

```python
n = 'Jim'
a = 16
w = 180.67
print('%s is %d years old and weights %.0f lbs.' % (n, a, w))
Jim is 16 years old and weights 181 lbs.
```

4.50   Additional string methods

In this section we were only able to show the most frequently used text string methods. The string class has many more methods which can be found in the official Python documentation at `https://docs.python.org/3.3/library/stdtypes.html#string-methods` (scroll down to section "String Methods").

4.51   The `string` library

The `string` library contains additional useful string functionality including string constants (such as all ASCII characters, all lowercase characters, all uppercase characters, all printable characters, all punctuation characters, ...), string formatting functionality, etc. More information can be found at `https://docs.python.org/3/library/string.html`.

4.52   Natural language toolkit (NLTK)

The Natural Language Toolkit (NLTK) available at `https://www.nltk.org/` is a leading platform for building Python programs to work with human language data.

NLTK is a free, open source, community-driven project which is suitable for linguists, engineers, students, educators, researchers, and industry users. NLTK provides easy-to-use interfaces to over 50 corpora and lexical resources such as WordNet, along with a suite of text processing libraries for classification, tokenization, stemming, tagging, parsing, and semantic reasoning, and wrappers for industrial-strength NLP libraries.

5   Variables and Types

5.1   Objectives

In this Section we will learn:

- That Python is a dynamically (weakly) typed language.
- About various types of variables (data types) in Python.
- How to create variables and initialize them with values.
- About dynamic type interpretation.
- How to determine the type of a variable at runtime.
- How to force (cast) a variable to have certain type.
- About *local* and *global* variables.
- That using global variables can lead to problems.
- About *shadowing* of variables.

5.2   Why learn about variables?

Variables are containers. The reason for using them is to store useful information such as values, text strings, etc. This encompasses many different scenarios, some of which you already encountered earlier in this textbook. For example, in Section 2 we used several times the fact that the value of π is stored in Numpy's variable named `pi`:

```python
import numpy as np
print(np.pi)
```
```
3.141592653589793
```

In Section 3 we typed `cmap='jet'`, `cmap='ocean'` or `cmap='viridis'` to store the name of a color map in a variable named `cmap`.

In Section 4 you learned how to parse text strings with the `for` loop:

```python
txt = 'Hello!'
for c in txt:
    print(c, end=' ')
```
```
H e l l o !
```

Here, the variable `c` is used to store the next character from the text string `txt` in each cycle of the `for` loop.

83

5.3 Statically vs. dynamically typed languages

In many languages including C/C++ and Java, the type of each variable must be declared in advance. We say that these languages are *statically (strongly) typed*. In contrast to that, Python is *dynamically (weakly) typed*. This means that the type of each variable is determined at runtime depending on the value that is assigned to it. In this section we will discuss various aspects of working with variables in Python ranging from trivial to advanced.

5.4 Types of variables (data types) in Python

In Python, variables can have the following basic types:

- text string (`str`) ... Sequence of characters enclosed in quotes.
  Example: `txt = "Hello!"`
- Boolean (`bool`) ... Either `True` or `False`.
  Example: `flag = True`
- integer (`int`) ... Integers are unlimited in size and have no maximum value.
  Example: `n = 255`
- real number (`float`) ... Real (floating point) numbers.
  Example: `val = 7.49`
- complex number (`complex`) ... Complex numbers with a real and imaginary part.
  Example: `z = 2 + 3j`

Python also has container data types for more advanced applications:

- list (`list`),
- tuple (`tuple`),
- dictionary (`dict`),
- set (`set`).

In this section we will focus on the basic data types, as well as on various general aspects of variables. Text strings were discussed in great detail in Section 4. Boolean values, variables and expressions will be discussed in Section 6. The container data types will be discussed in Section 7.

5.5 Using a non-initialized variable

Every variable must be initialized before use, otherwise the interpreter will complain:

```
print(5*w)
on line 1:
NameError: name 'w' is not defined
```

5.6 Various ways to create variables and initialize them with values

In Python, every variable is an object, and therefore the process of its initialization is different and more flexible than in statically typed languages. Let's look at integers for example. The most common way, of course, is to type something like:

```python
n = 255
print(n)
```
```
255
```

It is possible to initialize multiple variables with the same value at once:

```python
m = n = p = 0
print(m, n, p)
```
```
0 0 0
```

And it is also possible to pass the desired value to the constructor of the class `int` by typing:

```python
n = int(255)
print(n)
```
```
255
```

The value can be omitted, in which case the default value 0 will be used. In other words, the following line defines an integer variable `m` and initializes it with zero:

```python
m = int()
print(m)
```
```
0
```

As you already know, it is possible to use an existing variable to initialize the value of a new variable:

```python
a = 1
b = 2.5
c = 0.5
d = (a + b) / c
print('d =', d)
```
```
d = 7.0
```

Of course, apples cannot be mixed with oranges. When we try to add a number to a text string, then the interpreter rightfully complains:

```
a = 'My car is a Ferrari.'
b = 3.5
c = a + b
```
```
on line 3:
TypeError: Can't convert 'float' object to str implicitly
```

More to follow in the next subsection.

5.7   Casting variables to text strings

What we showed in the previous subsection can be taken one step further: It is possible to pass a value or variable of any basic type (int, float, bool, ...) or container type (list, tuple, ...) to the constructor of the text string class, converting it into a text string. This is called *casting*. The following example casts a real number to a text string:

```
x = 2.17
xstr = str(x)
xstr
```
```
'2.17'
```

Booleans can be cast to text strings in the same way:

```
flag = False
flagstr = str(flag)
flagstr
```
```
'False'
```

And it works for container data types as well. The last example here casts a list to a text string:

```
L = [2, 3, 5, 7, 11]
Lstr = str(L)
Lstr
```
```
'[2, 3, 5, 7, 11]'
```

86

5.8   Casting can be tricky sometimes

In the previous subsection we have shown that variables of all types can be cast to a text string. This also works the other way round, and other casts are possible, but only if such a cast makes sense. Sometimes the result can be surprising. For example, a Boolean variable can be initialized with a text string (nonempty string = `True`, empty string = `False`). So if the text string is `'False'`, the resulting Boolean will be `True`:

```
txt = 'False'
flag = bool(txt)
flag
```
```
True
```

And,

```
txt = ''
flag = bool(txt)
flag
```
```
False
```

For compatibility with C/C++ and other languages, Booleans can also be initialized with numbers (nonzero = `True`, zero = `False`):

```
num = 2.5
flag = bool(num)
flag
```
```
True
```

and

```
num = 0
flag = bool(num)
flag
```
```
False
```

One last example: A text string containing a list can be cast to a list, but the result might not be what one would expect. The result will not be the original list – instead, it will be a list of individual characters that form the text string:

```
txt = '[1, 2, 3]'
L = list(txt)
L
```
```
['[', '1', ',', ' ', '2', ',', ' ', '3', ']']
```

5.9 Inadmissible casting

If a text string contains a number, it can be cast to integer or float:

```
txt = '10'
val = int(txt)
val
```
```
10
```

However, if one tried to cast to a number a text string which does not represent a number, then the interpreter would rightfully complain:

```
txt = 'Monday'
val = int(txt)
val
```
```
on line 2:
ValueError: invalid literal for int() with base 10: 'Monday'
```

5.10 Dynamic type interpretation and its abuse

In Python, types of variables do not have to be declared in advance. Instead, the type of a variable is determined by the last value that was assigned to it. This also means that the type of a variable can change at runtime. Let us look at the following code:

```
a = 5
print(a)
a = 3.14
print(a)
a = 'Hello!'
print(a)
a = True
print(a)
```
```
5
3.14
Hello!
True
```

Initially, the type of the variable `a` was an integer (`int`), then it was changed to a real number (`float`), then it became a text string (`str`), and finally it became a Boolean variable (`bool`). However, the above code is abusing Python's flexibility, and doing so is a great way to shoot yourself in the leg. There is a good reason for much tighter type control in C/C++ and other languages – it makes the code less prone to mistakes.

Avoid abuse of dynamic type interpretation. A variable, once created for a purpose, should serve that purpose. If you need another variable for something else, just create another variable.

Also:

Have names of variables reflect their contents. For example, `phone_number` is a good name for a variable storing a phone number. Using the name `x17` for such a variable is not against any rules, but it makes your code unreadable.

In general, do not be too inventive with names of variables (any other aspects of your code). You should not leave anything for the reader to figure out – reading your code should not be an intellectual exercise.

5.11   Determining the type of a variable at runtime

The simplest two ways to determine the type of a variable at runtime is either to use the function `isinstance` or the function `type`. Let's begin with the former.

The function `isinstance(var, T)` returns `True` if the variable `var` has type `T` and `False` otherwise:

```
var = 'Hello!'
test = isinstance(var, int)
print(test)
```
```
False
```

And,

```
var = 42
test = isinstance(var, int)
print(test)
```
```
True
```

Here is a slightly more complex example which includes a function definition and an `if-elif-else` statement:

```python
def printtype(a):
    if isinstance(a, str):
        print('a is a text string.')
    elif isinstance(a, int):
        print('a is an integer.')
    elif isinstance(a, float):
        print('a is a real number.')
    elif isinstance(a, bool):
        print('a is a Boolean.')
    elif isinstance(a, list):
        print('a is a list.')
    elif isinstance(a, tuple):
        print('a is a tuple.')
    elif isinstance(a, dict):
        print('a is a dictionary.')
    elif isinstance(a, set):
        print('a is a set.')
    else:
        print('a has unknown type.')

var = 'Hello!'
printtype(var)
```
```
a is a text string.
```

Now let's call the function with a list:

```python
var = [1, 2, 3]
printtype(var)
```
```
a is a list.
```

The built-in function `type` will return the type of a variable directly:

```python
var = 'Hello!'
print(type(var))
```
```
<class 'str'>
```

90

Or for a real number:

```
var = 3.14
print(type(var))
<class 'float'>
```

One can also use the function `type` in conditions:

```
var = 3.14
if type(a) == float:
    print('a is a float.')
a is a float.
```

We will discuss functions in Section 8 and conditions in Section 10.

5.12 Operators +=, -=, *= **and** \=

The simplest way to increase the value of a numerical variable in Python by a given number is to use the operator +=:

```
v = 1
v += 3
print('v =', v)
v = 4
```

We can also subtract a number from a numerical variable:

```
v -= 1
print('New value of v is', v)
New value of v is 3
```

We can multiply a numerical variable with a number:

```
v *= 4
print('Now v is', v)
Now v is 12
```

And we can divide a numerical variable with a number:

91

```
v /= 6
print('Finally, v is', v)
Finally, v is 2
```

Analogous operators are available for `**` (power), `//` (floor division), `%` (modulo) etc.

5.13   Local and global variables

In the following subsections we are going to talk about functions. Functions will be discussed in Section 8. Feel free to jump there now, and return here later.

Variables defined inside a function are *local* to that function. This means that they 'live' inside of the function's body. If we try to access such a variable outside the function, even though the function was already called, the variable is unknown:

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

print(c)
on line 5:
NameError: name 'c' is not defined
```

5.14   Using global variables in functions can get you in trouble

When a variable is created in the main program (main scope), then it is *global*. This means that it 'lives' in the entire program, including in the bodies of all functions. Then in principle you can use it in any function without passing it as an argument. However:

```
Using variables in functions without passing them as arguments
is a bad programming practise.
```

That's what the following sample code does:

```
def printval():
    print('val =', val)
val = 5.0
printval()
5
```

Such a function is not self-contained: When one takes it and uses in some other module, it will throw an error because it can't operate without the variable `val`.

On the other hand, it is very easy to write the function correctly, passing the value `val` as an argument:

```python
def printval(out):
    print('val =', out)
val = 5
printval(val)
```
```
5
```

Now the function `printval()` is self-contained – it does not have any dependencies in the rest of the program.

5.15   Changing global variables in functions can get you in big trouble

As you already know, using a global variable in a function is a bad thing. But *changing* a global variable inside a function is *really bad*.

Changing a global variable inside a function is something that will get your job application tossed.

Here is a sample code which does that:

```python
def doubleval():
    val *= 2
val = 5
doubleval()
print('val =', val)
```
```
10
```

In the super-short code above you can see the definition of the variable `val` as well as the definition of the function `doubleval`. In a larger code, you would see neither of them. You would be looking, clueless, at an isolated line of code calling this function:

```python
doubleval()
```

This makes your code totally unreadable and worse – hidden inside the function is a landmine. Literally – like a landmine which is hidden beneath the surface and invisible, the change to a global variable is hidden inside the function `doubleval`. Having such a function in your software is likely to render your software buggy and dysfunctional very soon.

5.16   Shadowing of variables

Sometimes one may end up having two different variables of the same name in the code. This is more relevant to larger software projects, but we can illustrate it on a short program too:

```python
def add(a, b):
    c = a + b
    print('The value of c inside is', c)
    return c
c = 5
result = add(1, 2)
print('The value of c outside is', c)
```
```
The value of c inside is 3
The value of c outside is 5
```

We say that the global variable `c` is *shadowed* by the local variable `c` in the function `add()`.

```
If there is a name clash, the local variable has priority
over the global one.
```

When the above code is evaluated, the interpreter gives to the local variable `c` a different name. For the interpreter, the two variables are two completely different objects.

5.17   Callable variables

Python makes it possible to store functions in variables. These variables are then callable. We will discuss them in more detail in Subsection 8.24.

6 Boolean Values, Functions, Expressions, and Variables

6.1 Objectives

In this Section we will learn:

- About Boolean values and Boolean functions.
- About Boolean expressions and Boolean variables.
- How to use comparison operators for numbers and text strings.
- About the Boolean operations `and`, `or`, `not` and their *truth tables*.
- How conditions and the `while` loop work with Boolean values.
- How to calculate the value of π using the Monte Carlo method.
- How to use the `range` function with the `for` loop.

6.2 Why learn about Booleans?

Python has a built-in data type `bool` to represent Boolean values. These values are sometimes called *truth values* or *logical values*. In case you wonder where the name came from, George Boole (1815-1864) was an English mathematician who introduced the so-called *Boolean algebra* in his 1854 book *The Laws of Thought*. At that time, the Boolean algebra was merely an abstract theory without obvious practical applications. One hundred years later, it became the fundamental framework of computer science.

6.3 Boolean values

In Python, Boolean values and variables are most often used in conditions as well as in the `while` loop. A variable of type `bool` can have two values, `True` or `False`. Once such a value is assigned to a variable, the variable becomes a Boolean variable:

```python
flag = True
print(flag)
print(type(flag))
True
<class 'bool'>
```

And,

```python
flag2 = False
print(flag2)
print(type(flag2))
False
<class 'bool'>
```

95

6.4 Boolean functions

By *Boolean functions* we mean functions returning either `True` or `False`. Python has built-in Boolean functions – for example, in Subsection 5.11 you learned about the function `isinstance(var, T)` which returns `True` if the variable `var` has type `T` and `False` otherwise:

```python
var = 'Hello!'
test = isinstance(var, str)
print(test)
```
```
True
```

The Numpy library which you know well from Section 2 provides (among others) a Boolean function `equal` which compares two arrays elementwise, and returns an array of Boolean values:

```python
import numpy as np
a = [1, 2, 3]
b = [1, 2, 4]
result = np.equal(a, b)
print(result)
```
```
[True, True, False]
```

And last, you can easily define your own Boolean functions. For example, the following function `divisible(a, b)` returns `True` if a is divisible by b and `False` otherwise:

```python
def divisible(a, b):
    return a % b == 0

result = divisible(121, 11)
print(result)
```
```
True
```

6.5 Comparison operators for numbers

Expressions which evaluate to `True` or `False` are called *Boolean expressions*. There are many types of such expressions. Let's begin with comparisons between numbers:

```python
print(1 < 2.3)
```
```
True
```

96

Of course one can compare numerical variables as well:

```
a = 5
b = 10
print(a > b)
```
```
False
```

The modulo operator % can be used to check whether a number is even,

```
a = 22
print(a % 2 == 0)
```
```
True
```

etc.

Boolean expressions involving numbers or numerical variables mostly contain the following comparison operators:

Symbol	Meaning
>	greater than
>=	greater than or equal to
<=	less than or equal to
<	less than
==	equal to
!=	not equal to
<>	not equal to (same as !=)

6.6 Comparison operator == vs. assignment operator =

A common source of confusion for novice programmers is the difference between the == and = operators. The *comparison operator* == returns True or False and it is used to compare two values and determine if they are equal:

```
a = 1
print(a == 2)
```
```
False
```

The *assignment operator* = is used to assign a value to a variable, and it does not return anything:

97

```
print(a = 1)
```
```
on line 1:
TypeError: 'a' is an invalid keyword argument for this function
```

6.7 Comparison operators for text strings

Text strings can be compared using the same operators that are used for numbers. Since you know that Python is case-sensitive, the following result will not surprise you:

```
a = 'Hello!'
b = 'hello!'
print(a == b)
```
```
False
```

But the result of this comparison may be a bit surprising:

```
a = 'Monday'
b = 'Friday'
print(a > b)
```
```
True
```

Python compares strings using the ASCII codes of the characters. Since the ASCII code of the letter 'M' is

```
print(ord('M'))
```
```
77
```

and the ASCII code of 'F' is

```
print(ord('F'))
```
```
70
```

the outcome of 'Monday' > 'Friday' is the same as the outcome of 77 > 70 which is True.

6.8 Storing results of Boolean expressions in variables

The results of Boolean expressions are Boolean values True or False. They can be stored in variables which then automatically become Boolean variables. For example:

```
a = 1
result = a < 1
print(result)
```
```
False
```

One may need a bit of time to get used to this syntax, but it makes perfect sense. Above, `a < 1` is a Boolean expression which evaluates to `False`, and this Boolean value is then assigned to the variable `result`. Here is another example:

```
a = 13
result = 169 % a == 0
print(result)
```
```
True
```

Here, the Boolean expression `169 % a == 0` evaluates to `True` because 169 is divisible by 13. The value `True` is then assigned to the variable `result`.

6.9 Example: Checking if there exists a triangle with edge lengths a, b, c

When solving real-life problems, we often need to check two or more logical expressions simultaneously. For example, consider the simple task of checking whether three real numbers a, b, c can represent the lengths of the sides of a triangle:
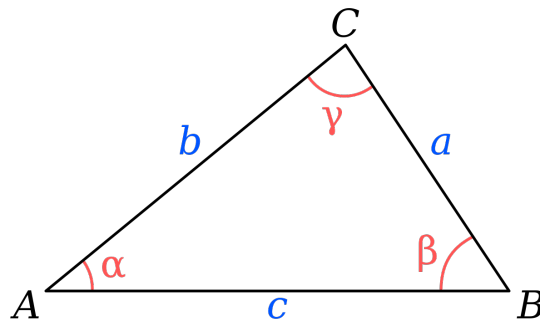
Fig. 52. Triangle with edges of length a, b, c.

Clearly, all three values must be greater than zero:

$$A = a > 0 \text{ and } b > 0 \text{ and } c > 0$$

99

Furthermore, the three values also need to satisfy the triangle inequality:

$$B = a + b > c \text{ or } b + c > a \text{ or } a + c > b$$

Hence the numbers a, b, c form an admissible input if the following Boolean expression `C` is `True`:

$$C = A \text{ and } B$$

Here is the corresponding Boolean function `checktriangle(a, b, c)` which returns `True` if there exists a triangle with edge lenghts a, b, c and `False` otherwise:

```python
def checktriangle(a, b, c):
    """
    Check if there exists a triangle with edge lengths a, b, c.
    """
    A = a > 0 and b > 0 and c > 0
    B = a + b > c or b + c > a or a + c > b
    return A and B

# Main program:
a = 3
b = 4
c = 5
result = checktriangle(a, b, c)
print(result)
```
```
True
```

Here, the text string enclosed in triple quotes is called *docstring*. We will talk about docstrings in more detail when we will discuss functions in Section 8.

As you can see, the logical operations `and` and `or` indeed are useful. And there is one more – `not`. Now let us revisit them one by one, starting with logical `and`.

6.10   Logical `and` and its truth table

If `A` and `B` are Boolean expressions, then `A and B` is `True` only if both `A` and `B` are `True`. Otherwise it is `False`. Here is an example:

```
a = 1
v1 = a > 0
v2 = a < 5
print(v1 and v2)
```
```
True
```

Here both `v1` and `v2` are `True`, so `v1 and v2` is `True`. Let's look at one additional example:

```
a = 6
v1 = a > 0
v2 = a < 5
print(v1 and v2)
```
```
False
```

Here `v1` is `True` and `v2` is `False`, therefore `v1 and v2` is `False`.

A table that summarizes the outcome of `A and B` for all possible combinations of the Boolean inputs `A` and `B` is called *truth table*. Here is how it looks for the `and` operator:

A	B	A and B
True	True	True
True	False	False
False	True	False
False	False	False

6.11 Logical `or` and its truth table

If `A` and `B` are Boolean expressions, then `A or B` is `True` if at least one of `A`, `B` is `True`. It is `False` if both `A`, `B` are `False`. An example:

```
a = 10
v1 = a < 0
v2 = a > 5
print(v1 or v2)
```
```
True
```

Here `v1` is `False` and `v2` is `True`, therefore `v1 or v2` is `True`. Another example:

```
a = 3
v1 = a < 0
v2 = a > 5
print(v1 and v2)
```
```
False
```

Here both v1 and v2 are False, therefore v1 or v2 is False.

The truth table of the or operator:

A	B	A or B
True	True	True
True	False	True
False	True	True
False	False	False

6.12   Negation not

Negation not is a logical operation that changes the value of a Boolean expression from True to False and vice versa:

```
a = 3
v1 = a < 0
print(not v1)
```
```
True
```

Here v1 is False, therefore not v1 is True.

The truth table of the not operator is very simple:

A	not A
True	False
False	True

6.13   Booleans in conditions and the while loop

Conditions will be discussed in more detail in Section 10 and the while loop in Section 11. But we would like to mention here that both conditions and the while loop are based on Booleans. For example, the if-else statement can accept a Boolean value

`True` which makes the condition automatically satisfied:

```python
if True:
    print('Condition is automatically satisfied.')
else:
    print('Condition is not satisfied.')
```
```
Condition is automatically satisfied.
```

It is hard to think about a meaningful use of that. But passing a Boolean variable (named, say, `DEBUG`) can be used to turn on or off debug prints throughout your program:

```python
if DEBUG:
    print('Debug mode: x =', x)
```
```
Debug mode: x = 4
```

Typing `while True` is a popular way to imitate the `do-while` or `repeat-until` loops which are not present in Python. The keyword `break` is then used to exit the loop. For example, the following short program will keep asking the user to enter his age until the user enters a positive integer:

```python
while True:
    a = input('Enter your age:')
    if a.isdigit() and int(a) > 0:
        print('Thank you.')
        break
    else:
        print('Please enter your age (positive integer).')
```

And of course, the most frequent case is using Boolean expressions. For example, the following condition will report whether a number is divisible by 10 or not:

```python
n = 100
if n % 10 == 0:
    print('Number', n, 'is divisible by 10.')
else:
    print('Number', n, 'is not divisible by 10.')
```
```
Number 100 is divisible by 10.
```

103

6.14 Example: Monte Carlo calculation of π

The Monte Carlo method is a popular method of scientific computing. Its idea is to employ a large number of random values to calculate approximately results that are difficult or impossible to obtain exactly. Sounds complicated? On the contrary! Monte Carlo methods are simple, and they leave all the heavy lifting to the computer to do. Their drawback is that usually they are very computationally expensive. To illustrate this, let's use a Monte Carlo method to calculate the value of π!

Fig. 53 shows a unit square $(0, 1) \times (0, 1)$:
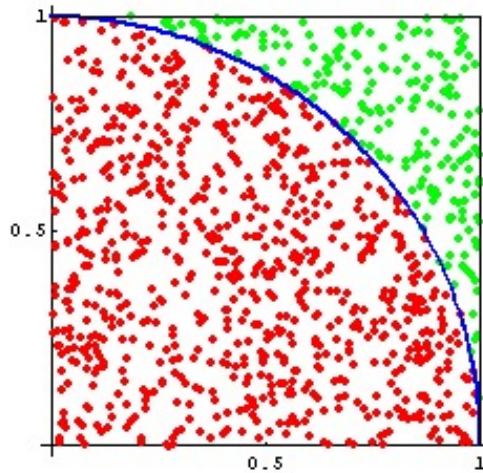
Fig. 53. Monte Carlo calculation of π.

The red part is a quarter-circle whose area is $C = \pi r^2/4 = \pi/4$. The area of the entire square is $A = 1$. We will shoot a large number N of random points into the square. We will count the number I of points which lie inside the quarter-circle. Then the ratio $I : N$ will be approximately the same as $C : A$. Hence

$$\frac{I}{N} \approx \frac{C}{A} = \frac{\pi}{4}$$

and therefore

$$\pi \approx \frac{4I}{N}$$

In summary, all we need to do is generate N random points (x, y) in the unit square, count the number I of points which lie in the quarter circle,

$$x^2 + y^2 \leq 1,$$

and then $4I/N$ will give us an approximate value of π.

The program and sample results

We will use the Random library that you know from Subsection 2.16:

```python
# Import the Random library:
import random as rn
# Number of points:
N = 100000
# Number of points inside the circle:
I = 0
# Generate N random points:
for i in range(N):
    # Generate a random number between 0 and 1:
    x = rn.random()
    # Generate a random number between 0 and 1:
    y = rn.random()
    # If (x, y) lies inside the circle, increase counter I:
    if x**2 + y**2 <= 1:
        I += 1
# Calculate and display the final result:
result = 4*I/N
print('Approximate value of pi =', result)
```

Sample results for various values of N are shown in the following table:

N	result
100	3.48
1000	3.272
10000	3.1576
100000	3.13764
1000000	3.14113
10000000	3.14253

Notice that a very large number of random points is needed to obtain the value of π accurate at least to two decimal digits – this is characteristic for Monte Carlo methods.

7 Lists, Tuples, Dictionaries, and Sets

7.1 Objectives

In this Section we will learn:

- How to work with lists, tuples, dictionaries, and sets.
- Important properties of these data structures.
- How to use methods of these classes to operate with them efficiently.
- The difference between mutable and immutable objects in Python.

7.2 Why learn about lists?

Lists are an extremely popular data structure in Python, and the language provides a wide range of functionality to work with them. In Section 5 we said that a variable is like a container which can hold a text string, a numerical value, a Boolean, and other things. Using the same language, a list is like a cargo train where these containers can be loaded one after another. The train can be as long as one needs, and its length can even change at runtime. One can use a list to store a huge amount of data, and process it using many built-in tools that we are going to explore in this section.

7.3 Creating a list

An empty list named, for example, `L` is created as follows:

```
L = []
```

A nonempty list is created by listing comma-separated items and enclosing them in square brackets. Let's create a list `L2` of integers:

```
L2 = [2, 3, 5, 7, 11]
```

Here is a list `cls` with the names of ten students:

```
cls = ['John', 'Pam', 'Emily', 'Jessie', 'Brian', 'Sam', 'Jim',
'Tom', 'Jerry', 'Alex']
```

Lists can contain virtually anything, including other lists. Here is a list named `pts` which contains the coordinates of four points in the XY plane:

```
pts = [[0, 0], [1, 0], [1, 1], [0, 1]]
```

And last, items in a list can have different types:

```
mix = ['Sidney', 5, 2.17, 1 + 2j, [1, 2, 3]]
```

As usual, Python gives you a lot of freedom – do not abuse it. Instead of throwing all data into one list, it pays off to have the data better organized.

7.4 Important properties of lists

In contrast to dictionaries and sets, lists can contain *repeated items*. This is a valid list:

```
X = [1, 2, 3, 1, 2, 3]
```

Further, lists are *ordered*. In other words, two lists containing the same items but ordered differently are not the same:

```
X = [1, 2, 3, 1, 2, 3]
Y = [1, 1, 2, 2, 3, 3]
print(X == Y)
```
```
False
```

7.5 Measuring the length of a list

You already know from Subsection 4.5 how to measure the length of text strings using the built-in function `len`. The same function can be used to measure the length of lists:

```
X = [1, 2, 3, 1, 2, 3]
print(len(X))
```
```
6
```

Lists have many other similarities to text strings – they can be added together, multiplied with integers, their items can be accessed via indices, they can be sliced, etc. You will see all that later in this section.

7.6 Appending new items to lists

The list method `append` will append a new item to the end of a list:

```
n = 7
X = [2, 3, 5]
X.append(n)
print(X)
```
```
[2, 3, 5, 7]
```

The original list can be empty:

```
name = 'Alyssa'
names = []
names.append(name)
print(names)
```
```
['Alyssa']
```

As mentioned earlier, a list can contain items of different types, including other lists:

```
point = [1, 2]
X = [2, 3, 5]
X.append(point)
print(X)
```
```
[2, 3, 5, [1, 2]]
```

And last, items can be appended to the end of a list without using the method `append`:

```
name = 'Hunter'
names = ['Alyssa', 'Zoe', 'Phillip']
names += [name]
print(names)
```
```
['Alyssa', 'Zoe', 'Phillip', 'Hunter']
```

Here, `[name]` is a one-item list containing the text string `name`. The line `names += [name]` adds the list `[name]` to the list `names`. We are going to talk more about adding lists in the next subsection.

7.7   Adding lists

You already know from Subsection 4.20 that text strings can be added using the operator + just like numbers. Lists can be added in the same way:

```
X = [2, 3, 5]
Y = [7, 11]
Z = X + Y
print(Z)
```
```
[2, 3, 5, 7, 11]
```

A list can be extended by adding another list to it using the operator +=:

```
X = [2, 3, 5]
Y = [7, 11]
X += Y
print(X)
```
```
[2, 3, 5, 7, 11]
```

7.8   Adding is not appending

Novice programmers sometimes by mistake append a list Y to an existing list X instead of adding Y to X. But this leads to a completely wrong result:

```
X = [2, 3, 5]
Y = [7, 11]
X.append(Y)
print(X)
```
```
[2, 3, 5, [7, 11]]
```

The correct way to add the list Y to the list X is:

```
X = [2, 3, 5]
Y = [7, 11]
X += Y
print(X)
```
```
[2, 3, 5, 7, 11]
```

7.9   Multiplying lists with integers

Do you still remember from Subsection 4.21 that a text string can be made N times longer by multiplying it with a positive integer N? Multiplication of lists with positive integers works analogously:

```
X = ['a', 'b', 'c', 'd']
Y = 3*X
print(Y)
```

```
['a', 'b', 'c', 'd', 'a', 'b', 'c', 'd', 'a', 'b', 'c', 'd']
```

The operator *= can be used to extend the list in place:

```
nums = [1, 0, 1]
nums *= 4
print(nums)
```

```
[1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1]
```

7.10   Parsing lists with the `for` loop

We met the `for` loop for the first time in Subsection 4.22 where it was used to parse text strings one character at a time. Analogously, it can be used to parse lists one item at a time:

```
names = ['Alyssa', 'Zoe', 'Phillip']
for name in names:
    print(name)
```

```
Alyssa
Zoe
Phillip
```

The `for` loop will be discussed in more detail in Section 9.

7.11   Accessing list items via their indices

In Subsection 4.24 you saw how individual characters in a text string can be accessed using their indices. Individual items in a list are accessed exactly in the same way:

```
L = ['f', 'a', 's', 't']
print('First item:', L[0])
print('Second item:', L[1])
print('Third item:', L[2])
print('Fourth item:', L[3])
```

110

```
First item: f
Second item: a
Third item: s
Fourth item: t
```

7.12 Slicing lists

We are not going to discuss list slicing in detail because lists are sliced in exactly the same way as text strings. This was explained in Subsection 4.29. But showing one small example cannot hurt:

```
L = ['c', 'o', 'f', 'f', 'e', 'e', 'm', 'a', 'k', 'e', 'r']
print('L[3:6] =', L[3:6])
print('L[:6] =', L[:6])
print('L[6:] =', L[6:])
print('L[:] =', L[:])
print('L[::2] =', L[::2])
print('L[5::-1] =', L[5::-1])
```
```
L[3:6] = ['f', 'e', 'e']
L[:6] = ['c', 'o', 'f', 'f', 'e', 'e']
L[6:] = ['m', 'a', 'k', 'e', 'r']
L[:] = ['c', 'o', 'f', 'f', 'e', 'e', 'm', 'a', 'k', 'e', 'r']
L[::2] = ['c', 'f', 'e', 'm', 'k', 'r']
L[5::-1] = ['e', 'e', 'f', 'f', 'o', 'c']
```

7.13 Creating a copy of a list – the wrong way and the right way

To create a new copy `Lnew` of a list `L`, it is not enough to just assign it to a new variable

```
Lnew = L
```

the way it works for text strings. Namely, this does not copy the list - it only makes `Lnew` point to the same place in memory as `L`. We will talk about this in more detail when discussing mutability of lists in Subsection 7.34. For now let's show what surprise one may get when "copying" a list in this way.

The code below "creates a copy" `Lnew` of a list `L` by typing `Lnew = L`. Then it appends a new item to the end of `Lnew`. But as you will see, the new item also appears in the original list `L`!

```
L = [1, 2, 3]
Lnew = L
Lnew.append(4)
print('Lnew =', Lnew)
print('L =', L)
```
```
Lnew = [1, 2, 3, 4]
L = [1, 2, 3, 4]
```

The correct way to create a copy `Lnew` of a list `L` is by slicing:

```
Lnew = L[:]
```

Let us make this tiny change in the code above, and you will see that the undesired item 4 will not appear in the original list `L` anymore:

```
L = [1, 2, 3]
Lnew = L[:]
Lnew.append(4)
print('Lnew =', Lnew)
print('L =', L)
```
```
Lnew = [1, 2, 3, 4]
L = [1, 2, 3]
```

7.14   Popping list items by index

Class `list` has a method `pop` which removes the last item from the list and returns it for further use:

```
team = ['John', 'Pam', 'Brian', 'Sam', 'Jim']
name = team.pop()
print(name)
print(team)
```
```
Jim
['John', 'Pam', 'Brian', 'Sam']
```

When called with an index, the method will pop the corresponding item instead of the last one. Since indices start from 0, using the index 0 will pop the first item:

112

```
team = ['John', 'Pam', 'Brian', 'Sam', 'Jim']
name = team.pop(0)
print(name)
print(team)
```
```
John
['Pam', 'Brian', 'Sam', 'Jim']
```

For illustration, let's also pop the second item using index 1:

```
team = ['John', 'Pam', 'Brian', 'Sam', 'Jim']
name = team.pop(1)
print(name)
print(team)
```
```
Pam
['John', 'Brian', 'Sam', 'Jim']
```

7.15   Deleting list items by index

Typing `del L[i]` will delete item with index `i` from list `L` and destroy it. For illustration, let's delete the first item from the list `team`:

```
team = ['John', 'Pam', 'Brian', 'Sam', 'Jim']
del team[0]
print(team)
```
```
['Pam', 'Brian', 'Sam', 'Jim']
```

7.16   Checking if an item is in a list

In Subsection 4.35 you learned that typing `substr in txt` will return `True` if the substring `substr` is present in the text string `txt`, and `False` otherwise. The keyword `in` can be used in the same way to check whether an item is present in a list:

```
team = ['John', 'Pam', 'Brian', 'Sam', 'Jim']
name = 'Sam'
print(name in team)
```
```
True
```

Now let's check for `'Xander'` and store the result in a Boolean variable `found`:

```
team = ['John', 'Pam', 'Brian', 'Sam', 'Jim']
name = 'Xander'
found = name in team
print(found)
```
```
False
```

7.17   Locating an item in a list

Now, let's say that we need to delete Brian from the team. You can see the list of names in the code below, but in reality you would not see it – you would not know where the name 'Brian' is located. This means that we first need to locate the name 'Brian' using the method index, and then delete the name using the keyword del:

```
team = ['John', 'Pam', 'Brian', 'Sam', 'Jim']
name = 'Brian'
i = team.index(name)
del team[i]
print(team)
```
```
['John', 'Pam', 'Sam', 'Jim']
```

If the list item is not found, the method index will throw an error:

```
team = ['John', 'Pam', 'Brian', 'Sam', 'Jim']
name = 'Eli'
i = team.index(name)
del team[i]
print(team)
```
```
on line 3:
ValueError: 'Eli' is not in list
```

7.18   Finding and deleting an item from a list

A safe way to delete an item from a list is to first check whether the item is there:

```
team = ['John', 'Pam', 'Brian', 'Sam', 'Jim']
name = 'Eli'
if name in team:
    i = team.index(name)
    del team[i]
    print(team)
else:
    print('Item not found.')
```
```
Item not found.
```

7.19 Finding and deleting all occurrences of an item

Only a minor change to the previous program is needed to find and delete all occurrences of a given item from a list:

```
team = ['Sam', 'John', 'Pam', 'Brian', 'Sam', 'Jim']
name = 'Sam'
while name in team:
    i = team.index(name)
    del team[i]
print(team)
```
```
['John', 'Pam', 'Brian', 'Jim']
```

7.20 Counting occurrences of items

Class list has a method count to count the occurrences of an item in the list:

```
team = ['Sam', 'John', 'Pam', 'Brian', 'Sam', 'Jim']
name = 'Sam'
n = team.count(name)
print(n)
```
```
2
```

7.21 Inserting list items at arbitrary positions

New item x can be inserted to a list L at position n by typing L.insert(n, x). In other words, after the operation, item x will have index n in the list L. For illustration, let us insert the name 'Daniel' at the fourth position (index 3) to a list of names team:

```
team = ['John', 'Pam', 'Brian', 'Sam', 'Jim']
name = 'Daniel'
team.insert(3, name)
print(team)
```

```
['John', 'Pam', 'Brian', 'Daniel', 'Sam', 'Jim']
```

7.22 Sorting a list in place

Class `list` provides a method `sort` to sort the list in place. By *in place* we mean that the original list is changed. If you need to create a sorted copy of a list, see Subsection 7.23.

In case you are aware of the existence of various sorting algorithms (BubbleSort, QuickSort, MergeSort, InsertionSort, ...), Python uses an algorithm called TimSort. This is a hybrid sorting algorithm derived from MergeSort and InsertionSort, designed to perform well on many kinds of real-world data. It was invented by Tim Peters in 2002 for use in the Python programming language. Here is an example:

```
L = ['Tom', 'Sam', 'John', 'Pam', 'Jim', 'Jerry', 'Jack',
'Daniel', 'Alex', 'Brian']
L.sort()
print(L)
```

```
['Alex', 'Brian', 'Daniel', 'Jack', 'Jerry', 'Jim', 'John',
'Pam', 'Sam', 'Tom']
```

7.23 Creating sorted copy of a list

The obvious solution to this task is to create a copy of the list and then sort it using the `sort` method:

```
X2 = X[:]
X2.sort()
```

However, Python makes this easier by providing a built-in function `sorted`. This function creates a copy of the original list, sorts it, and returns it, leaving the original list unchanged:

```
X2 = sorted(X)
```

116

7.24 Using key functions in sorting

Both the method `sort` and the function `sorted` accept an optional argument `key`. This is a function to be applied to all list items prior to sorting. For example, if one wants to disregard the case of characters while sorting a list of text strings, one can define `key=str.lower`. To illustrate this, first let's sort a sample list without the key:

```
L = ['f', 'E', 'd', 'C', 'b', 'A']
L.sort()
print(L)
```
```
['A', 'C', 'E', 'b', 'd', 'f']
```

And now let's use `key=str.lower`:

```
L = ['f', 'E', 'd', 'C', 'b', 'A']
L.sort(key=str.lower)
print(L)
```
```
['A', 'b', 'C', 'd', 'E', 'f']
```

Alternatively to `key=str.lower`, one could use any other string method (many were discussed in Section 4).

7.25 Using lambda functions in sorting

The so-called *lambda functions* are a quick and elegant way to define one's own sorting keys. Lambda functions will be discussed in detail in Subsection 8.18 but let's show a simple example now. Let's say that we want for some reason to sort a list of words according to the last character in each word. The corresponding lambda function which takes a text string and returns its last character has the form

```
lambda x: x[-1]
```

And now the code:

```
L = ['Peter', 'Paul', 'Mary']
L.sort(key=lambda x: x[-1])
print(L)
```
```
['Paul', 'Peter', 'Mary']
```

7.26 Reversing a list in place

Class `list` provides a method `reverse` to reverse the list in place:

```
L = ['Tom', 'Sam', 'John', 'Pam']
L.reverse()
print(L)
['Pam', 'John', 'Sam', 'Tom']
```

7.27 Creating reversed copy of a list

Analogously to creating a sorted copy of a list (Subsection 7.23), there are two ways to do this. First, one can just create a copy of the list and then reverse it using the `reverse` method:

```
X2 = X[:]
X2.reverse()
```

Or, one can call the built-in function `reversed`:

```
X2 = list(reversed(X))
```

Note: Since this function is designed for more general applications, it is necessary to cast the result to a list.

7.28 Zipping lists

Another useful operation with lists is their *zipping*. By this we mean taking two lists of the same length and creating a new list of pairs. Python does this using a built-in function `zip`. Example:

```
A = [1, 2, 3]
B = ['x', 'y', 'z']
print(zip(A, B))
[(1, 'x'), (2, 'y'), (3, 'z')]
```

If the lists are not equally long, the items at the end of the longer list are skipped:

```
A = [1, 2, 3, 4, 5]
B = ['x', 'y', 'z']
print(zip(A, B))
```

```
[(1, 'x'), (2, 'y'), (3, 'z')]
```

7.29   List comprehension

List comprehension is a quick and elegant way to perform some operation with all items in an iterable (text string, list, tuple, dictionary, ...), creating a new list in the process. This technique is genuinely Pythonic and extremely popular in the Python programming community. Let's begin with a simple example where we want to square all numbers in a list:

```
L = [1, 2, 3, 4, 5]
S = [x**2 for x in L]
print(S)
[1, 4, 9, 16, 25]
```

The next example converts a list of characters to a list of the corresponding ASCII codes:

```
L = ['b', 'r', 'e', 'a', 'k', 'f', 'a', 's', 't']
S = [ord(c) for c in L]
print(S)
[98, 114, 101, 97, 107, 102, 97, 115, 116]
```

Oh, wait – a text string is iterable, so one can use it in the list comprehension directly!

```
txt = 'breakfast'
S = [ord(c) for c in txt]
print(S)
[98, 114, 101, 97, 107, 102, 97, 115, 116]
```

7.30   List comprehension with `if` statement

It is possible make an `if` statement part of the list comprehension. Then the operation is only performed on some items in the list while other items are ignored (and left out).

A common application of this type of list comprehension is *filtering* – this means that one picks selected items from the list without changing them (although changing them is possible as well). As an example, let's select all even numbers from a list of integers L:

```
L = [2, 5, 8, 11, 12, 14, 16, 19, 20]
E = [n for n in L if n%2 == 0]
print(E)
[2, 8, 12, 14, 16, 20]
```

One can use list comprehension to extract from a list of payments all payments which were made in EUR:

```
P = ['125 USD', '335 EUR', '95 EUR', '195 USD', '225 EUR']
E = [x for x in P if x[-3:] == 'EUR']
print(E)
['335 EUR', '95 EUR', '225 EUR']
```

Or, one can use list comprehension to extract all integers from a text string:

```
txt = 'American Civil War lasted from April 1861 to May 1865'
L = txt.split()
nums = [int(word) for word in L if word.isdigit()]
print(nums)
[1861, 1865]
```

And as a last example, let's write a program which will select from list A all items which are not in list B:

```
A = ['b', 'o', 'd', 'y', 'g', 'u', 'a', 'r', 'd']
B = ['b', 'o', 'd', 'y']
D = [c for c in A if not c in B]
print(D)
['g', 'u', 'a', 'r']
```

7.31 List comprehension with `if` and `else`

In its fullest form, list comprehension makes it possible to incorporate conditional (ternary) expressions of the form

```
value_1 if boolean_expression else value_2
```

Importantly, a conditional expression is not an `if-else` statement. We will talk about this in more detail in Subsection 10.10. In list comprehension, they can be used as fol-

lows:

```
[conditional_expression for item in list]
```

Let's demonstrate this on an example where all characters in a list A which are not vowels will be replaced with '-':

```
A = ['b', 'o', 'd', 'y', 'g', 'u', 'a', 'r', 'd']
vowels = ['a', 'e', 'i', 'o', 'u']
D = [c if c in vowels else '-' for c in A]
print(D)
['-', 'o', '-', '-', '-', 'u', 'a', '-', '-']
```

For better readability, it is possible to enclose the conditional expression in parentheses. This is shown in the next example where vowels are replaced with 1s and all other characters with 0s:

```
A = ['b', 'o', 'd', 'y', 'g', 'u', 'a', 'r', 'd']
vowels = ['a', 'e', 'i', 'o', 'u']
D = [(1 if c in vowels else 0) for c in A]
print(D)
[0, 1, 0, 0, 0, 1, 1, 0, 0]
```

7.32   List comprehension with nested loops

List comprehension can use multiple `for` statements. Imagine that you have a list of numbers N, list of characters C, and you want to create a list of labels L by combining the numbers and characters together:

```
N = [1, 2, 3]
C = ['A', 'B']
L = [str(n) + c for n in N for c in C]
print(L)
['1A', '1B', '2A', '2B', '3A', '3B']
```

And if you also want to add some symbols, such as + and −, you can use a triple-nested loop:

```
N = [1, 2, 3]
C = ['A', 'B']
S = ['+', '-']
L = [str(n) + c + s for n in N for c in C for s in S]
print(L)
```
```
['1A+', '1A-', '1B+', '1B-', '2A+', '2A-', '2B+', '2B-', '3A+',
'3A-', '3B+', '3B-']
```

The result of a comprehension does not have to be a list. It can be another iterable, notably a tuple or a dictionary. We will talk about this in Subsections 7.45, 7.46 and 7.53.

7.33 Mutable and immutable objects

As you already know, almost everything in Python is an object. There are two types of objects – *mutable* and *immutable*. Immutable objects cannot be changed by functions. In other words, one can pass an immutable object into a function, the function can attempt to alter it, but after the function finishes, the object will remain unchanged.

Text strings (str) are immutable objects. To illustrate what we said above, we will pass a sample text string txt to a function redef which will try to change it:

```
def redef(x):
    x = 'I am redefining you!'

txt = 'Hello!'
redef(txt) # here the function attempts to change the string
print(txt) # but this line displays 'Hello!'
```
```
Hello!
```

Clearly the text string txt was not changed. To understand what is going on, let's print the memory locations of the variables txt and x prior and after the changes. You will see that when txt is passed to the function, the variable x points to the same memory address as txt. But as soon as x is overwritten, its location in the memory changes. Hence, the memory segment where txt is stored remains unaltered – meaning that txt does not change:

```
def redef(x):
    print('2. x is at', hex(id(x)))
    x = 'I am redefining you!'
    print('3. x is at', hex(id(x)))

txt = 'Hello!'
print('1. txt is at', hex(id(txt)))
redef(txt) # here the function attempts to change the string
print('4. txt is at', hex(id(txt)))
print(txt) # but this line displays 'Hello!'
```

```
1. txt is at 0x39063e8
2. x is at 0x39063e8
3. x is at 0x3906340
4. txt is at 0x39063e8
Hello!
```

Sometimes, novice programmers incorrectly interpret immutability as "the object cannot be changed". But that's not correct. Sure enough a text string can be changed:

```
txt = 'Hello!'
print(txt)
txt = 'Hi there!' # txt is being changed (overwritten) here
print(txt)
```

```
Hello!
Hi there!
```

Let's display the memory location of the variable txt before and after. You will see again that its memory location changes:

```
txt = 'Hello!'
print('1. txt is at', hex(id(txt)))
txt = 'Hi there!' # txt is being changed (overwritten) here
print('2. txt is at', hex(id(txt)))
```

```
1. txt is at 0x2eb1e68
2. txt is at 0x2eb1f10
```

This is the true meaning of immutability. Soon you will see that mutable objects can be altered while remaining at the same memory location. But in the meantime, let's

explore additional immutable objects:

Integers (`int`), real numbers (`float`), and complex numbers (`complex`) are immutable. Let's just show this using a real number. We will again pass it into a function which will try to change it:

```python
def redef(x):
    x = 0

z = 17.5
redef(z) # here the function attempts to reset z to 0
print(z) # but this line prints 17.5 again
```
```
17.5
```

Again, let's print the memory location of the numerical variable prior and after the change, and you will see that it will be different:

```python
z = 17.5
print('1. z is at', hex(id(z)))
z = 0
print('2. z is at', hex(id(z)))
```
```
1. z is at 0x8e0820
2. z is at 0x8dfba0
```

At this point you understand immutability. For completeness let's remark that Booleans (`bool`) and tuples (`tuple`) are immutable types as well. Mutable types include lists (`list`), dictionaries (`dict`), sets (`set`) and most custom classes.

In the next subsection we will look at mutability of lists in more detail.

7.34   Mutability of lists

With the understanding of immutability that you have from the previous subsection we can now offer two equivalent definitions of mutability:

1. An object is mutable if it can be changed inside functions.
2. An object is mutable if it can be altered while remaining at the same memory location.

Let us illustrate the first point by passing a list into a function `redef` which will append a new item to it.

```python
def redef(x):
    x.append(4)

L = [1, 2, 3]
redef(L) # here the function appends a new item 4 to L
print(L)
```

```
[1, 2, 3, 4]
```

To illustrate the second point, let's show that appending a new item to a list will not change the list's location in the memory:

```python
L = [1, 2, 3]
print('1. L is at', hex(id(L)))
L.append(4)
print('2. L is at', hex(id(L)))
```

```
1. L is at 0x1cc1128
2. L is at 0x1cc1128
```

Being able to check where objects are stored in computer memory can help you solve all sorts of mysteries. For example, the following experiment clearly reveals the source of the problem that was described in Subsection 7.13 (creating a copy of a list incorrectly):

```python
L = [1, 2, 3]
print('1. L is at', hex(id(L)))
K = L
print('2. K is at', hex(id(K)))
K.append(4)
print('3. L is at', hex(id(L)))
print('4. K is at', hex(id(K)))
```

```
1. L is at 0x27c4b90
2. K is at 0x27c4b90
3. L is at 0x27c4b90
4. K is at 0x27c4b90
```

125

As you can see, L and K really are just two names for the same list which is located at the same memory position `0x27c4b90`.

7.35   Tuples

Tuples are very similar to lists, with the following differences:

1. Tuples use parentheses `(...)` where lists use square brackets `[...]`.
2. Tuples cannot be changed in place – they can only be overwritten like text strings.
3. Tuples are immutable – they cannot be changed by functions.

These propertes make tuples perfect to represent sequences of data that does not change – such as the names of weekdays, or names of months:

```
months = ('January', 'February', 'March', 'April', 'May', \
'June','July', 'August', 'September', 'October', 'November', \
'December')
```

All items in this tuple are text strings, but a tuple can be heterogeneous – it can contain any combination of text strings, integers, real numbers, other tuples, functions, instances of classes, etc.

Working with tuples is similar to working with lists. Items in a tuple can be accessed via their indices:

```
print('First month:', months[0])
print('Second month:', months[1])
print('Third month:', months[2])
```
```
First month: January
Second month: February
Third month: March
```

One can use negative indices:

```
print('One before last month:', months[-2])
print('Last month:', months[-1])
```
```
One before last month: November
Last month: December
```

Tuples can be sliced like lists:

126

```
months[2:5]
```
```
('March', 'April', 'May')
```

The length of a tuple is obtained using the function `len()`:

```
len(months)
```
```
12
```

Let's now discuss the aspects of tuples which are different from lists.

7.36 Read-only property of tuples

If you search the web, you will find numerous discussions about when one should use tuples and when one should use lists.

As a matter of fact, the speed is not a significant factor – in other words, one cannot say that using tuples instead of lists wherever possible will speed up your code.

But one aspect of tuples is beneficial for sure – their read-only property. One cannot append, insert, pop or delete items from tuples:

```
days = ('Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun')
days.append('Another Sunday')
```
```
on line 2:
AttributeError: 'tuple' object has no attribute 'append'
```

So, if you have a sequence of data which should not be altered by anyone in the team, use a tuple.

7.37 Immutability of tuples

Tuples are immutable. In this, they are similar to text strings. A tuple cannot be changed by a function. It only can be overwritten, in which case it changes the location in the memory:

```
days = ('Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun')
print('1. days is at', hex(id(days)))
days = ('Lun', 'Mar', 'Mie', 'Jue', 'Vie', 'Sad', 'Dom')
print('2. days is at', hex(id(days)))
```
```
1. days is at 0x2a11e50
2. days is at 0x2f2fd00
```

For a more detailed discussion of mutable and immutable objects see Subsection 7.33.

7.38   Dictionaries

Sometimes one needs to store objects along with some related information. For example, these "objects" may be people, and we may need to store their phone number, age, address, family status, etc. Or, we may need to store a list of countries along with their GDP. Or a list of used cars with their price.

Well, you know that lists can contain other lists, so you could handle this:

```
cars = [['Nissan Altima', 10000], ['Toyota Camry', 15000],
['Audi S3', 25000]]
```

But, this is a poor man's solution. Searching for items and other operations would be cumbersome. Here is what it would take to find the value (second item) for a given key (first item):

```
def findvalue(D, key):
    for k, v in D:
        if k == key:
            return v

cars = [['Nissan Altima', 10000], ['Toyota Camry', 15000],
['Audi S3', 25000]]
findvalue(cars, 'Nissan Altima')
```
```
10000
```

Therefore, Python provides a dedicated data type called *dictionary*. Under the hood a dictionary is similar to a list of lists, but it has a lot of useful functionality which we will explore in the following subsections. To begin with, one does not need a function `findvalue`. Instead, one just types

```
cars = {'Nissan Altima':10000, 'Toyota Camry':15000,
'Audi S3':25000}
cars['Nissan Altima']
```
```
10000
```

7.39 Creating an empty and nonempty dictionary

An empty dictionary `D` is defined as follows:

```
D = {}
```

As you already saw, the dictionary for the above example is

```
cars = {'Nissan Altima':10000, 'Toyota Camry':15000,
'Audi S3':25000}
```

Note that dictionaries use curly braces `{...}` where lists use square brackets `[...]` and tuples parentheses `(...)`.

The length of the dictionary is the number of `key:value` pairs. It can be measured using the function `len`:

```
cars = {'Nissan Altima':10000, 'Toyota Camry':15000,
'Audi S3':25000}
print(len(cars))
3
```

7.40 Keys, values, and items

In the last example, the cars were the *keys* and their prices were the corresponding *values*. The list of all keys can be obtaining by typing

```
K = cars.keys()
print(K)
['Nissan Altima', 'Toyota Camry', 'Audi S3']
```

Similarly, one can also extract the list of all values:

```
V = cars.values()
print(V)
[10000, 15000, 25000]
```

The `key:value` pairs are called *items*. Each pair is a two-item tuple. A list of these tuples can be obtained by typing:

```
L = list(cars.items())
print(L)
```
```
[('Nissan Altima', 10000), ('Toyota Camry', 15000),
('Audi S3', 25000)]
```

7.41 Accessing values using keys

In lists and tuples it is clear which item is first, which one is second, etc. That's because they are *ordered*. Then one can access their items using indices, slice them, reverse them, etc.

None of this is possible with dictionaries.

Dictionaries are not ordered, so there is no first item and there is no last item. They do not have indices and they cannot be sliced. Instead, values are accessed using the keys directly. As you already saw before:

```
cars = {'Nissan Altima':10000, 'Toyota Camry':15000,
'Audi S3':25000}
print(cars['Audi S3'])
```
```
25000
```

In other words, the keys serve as indices.

7.42 Adding, updating, and deleting items

Here is a small phone book:

```
phonebook = {'Pam Pan':8806336, 'Emily Elk':6784346,
'Lewis Lux':1122345}
```

A `new_key:new_value` pair can be added to an existing dictionary `dict_name` by typing `dict_name[new_key] = new_value`. So, Silly Sam with the phone number 1234567 would be added as follows:

```
phonebook['Silly Sam'] = 1234567
print(phonebook)
```
```
{'Pam Pan': 8806336, 'Silly Sam':1234567, 'Emily Elk':6784346,
'Lewis Lux':1122345}
```

Dictionaries cannot contain repeated keys. Therefore, when adding a `key:value` pair whose `key` already exists in the dictionary, no new item will be created. Instead, only the `value` in the existing `key:value` pair will be updated:

```python
phonebook['Pam Pan'] = 5555555
print(phonebook)
```
```
{'Pam Pan': 5555555, 'Silly Sam':1234567, 'Emily Elk':6784346,
'Lewis Lux':1122345}
```

And last, items can be deleted using the keyword `del`:

```python
del phonebook['Lewis Lux']
print(phonebook)
```
```
{'Pam Pan': 5555555, 'Silly Sam':1234567, 'Emily Elk':6784346}
```

7.43   Checking for keys, values, and items

To check for a given `key`, one can type `key in dict_name`:

```python
print('Silly Sam' in phonebook)
```
```
True
```

It is also possible to use the slightly longer version `key in dict_name.keys()`. To check for a given `value`, one can type `value in dict_name.values()`:

```python
print(5555555 in phonebook.values())
```
```
True
```

And finally, one can check for the presence of a given `key:value` pair by typing `(key, value) in dict_name.items()`:

```python
print(('Emily Elk', 6784346) in phonebook.items())
```
```
True
```

7.44   Finding all keys for a given value

Note that while there is only one `value` for every `key`, multiple different `keys` may have the same `value`. Such as in this dictionary where the Toyota and the Subaru have

131

the same price:

```
cars = {'Nissan Altima':10000, 'Toyota Camry':15000,
'Audi S3':25000, 'Subaru Forester':15000}
```

Hence one has to parse all `items` in the dictionary to find all the `keys`:

```python
def findkeys(D, value):
    result = []
    for k, v in D.items():
        if v == value:
            result.append(k)
    return result

cars = {'Nissan Altima':10000, 'Toyota Camry':15000,
'Audi S3':25000, 'Subaru Forester':15000}
findkeys(cars, 15000)
```

```
['Toyota Camry', 'Subaru Forester']
```

But this task has a way more elegant solution which we can show you because you already know list comprehension. Recall that comprehension was discussed in Subsections 7.29 – 7.32.

7.45 Finding all keys for a given value using comprehension

Comprehension for a dictionary D works analogously to the comprehension for a list L, except that instead of `for x in L` one types `for k, v in D.items()`. A shorter solution to the task from the previous subsection is:

```python
def findkeys(D, value):
    return [k for k, v in D.items() if v == value]

cars = {'Nissan Altima':10000, 'Toyota Camry':15000,
'Audi S3':25000, 'Subaru Forester':15000}
findkeys(cars, 15000)
```

```
['Toyota Camry', 'Subaru Forester']
```

7.46 Reversing a dictionary using comprehension

Comprehension is a really powerful ally of the Python programmer. Reversing a dictionary is a one-liner:

```python
def reverse(D):
    return {v:k for k, v in D.items()}

cars = {'Nissan Altima':10000, 'Toyota Camry':15000,
'Audi S3':25000}
reverse(cars)
```
```
{10000:'Nissan Altima', 15000:'Toyota Camry',
25000:'Audi S3'}
```

7.47 Beware of repeated values

If the dictionary contains repeated `values`, reversing it becomes an ill-defined task with unpredictable results. For illustration, here is the code from the previous subsection again, except we added one more car whose price is 15000:

```python
def reverse(D):
    return {v:k for k, v in D.items()}

cars = {'Nissan Altima':10000, 'Toyota Camry':15000,
'Audi S3':25000, 'Subaru Forester':15000}
reverse(cars)
```
```
{10000:'Nissan Altima', 25000:'Audi S3',
15000:'Subaru Forester'}
```

But wait - where is the Camry?

The answer is that the pair `15000:'Toyota Camry'` was overwritten with the pair `15000:'Subaru Forester'` (you know from Subsection 7.42 that a dictionary cannot contain repeated `keys`). However, since the items in a dictionary are not ordered, the pair `15000:'Subaru Forester'` could very easily have been overwritten with the pair `15000:'Toyota Camry'`. Then not the Camry, but the Forester would be missing. Hence the outcome of the program depends on a concrete implementation of Python, and/or on the order we insert items in our dictionary, neither of which is a good thing.

7.48 Creating a dictionary from two lists

You already know how to decompose a dictionary D into a pair of lists `D.keys()` and `D.values()`. This is the reverse task. But it can be solved easily using the `zip` method you know from Subsection 7.28:

```python
makes = ['Nissan Altima', 'Toyota Camry', 'Audi S3']
prices = [10000, 15000, 25000]
D = dict(zip(makes, prices))
print(D)
```
```
{'Audi S3': 25000, 'Nissan Altima': 10000,
'Toyota Camry': 15000}
```

7.49 Reversing a dictionary using `zip`

Here is an alternative solution to the task from Subsection 7.46 based on zipping:

```python
def reverse(D):
    return dict(zip(D.values(), D.keys()))

cars = {'Nissan Altima':10000, 'Toyota Camry':15000,
'Audi S3':25000, 'Subaru Forester':15000}
reverse(cars)
```
```
cars = {10000:'Nissan Altima', 25000:'Audi S3',
15000:'Subaru Forester'}
```

As you can see, the Camry disappeared again. This is not a fault of the method – the problem is that the dictionary contains repeated values, as we discussed in Subsection 7.47.

7.50 Creating a copy of a dictionary

Dictionaries are mutable objects (see Subsection 7.33) and therefore one could make a mistake trying to create a copy `cars2` of `cars` by typing `cars2 = cars`. The code below does that:

```
cars = {'Nissan Altima':10000, 'Toyota Camry':15000,
'Audi S3':25000}
cars2 = cars
cars2['Volvo XC60'] = 35000
print(cars)
print(cars2)
```
```
{'Nissan Altima': 10000, 'Toyota Camry': 15000,
'Audi S3': 25000, 'Volvo XC60': 35000}
{'Nissan Altima': 10000, 'Toyota Camry': 15000,
'Audi S3': 25000, 'Volvo XC60': 35000}
```

You can see that when the "copy" `cars2` was altered, so was the original dictionary. The reason is that both `cars2` and `cars` point to the same location in the memory.

The correct way to create a copy is to type either `cars2 = dict(cars)` or `cars2 = cars.copy()`. Let's use the former approach in the following example. The example also shows that when `cars2` is altered, `cars` remains unchanged:

```
cars = {'Nissan Altima':10000, 'Toyota Camry':15000,
'Audi S3':25000}
cars2 = dict(cars)
cars2['Volvo XC60'] = 35000
print(cars)
print(cars2)
```
```
{'Nissan Altima': 10000, 'Toyota Camry': 15000,
'Audi S3': 25000}
{'Nissan Altima': 10000, 'Toyota Camry': 15000,
'Audi S3': 25000, 'Volvo XC60': 35000}
```

7.51 Merging dictionaries

Let's say that we have two dictionaries such as

```
cars = {'Nissan Altima':10000, 'Toyota Camry':15000,
'Audi S3':25000}
```

and

```
cars2 = {'Audi S3':30000, 'Volvo XC60':35000}
```

135

and we need to merge `cars2` into `cars`. The `dict` class has a method `update` for this:

```
cars.update(cars2)
print(cars)
```
```
{'Nissan Altima': 10000, 'Toyota Camry': 15000,
'Audi S3': 30000, 'Volvo XC60': 35000}
```

Notice that the price of the Audi is 30000 now – the reason is that the original `value` in `cars` was overwritten with the new `value` from `cars2` for the same `key`.

7.52 Adding dictionaries

Remember how text strings and lists can be added using the + symbol? It would be nice if this worked for dictionaries too, but unfortunately it does not (maybe it will, in a future version of Python). The classical way to add two dictionaries includes two steps:

1. Create a copy of the first dictionary.
2. Merge the other dictionary into it.

Here is an example:

```
newcars = dict(cars)
newcars.update(cars2)
print(newcars)
```
```
{'Nissan Altima': 10000, 'Toyota Camry': 15000,
'Audi S3': 30000, 'Volvo XC60': 35000}
```

Starting with Python version 3.5, it is possible to use a compact notation {**X, **Y} to add dictionaries X and Y. If there are shared keys, then `values` from Y will overwrite the corresponding `values` in X. Let's illustrate this on our cars:

```
newcars = {**cars, **cars2}
print(newcars)
```
```
{'Nissan Altima': 10000, 'Toyota Camry': 15000,
'Audi S3': 30000, 'Volvo XC60': 35000}
```

7.53 Composing dictionaries

This is another nice exercise for dictionary comprehension. Let's define two sample dictionaries – one English-Spanish,

136

```
en_es = {'fire':'fuego', 'water':'agua'}
```

and one English-French:

```
en_fr = {'fire':'feu', 'water':'eau'}
```

Assuming that all keys which are present in en_es are also present in en_fr, one can create the corresponding Spanish-French dictionary as follows:

```
es_fr = {v:en_fr[k] for k, v in en_es.items()}
print(es_fr)
```
```
{'fuego': 'feu', 'agua': 'eau'}
```

However, this simple approach will fail with a KeyError if the sets of keys in en_es and en_fr differ. Since the comprehension goes over all keys in en_es, the remedy is to include an additional condition checking that the key also is in en_fr:

```
en_es = 'fire':'fuego', 'water':'agua', 'and':'y'
en_fr = 'fire':'feu', 'water':'eau'
es_fr = {v:en_fr[k] for k, v in en_es.items() \
        if k in en_fr.keys()}
print(es_fr)
```
```
{'fuego': 'feu', 'agua': 'eau'}
```

7.54 Sets

Do you still remember your Venn diagrams, and set operations such as set union and set intersection? That's exactly what sets in Python are foor. The implementation of the set data type in Python follows closely their math definition. One needs to remember that:

– They are not ordered.
– They cannot contain repeated items.
– They are mutable.

In the rest of this section we will show you their most important properties, and some interesting things you can do with sets.

7.55 Creating sets

An empty set S can be created by typing

```
S = set()
```

The easiest way to create a nonempty set is to type:

```
S = {1, 2, 3}
print(S)
```
```
{1, 2, 3}
```

Notice that sets use curly braces, same as dictionaries. You can also pass a list into their constructor:

```
S = set([1, 2, 3, 1, 2, 3])
print(S)
```
```
{1, 2, 3}
```

As you can see, the repeated items were removed. And as a last example, one can initialize a set with a text string, in which case the set will contain all invididual characters without repetition:

```
S = set('independence')
print(S)
print(len(S))
```
```
{'i', 'c', 'e', 'd', 'n', 'p'}
6
```

As you can see, function `len` can be used to obtain the number of items in a set.

7.56 Adding and removing items

New element x can be added to a set S by typing `S.add(x)`:

```
S = set('independence')
S.add('q')
print(S)
```
```
{'i', 'c', 'e', 'd', 'n', 'p', 'q'}
```

An entire new set N can be added to an existing set S by typing `S.update(N)`:

```
S = set('independence')
S = set('nicer')
S.update(N)
print(S)
{'n', 'd', 'p', 'r', 'i', 'e', 'c'}
```

As you can see, the order of elements in a set can change arbitrarily while performing set operations.

There are two basic ways to remove an element x from a set S. First, typing

```
S.remove(x)
```

However, this will raise `KeyError` if the element is not in the set. Alternatively, one can also use

```
S.discard(x)
```

which will only remove the element if it's present. And last, typing

```
S.clear()
```

will remove all elements from the set S.

7.57 Popping random elements

Since sets are not ordered like lists, it is not possible to pop the first element, the last element, or an element with a given index. Instead, calling `S.pop()` will remove and return a random element. The function will throw a `KeyError` if the set is empty. Here is an example:

```
S = set('independence')
print(S.pop())
print(S.pop())
print(S)
'i'
'd'
{'n', 'p', 'e', 'c'}
```

139

To avoid the `KeyError`, you can always check whether a set is empty before popping an element:

```python
if len(S) != 0:
    x = S.pop()
```

7.58 Checking if an element is in a set

As you would expect, this is done using the keyword `in` as with lists, tuples, and dictionaries:

```python
S = set('independence')
x = 'e'
print(S)
print(x in S)
```
```
{'n', 'd', 'p', 'i', 'e', 'c'}
True
```

7.59 Checking for subsets and supersets

Let's have three sets

```python
A = {1, 2, 3, 4}
B = {3, 4, 5}
C = {3, 4}
```

One can check if C is a subset of A (meaning that every element of C is also present in A) by typing:

```python
C.issubset(A)
```
```
True
```

The same can be done using the symbol <=:

```python
C <= A
```
```
True
```

Next, one can check if A is a superset of B (meaning that every element of B is also present in A) by typing:

```
A.issuperset(B)
```
```
False
```

Equivalently, one can use the symbol >=:

```
A >= B
```
```
False
```

7.60   Intersection and union

Let's have the same three sets as in the previous subsection:

```
A = {1, 2, 3, 4}
B = {3, 4, 5}
C = {3, 4}
```

The intersection of A and B is the set of all elements which are both in A and in B:

```
I = A.intersection(B)
print(I)
```
```
{3, 4}
```

Equivalently, one can use the symbol &:

```
I = A & B
print(I)
```
```
{3, 4}
```

The union of A and B is the set of all elements which are in A or in B (or in both at the same time):

```
U = A.union(B)
print(U)
```
```
{1, 2, 3, 4, 5}
```

Equivalently, one can use the symbol |:

```
U = A | B
print(U)
```

```
{1, 2, 3, 4, 5}
```

7.61 Difference and symmetric difference

Here are the three sets from the previous subsection again:

```
A = {1, 2, 3, 4}
B = {3, 4, 5}
C = {3, 4}
```

The set difference of A and B is the set of all elements of A which are not in B:

```
D = A.difference(B)
print(D)
```
```
{1, 2}
```

Equivalently, one can use the symbol −:

```
D = A - B
print(D)
```
```
{1, 2}
```

The symmetric difference of A and B is the set of all elements of A which are not in B, and of all elements of B which are not in A:

```
SD = A.symmetric_difference(B)
print(SD)
```
```
{1, 2, 5}
```

Equivalently, one can use the symbol ^:

```
SD = A ^ B
print(SD)
```
```
{1, 2, 5}
```

We will stop here but there is additional set functionality that you can find in the official Python documentation at https://docs.python.org.

7.62 Using a set to remove duplicated items from a list

Let's close this section with a neat application of sets – converting a list to a set a set can be used to quickly and elegantly remove duplicated items from a list:

```python
L = [1, 2, 3, 1, 2, 3, 4]
L = list(set(L))
print(L)
```
```
{1, 2, 3, 4}
```

One has to keep in mind though that the conversion to a set and back to a list can change the order of items.

8 Functions

8.1 Objectives

In this Section we will learn

– How to define and call functions.
– Why it is important to write *docstrings*.
– About the difference between *parameters* and *arguments*.
– How to return multiple values.
– What one means by a *wrapper*.
– How to use parameters with default values.
– How to accept a variable number of arguments.
– How to define local functions within functions.
– About anonymous lambda functions.
– That functions can create and return other functions.
– That functions are objects, and how to take advantage of that.

8.2 Why learn about functions?

The purpose of writing functions is to make selected functionality easily reusable throughout the code. In this way one avoids code duplication, and brings more structure and transparency into the software.

8.3 Defining new functions

In Python, the definition of every new function begins with the keyword `def`, but one moreover has to add round brackets and the colon ':' at the end of the line. The following sample function adds two numbers and returns the result:

```python
def add(a, b):
    """
    This function adds two numbers.
    """
    return a + b
```

The round brackets in the function definition are mandatory even if the function does not have any input parameters, but the `return` statement can be omitted if not needed. Lines 2 - 4 contain a docstring. Docstrings are optional, but providing one with every function is very important. We will explain why in the following Subsection 8.4.

144

Note that the code above contains a function definition only – the function is not called. In other words, if you run that program, nothing will happen. In order to call the function and add some numbers with it, one needs to write one additional line such as:

```python
print('5 + 3 is', add(5, 3))
```

This will produce the following output:

```
5 + 3 is 8
```

In most cases, functions are defined to process some input parameters and to return some output values. However, this does not apply always. The following function does not take any arguments and it does not return anything:

```python
def print_hello():
    """
    This function displays 'Hello!'
    """
    print('Hello!')
```

8.4   Docstrings and function `help`

Every function should have a docstring – concise and accurate description of what it does. This is useful not only for other people who look at your software, but also for you. There is a famous saying that when you see your own code after several moths, it might as well be written by somebody else – and there is something to it. Showing other people a code where functions do not have docstrings will lead them to think that you probably are not a very experienced programmer.

In addition, Python has a built-in function `help` which displays the docstrings. This is incredibly useful in larger software projects where usually you do not see the source code of all functions that you are using. Here is a simple example:

```python
def print_hello():
    """
    This function displays 'Hello!'
    """
    print('Hello!')

help(print_hello)
```

```
Help on function print_hello:

print_hello()
    This function displays 'Hello!'
```

8.5 Function *parameters* vs. *arguments*

The words "parameter" and "argument" are often confused by various authors, so let's clarify them. When we talk about the *definition* of a function, such as

```python
def add(a, b):
    """
    This function adds two numbers.
    """
    return a + b
```

then we say that the function `add` has two *parameters* `a` and `b`. But when we are talking about *calling* the function, such as

```python
c = add(x, y)
```

then we say that the function was called with the *arguments* `x` and `y`.

8.6 Names of functions should reveal their purpose

You already know from Section 5 that Python is a dynamically (weakly) typed language. This means that one does not have to specify the types of newly created variables.

Analogously, one does not have to specify the types of function parameters. This means that the above function `add(a, b)` will work not only for numbers but also for text strings, lists, and any other objects where the operation '+' is defined. Let's try this:

```python
def add(a, b):
    """
    This function adds two numbers.
    """
    return a + b

word1 = 'Good '
word2 = 'Evening!'
print(add(word1, word2))
```
```
Good Evening!
```

Or this:

```python
def add(a, b):
    """
    This function adds two numbers.
    """
    return a + b

L1 = [1, 2, 3]
L2 = ['a', 'b', 'c']
print(add(L1, L2))
```
```
[1, 2, 3, 'a', 'b', 'c']
```

In the last two examples, the use of the function was incompatible with its original purpose which is described in the docstring. Statically typed languages such as C, C++ or Java are more strict and this is not possible. As usual, Python gives you plenty of freedom. Do not abuse it. Reading your code should not be an intellectual exercise. On the contrary – by looking at your code, the reader should effortlessly figure out what the code is doing. For this reason, instead of using a "smart" universal function such as the function add defined above, it would be better to define separate functions add_numbers, add_strings and add_lists whose names clearly state what they are meant to do.

8.7   Functions returning multiple values

Python functions can return multiple values which often comes very handy. For example, the following function time decomposes the number of seconds into hours,

minutes and seconds, and returns them as three values:

```
def time(s):
    """
    This function converts a number of seconds to hours,
    minutes and seconds.
    """
    h = s // 3600
    s %= 3600
    m = s // 60
    s %= 60
    return h, m, s
```

Technically, the returned comma-separated values h, m, s are a tuple, so the function could be written with (h, m, s) on the last line:

```
def time(s):
    """
    This function converts a number of seconds to hours,
    minutes and seconds.
    """
    h = s // 3600
    s %= 3600
    m = s // 60
    s %= 60
    return (h, m, s)
```

The tuple which is returned from the function can be stored in a single variable which then becomes a tuple:

```
t = time(3666)
print(t)
(1, 1, 6)
```

Or the returned tuple can be stored in three separate variables:

```
v1, v2, v3 = time(3666)
print(v1, v2, v3)
1  1  6
```

148

8.8   What is a *wrapper*?

Sometimes, when looking for some answers on the web, you may come across the word *wrapper*. By a wrapper one usually means a function that "wraps" around another function – meaning that it does not do much besides just calling the original function, but adding or changing a few things. Wrappers are often used for functions which one cannot or does not want to change directly. Let's show an example.

In Subsection 4.33 you learned how to use the `ctime` function of the `time` library to extract the current date and time from the system. The 24-character-long text string contains a lot of information:

```
Sat May 12 13:20:27 2018
```

But what if one only needs to extract the date? Here is a wrapper for the function `time.ctime` which does that:

```python
def date():
    """
    This function returns date in the form 'May 12, 2018'.
    """
    txt = time.ctime()
    return txt[4:10] + ',' + txt[-5:]

import time
print(date())
```
```
May 12, 2018
```

Another example of a simple wrapper will be shown in the next subsection. Later, in Subsection 18.1, we will take wrappers to the next level.

8.9   Using parameters with default values

Have you ever been to Holland? It is the most bicycle friendly place in the world. Imagine that you work for the Holland Census Bureau. Your job is to ask 10000 people how they go to work, and enter their answers into a database. The program for entering data into the database was written by one of your colleagues, and it can be used as follows:

```python
add_database_entry('John', 'Smith', 'walks')
```

or

```
add_database_entry('Louis', 'Armstrong', 'bicycle')
```

or

```
add_database_entry('Jim', 'Bridger', 'horse')
```

etc. Since you are in Holland, it can be expected that 99% of people are using the bicycle. In principle you could call the function `add_database_entry()` to enter each answer, but with 9900 bicyclists out of 10000 respondents you would have to type the word "bicycle" many times.

Fortunately, Python offers a smarter way to do this. One can define a new function

```
def enter(first, last, transport='bicycle'):
    """
    This function calls enter_into_database with
    transport='bicycle' by default.
    """
    enter_into_database(first, last, transport)
```

This is a simple (thin) wrapper to the function `add_database_entry()` that allows us to omit the third argument in the function call and autocomplete it with a default value which is `"bicycle"`. In other words, now we do not have to type `"bicycle"` for all the bicyclists:

```
enter('Louis', 'Armstrong')
```

Only if we meet a rare someone who uses a car, we can type

```
enter('Niki', 'Lauda', 'car')
```

A few things to keep in mind:

(1) Parameters with default values need to be introduced after standard (non-default) parameters. In other words, the following code will result into an error:

```
def add(a=5, b):
    """
    This function adds a, b with default value a=5.
    """
    return a + b
on line 1:
SyntaxError: non-default argument follows default argument
```

(2) When any of the optional arguments are present in the function call, it is a good practise to use their names to avoid ambiguity and make your code easier to read:

```
def add(x, a=2, b=3):)
    """
    This function adds x, a, b with default values a=2 and b=3.
    """
    return x + a + b

print(add(1, b=5)
```

In this case the value 1 will be assigned to x, a will take the default value 2, and b will be 5. The output will be

```
8
```

8.10   Functions accepting a variable number of arguments

The following function will multiply two numbers:

```
def multiply(x, y):
    """
    This function multiplies two values.
    """
    return x * y

print(multiply(2, 3))
6
```

But what if we wanted a function that can by called as multiply(2, 3, 6) or multiply(2, 3, 6, 9), multiplying a different number of values each time?

Well, the following could be done using a list:

```python
def multiply(L):
    """
    This function multiplies all values coming as a list.
    """
    result = 1
    for v in L:
        result *= v
    return result

print(multiply([2, 3, 6]))
```
```
36
```

But then one would have to use extra square brackets when calling the function. For two values, the code `multiply([2, 3])` would not be backward compatible with the original code `multiply(2, 3)`.

This problem can be solved using `*args`. Using `*args` is very similar to using a list, but one does not need to use the square brackets when calling the function afterwards:

```python
def multiply(*args):
    """
    This function multiplies an arbitrary number of values.
    """
    result = 1
    for v in args:
        result *= v
    return result

print(multiply(2, 3, 6))
```
```
36
```

8.11 `*args` **is not a keyword**

As a remark, let us mention that `args` is just a name for a tuple, and any other name would work as well:

```python
def multiply(*values):
    """
    This function multiplies an arbitrary number of values.
    """
    result = 1
    for v in values:
        result *= v
    return result

print(multiply(2, 3, 6))
```
```
36
```

However, using the name `args` in functions with a variable number of arguments is so common that using a different name might confuse a less experienced programmer. Therefore, we recommend to stay with `args`.

8.12 Passing a list as `*args`

When using `*args`, then `args` inside the function is a tuple (immutable type), not a list. So, using `*args` is safer from the point of view that the function could change the list, but it cannot change the `args`.

When we have many numbers which we don't want to type one after another, or when we want to pass a list L as `*args` for another reason, it can be done by passing `*L` to the function instead of `L`. The following example multiplies 1000 numbers:

```python
def multiply(*args):
    """
    This function multiplies an arbitrary number of values.
    """
    result = 1
    for v in args:
        result *= v
    return result

L = []
for i in range(1000):
    L.append(1 + i/1e6)
print(multiply(*L))
```
```
1.6476230382011265
```

8.13   Obtaining the number of `*args`

For some tasks one needs to know the number of the arguments which are passed through `*args`. Since `args` is a tuple inside the function, one can normally use the function `len`. This is illustrated on the function `average` below which calculates the arithmetic average of an arbitrary number of values:

```python
def average(*args):
    """
    This function calculates the average of an arbitrary
    number of values.
    """
    n = len(args)
    s = 0
    for v in args:
        s += v
    return s / n

average(2, 4, 6, 8, 10)
```
```
6.0
```

8.14   Combining standard arguments and `*args`

Let's say that we need a function which not only calculates the average of an arbitrary number of values, but in addition increases or decreases the result by a given offset. The easiest way is to put the standard parameter first, and `*args` second:

```python
def average(offset, *args):
    """
    This function calculates the average of an arbitrary
    number of values, and adds an offset to it.
    """
    n = len(args)
    s = 0
    for v in args:
        s += v
    return offset + s / n

average(100, 2, 4, 6, 8, 10)
```
```
106.0
```

You could also do it the other way round – put `*args` first and the offset second – but this is not natural and people would probably ask why you did it. When calling the function, you would have to keyword the standard argument to avoid ambiguity:

```python
def average(*args, offset):
    """
    This function calculates the average of an arbitrary
    number of values, and adds an offset to it.
    """
    n = len(args)
    s = 0
    for v in args:
        s += v
    return offset + s / n

average(2, 4, 6, 8, 10, offset=100)
```
```
106.0
```

8.15 Using keyword arguments `*kwargs`

The name `kwargs` stands for "keyword arguments". You already know from Subsection 8.10 that `args` is a tuple inside the function. Analogously, `kwargs` is a dictionary. In other words, the `**kwargs` construct makes it possible to pass a dictionary into a function easily, without using the full dictionary syntax. In Subsection 8.11 we mentioned that `args` is just a name and one can use any other name instead. The same holds about `kwargs`.

For illustration, the following function `displaydict` accepts a standard dictionary `D` and displays it, one item per line:

```python
def displaydict(D):
    """
    This function displays a dictionary, one item per line.
    """
    for key, value in D.items():
        print(key, value)

displaydict({'Jen':7023744455, 'Julia':7023745566,
'Jasmin':7023746677})
```

```
Jen 7023744455
Julia 7023745566
Jasmin 7023746677
```

The code below does the same using `**kwargs`. Notice that instead of the colon `:` one uses the assignment operator `=`. Also notice that the keys are not passed as text strings anymore – they are converted to text strings implicitly:

```python
def displaydict(**kwargs):
    """
    This function displays a list of keyword arguments,
    one item per line.
    """
    for key, value in kwargs.items():
        print(key, value)

displaydict(Jen=7023744455, Julia=7023745566,
Jasmin=7023746677)
```
```
Jen 7023744455
Julia 7023745566
Jasmin 7023746677
```

8.16   Passing a dictionary as `**kwargs`

In Subsection 8.12 you have seen how one can pass a list to a function through `*args`. Analogously, it is possible to pass a dictionary to a function through `**kwargs`, as we show in the following example:

```python
def displaydict(**kwargs):
    """
    This function displays a list of keyword arguments,
    one item per line.
    """
    for key, value in kwargs.items():
        print(key, value)

D = {'Jen':7023744455, 'Julia':7023745566, 'Jasmin':7023746677}
displaydict(**D)
```

```
Jen 7023744455
Julia 7023745566
Jasmin 7023746677
```

8.17 Defining local functions within functions

In Subsections 5.13 - 5.15 you learned about the importance of using local (as well as *not using global*) variables in functions. In short, defining variables on the global scope pollutes the entire code because the name of a global variable can clash with the definition of a function or variable defined elsewhere.

Using local functions is less common than using local variables. But if one needs to use some functionality more than once within the body of a function, and not elsewhere in the code, then one should create a local function for it. Typically, the local function would be a small, one-purpose function while the outer function might be rather large and complex. Here is a bit artificial example whose sole purpose is to illustrate the mechanics of this:

```python
def split_numbers(L):
    """
    This function splits a list of integers
    into two lists of even and odd numbers.
    """

    def is_even(v):
        """
        This function returns True if the number n
        is even, and False otherwise.
        """
        return v%2 == 0

    E = []
    O = []
    for n in L:
        if is_even(n):
            E.append(n)
        else:
            O.append(n)
    return E, O
```

157

```
nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
enums, onums = split_numbers(nums)
print("Even:", enums)
print("Odd:", onums)
```
```
Even: [2, 4, 6, 8, 10]
Odd: [1, 3, 5, 7, 9]
```

We will not copy the entire code once more here, but if we tried to call the function `is_even` outside the function `split_numbers`, the interpreter would throw an error message similar to this one:

```
on line 28:
NameError: name 'is_even' is not defined
```

8.18 Anonymous lambda functions

Python has a keyword `lambda` which makes it possible to define anonymous one-line functions. The most important thing about these functions is that they are *expressions* (in contrast to standard functions defined using the `def` statement). Therefore, lambda functions can be used where statements are not allowed. The following example defines a lambda function which for a given value of x returns `x**2`, and then calls it with `x = 5`:

```
g = lambda x: x**2
g(5)
```
```
25
```

Note that the keyword `lambda` is followed by the name of the independent variable (here x), a colon, and then an expression containing the independent variable.

Also note that the anonymous function is assigned to a variable named g, by which it loses its anonymity. This negates the purpose of the keyword `lambda` and it's not how lambda functions are normally used. We did it here for simplicity – just to be able to call the function and display the result without making things more complex right away.

Before getting to more realistic applications of anonymous functions, let's mention that (in rare cases) they can be defined without parameters:

```
import time
day = lambda : time.ctime()[:3]
day()
```

```
Wed
```

And they can be defined with more than one parameter if needed:

```
plus = lambda x, y: x + y
plus(3, 4)
7
```

8.19 Creating multiline lambdas

Although it is possible to create multi-line lambda functions, this is against the philosophy of Python and strongly discouraged. If you really must do it, search the web and you will find your example.

8.20 Using lambdas to create a live list of functions

Let's say that you have a large data file `D`, and for each value `v` in it you need to calculate `v**2`, `cos(v)`, `sin(v)`, `exp(v)` and `log(v)`. Here is a very small sample of your data file:

```
D = [2.1, 1.9, 3.2]
```

Then an elegant approach is to define a list containing the five functions,

```
import numpy as np
F = [lambda x: x**2, lambda x: np.cos(x), \
lambda x: np.sin(x), lambda x: np.exp(x), lambda x: np.log(x)]
```

and use a pair of nested `for` loops to apply all functions to all data points:

```
for v in D:
    for f in F:
        print(round(f(v), 5), end=' ')
    print()
```

Output:

```
4.41 -0.50485 0.86321 8.16617 0.74194
3.61 -0.32329 0.9463 6.68589 0.64185
10.24 -0.99829 -0.05837 24.53253 1.16315
```

As you could see, in this example the five lambda functions were truly anonymous – their names were not needed because they were stored in a list.

8.21 Using lambdas to create a live table of functions

Imagine that you want to create a table (dictionary) of functions where on the left will be their custom names and on the right the functions themselves as executable formulas. This can be done as follows:

```python
import numpy as np

def addfns(d, **kwargs):
    """
    Add into dictionary d an arbitrary number of functions.
    """
    for name, f in kwargs.items():
        d[name] = f

F = {}
addfns(F, square=lambda x: x**2, cube=lambda x: x**3, \
exp=lambda x: np.exp(x))

print(F['square'](2))
print(F['cube'](2))
print(F['exp'](2))
```
```
4
8
7.38905609893065
```

8.22 Using lambdas to create a live table of logic gates

Recall that there are seven types of basic logic gates:

- AND(a, b) = a and b,
- OR(a, b) = a or b,
- NOT(a) = not a,
- NAND(a, b) = not(AND(a, b)),
- NOR(a, b) = not(OR(a, b)),
- XOR(a, b) = (a and not b) or (not a and b),
- XNOR(a, b) = not(XOR(a, b)).

160

The previous example can easily be adjusted to create their table:

```python
def addfns(d, **kwargs):
    """
    Add into dictionary d an arbitrary number of functions.
    """
    for name, f in kwargs.items():
        d[name] = f

LG =
addfns(LG, AND=lambda a, b: a and b, OR=lambda a, b: a or b, \
        NOT=lambda a: not a, NAND=lambda a, b: not (a and b), \
        NOR=lambda a, b: not (a or b), \
        XOR=lambda a, b: (a and not b) or (not a and b), \
        XNOR=lambda a, b: not((a and not b) or (not a and b)))

x = True
y = True
print(LG['AND'](x, y))
print(LG['OR'](x, y))
print(LG['NOT'](x))
print(LG['NAND'](x, y))
print(LG['NOR'](x, y))
print(LG['XOR'](x, y))
print(LG['XNOR'](x, y))
```

```
True
True
False
False
False
False
True
```

8.23 Using lambdas to create a factory of functions

We have not talked yet about functions which create and return other functions. This is a very natural thing in Python which has many applications. To mention just one - it allows Python to be used for *functional programming*. This is an advanced programming paradigm which is very different from both procedural and object-oriented program-

ming. We will not discuss it in this course, but if you are interested, you can read more on Wikipedia (`https://en.wikipedia.org/wiki/Functional_programming`).

The following example defines a function `quadfun(a, b, c)` which returns a function $ax^2 + bx + c$, and shows two different ways to call it:

```python
def quadfun(a, b, c):
    """
    Return a quadratic function ax^2 + bx + c.
    """
    return lambda x: a*x**2 + b*x + c


f = quadfun(1, -2, 3)
print(f(1))
print(quadfun(1, -2, 3)(1))
```
```
2
2
```

Standard functions which are defined locally within other functions can be returned in the same way.

8.24 Variables of type 'function'

In the previous example, we created a quadratic function $x^2 - 2x + 3$ and assigned it to a variable named `f`. Hence this variable is callable (has type 'function'). You already know from Subsection 5.11 how to check the type of a variable using the built-in function `type`:

```python
print(type(f))
```
```
<class 'function'>
```

Python has a built-in function `callable` which can be used to check at runtime whether an object is a function:

```python
if callable(f):
    print('f is a function.')
else:
    print('f is not a function.')
```
```
f is a function.
```

162

8.25 Inserting conditions into lambda functions

Look at the sample functions `min`, `even` and `divisible`:

```
min = lambda a, b: a if a < b else b
even = lambda n: True if n%2 == 0 else False
divisible = lambda n, m: True if n%m == 0 else False
```

Here, `if-else` is not used as a statement, but as a *conditional expression* (ternary operator). We will talk more about this in Subsection 10.10. For now, the above examples give you an idea of how to incorporate conditions into lambda functions.

Notice that we named the anonymous functions right after creating them, which contradicts their purpose. We just did this here to illustrate the inclusion of the conditional statement without making things unnecessarily complex.

8.26 Using lambdas to customize sorting

As you know from Subsection 7.22, calling `L.sort()` will sort the list `L` in place. From Subsection 7.23 you also know that Python has a built-in function `sorted` which returns a sorted copy of the list `L` while leaving `L` unchanged. Both `sort` and `sorted` accept an optional argument `key` which is an anonymous function.

This function, when provided, is applied to all items in the list `L` prior to sorting. It creates a helper list. The helper list is then sorted, and the original list `L` simultaneously with it. This makes it possible to easily implement many different sorting criteria.

Let's look at an example that sorts a list of names in the default way:

```
L = ['Meryl Streep', 'Nicole Kidman', 'Julia Roberts']
S = sorted(L)
print(S)
```
```
['Julia Roberts', 'Meryl Streep', 'Nicole Kidman']
```

But it would be much better to sort the names according to the last name. This can be done by defining a `key` which splits each text string into a list of words, and picks the last item:

```
L = ['Meryl Streep', 'Nicole Kidman', 'Julia Roberts']
S = sorted(L, key=lambda w: w.split()[-1])
print(S)
```
```
['Nicole Kidman', 'Julia Roberts', 'Meryl Streep']
```

163

8.27 Obtaining useful information about functions from their attributes

Functions in Python are objects, which means that they have their own attributes (more about this will be said in Section 14). Three of the most widely used ones are `__name__` which contains the function's name, `__doc__` which contains the docstring (if defined), and `__module__` which contains the module where the function is defined.

Let's look, for example, at the built-in function `sorted`:

```python
fn = sorted
print(fn.__name__)
print('----------')
print(fn.__module__)
print('----------')
print(fn.__doc__)
```

```
sorted
----------
builtins
----------
Return a new list containing all items from the iterable in
ascending order.

A custom key function can be supplied to customize the sort
order, and the reverse flag can be set to request the result
in descending order.
```

This works in the same way for your custom functions (as long as the docstring is defined).

9 The `for` Loop

9.1 Objectives

This Section summarizes various aspects of using the `for` loop, ranging from trivial to more advanced. The `for` loop was already mentioned several times in this text, and we will give references to those subsections where relevant. Depending on what you already know, in this section you will either review or learn the following:

– How to parse text strings one character at a time.
– How to split a text string into a list of words.
– How to parse tuples and lists one item at a time.
– More about the built-in function `range`.
– About using the keyword `else` with a `for` loop.
– About list comprehension – a special form of the `for` loop.
– About iterators, and how to make them.

9.2 Why learn about the `for` loop?

The `for` loop is a fundamental construct that all imperative programming languages (Python, C, C++, Fortran, Java, ...) use to iterate over sequences. These sequences can have many different forms:

– Numbers such as 0, 1, 2, ...,
– Text characters in a word.
– Words in a text document.
– Lines, words, or text characters in a text file.
– Entries in a relational database,
– etc.

The `for` loop is not present in all programming languages though. For example, *pure functional* programming languages such as Dylan, Erlang, Haskell or Lisp iterate over sequences using recursion instead.

In some imperative languages such as C, C++, Fortran or Java the `for` loop is sometimes called the *counting* loop, because its primarily use is to iterate over sequences of numbers.

As usual, Python is very flexible – the `for` loop can naturally iterate over sequences of numbers, text characters, list / tuple / dictionary items, and over even more general sequences. We will talk about all this in he following subsections.

9.3 Parsing text strings

If you do not know the `for` loop yet, please read Subsections 4.22 and 4.23 now. In Subsection 4.22 you will see how the keyword `for` is used to create a `for` loop, and how the `for` loop can be used to parse text strings one character at a time. In Subsection 4.23 you will see how the `for` loop can be used to reverse text strings.

9.4 Splitting a text string into a list of words

In order to parse a text document (text string) one word at a time, it is convenient to split it into a list of individual words first. This was explained in detail in Subsection 4.41. In practise "the devil is in the detail" though, and therefore in Subsection 4.42 we explained in more detail how to take care of punctuation while splitting text strings. Last, lists and list operations were explained in Section 7.

9.5 Parsing lists and tuples (including comprehension)

Lists and tuples can be parsed with the `for` loop one item at a time analogously to how text strings are parsed one character at a time. This was explained in detail in Subsection 7.10. The `for` loop works in the same way with tuples – it does not distinguish between lists and tuples at all.

Importantly, list comprehension (which also applies to tuples) is just another form of the `for` loop. If you do not know how to use list comprehension yet, read Subsections 7.29 – 7.31 now.

9.6 Parsing two lists simultaneously

Sometimes one needs to go over two (or more) lists simultaneously. In this case the standard approach is to use the built-in function `zip` which you already know from Subsection 7.28. As an example, imagine that you have two lists of numbers, and you need to create a list of their pairs, but only accept pairs where the values are in increasing order. Here is the code:

```python
A = [1, 6, 3, 9]
B = [3, 2, 8, 5]
result = []
for x, y in zip(A, B):
    if x < y:
        result.append([x, y])
print(result)
[[1, 3], [3, 8]]
```

166

9.7 Iterating over sequences of numbers and the built-in function `range`

The built-in function `range` was first mentioned in Subsection 6.14. What we have not shown there though, is that `range(N)` creates a sequence of integers `0, 1, 2, ...,` `N-1`. The result is an object of type `range`,

```
s = range(5)
print(s)
isinstance(s, range)
```
```
range(5)
True
```

If needed, it can be cast to a list (or tuple) for display purposes:

```
list(range(5))
```
```
[0, 1, 2, 3, 4]
```

A `for` loop of the form

```
for n in range(N):
```

goes over the corresponding sequence of numbers one number at a time:

```
for i in range(5):
    print(i)
```
```
0
1
2
3
4
```

If needed, one can start the sequence with a nonzero integer:

```
list(range(10, 15))
```
```
[10, 11, 12, 13, 14]
```

It is possible to skip over numbers using an optional step argument. For example, the sequence of all odd numbers between 1 and 11 can be created as follows:

```
list(range(1, 12, 2))
```
```
[1, 3, 5, 7, 9, 11]
```

One has to be a bit careful when creating multiples though. Typing `range(1, 16, 3)` might appear to be the way to create the sequence of all multiples of 3 between 1 and 15, but that's not the case:

```
list(range(1, 16, 3))
```
```
[1, 4, 7, 10, 13]
```

A better way to achieve this goal is through list comprehension:

```
[1] + [3*n for n in range(1, 6)]
```
```
[1, 3, 6, 9, 12, 15]
```

And last, one can create descending sequences using negative step values. This is the sequence of even numbers 20, 18, ..., 2:

```
list(range(20, 0, -2))
```
```
[20, 18, 16, 14, 12, 10, 8, 6, 4, 2]
```

9.8   Parsing dictionaries (including comprehension)

Recall from Subsection 7.40 that a dictionary, in fact, is just a list of tuples of the form `(k, v)` where `k` is a `key` and `v` the corresponding `value`. From the same subsection you also know that the list of all keys in a dictionary named `D` can be extracted via `D.keys()`, the list of all values via `D.values()` and the list of all items via `D.items()`. The latter is a list of two-item tuples.

Hence, parsing a dictionary in fact means to parse a list. One has various options. To go through the list of all keys, type:

```
for k in D.keys():
```

The list of all values can be parsed via:

```
for v in D.values():
```

And to go through the list of all items, type:

```python
for k, v in D.items():
```

or

```python
for (k, v) in D.items():
```

Importantly, you should make yourself familiar with how comprehension is used for dictionaries. Unless you already did, make sure to read Subsection 7.45 where we showed how to use comprehension to find all keys for a given value, as well as Subsection 7.46 where we explained how to reverse a dictionary using comprehension.

9.9 Nested `for` loops

Loops can be *nested*, meaning that the body of a loop can contain another loop (whose body can contain another loop...). In this case the former is called *outer loop* and the latter is the *inner loop*. With each new nested level the indentation increases. This is shown in the following example which creates all combinations of given numbers and letters:

```python
nums = ['1', '2', '3', '4']
chars = ['A', 'B', 'C']
combinations = []
for n in nums:
    for c in chars:
        combinations.append(n + c)
print(combinations)
```
```
['1A', '1B', '1C', '2A', '2B', '2C', '3A', '3B', '3C', '4A',
'4B', '4C']
```

Let's also show an example of a triple-nested loop which creates all combinations of given numbers, letters and symbols:

```
nums = ['1', '2', '3', '4']
chars = ['A', 'B', 'C']
symbols = ['+', '-']
combinations = []
for n in nums:
    for c in chars:
        for s in symbols:
            combinations.append(n + c + s)
print(combinations)
```

```
['1A+', '1A-', '1B+', '1B-', '1C+', '1C-', '2A+', '2A-', '2B+',
'2B-', '2C+', '2C-', '3A+', '3A-', '3B+', '3B-', '3C+', '3C-',
'4A+', '4A-', '4B+', '4B-', '4C+', '4C-']
```

9.10   Terminating loops with the `break` statement

The `break` statement can be used to instantly terminate a `for` or `while` loop. It is useful in situations when it no longer makes sense to finish the loop. For example, let's say that we have a (very long) list of numbers, and it is our task to find out if all of them are positive. In this case, the best solution is to parse the list using a `for` loop, and terminate the loop when a negative number is found:

```
L = [1, 3, -2, 4, 9, 8, 4, 5, 6, 7]
allpositive = True
for n in L:
    if n < 0:
        allpositive = False
        break
print(allpositive)
```

```
False
```

If the `break` statement is used within nested loops, then it only terminates the nearest outer one. The following example illustrates that. It is analogous to the previous one, but it analyses a list of lists. You can see that the `break` statement only terminates the inner loop because the outer loop goes over the four sub-lists and displays the Boolean value of `allpositive` four times:

```
L = [[5, -1, 3], [2, 4], [9, 8, 3], [1, -4, 5, 6]]
for s in L:
    allpositive = True
    for n in s:
        if n < 0:
            allpositive = False
            break
    print(allpositive)
```
```
False
True
True
False
```

Importantly, the break statement is just a shortcut. It should be only used to prevent the computer from doing unnecessary operations. Do not get too inventive with this statement. Write your code in such a way that it would work without the break statement too. And finally – make sure to read Subsection 9.12 which is related to the break statement as well.

9.11 Terminating current loop cycle with continue

The continue statement is similar in nature to break, but instead of terminating the loop completely, it only terminates the current loop cycle, and resumes with the next one. As with the break statement, keep in mind that continue is a shortcut. Do not force it where you don't need it. Unfortunately, you will find many online tutorials which do exactly the opposite. Such as the following program which leaves out a given letter from a given text (we are printing in on red background because it's not a good code):

```
txt = 'Python'
skipchar = 'h'
for c in txt:
    if c == skipchar:
        continue
    print(c, end=' ')
```
```
P y t o n
```

The same result can be achieved way more elegantly without the continue statement:

171

```
txt = 'Python'
skipchar = 'h'
for c in txt:
    if c != skipchar:
        print(c, end=' ')
```

```
P y t o n
```

As a matter of fact, every `continue` statement can be replaced with a condition. So, the main benefit of using `continue` is that it "flattens" the code and makes it more readable when multiple nested conditions are present.

Let's illustrate this on a primitive translator from British to American English, whose sole purpose is to leave out the letter 'u' if it is following 'o', and when the next character is not 'n' or 's'. First, let's show the version with the `continue` statement:

```
txt = 'colourful and luxurious country home'
n = len(txt)
for i in range(n):
    if txt[i] != 'u':
        print(txt[i], end='')
        continue
    if i > 0 and txt[i-1] != 'o':
        print(txt[i], end='')
        continue
    if i < n-1 and txt[i+1] == 's':
        print(txt[i], end='')
        continue
    if i < n-1 and txt[i+1] == 'n':
        print(txt[i], end='')
```

```
colorful and luxurious country home
```

Below is the version without `continue`. Notice the complicated structure of the nested conditions:

172

```
txt = 'colourful and luxurious country home'
n = len(txt)
for i in range(n):
    if txt[i] != 'u':
        print(txt[i], end='')
    else:
        if i > 0 and txt[i-1] != 'o':
            print(txt[i], end='')
        else:
            if i < n-1 and txt[i+1] == 's':
                print(txt[i], end='')
            else:
                if i < n-1 and txt[i+1] == 'n':
                    print(txt[i], end='')
```
```
colorful and luxurious country home
```

For completeness let us add that the `continue` statement may only occur syntactically nested in a `for` or `while` loop, but not nested in a function or class definition or `try` statement within that loop.

9.12   The `for-else` statement

In Python, both the `for` loop and the `while` loop may contain an optional `else` branch. The `for-else` statement is sort of a mystery, and unfortunately the source of numerous incorrect explanations in online tutorials. There, one can sometimes read that the `else` branch is executed when the `for` loop receives an empty sequence to parse. Technically, this is true:

```
L = []
for n in L:
    print(n, end=' ')
else:
    print('The else branch was executed.')
```
```
The else branch was executed.
```

But the `else` branch is also executed when the sequence is not empty:

173

```
L = [1, 3, 5, 7]
for n in L:
    print(n, end=' ')
else:
    print('The else branch was executed.')
1 3 5 7 The else branch was executed.
```

So, why would one use the `else` branch at all?

Well, its purpose is different. As somebody suggested, it should have been named `nobreak` rather than `else`, and its purpose would be clear to everybody. Namely, the `else` branch is executed when the loop has finished regularly (not terminated with the `break` statement). If the loop is terminated with the `break` statement, the `else` branch is skipped. Effectively, this allows us to add code after the `for` loop which is skipped when the `break` statement is used. OK, but what is this good for? Let's show an example.

Imagine that you have a list of values, but some of them can be inadmissible (in our code below, inadmissible = negative). In particular, if all of them are inadmissible, the program cannot go on and needs to terminate with a custom error message. The `else` branch is perfect for catching this, and throwing that custom error message:

```
L = [-1, 2, -3, -4]
for n in L:
    # If admissible value was found, end the loop:
    if n > 0:
        break
else:
    raise Exception('Sorry, all values were negative.')
# Now some code that uses the admissible value:
import numpy as np
result = np.sqrt(n)
print('The result is', round(result, 4))
The result is 1.4142
```

Do not worry about the Exception – for now, it's there just to throw a custom error message. Exceptions will be discussed in detail in Section 12.

In the "old-fashioned" way, without the `else` branch, one would have to introduce an extra Boolean variable named (for example) `found` to find out whether an admissible value was found. And, one would have to use an extra condition to inspect the

value of `found`:

```python
L = [-1, 2, -3, -4]
found = False
for n in L:
    # If admissible value was found, end the loop:
    if n > 0:
        found = True
        break
if not found:
    raise Exception('Sorry, all values were negative.')
# Now some code that uses the admissible value:
import numpy as np
result = np.sqrt(n)
print('The result is', round(result, 4))
```
```
Result is 1.4142
```

In summary, the use of the `else` branch has saved us one variable and one condition.

9.13   The `for` loop behind the scenes – iterables and iterators

At the end of this section we would like to explain the technical details of the `for` loop, as well as the nature of objects this loop works with. For this, we will need to use some object-oriented terminology, so consider reading Section 14 first.

Let's begin with a simple `for` loop:

```python
s = 'Hi!'
for c in s:
    print(c, end=' ')
```
```
H i !
```

The text string `s` is an *iterable object* or just *iterable*, meaning that it has a method __iter__. Other examples of iterable objects in Python are lists, tuples, and dictionaries.

Calling `s.__iter__()` returns an *iterator*. Equivalently, the iterator can be obtained by calling `iter(s)` (the built-in function `iter` calls the __iter__ method of its argument implicitly).

Behind the scenes, the `for` statement in the above example calls `iter(s)`. For simplicity, let's call the returned iterator object `iterobj`. This object has the method

`__next__` which can access the elements in the container `s` one at a time. When there are no more elements, `__next__` raises a `StopIteration` exception which tells the for loop to terminate. The following example shows how the above `for` loop really works, by creating an iterator object `iterobj` from the text string `s`, and repeating the two lines

```
c = iterobj.__next__()
print(c, end=' ')
```

until the `StopIteration` exception is thrown:

```
s = 'Hi!'
iterobj = iter(s)
c = iterobj.__next__()
print(c, end=' ')
c = iterobj.__next__()
print(c, end=' ')
c = iterobj.__next__()
print(c, end=' ')
c = iterobj.__next__()
print(c, end=' ')
```
```
H i !
```
```
on line 9:
StopIteration
```

The last line 10 is not executed because the iteration is stopped on line 9. You can also iterate using the built-in function `next` which calls `__next__` implicitly:

```
s = 'Hi!'
iterobj = iter(s)
c = next(iterobj)
print(c, end=' ')
c = next(iterobj)
print(c, end=' ')
c = next(iterobj)
print(c, end=' ')
c = next(iterobj)
print(c, end=' ')
```
```
H i !
```

```
on line 9:
StopIteration
```

For completeness, let's perform the above exercise once more for a sample list `L = [1, 2, 3]`. First, let's create the corresponding iterator object and use its method `__next__` to iterate through `L`:

```
L = [1, 2, 3]
iterobj = iter(L)
n = iterobj.__next__()
print(n, end=' ')
n = iterobj.__next__()
print(n, end=' ')
n = iterobj.__next__()
print(n, end=' ')
n = iterobj.__next__()
print(n, end=' ')
```

```
1, 2, 3
```

```
on line 9:
StopIteration
```

And again, one can use the built-in function `next` to iterate without revealing the object-oriented nature of the iteration process:

```
L = [1, 2, 3]
iterobj = iter(L)
n = next(iterobj)
print(n, end=' ')
n = next(iterobj)
print(n, end=' ')
n = next(iterobj)
print(n, end=' ')
n = next(iterobj)
print(n, end=' ')
```

```
1, 2, 3
```

```
on line 9:
StopIteration
```

9.14 Making your own classes iterable

It is easy to add iterator behavior to your classes. Just add an __iter__ method which returns an object with a __next__ method. If your class already has a __next__ method, then __iter__ can just return self:

```python
class Reverse:
    """
    Iterable for looping over a sequence backwards.
    """
    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __iter__(self):
        return self

    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
```

And now let's use it:

```python
rev = Reverse('Hi!')
for c in rev:
    print(c, end=' ')
! i H
```

9.15 Generators

Generators are a simple and powerful tool for creating iterators. They are written like regular functions but use the yield statement whenever they want to return data. Each time the built-in function next is called on it, the generator resumes where it left off - it remembers all the data values and which statement was last executed. Anything that can be done with iterable objects as described in the previous section can also be done with generators. What makes generators so elegant is that the __iter__ and __next__ methods are created automatically. Let's show an example to make all this more clear.

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]

rev = reverse('Hi!')
for c in rev:
    print(c, end=' ')
```
! i H

9.16   Functional programming

The more you learn about iterators, the closer you get to functional programming. This is a fascinating programming style and Python provides modules `itertools` and `functools` to support it. To learn more, we recommend that you start with the *Functional Programming HOWTO* in the official Python documentation which can be found at

  `https://docs.python.org/dev/howto/functional.html#iterators`

10 Conditions

10.1 Objectives

Conditions have been mentioned in various contexts in Subsections 4.35, 4.44, 4.45, 4.46, 5.11, 6.13, 6.14, 7.18, 7.30, 7.31, 8.17, 8.24, 8.25, 9.10 and 9.11. Therefore, writing a separate Section about them may seem to be somewhat superfluous. Nevertheless, in this text we like to present every important concept of Python programming in a self-contained manner, and therefore we decided to include a Section on conditions. Feel free to skip it if you know everything about them. If you'll keep reading, in this Section we will review / learn the following:

- About Boolean (logical) values, operators, and expressions.
- That keyword `if` is not needed to implement conditions.
- About the `if` statement and types of values it accepts.
- About the optional `else` branch.
- About the full `if-elif-else` statement.
- About conditional (ternary) expressions.

10.2 Why learn about conditions?

Conditions allow computer programs to adapt to various types of events at runtime, ranging from calculated values to user or sensory input. They are part of all programming paradigms including imperative programming, functional programming, event-driven programming, object-oriented programming, etc.

10.3 Boolean values, operators, and expressions

Unless you are already familiar with Boolean values, operators, and expressions, now is the time to visit Section 6 and learn about them. In particular, in Subsection 6.13 we explained that conditions are based on Booleans.

10.4 Using conditions without the keyword `if`

You already know from Section 6 how to create truth tables. Therefore, it should not be hard for you to verify that the truth table of the Boolean expression

```
if A then B else C
```

is identical with the truth table of the expression

```
(A and B) or ((not A) and C)
```

This means that any language which defines Boolean operations automatically provides a way to handle `if-then-else` conditions, even when the keyword `if` is not used (such as for example in Prolog).

10.5   The `if` statement

In Python, the simplest way to conditionally execute code is to make it part of an `if` statement:

```
if <boolean_expression>:
    <do_something>
```

Note the indentation and the mandatory colon : after the Boolean expression. As an example:

```
if x > 0:
    print('The value of x is', x)
```

This code has several possible outputs. First, if x is positive (for example 1.5):

```
The value of x is 1.5
```

Second, if x is non-positive (for example 0):

Third, if x is of incompatible type:

```
TypeError: '>' not supported between 'str' and 'int'
```

And last, if x is undefined:

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

10.6 Types of values accepted by the `if` statement

When the interpreter finds in the code the keyword `if`, it looks for an expression behind it. If such an expression is not found, it throws an error. Otherwise the expression is evaluated.

It is worthwhile mentioning that not only `True` and `False` but also numbers, text strings, and even lists etc. are accepted as admissible values. Let's show some examples, starting with Booleans. The keyword `if` can be followed by a Boolean value:

```python
if True:
    print('This will always be printed.')
```

or

```python
if False:
    print('This will never be printed.')
```

One can use a Boolean variable:

```python
if flag:
    print('This will be printed if flag is True.')
```

Boolean expressions are admissible as well,

```python
if 'A' in txt:
    print("This will be printed if string txt contains 'A'.")
```

and so are Boolean functions:

```python
if isinstance(n, int):
    print('This will be printed if n is an integer.')
```

The keyword `if` can be followed by a number (integer, float or complex):

```python
if v:
    print('This will be printed if v is not zero.')
```

One can use a text string:

```
if txt:
    print('This will be printed if string txt is not empty.')
```

Even a list, tuple or a dictionary can be used:

```
if L:
    print('This will be printed if list L is not empty.')
```

However, using numbers, text strings, lists and such decreases code readability. It takes a little effort to be more explicit and convert these into Boolean expressions. For example, the last example can be improved as follows:

```
if len(L) != 0:
    print('This will be printed if list L is not empty.')
```

10.7   The optional `else` branch

The keyword `else` is optional and can be used for code to be evaluated when the condition is not satisfied:

```
if <boolean_expression>:
    <do_something>
else:
    <do_something_else>
```

Here is an example:

```
invitations = ['Eli', 'Elsa', 'Eva']
name = 'Anna'
if name in invitations:
    print(name + ' was invited.')
else:
    print(name + ' was not invited.')
Anna was not invited.
```

10.8   The full `if-elif-else` statement

The `elif` statement can be used to simplify cases with more than two options. If you think that "elif" sounds a lot like "else if" then you are absolutely right! In general,

183

```
if <boolean_expression_1>:
    <do_action_1>
elif <boolean_expression_2>:
    <do_action_2>
else:
    <do_action_3>
```

means exactly the same as

```
if <boolean_expression_1>:
    <do_action_1>
else:
    if <boolean_expression_2>:
        <do_action_2>
    else:
        <do_action_3>
```

Clearly, the latter involves more indentation. Exactly – the purpose of `elif` is to make complex conditions more flat. This becomes easier to see as the number of cases grows. Let us therefore show an example with five cases.

10.9   Example – five boxes of apples

Imagine that you have five wooden boxes:

- Box A is for apples that weight under 5 ounces.
- Box B is for apples that weight at least 5 but less than 10 ounces.
- Box C is for apples that weight at least 10 but less than 15 ounces.
- Box D is for apples that weight at least 15 but less than 20 ounces.
- Box E is for apples that weight at least 20 ounces.

Your task is to write a function `box(weight)` that, given the `weight` of an apple, chooses the correct box for it and returns the corresponding letter:

```python
def box(weight):
    if weight < 5:
        return 'A'
    elif weight < 10:
        return 'B'
    elif weight < 15:
        return 'C'
    elif weight < 20:
        return 'D'
    else:
        return 'E'
```

Here is the same code without the `elif` statement:

```python
def box(weight):
    if weight < 5:
        return 'A'
    else:
        if weight < 10:
            return 'B'
        else:
            if weight < 15:
                return 'C'
            else:
                if weight < 20:
                    return 'D'
                else:
                    return 'E'
```

In summary, the `elif` statement simplifies conditions with multiple cases.

10.10 Conditional (ternary) expressions

Python (as well as C/C++ and other languages) supports a flexible way of working with conditions called *conditional (ternary) expressions*. In Python, these expressions have the form

```
value_1 if boolean_expression else value_2
```

Unlike the `if`, `if-else` or `if-elif-else` statements, conditional expressions are not statements. This means that they can be inserted into formulas and text strings, lists, tuples, dictionaries, etc. In the following example, a conditional expression is used in a price calculation:

```python
discount = True
percent = 20
base = 1000
final_price = base * (1 - percent / 100) if discount else base
print('Your price is', final_price)
```

```
Your price is 800.0
```

And,

```python
discount = False
percent = 20
base = 1000
final_price = base * (1 - percent / 100) if discount else base
print('Your price is', final_price)
```

```
Your price is 1000
```

In this example, a conditional expression is used within a text string:

```python
name = 'Zoe'
age = 16
txt = name + (' is a child.' if age < 18 else ' is an adult.')
print(txt)
```

```
Zoe is a child.
```

And,

```python
name = 'Nathan'
age = 21
txt = name + (' is a child.' if age < 18 else ' is an adult.')
print(txt)
```

```
Nathan is an adult.
```

10.11   Nested conditional expressions

Conditional expressions can be nested, which makes it possible to handle situations with more than two options:

```python
name = 'Mark'
age = 14
txt = name + (' is a child.' if age < 13 else \
(' is a teenager.' if age < 18 else ' is an adult.'))
print(txt)
```
```
Mark is a teenager.
```

Using parentheses is desirable. Here is what happens when we remove them:

```python
name = 'Mark'
age = 14
txt = name + ' is a child.' if age < 13 else \
' is a teenager.' if age < 18 else ' is an adult.'
print(txt)
```
```
 is a teenager.
```

10.12   Famous workarounds

Ternary conditional expressions were not present in older versions of Python (below version 2.5). Therefore programmers developed clever tricks that emulate their behavior. These tricks are not needed today anymore, but you may find the code in older Python programs. Therefore it's good to know how they work. Let's get back to Zoe from Subsection 10.10 who is 16 years old:

```python
name = 'Zoe'
age = 16
txt = name + (' is an adult.', ' is a child.')[age < 18]
print(txt)
```
```
Zoe is a child.
```

And here is Nathan who with 21 years is an adult:

```python
name = 'Nathan'
age = 21
txt = name + (' is an adult.', ' is a child.')[age < 18]
print(txt)
```

```
Nathan is an adult.
```

To understand how this works, notice that `True` works as 1 and `False` as 0 when used as an index:

```
L = ['A', 'B']
print(L[False])
A
```

and

```
L = ['A', 'B']
print(L[True])
B
```

If you insisted on calling this a dirty trick, we would probably not object.

11 The 'While' Loop

11.1 Objectives

The `while` loop was mentioned in various contexts in Subsections 6.3, 6.13, 7.19, 9.10, 9.11 and 9.12. It is very similar to the `for` loop that you already know from Section 9, with one main difference. In other words, there is only one thing you need to learn about the `while` loop. In this Section we will show you:

 – The main difference between the `for` and `while` loops.
 – How to terminate a `while` loop with a `break` statement.
 – How to measure the execution time of your programs.
 – How to use the `continue` statement to skip the rest of a `while` loop.
 – How to emulate the `do-while` loop which Python does not have.
 – How to use the `while` loop to solve an equation that cannot be solved on paper.

11.2 Why learn about the `while` loop?

Every imperative programming language has two types of loops that serve different purposes: The *for (counting) loop* is used when the number of cycles is known in advance. Such as, when one needs to parse all items in a list, or when one just needs to repeat something 10 times.

On the contrary, the *while (conditional) loop* is open-ended. Instead of a number of repetitions, it has a condition and keeps going while the condition is satisfied.

11.3 Syntax and examples

The `while` loop in Python has the following general format:

```python
while <boolean_expression>:
    <do_something>
```

There is a mandatory indent which is the same as for the `for` loop, conditions, and functions. Moreover, there is a mandatory colon : after the Boolean expression. The `while` loop can accept the same types of Boolean expressions as conditions. Let's begin with the Boolean values `True`,

```python
while True:
    print('This will be printed infinitely many times.')
```

and `False`,

```
while False:
    print('This will never be printed.')
```

Both these cases are actually useful in practise – the former in combination with the `break` statement and the later to temporarily disable code, for example for debugging purposes.

Let's do one more example, now with an inequality (Boolean expression). Did you know that when adding $1/1, 1/2, 1/3, 1/4, 1/5$ etc. one can eventually exceed any real number? Let's do this: For a given real number N we will be adding these fractions in a loop until their sum exceeds N, and we'll count how many fractions we needed:

```
N = 10
c = 0
total = 0
while total < N:
    c += 1
    total += 1 / c
print('Total =', total)
print('c =', c)
```
```
Total = 10.000043008275778
c = 12367
```

Do not choose N too high though – already with N = 20 the computer will work for several minutes.

11.4 How to time your programs

Measuring execution time of functions and programs is a bit more tricky than it appears. The reason is that there are other processes running on your computer. If you measure the wall time (time on your clock), then of course the time taken by these processes is included.

The wall time can be measured using the built-in function `time` from the `time` library. Just calling `time.time()` is not very helpful because it returns the total time since the beginning of the epoch:

```
import time
print(time.time())
```
```
1532160646.6236913
```

However, when called once at the beginning and then at the end of the program, it yields its exact duration in seconds (wall time). For illustration, let's measure the execution time of the program from Subsection 11.3:

```python
import time
N = 18
c = 0
total = 0
start = time.time()
while total < N:
    c += 1
    total += 1 / c
print('Total =', total)
print('c =', c)
print('Computation took', time.time() - start, 's.')
```

```
Total = 18.00000000371793
c = 36865412
Computation took 13.102287769317627 s.
```

If you are interested in measuring the time the CPU spends with just your function or program (and not other processes), use the function `process_time` instead:

```python
import time
N = 18
c = 0
total = 0
start = time.process_time()
while total < N:
    c += 1
    total += 1 / c
print('Total =', total)
print('c =', c)
print('Computation took', time.process_time() - start, 's.')
```

```
Total = 18.00000000371793
c = 36865412
Computation took 12.925883286 s.
```

As you can see, the process time is slightly shorter than the wall time.

191

And finally - the result of the timing will be slightly different every time your program is executed. Therefore, to have a realistic understanding of how much time your program really takes, run it multiple times.

11.5   The `break` statement

You already saw the `break` statement in Subsection 9.10 where we used it to terminate the `for` loop. It can be used to exit a `while` loop at any time too, no questions asked. If multiple loops are embedded, then it only exits the closest outer one. The `break` statement can be used, for example, as an emergency brake to prevent a `while` loop from running forever. The computation length in the previous example starts to be a problem with N around 20. Therefore, let's set the timeout to 30 seconds (wall time):

```python
import time
N = 20
c = 0
total = 0
timeout = 30
start = time.time()
while total < N:
    c += 1
    total += 1 / c
    if time.time() - start > timeout:
        print('Computation took too long, exiting.')
        break
print('Total =', total)
print('c =', c)
```
```
Computation took too long, exiting.
Total = 18.63800087724099
c = 69774921
```

11.6   The `continue` statement

The `continue` statement which you learned for the `for` loop in Subsection 9.11 works for the `while` loop analogously – it skips the rest of the current cycle and begins with the next one.

For illustration, let us write a program to generate triplets of random numbers between zero and one until all three of them are greater than 0.9. We also want to know how many attempts were needed. Clearly, if the first random number is less than or equal to 0.9, it does not make sense to generate the other two. That's when we use

192

`continue`. Then again, if the second random number is less than or equal to 0.9, it is time to use `continue`. At the end, when all three random numbers are above 0.9, we use `break` to terminate the loop. Here is the code:

```python
import random as rn
counter = 0
while True:
    counter += 1
    a1 = rn.random()
    if a1 <= 0.9:
        continue
    a2 = rn.random()
    if a2 <= 0.9:
        continue
    a3 = rn.random()
    if a3 <= 0.9:
        continue
    print("Finally the desired triplet:")
    print(a1, a2, a3)
    print("It took", counter, "attempts to get it!")
    break
```

Note that `while True` was used here to create an infinite loop. Sample output:

```
Finally the desired triplet:
0.921077479893 0.956493808495 0.917136354634
It took 1168 attempts to get it!
```

11.7   Emulating the `do-while` loop

In practice one often needs to perform some action once before repeating it in the `while` loop. Therefore some languages (C, C++, Java, ...) have the `do-while` loop. Python does not have it because it is easy to emulate – we are going to show you how.

As an example, let's say that we need to generate random numbers between zero and one as long as they are less than 0.75. When the number is greater or equal to 0.75, the loop will stop. First, here is a cumbersome way to do this using the standard `while` loop. Notice that the code for generating random numbers is there twice:

```
import random as rn
a = rn.random()
while a < 0.75:
    print(a)
    a = rn.random()
```

A popular way to circumvent this problem and only have the essential code in the loop once, is to use an infinite loop combined with a `break` statement:

```
import random as rn
while True:
    a = rn.random()
    print(a)
    if a >= 0.75:
        break
```

Sample output:

```
0.383771890647
0.068637471541
0.0850385942776
0.682442836322
0.168282761575
0.121694025066
0.185099163429
0.606740825997
0.209771333501
0.186191737708
```

11.8 The `while-else` statement

The `while` loop can be enhanced with a completion (nobreak) clause `else` in the same way as the `for` loop (see Subsection 9.12). The `else` branch is executed always, even when the `while` loop does not perform a single cycle:

```
while False:
    pass
else:
    print('While loop finished without break.')
print('Code after the while loop.')
```

```
While loop finished without break.
Code after the while loop.
```

But importantly, it will be skipped when the loop is terminated with the `break` statement:

```python
while True:
    break
else:
    print('While loop finished without break.')
print('Code after the while loop.')
```
```
Code after the while loop.
```

The `else` branch can obviously be used to acknowledge a successful completion of the `while` loop without the need to use the `break` statement. But one can also designate `break` to be the successful ending, and then the `else` branch can be used for an error message. This is illustrated on the following example where the user is asked to keep entering positive integers, until s/he enters `'x'` to finish:

```python
L = []
a = '0'
while a.isdigit():
    a = input("Enter a positive integer, or 'x' to finish:")
    if a == 'x':
        break
    if a.isdigit():
        L.append(int(a))
else:
    raise Exception("You should have entered positive integers
    or 'x'!\nSelf-destructing in 60 seconds...")
print('Thank you, you entered', L)
```
```
Exception: You should have entered positive integers or 'x'!
Self-destructing in 60 seconds...
```

11.9 Abusing the `while` loop

Codes like this have seen the light of the world too many times:

```
counter = 0
while counter < 20:
    # Do something with the counter, here
    # for simplicity we just print it:
    print(counter)
    # Increase the counter:
    counter += 1
```

Although there is nothing syntactically wrong, a person who writes such a code prob-
ably does not understand why programming languages have two types of loops. A
better code for the same would be:

```
for counter in range(20):
    # Do something with the counter, here
    # for simplicity we just print it:
    print(counter)
```

11.10   Solving an unsolvable equation

As your programming skills get stronger, you can tackle more difficult problems. In
this subsection we want to demostrate what scientific computing is all about, by solv-
ing an equation that cannot be solved "on paper". The task is to find an angle x that is
equal to its own cosine. In other words, we need to solve the equation

$$\cos(x) = x.$$

As the first step, let us visualize the graphs so that we know where approximately the
solution can be expected:

```
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(0, np.pi/2, 100)
y = np.cos(x)
z = x
plt.clf()
plt.plot(x, y, label='cos(x)')
plt.plot(x, z, label='x')
plt.legend()
plt.show()
```

The output is shown in Fig. 54.
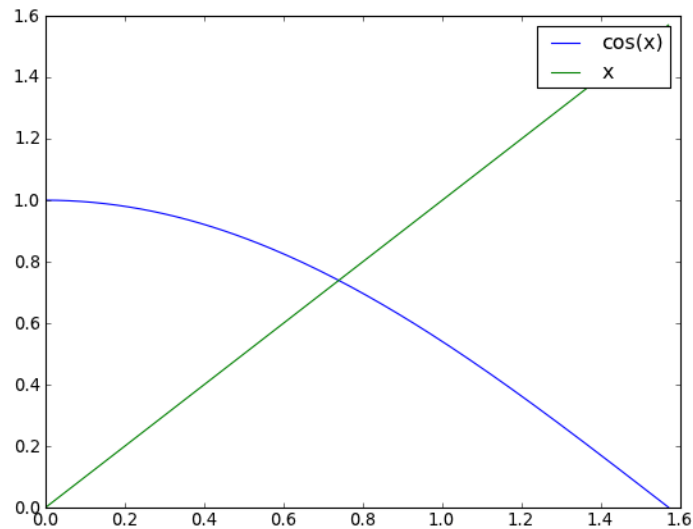
Fig. 54. Graphs of the functions $\cos(x)$ and x in the interval $(0, \pi/2)$.

The solution that we are after is the value of x where the two graphs intersect. Looking at the picture, clearly the solution is somewhere between 0.6 and 0.8. An engineer would not be extremely happy with such an answer though – we need to make this guess much more accurate.

Looking at the graphs again, we can see that in order to reach the intersection point, we can depart from zero and march with very small steps to the right while the value of $\cos(x)$ is greater than the value of x. We stop as soon as $\cos(x)$ becomes less than x. Clearly, the `while` loop needs to be used as we do not know exactly how many steps will be taken.

Let's do this. Our step will be called `dx`, and our marching point on the x-axis will be called simply `x`. We will also count steps via the variable `n`. Choosing `dx = 1e-6` will give us the result accurate to five decimal digits:

```python
import numpy as np
x = 0
dx = 1e-6
n = 0
while np.cos(x) > x:
    x += dx
    n += 1
print('Result is approximately', x)
print('Steps made:', n)
```

197

```
Result is approximately 0.739086
Steps made: 739086
```

In other words, the `while` loop ran 739086 times!

11.11   Numerical methods and scientific computing

Note that the numerical method that we used was rather naive and took too much CPU time. Scientific computing is an exciting field where researchers develop new methods that can solve problems like this more accurately and using less CPU time. The problem we just solved belongs to a wider category of *rootfinding problems*. There are much more powerful methods than the one we showed. In particular, the so-called *Newton's method* can solve it in just 5 steps (compare to 739086). One of NCLab's main objectives is to ease access to scientific computing via its Creative Suite.

12 Exceptions

12.1 Objectives

In this Section we will learn:

- Why one should write resilient software which won't fail with wrong input data.
- To be especially careful when working with user input data.
- About catching exceptions with the `try-except` statement.
- About `ZeroDivisionError`, `TypeError` and other types of exceptions.
- About the built-in function `assert` and the `AssertionError` exception.
- How to raise exceptions with the `raise` statement.
- The full form of the `try-except-else-finally` statement.

12.2 Why learn about exceptions?

Imagine that you are responsible for a software module which processes user data. The data is coming from another software module or from the user directly. Part of the data is a list of ages of a group of people, and you need to calculate their average. This is easy:

```python
def average(L):
    """
    Calculate average of the numbers in the list L.
    """
    a = 0
    for x in L:
        a += x
    return a / len(L)
```

A quick test to confirm that the function works correctly:

```python
ages = [1, 2, 6]
print(average(ages))
```
```
3.0
```

It is a mistake of novices to think that the story ends here. On the contrary, merely it begins here. You won't get commended for writing a simple function like this. But you will get a lot of complaints when your function fails. Let's show some examples. First, what if the list `ages` happens to arrive empty?

```
ages = []
print(average(ages))
```
```
Traceback (most recent call last):
  File "<string>", line 11, in <module>
  File "<nclab>", line 8, in average
ZeroDivisionError: division by zero
```

Oops, not good. Of course this is not your fault but life is not fair. Next, what if some of the ages arrive as text strings (although they shouldn't)?

```
ages = [1, '2', 6]
print(average(ages))
```
```
Traceback (most recent call last):
  File "<string>", line 1, in <module>
  File "<nclab>", line 7, in average
TypeError: unsupported operand type(s) for +=: 'int' and 'str'
```

Sometimes, unbelievable things can happen in computer programming – more precisely, they sound unbelievable until you get used to them. Such as, a colleague or a user who is supposed to provide a list can send you something which is not a list at all. Let's see what happens if your function receives a dictionary. After all, these should be the ages of people, right?

```
ages = {'Peter':23, 'Paul':16}
print(average(ages))
```
```
Traceback (most recent call last):
  File "<string>", line 1, in <module>
  File "<nclab>", line 4, in average
TypeError: unsupported operand type(s) for +=: 'int' and 'str'
```

Or, somebody will provide an integer instead of a list by mistake:

```
ages = 1
print(average(ages))
```
```
Traceback (most recent call last):
  File "<string>", line 1, in <module>
  File "<nclab>", line 3, in average
TypeError: 'int' object is not iterable
```

Guess who will be blamed if a customer decides against buying the software because of a few crashes which occurred in a function you wrote. Nobody will ask what caused them. There is a famous saying in computer science which has a lot of truth to it:

"A good programmer is someone who looks both ways before crossing a one-way street." (Doug Linder)

On the other hand, the good news is that Python makes it easy to fortify your software (and protect yourself) by catching and handling exceptions without letting the program crash. That's what we want to show you in this section.

12.3 Catching exceptions – statement `try-except`

Let's stay with the previous example for a while. One could try to anticipate all things that might go wrong, but the users are always one step ahead in their creativity. Therefore, Python provides *exceptions* whose philosophy is the opposite – instead of worrying in advance, let the code run and only take a corrective action if something goes wrong. We have seen in Subsection 12.2 that `ZeroDivisionError` and `TypeError` can be expected. Therefore, let's fortify the function `average` against them:

```python
def average(L):
    """
    Calculate average of the numbers in the list L.
    Return False if the calculation fails.
    """
    try:
        a = 0
        for x in L:
            a+=x
        return a / len(L)
    except ZeroDivisionError:
        print('Warning: ZeroDivisionError in average().')
        print('Returning False.')
        return False
    except TypeError:
        print('Warning: TypeError in average().')
        print('Returning False.')
        return False
```

In the above program, the original code is in the `try` branch. This branch can be followed by as many `except` branches as you wish, that will handle different sorts of problems. Of course none of them will be executed if the `try` branch passes without generating an exception. We will give a list of the most frequently used exceptions in the following subsection.

It is needless to say that the function has improved. It works as before when the input data is correct:

```
ages = [1, 2, 6]
print(average(ages))
3.0
```

But with missing input data it does not crash anymore! Instead, it issues a warning and returns `False`:

```
ages = []
print(average(ages))
Warning: ZeroDivisionError occured in average().
Returning False.
False
```

The same if the data contain items of a wrong type:

```
ages = [1, '2', 6]
print(average(ages))
Warning: TypeError occured in average().
Returning False.
False
```

Next let's look at various other types of exceptions that one can catch.

12.4 List of common exceptions

Here is a list of the most frequently used exceptions for reference:

- `Exception`: Base class for all exceptions. It can be used to catch a general exception.This will be discussed in Subsection 12.5.
- `StopIteration`: Raised when the `next` method of an iterator does not point to any object. For iterators see Subsection 9.13.

- `StandardError` Base class for all built-in exceptions except `StopIteration` and `SystemExit`.
- `ArithmeticError` Base class for all errors that occur for numeric calculation.
- `OverflowError` Raised when a calculation exceeds maximum limit for a numeric type.
- `FloatingPointError` Raised when a floating point calculation fails.
- `ZeroDivisionError` Raised when division or modulo by zero takes place for all numeric types.
- `AssertionError` Raised in case of failure of the `assert` statement. This will be discussed in Subsection 12.6.
- `AttributeError` Raised in case of failure of attribute reference or assignment.
- `EOFError` Raised when there is no input from either the `raw_input` or `input` function and the end of file is reached.
- `ImportError` Raised when an import statement fails.
- `KeyboardInterrupt` Raised when the user interrupts program execution, usually by pressing CTRL+C.
- `LookupError` Base class for all lookup errors.
- `IndexError` Raised when an index is not found in a sequence.
- `KeyError` Raised when the specified key is not found in the dictionary.
- `NameError` Raised when an identifier is not found in the local or global namespace.
- `UnboundLocalError` Raised when trying to access a local variable in a function or method but no value has been assigned to it.
- `EnvironmentError` Base class for all exceptions that occur outside the Python environment.
- `IOError` Raised when an input/output operation fails, such as the print statement or the `open` function when trying to open a file that does not exist.
- `OSError` Raised for operating system-related errors.
- `SyntaxError` Raised when there is an error in Python syntax.
- `IndentationError` Raised when indentation is not specified properly.
- `SystemError` Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.
- `SystemExit` Raised when Python interpreter is quit by using the `sys.exit` function. If not handled in the code, causes the interpreter to exit.
- `TypeError` Raised when an operation or function is attempted that is invalid for the specified data type.
- `ValueError` Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
- `RuntimeError` Raised when a generated error does not fall into any category.
- `NotImplementedError` Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.

12.5 Catching a general exception

If one does not need to perform different corrective actions based on different types of exceptions, it is often enough to only catch the general exception:

```python
def average(L):
    """
    Calculate average of the numbers in the list L.
    Return False if the calculation fails.
    """
    try:
        a = 0
        for x in L:
            a+=x
        return a / len(L)
    except Exception as e:
        print('Warning in average(): ' + str(e))
        print('Returning False.')
        return False
```

When called with missing input data, the function now responds as follows:

```python
ages = []
print(average(ages))
Warning in average(): division by zero
Returning False.
False
```

And when the data is wrong:

```python
ages = [1, '2', 6]
print(average(ages))
Warning in average(): unsupported operand type(s) for +=:
'int' and 'str'
Returning False.
False
```

12.6 Function `assert`

The built-in function `assert` raises an exception when the Boolean statement used as its argument is `False`. This is a quick way to make sure that the program does not

continue with incorrect data:

```python
def myfunction(x):
    """
    Returns 1/x. If x == 0, throws AssertionError.
    """
    assert(x != 0)
    return 1/x

print(myfunction(0))
```

```
Traceback (most recent call last):
  File "<string>", line 8, in <module>
  File "<nclab>", line 5, in myfunction
AssertionError
```

Alternatively, the AssertionError exception can be caught using the try command as usual:

```python
import numpy as np
def myfunction(x):
    """
    Returns 1/x. If x == 0, returns np.nan.
    """
    try:
        assert(x != 0)
        return 1/x
    except AssertionError:
        print('Warning in myfunction(): division by zero')
        print('Returning nan.')
        return np.nan

print(myfunction(0))
```

```
Warning in myfunction(): division by zero
Returning nan.
nan
```

205

12.7 Throwing exceptions – statement `raise`

Python does not have a command to stop program execution. If you need to terminate your program prematurely, for example because of wrong input data, the best thing is to raise an exception using the `raise` statement.

Let's illustrate this on a function which is undefined for $x = 3$ and $x = -3$:

```python
def f(x):
    """
    Returns 1/(x**2 - 9).
    If x == 3 or x == -3, raises ValueError.
    """
    if x == 3 or x == -3:
        raise ValueError('Division by zero detected.')
    return 1 / (x**2 - 9)
```

When the function `f(x)` is called with 3 or -3, the output will be

```
Traceback (most recent call last):
  File "<string>", line 9, in <module>
  File "<nclab>", line 6, in myfunction
ValueError: Division by zero detected.
```

12.8 The full statement `try-except-else-finally`

You already know from Subsection 12.3 that the `try-except` statement can contain multiple `except` branches. Moreover, it can contain optional branches `else` and/or `finally`. Your understanding of the `else` branch in `for` loops (Subsection 9.12) and in `while` loops (Subsection 11.8) will make this very simple.

The `else` branch: The code is executed only if no exceptions were raised in the `try` branch. This is analogous to `for-else` and `while-else` where the `else` branch is executed only if no `break` was used. Code executed in this branch is just like normal code: if there is an exception, it will not be automatically caught (and probably stop the program).

The `finally` branch: The code always executes after the other blocks, even if there was an uncaught exception (that didnt cause a crash) or a `return` statement in one of the other branches. Code executed in this branch is just like normal code: if there is an exception, it will not be automatically caught.

13 File Operations

13.1 Objectives

In this Section we will learn

- About the NCLab file system and the security of your data in NCLab.
- How to open files via the built-in function `open` and the `with` statement.
- Various modes to open files (for reading, writing, appending, etc.).
- About Latin-1 and UTF-8 encoding of text files.
- How to parse files one line at a time, and other ways to read data from a file.
- To consider file size and the amount of available runtime memory.
- How to use the file pointer and work on the character level.
- About various ways to write data to a file.
- That things can go wrong, and one should use exceptions when working with files.

In this section we will mostly work with text files. Binary files will be discussed later in Subsections 13.30 – 13.33.

13.2 Why learn about files?

So far you learned about using various programming techniques to process data in the form of text strings, variables, lists, dictionaries, etc. But all such data, in one way or another, comes from files. Whether stored on the hard disk of your computer or on a cloud server, files are an indivisible part of computer programming, and every software developer must be fluent in working with them.

13.3 NCLab file system and security of your data

The file manager *My Files* whose icon is present on your NCLab Desktop gives you access to your own NCLab file system which works similarly to the Windows, Linux and other file systems. You can create files and folders, rename them, move them, and delete them. You can also upload files from your hard disk, flash drive, or from the web. Thanks to a generous quota you should never be worried about running out of space. On the backend, your files are stored in a Mongo distributed database system at Digital Ocean, and all data is backed up every day. The communication between your browser and the cloud servers is encrypted for maximum security.

207

13.4 Creating sample text file

Depending on the PDF viewer you are using to read this text, you should be able to select the block below with your mouse, and use CTRL+C to copy it to clipboard:

```
              #             #
              #             #
########  ########  #     ########  ########
#      #  #         #            #  #      #
#      #  #         #     ########  #      #
#      #  #         #     #      #  #      #
#      #  ########  ###   ########  ########
```

Next, go to the NCLab Desktop, open My Files, and create a new folder named "sand-box". We will be using it throughout this section. Then, open the Creative Suite, and under Accessories launch Notepad. Paste the text there via CTRL+V, and you should see something like this:
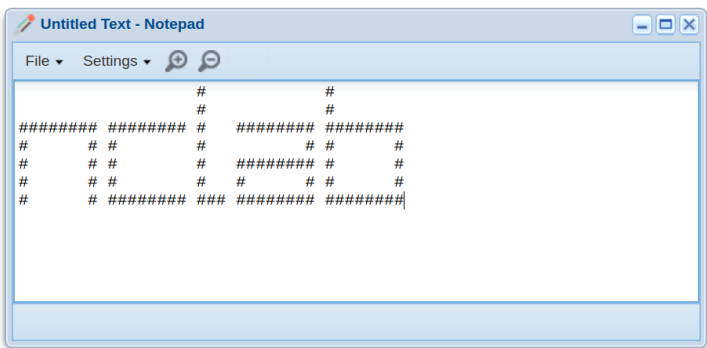
Fig. 55. Notepad is a simple text editor in NCLab.

Then use "Save in NCLab" in the Notepad's File menu to save the file under the name "logo" to your newly created folder "sandbox". Notepad will automatically add an extension ".txt" to it (which will not be displayed by the file manager – like in Windows). The text "Untitled Text" in the top part of the Notepad window will change to "logo":
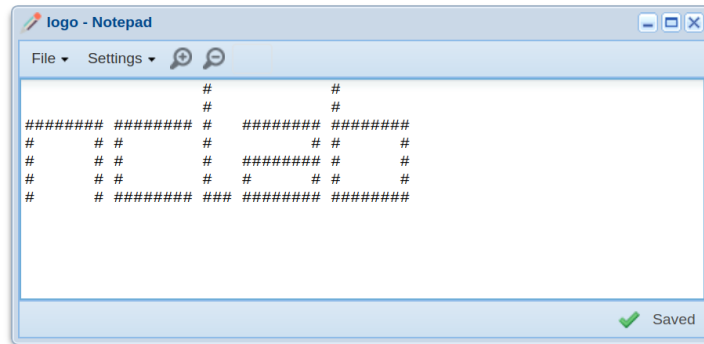
Fig. 56. Your sample text file "logo.txt" is ready to use.

13.5   Creating sample Python file

So far it did not matter where your Python code was located, and it was even OK to run any program in unsaved mode in the Python app. This is different when working with files. The reason is that when opening a file, the interpreter will look for it in the current folder where your Python source code is saved.

Therefore, let's go to the Creative Suite, launch a new Python worksheet, erase the demo code, and type there a single line `f = open('logo.txt', mode='r')` Then use "Save in NCLab" in the File menu to save the file under the name "test" to the folder "sandbox" where you already have your file "logo.txt". The Python app will automatically add an extension ".py" to it which will not be displayed by the file manager. This is how it should look like:

Fig. 57. Sample file "test.py".

209

13.6 Opening a text file for reading – method `open`

In the previous subsection you already met the built-in function `open` which can be used to open files:

```
f = open('logo.txt', mode='r')
```

The parameter name `mode` can be left out for brevity, so usually you will see the function `open` used as follows:

```
f = open('logo.txt', 'r')
```

The first argument `'logo.txt'` is the file name. The interpreter will look for it in the directory where your Python source code is located. Using subfolders such as `'data/logo.txt'` is OK as well (assuming that there is a subfolder `'data'` in your currect directory which contains the file `'logo.txt'`).

The next argument `'r'` means "for reading in text mode". Various other modes to open a file will be mentioned in Subsection 13.13.

The function `open` has an optional third parameter `encoding` whose default value is `'utf-8'`. We will talk about encoding in Subsection 13.11.

Finally, the variable `f` will store a pointer to the file contents. As everything else in Python, `file` is a class. In the following subsections we will explain various methods of this class which can be used to work with data in files.

13.7 Closing a file – method `close`

The `file` class has a method `close` to close the file. An open file named `f` can be closed by typing

```
f.close()
```

Nothing happens when one tries to close a file which is not open.

13.8 Checking whether a file is closed

To check whether a file `f` is closed, one can use the attribute `closed` of the `file` class:

```
f.closed
```

which is either `True` or `False`.

13.9 Using the `with` statement to open files

The preferred way to open files is using the `with` statement:

```python
with open('logo.txt', 'r') as f:
```

The body of the `with` statement is indented analogously to loops and conditions. There is no need to use the `close` method – the file will be closed automatically, even if any exceptions are raised. This is a good alternative to `try-finally` blocks which were introduced in Subsection 12.8. More details on the usage of the `with` statement can be found in the PEP 343 section of the official Python documentation at `https://www.python.org/dev/peps/pep-0343/`.

13.10 Getting the file name

The `file` class has an attribute `name` which can be used to obtain the filename:

```python
with open('logo.txt', 'r') as f:
    print(f.name)
```
```
logo.txt
```

13.11 A word about encoding, Latin-1, and UTF-8

In Subsections 4.26 – 4.28 we discussed the ASCII table which translates Latin text characters and other widely used text symbols into decimal integer codes and vice versa. The decimal integer codes are then translated into the corresponding binary numbers (zeros and ones) before they are saved to the disk. For example, the ASCII text string

```
Hello!
```

becomes

```
010010000110010101101100011011000110111100100001
```

Every 8-bit (1-Byte) segment in this sequence represents one ASCII character. For example, `01001000` is `'H'` etc.

The process of converting text into a binary sequence is called *encoding*. It only applies to text files, not to binary files. The above example, up to some technical details, describes Latin-1 (ISO-8859-1) encoding. This was the standard encoding for Western European languages until the advent of Unicode (UTF-8) encoding in 1991.

Unicode – which for simplicity can be understood as "extended ASCII" – was created in order to accommodate accents (ñ, ü, ř, ç, ...), Japanese, Chinese and Arabic symbols, and various special characters coming from other languages. While the original 8-bit Latin-1 (ISO-8859-1) only can encode 256 characters, Unicode (UTF-8) can encode 1,114,112. It uses a variable number of bits for various characters, trying to be as economical as possible. The exact scheme is beyond the scope of this textbook but you can find it on the web.

Importantly, UTF-8 is backward compatible with Latin-1 (ISO-8859-1) in the sense that UTF-8 codes of ASCII characters are the same as their Latin-1 (ISO-8859-1) codes. In other words, it does not matter whether UTF-8 or Latin-1 (ISO-8859-1) encoding is used for ASCII text.

Importantly, UTF-8 encoding is the standard in Python 3, so you don't have to worry about using language-specific non-ASCII characters in your text strings. We will stop here for now, but feel free to learn more about encoding in the official Python documentation online.

13.12   Checking text encoding

Class `file` has an attribute `encoding` which can be used to obtain information about text encoding:

```
with open('logo.txt', 'r') as f:
    print(f.encoding)
utf-8
```

The `file` class has a number of other useful attributes besides `name` and `encoding`. Feel free to learn more at `https://docs.python.org/3/library/index.html`.

13.13   Other modes to open a file

For reference, here is an overview of various modes to open a file in Python:

- `'r'`: This is the default mode. It opens the file for reading as text (in UTF-8 encoding). Starts reading the file from the beginning. If the file is not found, `FileNotFoundError` exception is raised.
- `'rb'`: Like `'r'` but opens the file for reading in binary mode.
- `'r+'`: Opens the file for reading and writing. File pointer is placed at the beginning of the file (the file pointer will be discussed in Subsection 13.23).
- `'w'`: This mode opens the file for writing text. If the file does not exist, it creates a new file. If the file exists, it truncates the file (erases all contents).

212

- – 'wb': Like 'w' but opens the file for writing in binary mode.
- – 'w+': Like 'w' but also allows to read from file.
- – 'wb+': Like 'wb' but also allows to read from file.
- – 'a': Opens file for appending. Starts writing at the end of file. If the file does not exist, creates a new file.
- – 'ab': Like 'a' but in binary format.
- – 'a+': Like 'a' but also allows to read from file.
- – 'ab+': Like 'ab' but also allows to read from file.
- – 'x': Creates a new file. If the file already exists, the operation fails.

13.14 Reading the file line by line using the `for` loop

Great, now let's do something with our sample file "logo.txt"!

To begin with, we will just display its contents. Python has a convenient way of reading an open text file one line at a time via the `for` loop. Adjust your sample Python file "test.py" as follows:

```python
with open('logo.txt', 'r') as f:
    for line in f:
        print(line)
```

Here `line` is not a keyword – it is a name for the text string representing the next line in the file `f`, and we could use some other name if we wanted.

But when you run the code, you will see this strange output:

```
                #               #

                #               #

########  ########  #     ########  ########

#         #  #      #             #  #         #

#         #  #      #     ########  #         #

#         #  #      #     #         #  #         #

#         #  ########  ###  ########  ########
```

213

Well, that definitely does not look right! We will look into this mystery in the following subsection. But first let's make sure that indeed every line extracted from the file f is a text string:

```python
with open('logo.txt', 'r') as f:
    for line in f:
        print(type(line))
```
```
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
```

13.15   The mystery of empty lines

To solve the mystery of empty lines from the previous subsection, let's use the built-in function repr that you know from Subsection 4.10:

```python
with open('logo.txt', 'r') as f:
    for line in f:
        print(repr(line))
```
```
'                  #                #\n'
'                  #                #\n'
'######## ######## #    ######## ########\n'
'#      # #        #            # #      #\n'
'#      # #        #    ######## #      #\n'
'#      # #        #    #      # #      #\n'
'#      # ######## ### ######## ########'
```

Oh, now this makes more sense! Each line extracted from the file (with the exception of the last one) contains the newline character \n at the end, and the print function adds one more by default. As a result, the newline characters are doubled, which causes the empty lines to appear.

This problem can be cured in two different ways. First, we can prevent the print function from adding the newline character \n after each line:

214

```
with open('logo.txt', 'r') as f:
    for line in f:
        print(line, end='')
```

```
                 #              #
                 #              #
######## ######## #    ######## ########
#      # #        #           # #       #
#      # #        # ######## #        #
#      # #        # #      # #        #
#      # ######## ### ######## ########
```

But this does not remove the newline characters from the text strings, which might cause other problems. So let's remove them using the string method `rstrip` from Subsection 4.11:

```
with open('logo.txt', 'r') as f:
    for line in f:
        line = line.rstrip()
        print(line)
```

```
                 #              #
                 #              #
######## ######## #    ######## ########
#      # #        #           # #       #
#      # #        # ######## #        #
#      # #        # #      # #        #
#      # ######## ### ######## ########
```

Note that we cannot use the string method `strip` here because some of the lines contain empty spaces on the left which we need to remain there.

13.16   Reading individual lines – method `readline`

The `file` class has a method `readline` which only reads one line from the file. This makes reading from files more flexible. For example, one might want to only read the first three lines:

215

```python
with open('logo.txt', 'r') as f:
    for i in range(3):
        line = f.readline()
        line = line.rstrip()
        print(line)
```

```
                #               #
                #               #
######## ######## #     ######## ########
```

Or, the file might have a two-line header that needs to be skipped:

```
% Created by Some Awesome Software
% Date: January 1, 1111
                #               #
                #               #
######## ######## #     ######## ########
#        # #      #            # #        #
#        # #      #     ######## #        #
#        # #      #     #      # #        #
#        # ######## ### ######## ########
```

To skip the header, one just needs to call `readline` two times before using the `for` loop:

```python
with open('logo.txt', 'r') as f:
    dummy = f.readline()
    dummy = f.readline()
    for line in f:
        line = line.rstrip()
        print(line)
```

```
                #               #
                #               #
######## ######## #     ######## ########
#        # #      #            # #        #
#        # #      #     ######## #        #
#        # #      #     #      # #        #
#        # ######## ### ######## ########
```

216

When the end of file is reached and there is no next line to read, `readline` returns an empty string:

```python
with open('logo.txt', 'r') as f:
    for line in f:
        pass
    line = f.readline()
    print(line)
```
```

```

Finally let's mention that one can call the method as `f.readlines(n)` where `n` is the number of bytes (characters) to be read from the file `f`. In other words, it is possible to parse the file `f`, reading just the first `n` characters from each line.

13.17   Reading the file line by line using the `while` loop

The method `readline` from the previous subsection can be combined with the `while` loop as well:

```python
with open('logo.txt', 'r') as f:
    while True:
        line = f.readline()
        if not line:
            break
        line = line.rstrip()
        print(line)
```
```
              #             #
              #             #
########  ########  #    ########  ########
#       # #        #             # #        #
#       # #        #    ######## #        #
#       # #        #    #      # #        #
#       # ######## ### ######## ########
```

Notice how `if not line` was used to detect that end of file was reached. This works because when the end of file is reached, `line` is an empty string, and `if not` applied to an empty string yields `True` (see Subsection 10.6).

13.18 Reading the file using `next`

The `file` class is iterable. Therefore it has the method `__next__` and it can also be parsed with the built-in function `next` (see Subsections 9.13 - 9.16). Both can be used analogously to `readline` to access individual lines or parse the file line-by-line. This time we will enumerate the lines in the file "logo.txt".

First let's do this using the method `__next__`:

```python
with open('logo.txt', 'r') as f:
    for i in range(7):
        print('Line', str(i+1) + ':  ' + f.__next__().rstrip())
```

```
Line 1:                     #              #
Line 2:                     #              #
Line 3:   ######## ######## #   ######## ########
Line 4:   #      # # #      #        # #       #
Line 5:   #      # # #      # ######## #       #
Line 6:   #      # # #      # #       # #      #
Line 7:   #        # ######## ### ######## ########
```

Alternatively, one can use `next`:

```python
with open('logo.txt', 'r') as f:
    for i in range(7):
        print('Line', str(i+1) + ':  ' + next(f).rstrip())
```

```
Line 1:                     #              #
Line 2:                     #              #
Line 3:   ######## ######## #   ######## ########
Line 4:   #      # # #      #        # #       #
Line 5:   #      # # #      # ######## #       #
Line 6:   #      # # #      # #       # #      #
Line 7:   #        # ######## ### ######## ########
```

13.19 Reading the file as a list of lines – method `readlines`

The `file` class has a method `readlines` which makes it possible to read the entire file at once, and returns a list of lines (as text strings):

```python
with open('logo.txt', 'r') as f:
    L = f.readlines()
    print(L)
```

```
[’                     #                  #\n’,
 ’                     #                  #\n’,
 ’########  ########  #    ########  ########\n’,
 ’#         #  #       #             #  #        #\n’,
 ’#         #  #       #    ########  #         #\n’,
 ’#         #  #       #    #         #  #        #\n’,
 ’#         #  ########  ###  ########  ########’]
```

And there is an even shorter way to do this:

```python
with open('logo.txt', 'r') as f:
    L = list(f)
    print(L)
```

```
[’                     #                  #\n’,
 ’                     #                  #\n’,
 ’########  ########  #    ########  ########\n’,
 ’#         #  #       #             #  #        #\n’,
 ’#         #  #       #    ########  #         #\n’,
 ’#         #  #       #    #         #  #        #\n’,
 ’#         #  ########  ###  ########  ########’]
```

As you can see, in neither case were the text strings cleaned from the newline characters at the end. But there is something more important to say about this approach, which we will do in the following subsection.

13.20   File size vs available memory considerations

By RAM (Random-Access Memory) we mean runtime memory, or in other words the memory where the computer stores data while running programs. As opposed to hard disk space which is cheap, RAM is expensive, and therefore standard computers do not have much. Usually, as of 2018, standard desktop computers or laptops can have anywhere between 1 GB and 8 GB of RAM.

The file method `readlines` from the previous subsection will paste the entire contents of the file into the RAM, without worrying about the file size. So, one has to be careful here. If the size of the file is several GB, then there is a real possibility of running out of memory. Therefore, always be aware of the file size when working with files, and only use methods such as `readlines` when you are sure that the file is small.

Importantly, reading files with the `for` loop does not have this problem and it is safe even for very large files.

13.21 Reading the file as a single text string – method `read`

The `file` class has a method `read` which returns the entire contents of the text file as a single text string:

```python
with open('logo.txt', 'r') as f:
    txt = f.read()
    print(txt)
```

```
              #              #
              #              #
########  ########  #      ########  ########
#       #  #        #             #  #         #
#       #  #        #      ########  #         #
#       #  #        #      #      #  #         #
#       #  ########  ###  ########  ########
```

Since the entire file is pasted into the RAM at once, this method obviously suffers from the same problems with large files as the method `readlines`. The method `read` will read the file from the current position of the file pointer until the end of the file. The method `read` accepts an optional integer argument which is the number of characters to be read. An example will be shown in Subsection 13.23.

13.22 Rewinding a file

Sometimes one needs to go twice through the contents of a file. To illustrate this, let's create a new text file "ages.txt" in the folder "sandbox":
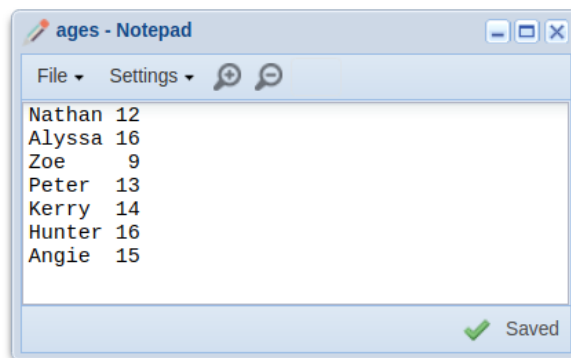
Fig. 58. Text file "ages.txt" containing names and ages of people.

This sample file is very small, and therefore it can be pasted to RAM. But imagine a real-life scenario where the file size is several GB, so you only can read it one line at a time. Your task is to find the person (or multiple persons) with the highest age, and return their name(s) as a list.

The easiest solution has three steps:

1. Pass through the file f once to figure out the highest age ha.
2. Rewind the file f. This can be done by calling f.seek(0).
3. Make a second pass and collect all people whose age matches ha.

Typing f.seek(0) will move the file pointer to the initial position at the beginning of the file f. We will talk about the file pointer and this method in more detail in the next subsection. And here is the corresponding code:

```python
with open('ages.txt', 'r') as f:
    # First get the highest age 'ha':
    ha = 0
    for line in f:
        line = line.rstrip()
        name, age = line.split()
        age = int(age)
        if age > ha:
            ha = age
    # Rewind the file:
    f.seek(0)
    # Next collect all people whose age matches 'ha':
    L = []
    for line in f:
        line = line.rstrip()
        name, age = line.split()
        age = int(age)
        if age == ha:
            L.append(name)

print('Highest age is', ha)
print('People:', L)
```
```
Highest age is 16
People: ['Alyssa', 'Hunter']
```

13.23   File pointer, and methods `read`, `tell` and `seek`

On the most basic level, a text file is a sequence of characters, and in some cases one needs to work with this sequence on a character-by-character basis. The methods `seek`, `read` and `tell` serve this purpose. But before we show how they are used, let's introduce the *file pointer*.

File pointer

The file pointer is an integer number which corresponds to the current position in the text file. When the file is opened for reading, the pointer is automatically set to 0. Then it increases by one with each new character read (or skipped).

Method `read`

From Subsection 13.21 you know that the method `read` accepts an optional integer argument which is the number of characters to be read. So, after calling `f.read(6)` the pointer will increase to 6. After calling once more `f.read(3)`, it will increase to 12 etc.

Method `tell`

The position of the pointer in an open file `f` can be obtained by typing `f.tell()` (some exceptions apply when the file is used as an iterator – see Subsection 13.24). In the following example, we will read the first line from the file "ages.txt" from Subsection 13.23,

```
Nathan 12
```

and look at the positions of the file pointer. More precisely, we will open the file and read the first 6 characters, then 3 more, and finally the newline character \n at the end of the line:

```
with open('ages.txt', 'r') as f:
    pos = f.tell()
    print(pos)
    name = f.read(6)
    print(repr(name))
    pos = f.tell()
    print(pos)
    age = f.read(3)
    print(repr(age))
    pos = f.tell()
    print(pos)
    c = f.read(1)
    print(repr(c))
    pos = f.tell()
    print(pos)
```

```
0
'Nathan'
6
' 12'
9
'\n'
10
```

Method seek

You already know the method seek from Subsection 13.22. More precisely, you know that f.seek(0) will reset the position of the file pointer to 0, which effectively rewinds the file to the beginning. But this method has more uses:

- Typing f.seek(n) which is the same as f.seek(n, 0) will set the pointer in the file f to position n (counted from the beginning of the file).
- Typing f.seek(n, 1) will move the pointer forward by n positions (counted from the current position). This number can be negative, moving the file pointer backward.
- Typing f.seek(n, 2) will set the pointer to position n (counted from the end of the file backward).

For illustration, let's improve the previous example and use f.seek(1, 1) to skip the empty character between the words "Nathan" and "12":

223

```python
with open('ages.txt', 'r') as f:
    pos = f.tell()
    print(pos)
    name = f.read(6)
    print(repr(name))
    pos = f.tell()
    print(pos)
    f.seek(1, 1)
    age = f.read(2)
    print(repr(age))
    pos = f.tell()
    print(pos)
    c = f.read(1)
    print(repr(c))
    pos = f.tell()
    print(pos)
```

```
0
'Nathan'
6
'12'
9
'\n'
10
```

As a last example, let's illustrate using `seek` with a negative offset to move the file pointer backward. We will read the first 9 characters in the file "ages.txt", then move the pointer back by 2 positions, and read two characters again:

```python
with open('ages.txt', 'r') as f:
    line = f.read(9)
    print(repr(line))
    f.seek(-2, 1)
    age = f.read(2)
    print(repr(age))
```

```
'Nathan 12'
'12'
```

This is enough for the moment, and we will return to the method `seek` again in Subsection 13.28 in the context of writing to files.

13.24 Using `tell` while iterating through a file

The method `tell` is disabled when the file is used as an iterator (parsed using a `for` loop, using the method `__next__`, or using the built-in function `next`). Here is an example where `tell` behaves in an unexpected way because of this:

```
with open('ages.txt', 'r') as f:
    for line in f:
        print(f.tell())
```
```
69
69
69
69
69
69
69
```

The reason is that a read-ahead buffer is used to increase efficiency. As a result, the file pointer advances in large steps across the file as one iterates over lines.

One may try to circumvent this problem by avoiding the `for` loop, but unfortunately also using `readline` disables `tell`:

```
with open('ages.txt', 'r') as f:
    while True:
        print(f.tell())
        line = f.readline()
        if not line:
            break
```
```
0
69
69
69
69
69
69
69
```

In short, trying to combine `tell` with optimized higher-level file operations is pretty much helpless. If you need to solve a task like this, the easiest way out is to define your

own function `readline` which is less efficient than the built-in version but does not disable `tell`:

```python
def myreadline(f):
    """
    Analogous to readline but does not disable tell.
    """
    line = ''
    while True:
        c = f.read(1)
        if c == '\n' or c == '':
            return line
        line += c

with open('ages.txt', 'r') as f:
    while True:
        print(f.tell())
        line = myreadline(f)
        if not line:
            break
        print(line)
```

```
0
Nathan 12
10
Alyssa 16
20
Zoe     9
30
Peter   13
40
Kerry   14
50
Hunter 16
60
Angie   15
69
```

13.25 Another sample task for `tell`

Imagine that your next task is to look for a particular word (name, number, ...) in the file, and return the list of all lines where the word is present. But the result should not be a list of text strings – instead, it should be a list of pointer positions which correspond to the beginning of each line. To be concrete, let's say that in the file "ages.txt" we must find the beginnings of all lines which contain the character '1'.

The best way to solve this task is to remember the position of the pointer before reading each line. But as you know from the previous subsection, this will be tricky because you will not be able to use the built-in method `readline`. Again, the best solution is to use your own method `readlines` (which was defined in the previous subsection). Here is the main program:

```python
s = '1'
with open('ages.txt', 'r') as f:
    while True:
        n = f.tell()
        line = myreadline(f)
        if not line:
            break
        if s in line:
            print(n)
```

```
0
Nathan 12
10
Alyssa 16
30
Peter  13
40
Kerry  14
50
Hunter 16
60
Angie  15
```

13.26 Writing text to a file – method `write`

In Subsection 13.13 you have seen various modes which can be used to open a file for reading, writing or appending. Mode 'w' will create a new text file if a file with the

given name does not exist. If the file exists, it will be opened and truncated (file pointer set to the beginning).

When writing to a file, it matters where the Python source file is located, because the file will be created in the current directory (this is the same as when opening a file for reading – see Subsection 13.5). Of course one can use an absolute path but in our case this is not needed.

Let's replace the code in the file "test.py" in folder "sandbox" with

```python
with open('greeting.txt', 'w') as f:
    f.write('Hello!')
```

After running the program, a new file named "greeting.txt" will appear in that folder:
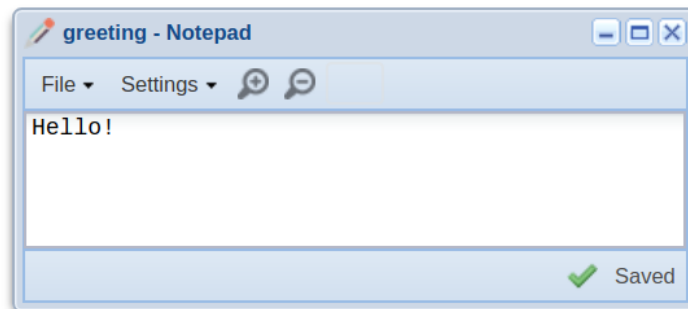
Fig. 59. Newly created file "greeting.txt".

Note that the `write` method does not add the newline character `\n` at the end of the text string. The code

```python
with open('greeting.txt', 'w') as f:
    f.write('Hello!')
    f.write('Hi there!')
    f.write('How are you?')
```

will result into a one line text file:

228
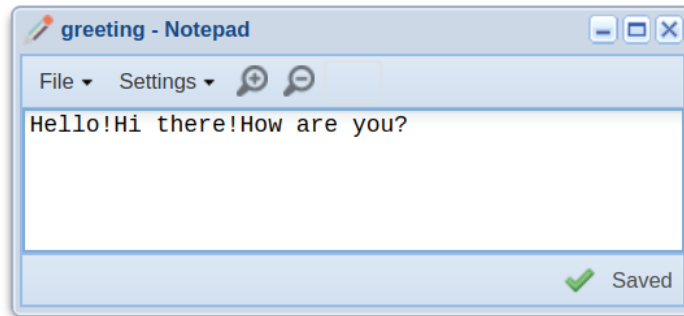
Fig. 60. The `write` method does not add the newline character at the end of the text string.

If we want to have multiple lines, we have to insert newline characters manually:

```python
with open('greeting.txt', 'w') as f:
    f.write('Hello!\n')
    f.write('Hi there!\n')
    f.write('How are you?')
```

The result is now what one would expect:
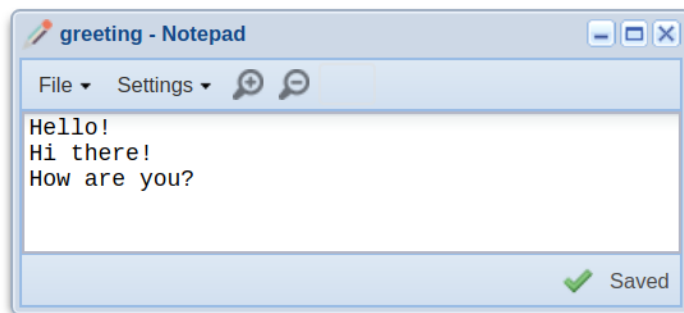
Fig. 61. Newline characters must be inserted in the text strings manually.

13.27 Writing a list of lines to a file – method `writelines`

Staying with the previous example, the three lines can be written into the file elegantly using a list and the method `writelines`. This method writes a list of text strings into the given file:

```python
L = ['Hello!', 'Hi there!', 'How are you?']
with open('greeting.txt', 'w') as f:
    f.writelines(L)
```

A bit disappointingly, newline characters are still not added at the end of the lines:
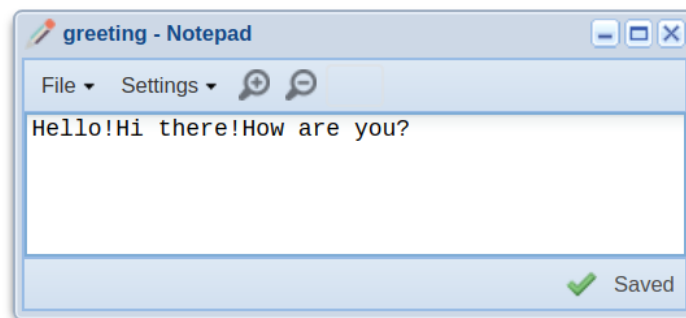
Fig. 62. Newline characters are not added automatically.

But after adding the newline characters manually,

```python
L = ['Hello!\n', 'Hi there!\n', 'How are you?']
with open('greeting.txt', 'w') as f:
    f.writelines(L)
```

one obtains the desired result:
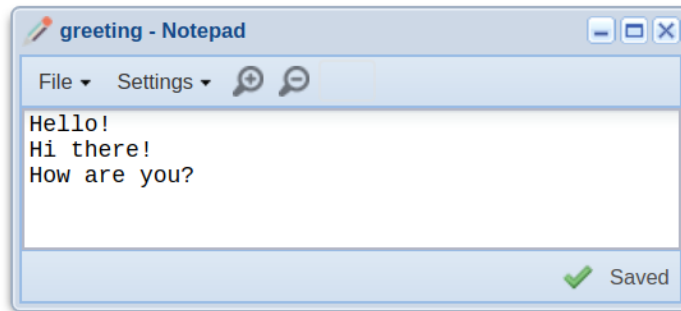
Fig. 63. Adding newline characters produces the desired result.

13.28 Writing to the middle of a file

In Subsection 13.23 we used the method `seek` to read from various parts of the file. But what happens if we want to write to some place in the middle of an open file? Will the new text be inserted or overwrite the text that is already there? We will show what will happen. Let's begin with introducing a sample file "news.txt":
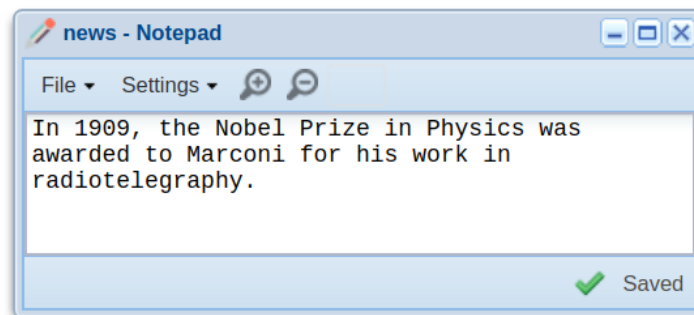
Fig. 64. Sample file "news.txt".

The name "Marconi" starts at position 51. Let's open the file in the `r+` mode, move the pointer to this position, and write there Marconi's first name, "Guglielmo ":

```python
with open('news.txt', 'r+') as f:
    f.seek(51, 0)
    f.write('Guglielmo ')
```

231

However, when the code is run, the contents of the file "news.txt" starting at position 51 is overwritten with the new text:
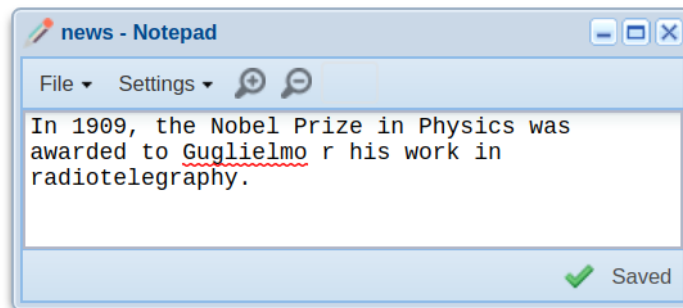
Fig. 65. Text beginning at position 51 was overwritten with the new text.

So this did not work. As a matter of fact, the text is a sequence of zeros and ones on the disk. It behaves like a sentence written with a pencil on paper. You can erase and overwrite part of it, but inserting new text in the middle is not possible. We will have to read the whole file as a text string, slice it, insert the first name in the middle, and then write the new text string back to the original file:

```python
with open('news.txt', 'r') as f:
    txt = f.read()
with open('news.txt', 'w') as f:
    f.write(txt[:51] + 'Guglielmo ' + txt[51:])
```

Finally, the desired result:
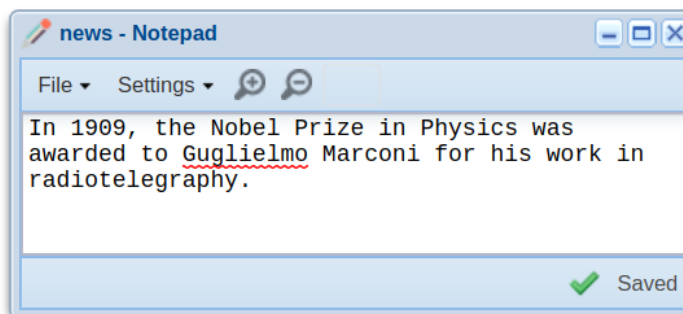
Fig. 66. Inserting new text required reading the whole file as a text string, slicing it, and then writing the result back into the original file.

232

13.29   Things can go wrong: using exceptions

A number of things can fail when working with files. When reading files:

– The file you want to open may not exist.
– You may not have sufficient privileges to open the file.
– The file may be corrupted.
– You may run out of memory while reading the file.

When writing to files:

– The folder where you want to create the new file may not exist or be read-only.
– You may not have sufficient privileges to write to the folder.
– You may run out of disk space while writing data to a file, perhaps by using an infinite loop by mistake, etc.

The most frequent exceptions related to working with files are:

– `FileExistsError` is raised when trying to create a file or directory which already exists.
– `FileNotFoundError` is raised when a file or directory is requested but doesn't exist.
– `IsADirectoryError` is raised when a file operation is requested on a directory.
– `PermissionError` is raised when trying to run an operation without the adequate access rights.
– `UnicodeError` is raised when a Unicode-related encoding or decoding error occurs.
– `OSError` is raised when a system function returns a system-related error, including I/O failures such as "file not found" or "disk full".
– `MemoryError` is raised when an operation runs out of memory.

It always is a good idea to put the code that works with the file into a `try` branch. (The `try-except` statement including its full form `try-except-else-finally` was discussed in Section 12.) An example:

```python
try:
    with open('myfile.txt', 'r') as f:
        txt = f.read()
except FileNotFoundError:
    print("File myfile.txt was not found.")
    ... take a corrective action ...
except MemoryError:
    print("Ran out of memory while reading file myfile.txt.")
    ... take a corrective action ...
```

233

Recall from Subsection 12.8 that optional `else` block can be added for code to be executed if no exceptions were raised, and a `finally` block for code to be executed always, no matter what happens in the previous blocks.

13.30 Binary files I – checking byte order

Binary files are suitable for storing binary streams (sequences of 0s and 1s) which not necessarily represent text – such as images or executable files. Before reading from or writing to a binary file, one needs to check the native byte order of the host platform. It can be either little-endian (bits are ordered from the little end = least-significant bit) or big-endian (bits are ordered from the big end = most-significant bit).

Big-endian is the most common format in data networking; fields in the protocols of the Internet protocol suite, such as IPv4, IPv6, TCP, and UDP, are transmitted in big-endian order. For this reason, big-endian byte order is also referred to as *network byte order*. Little-endian storage is popular for microprocessors, in part due to significant influence on microprocessor designs by Intel Corporation. We will not go into more detail here but you can easily find more information online.

To check the byte order, import the `sys` module and print `sys.byteorder`:

```
import sys
print(sys.byteorder)
'little'
```

Hence in our case, the byte order is little-endian.

13.31 Binary files II – writing unsigned integers

Data written to binary files, even raw binary sequences, can always be represented as unsigned (= positive) integers. For instance, it can be 8-bit (1-Byte) segments. Or 2-Byte segments. Or entries consisting of one 2-Byte and one 4-Byte segment. Let's say that our data entries consist of a 2-Byte `id` and a 4-Byte `value` (the latter scenario). To packetize the data into binary format, one needs to import the `struct` module:

```
import struct
```

This module performs conversions between Python values and C structs represented as Python `bytes` objects. It is described at the Python Standard Library documentation page `https://docs.python.org/3/library/struct.html`.

Next, let's open a binary file "datafile.bin" for writing, packetize a sample data pair `id = 345, value = 6789` and write it to the file. The file will be automatically closed at the end of the `with` statement:

```python
import struct
id = 345
value = 6789
with open('datafile.bin', 'wb') as f:
    data = struct.pack('<HI', id, value)
    f.write(data)
```

Here, `'<HI'` is a formatting string where < means little-endian byte order (> would be used for big-endian), `H` means a 2-Byte unsigned integer, and `I` means a 4-Byte unsigned integer. The following table summarizes the most widely used unsigned data formats:

Symbol	Python data type	Length in Bytes
B	unsigned integer	1
H	unsigned integer	2
I	unsigned integer	4
Q	unsigned integer	8

For a complete list of symbols see the above URL.

Writing to and reading from binary files can be simplified using the Pickle module – this will be discussed in Subsection **??**.

13.32 Binary files III – writing bit sequences

In the previous subsection we saw how to write to binary files unsigned integers of various lengths (8, 16, 32 and 64 bits). Now, let's say that we want to write 8-bit binary sequences. These may be parts of a long binary sequence coming from an image, and not represent numbers at all. But for the sake of writing to a binary file, we will convert them into unsigned integers. This is easy. For instance, the bit sequence `'11111111'` represents unsigned decimal integer 255:

```python
binseq = '11111111'
numrep = int(binseq, 2)
print(numrep)
```

The value 2 in `int(binseq, 2)` stands for base-2 (binary) number. Using the code from the previous subsection, an 8-bit binary sequence `binseq` can be written to a binary file "datafile.bin" as follows:

```python
import struct
binseq = '11111111'
with open('datafile.bin', 'wb') as f:
    data = struct.pack('<B', int(binseq, 2))
    f.write(data)
```

Here < means little-endian byte order and `B` stands for 1-Byte (8-bit) unsigned integer.

As another example, let's write a 32-bit sequence `110111101100011101100011100111001 1011`:

```python
import struct
binseq = '11011110110001110110001110011011'
with open('datafile.bin', 'wb') as f:
    data = struct.pack('<I', int(binseq, 2))
    f.write(data)
```

Here `I` stands for 4-Byte (32-bit) unsigned integer. On the way, the binary sequence was converted into an integer value 3737609115 (which really does not matter).

Finally, the binary sequence may be coming not as a text string but directly as a base-2 integer, such as `val=0b11011110110001110110001110011011`. In this case, the conversion step using `int` is not needed, and one can save the number to the file directly:

```python
import struct
val = 0b11011110110001110110001110011011
with open('datafile.bin', 'wb') as f:
    data = struct.pack('<I', val)
    f.write(data)
```

Obviously, a `for` loop can be used to write as many 8-bit, 16-bit, 32-bit or 64-bit sequences as needed.

13.33 Binary files IV – reading byte data

Let's get back to Subsection 13.31 where we created a binary file "datafile.bin" and wrote two unsigned integers `id = 345` (2 Bytes) and `value = 6789` (4 Bytes) to it. The two unsigned integers can be retrieved from the file by opening it for binary reading, read the 2-Byte and 4-Byte byte strings, and convert them into unsigned integers as follows:

```python
with open('datafile.bin', 'rb') as f:
    data = f.read(2)
    id = int.from_bytes(data, byteorder='little', signed=False)
    print(id)
    data = f.read(4)
    value = int.from_bytes(data, byteorder='little', signed=False)
    print(value)
```
```
345
6789
```

Note: We are using the fact that our system is little-endian (see Subsection 13.30). When the file is open for binary reading using the flag `'rb'`, the method `read` returns the so-called *byte string*. Typing `read(1)` returns a byte string of length 1 Byte, `read(2)` of length 2 Bytes etc. The byte string is not a regular text string. One needs to know what the byte string represents in order to decode it correctly. In our case, we knew that both byte strings represented unsigned integers, therefore the method `from_bytes` of class `int` could be used to decode it.

Now let's get back to the example from Subsection 13.32 where we saved the binary sequence `'11011110110001110110001110011011'` to the file "datafile.bin". Here is how to read it back:

```python
with open('datafile.bin', 'rb') as f:
    data = f.read(4)
    value = int.from_bytes(data, byteorder='little', signed=False)
    print(bin(value))              # binary (base-2) integer
    print(str(bin(value))[2:])   # text string of 0s and 1s
```
```
0b11011110110001110110001110011011
11011110110001110110001110011011
```

237

14   Object-Oriented Programming I - Introduction

14.1   Objectives

- Understand the philosophy of object-oriented programming (OOP).
- Understand how OOP differs from procedural (imperative) programming.
- Understand the advantages and disadvantages of OOP.
- Learn about classes, objects, attributes, and methods.
- Learn the syntax of defining and instantiating classes in Python.

More advanced concepts such as inheritance, class hierarchy, polymorphism, and multiple inheritance will be discussed in the next Section 15.

14.2   Why learn about OOP?

Throughout this course, we have been putting emphasis on designing elegant and efficient algorithms and writing clean, transparent code. This included:

- Isolating reusable functionality from the rest of the code by creating *custom functions*.
- Isolating data from the rest of the code by using *local variables*.

Object-oriented programming brings these good programming practices to perfection by isolating functions and the data they work with from the rest of the code using a single entity - a *class*. This approach, called *encapsulation*, is the main idea of OOP. Encapsulation makes the code extremely well structured and as a result, even large software projects become easy to manage.

14.3   Procedural (imperative) programming and OOP

The programming style we have been using so far is called *procedural (imperative) programming*. The name comes from using *procedures* to solve various tasks that lead to the solution of the problem at hand. Procedure is an activity, a sequence of operations done with data that is supplied to it. Importantly, *procedures do not own the data they operate with.*

This, however, means that the data must be stored somewhere else. In other words, the range of validity of the data extends beyong the borders of the function where it is processed. Can you see an analogy to using local and global variables? It is great to keep variables local, and in the same way it would be great to make data local to procedures that operate with them. But wait, this is called *object-oriented programming!*

14.4 Main benefits of OOP in a nutshell

When you grasp the concept of OOP, it is easy to "fall in love" with it. There are programmers who worship OOP and make it their preferred tool for solving everything. We would like to caution you against such an attitude. Every tool was designed with some purpose in mind, and there is no tool which would be the best solution to everything. OOP brings great advantages, but it also has disadvantages. Let's begin with the former:

- *Easier troubleshooting:* When a procedural code breaks down, it can take lots of time to find where it happened. In an object-oriented code, it is clear which object broke down, so the bug is easier to find.
- *Easier code reuse:* An object-oriented code can easily be reused through class inheritance. Encapsulated classes are also easier to port from one code to another.
- *Flexibility through polymorphism:* This is a truly sweet and unique advantage of OOP - the same code can do different things based on the context. For example, calling `X.make_sound()` will have a different outcome when `X=bee` and when `X=lion`.
- *Efficient problem-solving:* You already know that the best way to solve complex problems is to break them down into simple ones. Creating separate classes to handle the simple problems brings this approach to perfection.

14.5 There are disadvantages, too

As with everything - since there are people who love OOP, there also are people who hate it. It is good to listen to both camps because usually both have something of value to say. Here are the typical complaints against OOP:

- Object-oriented programs are longer than procedural ones.
- Object-oriented programs require more work to plan and write.
- Object-oriented programs are slower and require more resources.

Here is a famous comment about OOP (which is not entirely untrue):

"You want a banana but you get a gorilla holding the banana, and the entire jungle with it."

14.6 Procedural vs. object-oriented thinking

Let's present an example which illustrates the difference between procedural and object-oriented thinking. Imagine that you are moving and all your things need to be packed, boxed, loaded on a truck, hauled 1000 miles, unloaded, and brought into your new home.

Procedural thinking: "Do it yourself!"

You (the procedure) go rent a truck, buy cardboard boxes, etc. The truck, the boxes and other moving supplies represent data that the procedure does not own. Then you pack everything yourself, load it on the truck, drive to the other city, unload your things, carry them into your new home, and take out of the boxes. Then you go return the truck and throw away the paper boxes.

Object-oriented thinking: "Delegate the work!"

An object-oriented approach to solving the same task is very different. Instead of doing the work yourself, you call a moving company = create an instance `mc` of a class `MovingCompany`. The object `mc` comes with a truck, boxes, wraps, dolly, and other moving supplies automatically. They even have their own crew of big guys who can lift heavy things. In other words, you as the main program do not have to deal with the technical details of moving. This makes your life much easier, and you are free to do other things and solve other tasks. The movers will arrive at your home, and you will just call their methods `mc.pack()`, `mc.box()`, `mc.load()`, `mc.haul()`, `mc.unload()`, `mc.carry()` and `mc.unpack()`.

14.7   Classes, objects, attributes and methods

Let us stay with the moving example for one more moment. There are many moving companies and all of them do basically the same thing. So, we can talk about a moving company on a general level, without having a concrete one in mind. When we do that, we talk about a *concept*. In object-oriented terminology, such a concept is a *class*.

A *moving company* has both the *hardware* (truck, boxes, dolly, straps, furniture covers, ...) and *skills to use it* (packing, boxing, organizing, loading, unloading, driving the truck, ...). In OOP terminology, a *class* owns both the *data* and the *functionality that uses the data*. The hardware is called *attributes* of the class, and the skills are the *methods* of the class.

But a *class* as an abstract concept will not get your things moved. For that you need a concrete moving company with a name and a phone number. Such as the company `mc` from the previous subsection. Such a concrete representation of a class is then called an *object* or an *instance of the class*.

Technically, *methods* belong to the *class* because that's where they are defined, and not to the instance (object). The object just uses them. But in OOP we often say that "an object uses its methods to ...". The methods of a class can operate on data owned by the class (the moving company's own equipment) as well as on data that does not belong to the class (your things which are being moved).

240

The situation is similar with *attributes* (data). The data can be represented as variables, lists, tuples, dictionaries, etc. They are introduced when the class is defined, but at that time they do not store any concrete values yet. Only after the class is instantiated, the variables in the object are initialized with concrete values.

14.8   Defining class `Circle`

Let's see how all this is implemented in Python. We will say goodbye to the movers and look at some geometry instead. Our first class will be named `Circle`. The purpose of this class is to allow us to easily create many different circles, calculate their areas and perimeters, and plot them with Matplotlib. Hence the class `Circle` will have the following attributes and methods:

Attributes:

- Radius `R`,
- center point coordinates `Cx`, `Cy`,
- two arrays `ptsx`, `ptsy` with the X and Y coordinates of the points on the perimeter of the circle, for Matplotlib (see Subsection 3.7).

Methods:

- `__init__(self, r, cx, cy)` ... initializer,
- `area(self)` ... calculate and return the area of the circle,
- `perimeter(self)` ... calculate and return the perimeter,
- `draw(self)` ... draw itself using Matplotlib.

The method __init__ is an *initializer* which is not exactly the same as a *constructor* in other object-oriented languages. But many Python programmers call it a constructor for simplicity, and we might sometimes do it as well. The initializer has two roles:

- Add new attributes to the class when the class is defined.
- Initialize them with concrete values when the class is instantiated.

The first parameter `self` must be included in each method of the class. This is a reference to the concrete instance through which all data and methods of the instance are accessed.

For completeness we should add that it is possible to use a different name than `self`, but do not do this unless your goal is to confuse everybody who will ever read your code.

Class `Circle` can be defined by typing

```
class Circle:
```

but sometimes you may see

```
class Circle(object):
```

which is also correct. This heading is followed by the definition of the initializer and other methods. Let's show the complete code now and then we will explain it:

```
class Circle:
    """
    Circle with given radius R and center point (Cx, Cy).
    Default plotting subdivision: 100 linear edges.
    """

    def __init__(self, r, cx, cy, n = 100):
        """
        The initializer adds and initializes the radius R,
        and the center point coordinates Cx, Cy.
        It also creates the arrays of X and Y coordinates.
        """
        self.R = r
        self.Cx = cx
        self.Cy = cy
        # Now define the arrays of X and Y coordinates:
        self.ptsx = []
        self.ptsy = []
        da = 2*np.pi/self.n
        for i in range(n):
            self.ptsx.append(self.Cx + self.R * np.cos(i * da))
            self.ptsy.append(self.Cy + self.R * np.sin(i * da))
        # Close the polyline by adding the 1st point again:
        self.ptsx.append(self.Cx + self.R)
        self.ptsy.append(self.Cy + 0)
```

```python
    def area(self):
        """
        Calculates and returns the area.
        """
        return np.pi * self.R**2

    def perimeter(self):
        """
        Calculates and returns the perimeter.
        """
        return 2 * np.pi * self.R

    def draw(self, label):
        """
        Plots the circle using Matplotlib.
        """
        plt.axis('equal')
        plt.plot(self.ptsx, self.ptsy, label = label)
        plt.legend()
```

Notice that the methods are indented. You already know that the initializer __init__
both adds attributes to the class and initializes them with values when the class is in-
stantiated. In particular, the lines

```python
        self.R = r
        self.Cx = cx
        self.Cy = cy
```

add to the class `Circle` the attributes `R` (radius), and `Cx`, `Cy` (center point coordinates).
In the next subsection you will see how they are initialized with concrete values `r`, `cx`,
`cy` when the class is instantiated. The arrays `ptsx` and `ptsy` are first created empty
and then filled with the X and Y coordinates of the perimeter points.

Notice that all methods must have a mandatory first parameter `self`. This param-
eter is used by Python at runtime to pass an instance of the class into the method:

```python
    def __init__(self, r, cx, cy, n = 100):
    def area(self):
    def perimeter(self):
    def draw(self, label):
```

243

Correspondingly, all attributes and methods must be used with the prefix `self` in the class methods.

Finally, notice that inside the `for` loop, the code uses the parametric equation of the circle which was introduced in Subsection 3.19. The code also uses standard math formulas for the area πR^2 and perimeter $2\pi R$.

14.9 Using class `Circle`

With such a nice class in hand, it is easy to create many different circles, calculate their areas and perimeters, and plot them:

```python
# Create an instance of class Circle named C1:
C1 = Circle(1, 0, 0)
# Display the area and perimeter:
print("Area and perimeter of circle C1:", \
C1.area(), C1.perimeter())

# Create an instance of class Circle named C2:
C2 = Circle(0.5, 1, 0)
# Display the area and perimeter:
print("Area and perimeter of circle C2:", \
C2.area(), C2.perimeter())

# Create an instance of class Circle named C3:
C3 = Circle(0.25, 1.5, 0)
# Display the area and perimeter:
print("Area and perimeter of circle C3:", \
C3.area(), C3.perimeter())

# Finally, display all circles using Matplotlib:
plt.clf()
C1.draw("First circle")
C2.draw("Second circle")
C3.draw("Third circle")
plt.show()
```

In the above code, line 2 creates an instance of the class `Circle` named `C1`:

```python
C1 = Circle(1, 0, 0)
```

244

The arguments `1, 0, 0` are passed directly to the initializer of the class as the radius `R` and the center point coordinates `Cx` and `Cy`. The subdivision `n` is skipped so it will have the default value 100. Importantly, notice that the initializer is defined with the first parameter `self`,

```python
def __init__(self, r, cx, cy, n = 100):
```

which is omitted when the initializer is called. This is the case not only for the initializer but also for all other methods. The method `area` is defined with a single parameter `self`,

```python
def area(self):
```

but it is called without any arguments:

```python
C1.area()
```

The same holds for the method `perimeter`. The method `draw` is defined with two parameters `self` and `label`,

```python
def draw(self, label):
```

but it is only called with one argument - the label:

```python
C1.draw("First circle")
```

Line 5 shows how the methods of the class are called – just like usual functions, except the name of the method follows the name of the instance and a period ' . '. The values of the attributes can be accessed in the same way, but it is not needed in this example.
Finally, here is the text output:

```
Area and perimeter of circle 1: 3.14159265359 6.28318530718
Area and perimeter of circle 2: 0.785398163397 3.14159265359
Area and perimeter of circle 3: 0.196349540849 1.57079632679
```

The Matplotlib plot of the three circles is shown in Fig. 67.
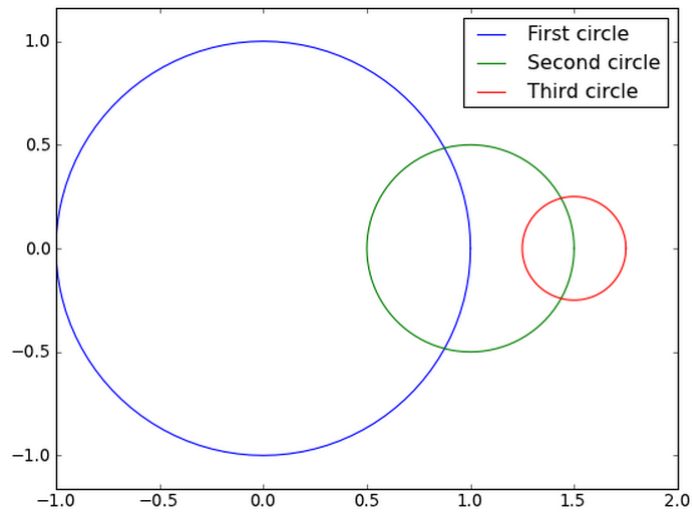
Fig. 67. Three instances of the class `Circle`.

15 Object-Oriented Programming II - Class Inheritance

We already know from Section 14 that the main idea of Object-Oriented Programming (OOP) is *encapsulation*. A *class* is a container that keeps together some data and functions which work with the data. This brings a lot of structure and clarity into the software. The data are called *attributes* and the functions are called *methods*. You also know that a *class* is an abstract concept – such as a blueprint of a house. An *object* is then a concrete *instance* (materialization) of the class – such as a concrete house which is built using the blueprint.

15.1 Objectives

In this Section we will venture further into OOP and explore *class inheritance*. We will:

– Understand the benefits of class inheritance and when it should be used.
– Learn the syntax of creating descendant classes (subclasses) in Python.
– Learn how methods of the parent class are called from the descendant.
– Learn how to correctly write the initializer of the subclass.
– Learn how to design a good class hierarchy.

15.2 Why learn about class inheritance?

Class inheritance is an indivisible part of OOP. It is a powerful way to reuse functionality which can bring unparalleled structure and efficiency to your software projects.

15.3 Hierarchy of vehicles

Before we get to coding, let's briefly talk about vehicles. All vehicles have some things in common, such as

– weight,
– maximum velocity,
– maximum occupancy.

But then they differ in some aspects:

– some vehicles move on the ground (bike, car, truck, bus...),
– some move in the air (plane, rocket, ...),
– some move on the water (jet ski, boat, ship, ...),
– and some even move under water (submarine).

This is illustrated in Fig. 68.
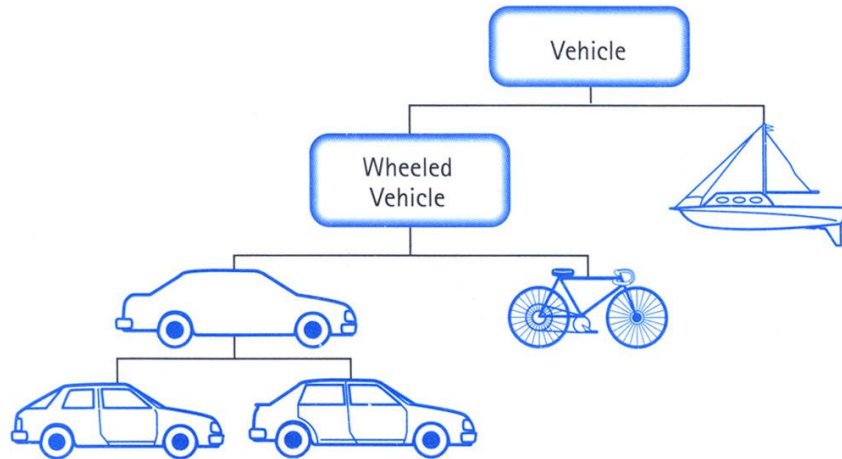
Fig. 68. Hierarchy of vehicles.

All descendants (subclasses) of a class will automatically inherit all its attributes and methods. But moreover, they may add additional attributes and/or methods of their own.

So, the base class `Vehicle` should only have universal attributes which are applicable to all vehicles - `weight`, `max_speed`, `max_occupancy`, etc. It also should only have general methods such as `accelerate()` and `decelerate()` which apply to all vehicles.

Then, a subclass `Wheeled_Vehicle` of the class `Vehicle` will inherit the attributes `weight`, `max_speed`, `max_occupancy`. It will also inherit the methods `accelerate()` and `decelerate()`. These methods may need to be redefined in the subclass because different vehicles may accelerate and decelerate differently. This is called *polymorphism* and we will discuss it in Subsection 16.10. The subclass may need some new attributes including `number_of_wheels` and new methods such as `change_tire()`.

A different subclass `Vessel` of the class `Vehicle` will also inherit the attributes `weight`, `max_speed`, `max_occupancy`. It will also inherit the methods `accelerate()` and `decelerate()` which may need to be redefined. But it will have neither the attribute `number_of_wheels` nor the method `change_tire()`. Instead, it may have a new attribute `displacement` (the volume of water it displaces) and a new method `lift_anchor()`.

15.4 Class inheritance

In Subsections 14.8 and 14.9 we have defined a class `Circle` which allows us to easily create many different circles, calculate their areas and perimeters, and plot them with Matplotlib. Now we would like to create analogous classes also for polygons, triangles, quadrilaterals, rectangles and squares.

We surely could copy and paste the class `Circle` five more times, and rename and tweak each copy to work for the particular shape. But that would be a horrible way to do it! The same can be done extremely elegantly and efficiently using class inheritance.

First, like with the vehicles, we need to figure out what attributes and methods we should include with the base class. In this case, let's call it `Geometry`. Hence, what will the classes `Circle`, `Polygon`, `Triangle`, `Quad`, `Rectangle` and `Square` have in common and where will they differ?

For sure, all these classes will need the arrays `ptsx` and `ptsy` of X and Y coordinates for Matplotlib. And in each class, the method `draw` will be identical to the one of the `Circle`. The other two methods `area` and `perimeter` will differ from class to class.

15.5 Base class `Geometry`

Based on the considerations from the previous subsection, the base class `Geometry` can be defined as follows:

```python
class Geometry:
    """
    Base class for geometrical shapes.
    """
    def __init__(self):
        """
        Initializer creating empty lists of X and Y coordinates.
        """
        self.ptsx = []
        self.ptsy = []

    def draw(self, label):
        """
        Display the shape using Matplotlib.
        """
        plt.axis('equal')
        plt.plot(self.ptsx, self.ptsy, label=label)
        plt.legend()
```

This base class really cannot do much, but it is fine. The class `Geometry` will not be instantiated. We will use it to derive several descendant classes from it. The benefit of class inheritance here is that if we decide in the future to change the plotting mechanism, we will just have to change it in one place – in the base class `Geometry` – and the change will propagate automatically to all its descendants.

15.6   Hierarchy of geometrical shapes

Let's review some basic facts from geometry, because this is going to help us design a good hierarchy of our classes which is shown in Fig. 69.
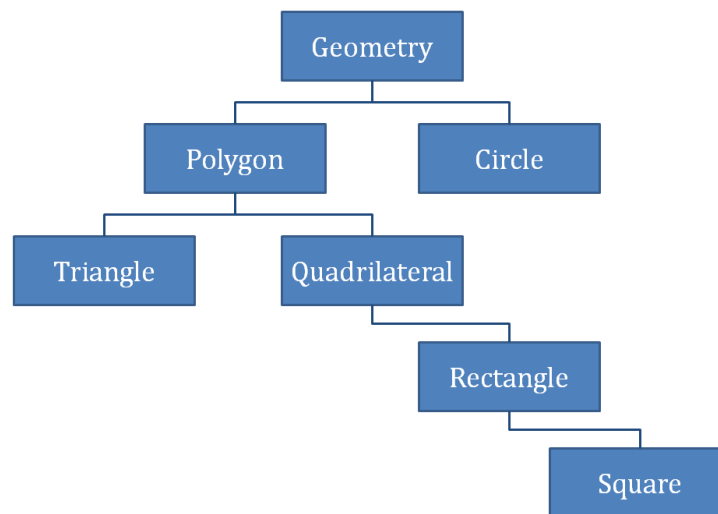
Fig. 69. Hierarchy of geometry classes.

First, a (closed oriented) polygon is formed by a sequence of points in the XY plane called *vertices* which are connected with a polyline into a closed loop. So, a triangle is a special case of a polygon with three vertices, and a quadrilateral is a special case of a polygon with four vertices. Therefore, the classes `Triangle` and `Quadrilateral` should naturally be descendants of the class `Polygon`.

Further, a rectangle is a special case of a quadrilateral whose adjacent edges have right angles between them. Hence the class `Rectangle` should be inherited from the class `Quadrilateral`.

And last, a square is a special case of a rectangle whose edges are equally long. Thus the class `Square` should be derived from the class `Rectangle`.

250

The decision where to place the class `Circle` is a bit more delicate. One could argue that in computer graphics, circles are just regular polygons with a large number of short edges. Based on this, one could derive class `Circle` from the class `Polygon`. Then, methods `area` and `perimeter` defined in the class `Polygon` would automatically work in the class `Circle`.

However, these methods would give imprecise results because a regular polygon approximating a circle has a slightly smaller area and perimeter than the real circle. And moreover, such class `Circle` would be different from the class `Circle` which was defined in Subsection 14.8. Therefore, we prefer to take the viewpoint that circles are geometrically genuinely different from polygons, and we will derive the class `Circle` directly from the base class `Geometry`.

Finally, notice that the less general class always is derived from a more general one, or in other words, that the process of class inheritance is making classes more specialized. In the following subsections we will explain each step of the inheritance scheme shown in Fig. 69 above.

15.7 Deriving class `Polygon` from class `Geometry`

Class `Polygon` can be inherited from the class `Geometry` by typing

```python
class Polygon(Geometry):
```

Its full definition is shown below. The attributes `ptsx` and `ptsy` inherited from the class `Geometry` are not repeated, nor is the method `draw`. If needed, the inherited method `draw` could be redefined in the subclass, but it is not necessary in our case. Redefining methods in subclasses is called *polymorphism* and we will discuss it in more detail in Subsection 16.10. Importantly, notice that the initializer of the superclass `Geometry` is called via the built-in function `super()` at the beginning of the body of the initializer of the subclass `Polygon`:

```python
super().__init__(self)
```

The purpose of this call is to initialize all attributes inherited from the superclass.

In order to calculate the area of a counter-clockwise (CCW) oriented polygon, we use a classical algorithm which constructs oriented trapezoids below the edges, and adds their (oriented) areas. This algorithm can be found easily online. The perimeter is calculated simply by adding up the lengths of all edges:

```python
class Polygon(Geometry):
    """
    Class Polygon represents a general polygon
    with counter-clockwise (CCW) oriented boundary.
    """

    def __init__(self, L):
        """
        Initializer: here L is list of vertices.
        """
        super().__init__(self)
        # Convert L into plotting arrays:
        for pt in L:
            self.ptsx.append(pt[0])
            self.ptsy.append(pt[1])
        # To close the loop in plotting:
        pt = L[0]
        self.ptsx.append(pt[0])
        self.ptsy.append(pt[1])

    def area(self):
        """
        Calculate the area of a general oriented polygon.
        """
        ymin = min(self.ptsy)
        # The area of an oriented polygon is the sum of
        # areas of oriented trapezoids below its edges:
        m = len(self.ptsx) - 1   # Number of trapezoids
        s = 0
        for i in range(m):
            s -= (self.ptsx[i+1] - self.ptsx[i]) \
                * (self.ptsy[i+1] + self.ptsy[i] \
                - 2*ymin) / 2.
        return s
```

```
    def perimeter(self):
        """
        Calculate the perimeter of a general oriented polygon.
        """
        l = 0
        m = len(self.ptsx) - 1
        for i in range(m):
            l += sqrt((self.ptsx[i+1] - self.ptsx[i])**2
                + (self.ptsy[i+1] - self.ptsy[i])**2)
        return l
```

Finally, we already mentioned this, but notice that the definition of the class `Polygon` does not repeat the definition of the method `draw` – this method is inherited from the class `Geometry` and will work without any changes.

15.8   Deriving classes `Triangle` and `Quad` from class `Polygon`

The `Polygon` class defined above is very general. Of course we could use it directly to render triangles, quads, rectangles, and squares. But these geometries are simpler and thus we also want their instantiation to be simpler. Let us begin with the class `Triangle`.

This class takes three points `a`, `b`, `c` as parameters, forms a list containing these three points, and then passes the list to the initializer of the parent class `Polygon`. Notice the way the initializer of the `Polygon` class is called from its descendant `Triangle` via the function `super()`:

```
class Triangle(Polygon):
    """
    General triangle with CCW oriented boundary.
    """
    def __init__(self, a, b, c):
        L = [a, b, c]
        super().__init__(self, L)
```

The methods `area`, `perimeter` and `draw` do not have to be redefined - they are inherited from the class `Polygon` and will work correctly.

The class `Quad` can be created analogously:

```
class Quad(Polygon):
    """
    General quadrilateral with CCW oriented boundary.
    """
    def __init__(self, a, b, c, d):
        L = [a, b, c, d]
        super().__init__(self, L)
```

Same as with the class `Triangle`, the methods `area`, `perimeter` and `draw` are inherited from the class `Polygon` and will work correctly without any changes.

15.9 Deriving class `Rectangle` from class `Quad`

As we explained in Subsection 15.6, rectangle is a special case of a quadrilateral. We will define it using just two numbers a, b for the edge lengths, as a CCW oriented quadrilateral with four points `(0, 0)`, `(a, 0)`, `(a, b)` and `(0, b)`:

```
class Rectangle(Quad):
    """
    Rectangle with dimensions (0, a) x (0, b).
    """
    def __init__(self, a, b):
        super().__init__(self, [0, 0], [a, 0], [a, b], [0, b])
```

The methods `area`, `perimeter` and `draw` are inherited from the class `Quad` and will just work without any changes.

15.10 Deriving class `Square` from class `Rectangle`

Square is a special case of rectangle with equally-long edges. Therefore we can derive class `Square` from the class `Rectangle` by just redefining the initializer:

```
class Square(Rectangle):
    """
    Square with dimensions (0, a) x (0, a).
    """
    def __init__(self, a):
        super().(self, a, a)
```

Again, the methods `area`, `perimeter` and `draw` are inherited from the class `Rectangle` and will work without any changes.

15.11   Deriving class `Circle` from class `Geometry`

As we explained in Subsection 15.6, circle is a non-polygonal shape which therefore should not be derived from the class `Polygon` but rather directly from the class `Geometry`. This time we will add new attributes `R` (radius) and `Cx`, `Cy` (coordinates of the center point). Since you already know the class `Circle` from Subsection 14.8, there is no need to explain the initializer `__init__` or the methods `area` and `perimeter` again:

```python
class Circle(Geometry):
    """
    Circle with given radius R and center point (Cx, Cy).
    Default plotting subdivision: 100 linear edges.
    """

    def __init__(self, r, cx, cy, n = 100):
        """
        The initializer adds and initializes the radius R,
        and the center point coordinates Cx, Cy.
        It also creates the arrays of X and Y coordinates.
        """
        super().__init__(self)
        self.R = r
        self.Cx = cx
        self.Cy = cy
        # Now define the arrays of X and Y coordinates:
        self.ptsx = []
        self.ptsy = []
        da = 2*np.pi/self.n
        for i in range(n):
            self.ptsx.append(self.Cx + self.R * np.cos(i * da))
            self.ptsy.append(self.Cy + self.R * np.sin(i * da))
        # Close the polyline by adding the 1st point again:
        self.ptsx.append(self.Cx + self.R)
        self.ptsy.append(self.Cy + 0)
```

```python
    def area(self):
        """
        Calculates and returns the area.
        """
        return np.pi * self.R**2

    def perimeter(self):
        """
        Calculates and returns the perimeter.
        """
        return 2 * np.pi * self.R
```

Notice that this time we also added the methods `area` and `perimeter` because only the method `draw` is inherited from the class `Geometry`.

15.12 Sample application

Finally, let us put all the classes to work! We will create sample instances, inquire about their areas and perimeters, and ask them to display themselves:

```python
# Create a triangle:
T = Triangle([-2, -0.5], [0, -0.5], [-1, 2])
print("Area and perimeter of the triangle:", \
T.area(), T.perimeter())

# Create a quad:
Q = Quad([-3, -1], [0, -1], [-1, 1], [-2, 1])
print("Area and perimeter of the quad:", \
Q.area(), Q.perimeter())

# Create a rectangle:
R = Rectangle(3, 1)
print("Area and perimeter of the rectangle:", \
R.area(), R.perimeter())

# Create a square:
S = Square(1.5)
print("Area and perimeter of the square:", \
S.area(), S.perimeter())
```

```
# Create a circle:
C = Circle(2.5, 0, 0.5)
print("Area and perimeter of the circle:", \
C.area(), C.perimeter())

# Plot the geometries:
plt.clf()
T.draw("Triangle")
Q.draw("Quad")
R.draw("Rectangle")
S.draw("Square")
C.draw("Circle")
plt.ylim(-3, 4)
plt.legend()
plt.show()
```

The graphical output is shown in Fig. 70.
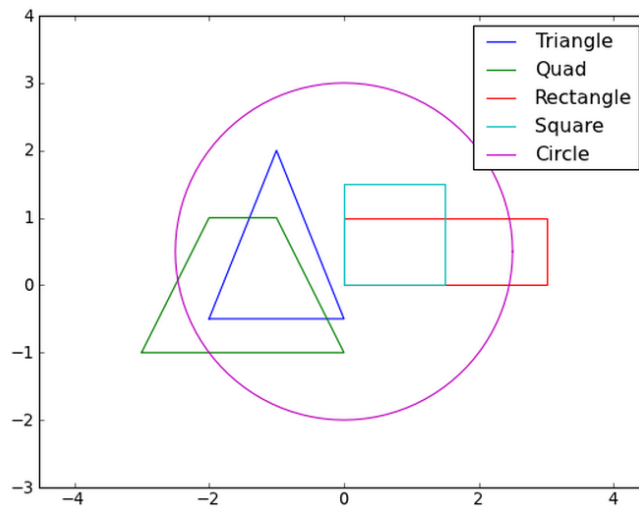
Fig. 70. Visualization of the instances created above.

And finally, here is the text output:

```
Area and perimeter of the triangle: 2.5 7.38516480713
Area and perimeter of the quad: 4.0 8.472135955
Area and perimeter of the rectangle: 3.0 8.0
Area and perimeter of the square: 2.25 6.0
Area and perimeter of the circle: 19.6349540849 15.7079632679
```

At this point you have a solid theoretical understanding of the principles of OOP and class inheritance. Some additional, more advanced aspects will be discussed in the following section. But most importantly, you need more training. Therefore we recommend that you take the Python II course in NCLab which will guide you through two additional projects – you will create your own object-oriented version of the Python Turtle (famous drawing program based on the educational programming language Logo) and build an object-oriented Geometry Editor. Visit NCLab's home page `http://nclab.com` to learn more!

16 Object-Oriented Programming III - Advanced Aspects

In this Section we will explore some more advanced aspects of OOP.

16.1 Objectives

We will learn about:

- Object introspection:
  - How to check if an object is instance of a given class.
  - How to use the built-in function `help` to inspect classes.
  - How to obtain class name from a given object.
  - How to check if a class is subclass of another class.
  - How to check if an object has a given attribute or method, etc.
- Polymorphism.
- Multiple inheritance.

16.2 Why learn about polymorphism and multiple inheritance?

Polymorphism is probably the most beautiful aspect of object-oriented programming. You already know how to create subclasses. We will show you that it is possible to redefine the same method in each subclass to work differently. This opens the door to amazing algorithms where the code can do very different things based on the object which is used. As for multiple inheritance, we will show you why it is good to know about it, but also why one should be extremely cautious with it.

16.3 Inspecting objects with `isinstance`

Python has a built-in function `isinstance` which can be used to determine if an object is an instance of a given class or its superclass. Typing `isinstance(a, C)` will return `True` if object `a` is an instance of class `C` or its superclass, and `False` otherwise. This function is often used when checking the sanity of user data.

As you know, all types of variables such as `bool`, `int`, `float`, `complex`, `str` etc. are classes. For example, one can check whether a variable `var` is a Boolean as follows:

```python
var = 10 < 6
flag = isinstance(var, bool)
print(flag)
True
```

Or, one can check if a variable `t` is a text string:

```python
t = 25
flag = isinstance(t, str)
print(flag)
```
```
False
```

In the same way one can check instances of other built-in types including `list`, `tuple`, `dict`, `set` etc:

```python
L = [1, 2, 3]
flag = isinstance(L, list)
print(flag)
```
```
True
```

16.4 Inspecting instances of custom classes with `isinstance`

One has to be a bit careful here. Let's return for a moment to our classes `Geometry`, `Circle`, `Polygon`, `Triangle`, `Quad`, `Rectangle` and `Square` from Section 15 whose hierarchy was shown in Fig. 69:
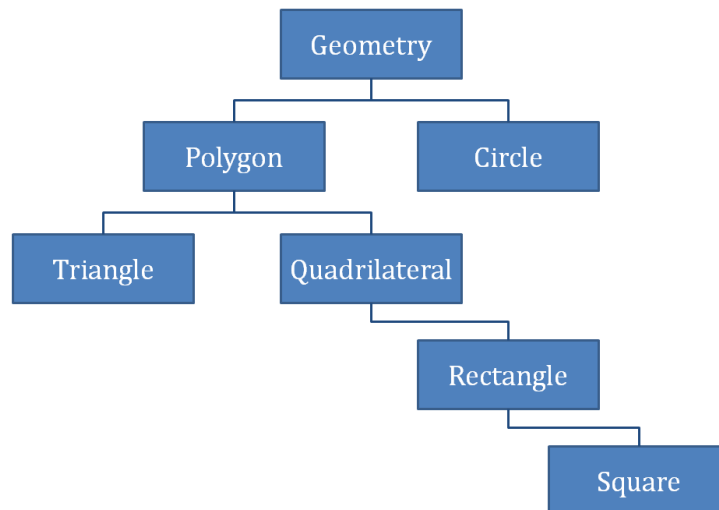
Fig. 71. Hierarchy of geometry classes.

260

As we mentioned in the previous subsection, typing `isinstance(a, C)` will return `True` if object `a` is an instance of class `C` or its superclass, and `False` otherwise. The part "or its superclass" is important. If one takes the name of the function `isinstance` literally, one might be surprised by the fact that a square checks as an instance of the class `Polygon`:

```python
s = Square(2)
flag = isinstance(s, Polygon)
print(flag)
```
```
True
```

When working with a custom hierarchy of classes, one might want the superclasses to be excluded. In the following subsection we will show you how.

16.5   Inspecting objects with `type`

If one needs to check whether an object `a` is an instance of a class `C` but not its superclass, one can use the built-in function `type`. The call `type(a)` returns the class this object is an instance of. Returning to the example from the previous subsection:

```python
s = Square(2)
flag = type(a) == Polygon
print(flag)
```
```
False
```

But the same test passes for the class `Square` as it should:

```python
s = Square(2)
flag = type(a) == Square
print(flag)
```
```
True
```

16.6   Inspecting classes with `help`

The built-in function `help` which you already know can be used to obtain very detailed information about any class, both built-in and custom. For illustration, let's get help on the class `list`:

```python
help(list)
```

The output is very long:

```
Help on class list in module builtins:

class list(object)
 |  list() -> new empty list
 |  list(iterable) -> new list initialized from iterable's
    items
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __delitem__(self, key, /)
 |      Delete self[key].
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getitem__(...)
 |      x.__getitem__(y) <==> x[y]
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __iadd__(self, value, /)
 |      Implement self+=value.
 |
 |  __imul__(self, value, /)
 |      Implement self*=value.
```

```
|  __init__(self, /, *args, **kwargs)
|      Initialize self.  See help(type(self)) for accurate
|      signature.
|
|  __iter__(self, /)
|      Implement iter(self).
|
|  __le__(self, value, /)
|      Return self<=value.
|
|  __len__(self, /)
|      Return len(self).
|
|  __lt__(self, value, /)
|      Return self<value.
|
|  __mul__(self, value, /)
|      Return self*value.n
|
|  __ne__(self, value, /)
|      Return self!=value.
|
|  __new__(*args, **kwargs) from builtins.type
|      Create and return a new object.  See help(type) for
|      accurate signature.
|
|  __repr__(self, /)
|      Return repr(self).
|
|  __reversed__(...)
|      L.__reversed__() -- return a reverse iterator over the
|      list
|
|  __rmul__(self, value, /)
|      Return self*value.
|
|  __setitem__(self, key, value, /)
|      Set self[key] to value.
```

```
|  __sizeof__(...)
|      L.__sizeof__() -- size of L in memory, in bytes
|
|  append(...)
|      L.append(object) -> None -- append object to end
|
|  clear(...)
|      L.clear() -> None -- remove all items from L
|
|  copy(...)
|      L.copy() -> list -- a shallow copy of L
|
|  count(...)
|      L.count(value) -> integer -- return number of
|      occurrences of value
|
|  extend(...)
|      L.extend(iterable) -> None -- extend list by appending
|      elements from the iterable
|
|  index(...)
|      L.index(value, [start, [stop]]) -> integer -- return
|      first index of value.
|      Raises ValueError if the value is not present.
|
|  insert(...)
|      L.insert(index, object) -- insert object before index
|
|  pop(...)
|      L.pop([index]) -> item -- remove and return item at
|      index (default last).
|      Raises IndexError if list is empty or index is out of
|      range.
|
|  remove(...)
|      L.remove(value) -> None -- remove first occurrence of
|      value.
|      Raises ValueError if the value is not present.
```

264

```
|  reverse(...)
|      L.reverse() -- reverse *IN PLACE*
|
|  sort(...)
|      L.sort(key=None, reverse=False) -> None -- stable sort
|      *IN PLACE*
|
|  ----------------------------------------------------------------
|  Data and other attributes defined here:
|
|  __hash__ = None
```

16.7   Obtaining class and class name from an instance

Every class in Python has a (hidden) attribute __class__ which makes it possible to retrieve the class C corresponding to a given instance a. Typing a.__class__ gives exactly the same result as typing type(a). The name of the class, as a text string, can be retrieved by typing a.__class__.__name__. This is illustrated in the following example which uses the class Circle which we defined in Subsection 14.8:

```
a = Circle(1, 0, 0)
c1 = type(a)
c2 = a.__class__
name = a.__class__.__name__
print(c1)
print(c1)
print(name)
```
```
<class 'Circle'>
<class 'Circle'>
Circle
```

16.8   Inspecting classes with issubclass

Python has a built-in function issubclass(A, B) which returns True if class A is subclass of the class B and False otherwise. The following example shows how this function can be used with our custom geometry classes from Section 15:

```
flag1 = issubclass(Triangle, Polygon)
flag2 = issubclass(Triangle, Circle)
print(flag1)
print(flag2)
```

```
True
False
```

However, in the code we may need to ask this question on the level of instances rather than classes. The solution is to first retrieve the class from each instance, and only then use `issubclass`:

```
t1 = Triangle([0, 0], [1, 0], [0, 1])
p1 = Polygon([[0, 1], [2, 0], [3, 2], [2, 3]])
c1 = Circle(2, 1, 1)
flag1 = issubclass(t1.__class__, p1.__class__)
flag2 = issubclass(t1.__class__, c1.__class__)
print(flag1)
print(flag2)
```

```
True
False
```

16.9   Inspecting objects with `hasattr`

Python has a useful built-in function `hasattr(a, name)` which returns `True` if object a has attribute or method `name`, and `False` otherwise. Here, `name` must be a text string. For instance, let's inquire about the class `Circle`:

```
c1 = Circle(2, 1, 1)
flag1 = hasattr(c1, 'R')
flag2 = hasattr(c1, 'ptsx')
flag3 = hasattr(c1, 'X')
flag4 = hasattr(c1, 'area')
flag5 = hasattr(c1, 'volume')
flag6 = hasattr(c1, 'draw')
print(flag1)
print(flag2)
print(flag3)
print(flag4)
print(flag5)
print(flag6)
```

```
True
True
False
True
False
True
```

16.10 Polymorphism - Meet the Lemmings

Have you ever heard about the Lemmings? The Lemmings is a legendary 8-bit game created by Mike Dailly and David Jones in 1991 for Amiga computers. You can read all about it on Wikipedia, and you can even play it today in your web browser at `https://www.elizium.nu/scripts/lemmings/if` you can survive some ads.

Fig. 72. The Lemmings.

The reason why we mentioned the Lemmings is that this is a great way to explain *polymorphism*.

Formally speaking, polymorphism is an ability of derived classes to redefine the methods of the ancestor class. Hmm, that's not very clear - is it? Fortunately, here come the Lemmings! Although all of them look alike, there are eight types: Digger, Miner, Basher, Builder, Blocker, Exploder, Floater, and Climber.

In the program below, each of them is an individual class, derived from the same base class `Lemming`. Notice that the base class defines a method `work` which is then

redefined in each subclass to do something different:

```python
class Lemming:
    def work(self): print("I do nothing.")
class Digger(Lemming):
    def work(self): print("I dig!")
class Miner(Lemming):
    def work(self): print("I mine!")
class Basher(Lemming):
    def work(self): print("I bash!")
class Builder(Lemming):
    def work(self): print("I build!")
class Blocker(Lemming):
    def work(self): print("I block!")
class Exploder(Lemming):
    def work(self): print("I explode!")
class Floater(Lemming):
    def work(self): print("I float!")
class Climber(Lemming):
    def work(self): print("I climb!")
```

Now let's create an instance of each subclass, and call the method `work` for each one:

```python
L = [Digger(), Miner(), Basher(), Builder(), Blocker(), \
     Exploder(), Floater(), Climber()]
for l in L:
    l.work()
```

```
I dig!
I mine!
I bash!
I build!
I block!
I explode!
I float!
I climb!
```

As you can see, the outcomes of the same code `l.work()` are different. This is the true meaning of polymorphism.

16.11 Polymorphism II - Geometry classes

We have already used polymorphism before without mentioning it. In Section 15 we created a base class `Geometry` whose initializer was very simple:

```python
def __init__(self):
    """
    Initializer creating empty lists of X and Y coordinates.
    """
    self.ptsx = []
    self.ptsy = []
```

Then we derived the classes `Polygon`, `Circle`, `Triangle`, `Quad`, `Rectangle` and `Square` from it, redefining the initializer in each subclass. For example, in the subclass `Polygon` the initializer converted a list of points `L` into the arrays `ptsx` and `ptsy`:

```python
def __init__(self, L):
    """
    Initializer: here L is list of vertices.
    """
    super().__init__(self)
    # Convert L into plotting arrays:
    for pt in L:
        self.ptsx.append(pt[0])
        self.ptsy.append(pt[1])
    # To close the loop in plotting:
    pt = L[0]
    self.ptsx.append(pt[0])
    self.ptsy.append(pt[1])
```

In the subclass `Square`, the initializer took just one number `a` which was the size of the square, and called the initializer of the superclass `Rectangle`:

```python
def __init__(self, a):
    super().(self, a, a)
```

We will not list all the remaining initializers here, but you can find them in Section 15.

Finally, let us mention that Python makes polymorphism so easy, that one could easily miss how big deal this is. For comparison:

269

- C++ introduces keyword "virtual" and special virtual methods to do the same thing.
- Java does not use the keyword "virtual" but it has a special type of class (abstract class).

16.12   Multiple inheritance I - Introduction

By *multiple inheritance* we mean that a descendant class (say B) is derived from two or more ancestor classes (say A1, A2, ...), inheriting the attributes and methods from both/all of them. The syntax is what you would expect - instead of typing

```
class B(A):
```

where A is the ancestor class, one types

```
class B(A1, A2):
```

where A1, A2 are two ancestor classes in this case.

Although multiple inheritance sounds like a cool thing, its practical usefulness is extremely limited. It is good to know that it exists, but experienced software developers do not recommend it. Let's explain why.

Most of the time, multiple inheritance is used without proper understanding. It may seem like an elegant way to easily aggregate attributes and/or methods from various classes together. Basically, to "add classes together". But this is fundamentally wrong.

It is OK to think about a Car as being a composition of various parts including Engine, Wheels, Carburetor, AirFilter, SparkPlugs, Exhaust, etc. But then one should just create class Car and have instances of all these other classes in it - this is not a case for multiple inheritance from the classes Engine, Wheels, Carburetor, etc.

A more justified case for multiple inheritance would be a class FlyingCar which would have the classes Car and Airplane as ancestors, attributes Wheels and Wings, and methods `drive()` and `fly()`. But you can already see problems - for example, what about Engine and Wheels? Will they be coming from the Car class or from the Airplane class?

In general, a good justification for multiple inheritance is extremely hard to find.

On top of this, the next subsection presents the so-called *Diamond of Dread*, a classical problem of multiple inheritance you should know about. But to finish on a positive note, Subsection 16.12 will present a working example of multiple inheritance where you will be able to see exactly how it's done.

16.13 Multiple inheritance II - The Diamond of Dread

The so-called *Diamond of Dread* is the classical problem of multiple inheritance.
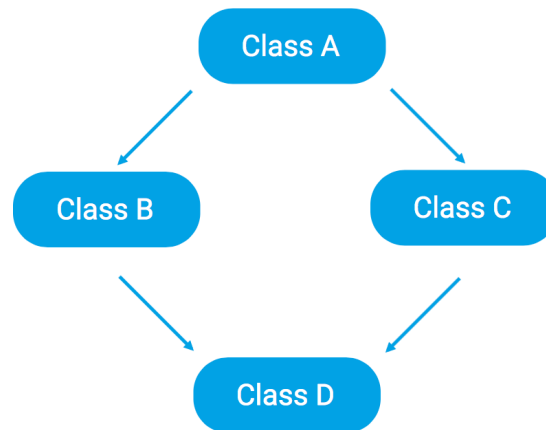
Fig. 73. Diamond of Dread.

In short - if one creates two descendants B, C of the same base class A, and then defines a new class D from B, C using multiple inheritance, there will be conflicts between attributes and/or methods coming from the classes B and C.

How to avoid the Diamond of Dread problem:

When creating a new class D using multiple inheritance from the classes B and C, make sure that the ancestors B and C are completely independent classes which do not have a common ancestor themselves. However, even if the classes B, C do not have a common ancestor, but have attributes or methods with the same names, there is going to be a conflict on the level of the class D.

16.14 Multiple inheritance III - Leonardo da Vinci

You certainly heard about Leonardo da Vinci, a universal Renaissance genius whose areas of interest included engineering, science, painting, music, architecture, and many others.

271

Fig. 74. Leonardo da Vinci.

In order to illustrate multiple inheritance, we allowed ourselves to create just two base classes:

– `Engineer` who has an attribute `invention` and method `invent`,
– `Painter` who has an attribute `painting` and method `paint`.

The program below creates a new class `Genius` as a descendant of these two classes. This class will have both attributes `invention` and `painting`, and both methods `invent` and `paint`. In particular, pay attention to the complicated-looking constructor which uses `*args` and `**kwargs`. You do not need to learn it, just remember that it's part of the story. Recall that the usage of `*args` and `**kwargs` was explained in Section 8.

```python
class Engineer:
    def __init__(self, *args, **kwargs):
        super(Engineer, self).__init__(*args, **kwargs)
        self.invention = kwargs['invention']
    def invent(self):
        return self.invention

class Painter:
    def __init__(self, *args, **kwargs):
        super(Painter, self).__init__()
        self.painting = kwargs['painting']
    def paint(self):
        return self.painting
```

```python
class Genius(Engineer, Painter):
    def __init__(self, *args, **kwargs):
        super(Genius, self).__init__(*args, **kwargs)
    def create(self):
        print("I am inventing " + self.invent() + ".")
        print("I am painting " + self.paint() + ".")

# Main program:
Leonardo = Genius(invention="an airplane", painting="Mona Lisa")
Leonardo.create()
```

```
I am inventing an airplane.
I am painting Mona Lisa.
```

17 Recursion

17.1 Objectives

In this Section we will show you:

- That recursion is a concept that extends way beyond computer programming.
- How to recognize whether or not a task is suitable for recursion.
- How to implement recursion correctly.
- What can happen if recursion is not done right.

17.2 Why learn about recursion?

"To iterate is human, to recurse divine." (L. Peter Deutsch)

This classical programming quote is 100% true - recursion is a beautiful and powerful tool. On the other hand, it is important to know that as any other tool, it is suitable for some tasks but not for all.

17.3 Introduction

Many websites describe recursion more or less as follows:

- *"Programming technique where a function can call itself..."*
- *"The process in which a function calls itself..."*

But recursion is much more. It can be found in many areas ranging from art and linguistics to mathematics and computer science. The following image shows an example of recursion in art: – the so-called *Russian dolls*.

Fig. 75. Russian dolls - an example of recursion in art.

Notice that they are all very similar, almost identical, just getting smaller and smaller. In fact, they can be inserted in each other so that at the end the largest one contains all of them.

17.4 Is your task suitable for recursion?

Let's talk about using recursion to solve certain tasks. An important skill that you need to acquire is to recognize whether a task is suitable for recursion or not. All tasks which are suitable for recursion are similar:

– Part of the task can be solved by any method.
– The rest leads to exactly the same task, just smaller.
– Typically, recursive tasks are repetitive in nature.

Let's show a few examples.

Example 1: Eating your lunch is recursion

Imagine that your "task" is to eat your food. If you don't like chili, picture your favorite food now. How do you eat it? Well, first you check: Is there any more left? If so, you eat a bite. Then there is a bit less food to eat. But you can start the original task again: Eat your food!

Fig. 76. Eating your lunch is recursion.

Example 2: Cleaning up is recursion

Your next task is to put all the Lego pieces back into the box. First you check the floor: Are there some pieces still lying around? If so, collect a few and put them into the box. Then there are fewer loose pieces on the floor. So your can start the original

275

task again: Put all the Lego pieces back into the box!

Fig. 77. Cleaning up is recursion.

Example 3: Reading a book is recursion

The third and last example is about reading a book. Your progress will be measured by the number of unread pages. First you check: Are there any more pages left to read? If so, read a few pages from where you left off. After that, fewer pages remain, so your task got smaller. But it is still the same: Read the rest of the book!

Fig. 78. Reading a book is recursion.

17.5 Calculating the factorial

The factorial $n!$ of a positive integer n is the product $n \cdot (n-1) \cdot (n-2) \ldots 2 \cdot 1$, and moreover the factorial $0!$ is 1. This is a perfect task for recursion because $n!$ is $n \cdot (n-1)!$ which means that the task can easily be reduced to the same task, just smaller. Here is the corresponding Python code, with a sample calculation of $5!$ at the end:

```
def factorial(n):
    if n > 0:
        return n * factorial(n-1)
    return 1

print(factorial(5))
```

```
120
```

On line 3, the function `factorial` calls itself – this is the *recursive call*. For every recursive call, a new copy of the function is created in computer memory.

The program looks extremely simple, but there is one important thing which is easy to miss – notice that *the recursive call is made from the body of a stopping condition*. Every recursive algorithm needs a stopping condition. Without one, it instantly turns into an infinite loop. In fact, obtaining an infinite loop is the most frequent bug in the design of recursive algorithms. We will talk about it in more detail in the next subsection.

But before we get there, let's take a closer look at the previous program. In order to gain more insight into what is happening inside, we will add some debugging prints into it. First let's run the program, and then we will discuss the output:

```
def factorial(n):
    if n > 0:
        print("Calling factorial(" + str(n-1) + ")")
        fact = factorial(n-1)
        print("Obtained factorial(" + str(n-1) \
        + ") which is " + str(fact))
        print("Returning " + str(n) + " * factorial(" \
        + str(n-1) + ") which is " + str(n*fact))
        return n * fact
    print("Recursion finished.")
    return 1

print(factorial(5))
```

```
Calling factorial(4)
Calling factorial(3)
Calling factorial(2)
Calling factorial(1)
Calling factorial(0)
Recursion finished.
Obtained factorial(0) which is 1
Returning 1 * factorial(0) which is 1
Obtained factorial(1) which is 1
Returning 2 * factorial(1) which is 2
Obtained factorial(2) which is 2
Returning 3 * factorial(2) which is 6
Obtained factorial(3) which is 6
Returning 4 * factorial(3) which is 24
Obtained factorial(4) which is 24
Returning 5 * factorial(4) which is 120
120
```

The first call to the function occurs on the last line of the program as `factorial(5)`. With `n = 5` the condition on line 2 is satisfied, and therefore the function calls itself on line 4 as `factorial(4)`.

At this moment, a new copy of the function is created in computer memory and the value of `n` is 4. Then again, the condition on line 2 (of the copy) is satisfied, and the function calls itself on line 4 as `factorial(3)`.

At this moment, a new copy of the function is created in computer memory and the value of `n` is 3. This goes on until a new copy of the function is created and the value of `n` is 0.

This is a very important moment because the condition on line 2 is not satisfied. Hence the fifth recursive copy of the function displays `"Recursion finished"`, returns 1, and ends.

At this moment, we are back on line 4 of the fourth recursive copy of the function, with `fact = 1`. The rest of the body of the condition is executed, and because `fact = 1` and `n = 1`, the return statement returns 1. Then the fourth recursive copy of the function ends.

At this moment, we are back on line 4 of the third recursive copy of the function, with `fact = 1`. The rest of the body of the condition is completed, and because `fact = 1` and `n = 2`, the return statement returns 2. Then the third recursive copy of the function ends.

This goes on until also the second and first recursive copies of the function are finished, and finally, also the main call to the function on the last line of the program ends.

17.6 Stopping condition and infinite loops

To illustrate the importance of the stopping condition, let us remove it from the previous program:

```python
def factorial(n):
    print("Calling factorial(" + str(n-1) + ")")
    fact = factorial(n-1)
    print("Obtained factorial(" + str(n-1) \
    + ") which is " + str(fact))
    print("Returning " + str(n) + " * factorial(" \
    + str(n-1) + ") which is " + str(n*fact))
    return n * fact
    print("Recursion finished.")
    return 1

print(factorial(5))
```

Then, a new copy of the function is automatically created on line 3, which means an infinite loop. Here is the corresponding output

```
Calling factorial(4)
Calling factorial(3)
Calling factorial(2)
Calling factorial(1)
Calling factorial(0)
Calling factorial(-1)
Calling factorial(-2)
Calling factorial(-3)
...
[OUTPUT TRUNCATED]
```

Python then throws a `RecursionError`:

```
Traceback (most recent call last):
  File "<string>", line 10, in <module>
  File "<nclab>", line 3, in factorial
  File "<nclab>", line 3, in factorial
  File "<nclab>", line 3, in factorial
  [Previous line repeated 987 more times]
  File "<nclab>", line 2, in factorial
RecursionError: maximum recursion depth exceeded while
getting the str of an object
```

17.7   Parsing binary trees

Unfortunately, most examples which are used to explain recursion are so simple that they can be solved more naturally without recursion. For example, the factorial from the previous subsection can be calculated non-recursively using a simple `for` loop:

```python
def factorial(n):
    result = 1
    for i in range(2, n+1):
        result *= i
    return result

print(factorial(5))
```
```
120
```

The beauty and power of recursion shows itself in more advanced applications. One of such applications are binary trees.
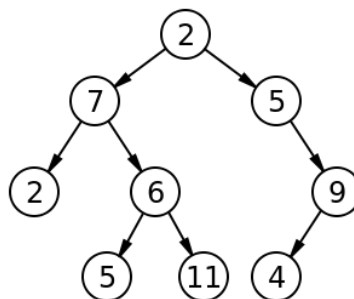

Fig. 79. Sample binary tree.

280

A binary tree "grows" from the top down. The circles are called *nodes*. Each node contains a number, and it can have up to two branches (hence the name *binary tree*). The number of levels in called *depth* – the sample binary tree in Fig. 79 has depth 4. Usually, the shape and depth of the binary tree are unknown.

One way to represent such a binary tree in Python is using embedded lists. Each node is a three-item list where the first item is the number, the second item is the left branch, and the third item is the right branch. An empty list is used for an empty branch. Let's show some examples.

In Fig. 79, the node with the number 2 is without branches, and therefore it is represented as [2, [], []]. The other nodes without branches just differ in the number. The node with the number 9 only has the left branch, and therefore it is represented by the list [9, [4, [], []], []]. The node with the number 6 has both branches, and therefore it has the form [6, [5, [], []], [11, [], []]]. The entire tree can then be written as [2, [7, [2, [], []], [6, [5, [], []], [11, [], []]]], [5, [], [9, [4, [], []], []]]].

Now imagine that our task is to parse such a binary tree and add all the numbers. This would be extremely difficult without recursion. On the other hand, a recursive algorithm is very simple:

- The list (let's call it t) has three items t[0] (the number), t[1] (the list representing the left branch), and t[2] (the list representing the right branch).
- If t is empty, return 0.
- Otherwise add to the value t[0] all values in the left branch t[1] and all values in the right branch t[2], and return the result.

This is the corresponding code. Take a minute to verify that indeed all the values in the tree from Fig. 79 add up to 51!

```python
def addtree(t):
    """
    Add recursively all values in a binary tree.
    """
    if t != []: return t[0] + addtree(t[1]) + addtree(t[2])
    return 0

# Main program:
T = [2, [7, [2, [], []], [6, [5, [], []], [11, [], []]]], [5,
    [], [9, [4, [], []], []]]]
print(addtree(T))
```
```
51
```

18 Decorators

The goal of this Section is to introduce *decorators* – elegant wrappers which can dynamically alter a function, method or even a class without changing their source code.

18.1 Introduction

We already know how to write simple wrappers from Subsections 8.8 and 8.9. Decorators, however, take wrapping to an entirely new level. They are very powerful and simple at the same time. You will like them for sure.

Before we begin, let's recall how a wrapper works. Say that we have a function `myfun` which does something. A wrapper is another function which calls `myfun` but it does some additional things before and/or after the call. At least that's what you know from Section 8. Now let's see what is possible with decorators.

For simplicity, our function `myfun(n)` will just add numbers `0, 1, ..., n` and return the result. Here it is:

```python
def myfun(n):
    s = 0
    for i in range(n+1):
        s += i
    return s
```

Let's make a sample call:

```python
res = myfun(10**7)
print("Result:", res)
```
```
Result: 50000005000000
```

So far so good? OK. Now let's say that we want to time it. All we need to do is place one additional line, a decorator `@report_time` before the function definition:

```python
@report_time
def myfun(n):
    s = 0
    for i in range(n+1):
        s += i
    return s
```

The call to the decorated function will now provide additional timing information:

```
res = myfun(10**7)
print("Result:", res)
```

```
Time taken: 0.80092 sec.
Result: 50000005000000
```

Don't worry, in the next subsection we will tell you how the decorator was created. But before that let's see another example.

Let's say that we want to add debugging information to the function – whenever the function is called, we want to know about it, and also what value(s) it returns. This can be done by adding another decorator @debug to it:

```
@debug
def myfun(n):
    s = 0
    for i in range(n+1):
        s += i
    return s
```

Calling the decorated function now will provide debugging information:

```
res = myfun(10**7)
print("Result:", res)
```

```
Entering function 'myfun'.
The function returns 50000005000000
Leaving function 'myfun'.
Result: 50000005000000
```

And finally, what if we wanted to both time and debug the function? Easy - just add both decorators!

```
@report_time
@debug
def myfun(n):
    s = 0
    for i in range(n+1):
        s += i
    return s
```

Here is the result of calling the twice decorated function:

```
res = myfun(10**7)
print("Result:", res)
```
```
Entering function 'myfun'.
The function returns 50000005000000
Leaving function 'myfun'.
Time taken: 0.8674440000000001 sec.
Result: 50000005000000
```

The simplicity of use of decorators is amazing. Now let's learn how to create them!

18.2   Building a timing decorator

In order to built decorators, we will need some things you learned in Section 8:

- How to write functions which accept a variable number of arguments (Subsections 8.10 - 8.16).
- How to define local functions within functions (Subsection 8.17).
- How to return a function from another function (Subsection 8.23).
- How to store functions in variables (Subsection 8.24).

This looks like a lot, but don't worry. Even if you are not familiar with all these concepts, you may read on and chances are that you will be able to understand. We will build the timing decorator in several steps.

First, as you know from Subsection 11.4, the function `process_time` from the `time` library can be used to time functions. Let's use it to time our function `myfun`:

```
import time
start = time.process_time()
val = myfun(10**7)
end = time.process_time()
print("Time taken:", end - start, "sec.")
```
```
Time taken: 0.8674440000000001 sec.
```

As the next step, let's enclose the timing functionality in a new function named for example `func_wrapper`. This function is the extension of the original function `myfun` that matters. Notice that it returns the result `val` returned by `myfun`. In the main program, we will call the wrapper instead of the original function `myfun` to obtain the

284

timing info:

```python
def myfun(n):
    s = 0
    for i in range(n+1):
        s += i
    return s

import time
def func_wrapper(n):
    start = time.process_time()
    val = myfun(n)
    end = time.process_time()
    print("Time taken:", end - start, "sec.")
    return val

res = func_wrapper(10**7)
print("Result:", res)
```

```
Time taken: 0.804517757 sec.
Result: 50000005000000
```

The next step is a technical trick: We define a new function `report_time(func)` which will accept an arbitrary function `func`, create the timing wrapper `func_wrap` for it, and return it:

```python
import time
def report_time(func):
    def func_wrapper(n):
        start = time.clock()
        val = func(n)
        end = time.clock()
        print("Time taken:", end - start, "sec.")
        return val
    return func_wrapper
```

It is important to realize that `report_time(myfun)` is a function – the one that is returned from `report_time`. This is the wrapped `myfun` which prints timing information. To break it down, it can be called as follows:

```
wrapped_myfun = report_time(myfun)
res = wrapped_myfun(10**7)
print("Result:", res)
```
```
Time taken: 0.804517757 sec.
Result: 50000005000000
```

And now the last step - we will replace `wrapped_myfun` with `myfun` which effectively redefines the original function with its timing wrapper. Now the function can be called as `myfun`, but actually executed is its timing wrapper:

```
myfun = report_time(myfun)
res = myfun(10**7)
print("Result:", res)
```
```
Time taken: 0.804517757 sec.
Result: 50000005000000
```

That's it, we are done! The line `@report_time` which is added in front of the definition of the function `myfun` is just syntactic sugar – a cosmetic replacement of the line `myfun = report_time(myfun)`. Here is the code:

```
import time
def report_time(func):
    def func_wrapper(n):
        start = time.clock()
        val = func(n)
        end = time.clock()
        print("Time taken:", end - start, "sec.")
        return val
    return func_wrapper


@report_time
def myfun(n):
    s = 0
    for i in range(n+1):
        s += i
    return s


res = myfun(10**7)
print("Result:", res)
```

```
Time taken: 0.804517757 sec.
Result: 50000005000000
```

Oh, and one more thing: The decorator, as defined now, will only work for functions which accept one argument. It is easy to adjust it to work for functions which accept any number of arguments (or no arguments) by replacing the two occurrences of n with *args, **kwargs as follows:

```python
import time
def report_time(func):
    def func_wrapper(*args, **kwargs):
        start = time.clock()
        val = func(*args, **kwargs)
        end = time.clock()
        print("Time taken:", end - start, "sec.")
        return val
    return func_wrapper

@report_time
def myfun(n):
    s = 0
    for i in range(n+1):
        s += i
    return s

res = myfun(10**7)
print("Result:", res)
```
```
Time taken: 0.804517757 sec.
Result: 50000005000000
```

In summary, the most important step is to define the wrapper func_wrap – and this is easy. The rest are only technical details.

18.3   Building a debugging decorator

In this subsection we will define the debugging decorator which we introduced in Subsection 18.1. The procedure is much the same as for the timing decorator, so let's just show the result:

```
def debug(func):
    def func_wrapper(*args, **kwargs):
        print("Entering function '" + func.__name__ + "'.")
        val = func(*args, **kwargs)
        print("The function returns", val)
        print("Leaving function '" + func.__name__ + "'.")
        return val
    return func_wrapper


@debug
def myfun(n):
    s = 0
    for i in range(n+1):
        s += i
    return s


res = myfun(10**7)
print("Result:", res)
```

```
Entering function 'myfun'.
The function returns 50000005000000
Leaving function 'myfun'.
Result: 50000005000000
```

Note that since `function` is a class, object introspection was used to obtain its name via `func.__name__` (see Subsection 16.7 for more details).

18.4 Combining decorators

Let's return for a moment to the function `myfun` and the decorator `report_time` from Subsection 18.2, and the debugging decorator `debug` from Subsection 18.3. As you know, the real line of code which stands behind

```
@debug
```

is

```
myfun = debug(myfun)
```

It redefines the original function to its debugging wrapper. There is nothing to stop us from redefining the wrapped function once more to its timing wrapper:

```
myfun = debug(myfun)
myfun = report_time(myfun)
```

At this moment, the function is wrapped twice. Using the decorator syntax, the same can be written more beautifully as

```
@report_time
@debug
def myfun(n):
    ...
```

This can be done with as many wrappers as needed. Just notice the order – the decorator listed first is the most outer one – applied last.

18.5   Decorating class methods

Class methods can be decorated just like standard functions. While every class method has the reference `self` to the current object as its first parameter, this is taken care of by the variable parameter list `*args, **kwargs`. Hence, here are our timing and debugging wrappers again, with no changes required for wrapping class methods:

```
import time
def report_time(func):
    def func_wrapper(*args, **kwargs):
        start = time.clock()
        val = func(*args, **kwargs)
        end = time.clock()
        print("Time taken:", end - start, "sec.")
        return val
    return func_wrapper
```

```
def debug(func):
    def func_wrapper(*args, **kwargs):
        print("Entering function '" + func.__name__ + "'.")
        val = func(*args, **kwargs)
        print("The function returns", val)
        print("Leaving function '" + func.__name__ + "'.")
        return val
    return func_wrapper
```

Application of the wrappers is very easy using the same decorator syntax as for functions. Do you still recall the class `Circle` which we defined In Subsection 14.8? In order to apply the debugging wrapper to its methods `area` and `perimeter`, just insert the line `@debug` in front of each. And to apply the timing wrapper to the method `draw`, use the line `@report_time`:

```
class Circle:
    """
    Circle with given radius R and center point (Cx, Cy).
    Default plotting subdivision: 100 linear edges.
    """
    def __init__(self, r, cx, cy, n = 100):
        """
        The initializer adds and initializes the radius R,
        and the center point coordinates Cx, Cy.
        It also creates the arrays of X and Y coordinates.
        """
        self.R = r
        self.Cx = cx
        self.Cy = cy
        # Now define the arrays of X and Y coordinates:
        self.ptsx = []
        self.ptsy = []
        da = 2*np.pi/self.n
        for i in range(n):
            self.ptsx.append(self.Cx + self.R * np.cos(i * da))
            self.ptsy.append(self.Cy + self.R * np.sin(i * da))
        # Close the polyline by adding the 1st point again:
        self.ptsx.append(self.Cx + self.R)
        self.ptsy.append(self.Cy + 0)
```

```python
    @debug
    def area(self):
        """
        Calculates and returns the area.
        """
        return np.pi * self.R**2

    @debug
    def perimeter(self):
        """
        Calculates and returns the perimeter.
        """
        return 2 * np.pi * self.R

    @report_time
    def draw(self, label):
        """
        Plots the circle using Matplotlib.
        """
        plt.axis('equal')
        plt.plot(self.ptsx, self.ptsy, label = label)
        plt.legend()
```

18.6   Passing arguments to decorators

It is possible to pass arguments to a decorator by creating a wrapper around it. The sole purpose of this extra wrapper is to take the arguments and pass them to the original decorator. Let's illustrate this on our decorator `report_time`. This decorator measures process time by default. Imagine that we want to be able to pass an argument into it to choose whether wall time or process time will be used. The name of the extra wrapper will be `timer(walltime=False)`:

291

```python
import time
def timer(walltime=False):
    def report_time(func):
        def func_wrapper(*args, **kwargs):
            if walltime:
                start = time.time()
                print("Measuring wall time.")
            else:
                start = time.process_time()
                print("Measuring process time.")
            val = func(*args, **kwargs)
            if walltime: end = time.time()
            else: end = time.process_time()
            print("Time taken:", end - start, "sec.")
            return val
        return func_wrapper
    return report_time
```

When used without arguments or with `walltime=False`, `timer` will measure process time:

```python
@timer()
def myfun(n):
    s = 0
    for i in range(n+1):
        s += i
    return s

res = myfun(10**7)
print("Result:", res)
```

```
Measuring process time.
Time taken: 0.666784343 sec.
Result: 50000005000000
```

However, when used with `walltime=True`, it will measure wall time:

```python
@timer(walltime=True)
def myfun(n):
    s = 0
    for i in range(n+1):
        s += i
    return s

res = myfun(10**7)
print("Result:", res)
```
```
Measuring wall time.
Time taken: 0.8023223876953125 sec.
Result: 50000005000000
```

18.7   Debugging decorated functions

As you know from Subsection 8.27, useful information can be retrieved from functions via their attributes. However, the decorator redefines the original function with its wrapper. This causes a problem for debugging because the attributes such __name__, __doc__ (docstring) or __module__ are now those of the wrapper and not of the original function. Look at this:

```python
print(myfun.__name__)
```
```
func_wrapper
```

Fortunately, this can be fixed using the function `wraps` from the `functools` module as follows:

```python
import functools as ft

import time
def timer(walltime=False):
    def report_time(func):
        @ft.wraps(func)
        def func_wrapper(*args, **kwargs):
            ...
        return func_wrapper
    return report_time
```

Now the attributes of the function `myfun` are correct:

```
print(myfun.__name__)
myfun
```

18.8   Python Decorator Library

The decorators are still in active development, so it is always a good idea to search for the newest information online. Besides a number of various online tutorials, we would like to bring to your attention a (uncesored) collection of wrappers at

```
https://wiki.python.org/moin/PythonDecoratorLibrary
```

This collection contains various interesting decorators, including (we just selected a few):

– Counting function calls,
– type enforcement (accepts/returns),
– profiling / coverage analysis,
– line tracing individual functions,
– synchronization,
– asynchronous calls,
– etc.

19 Selected Advanced Topics

The goal of this Section is to introduce selected advanced programming techniques in Python. Since Python is a modern language which is still evolving, we recommend that after reading about a technique here, you also search it on the web for possible updates.

19.1 Objectives

We will learn about:

– Maps and filters.
– The built-in function `reduce`.
– Creating shallow and deep copies.

19.2 Maps

You already know list comprehension well from Subsections 7.29 - 7.32. You also know that comprehension works for any iterable, not only for lists. The difference between a map and a comprehension is rather subtle, and many programmers prefer comprehension for being more Pythonic. But it is a good idea to learn about maps anyway because there are many other people who use them.

The built-in function `map(fun, iterable)` applies the function `fun` to each item of the iterable (text string, list, tuple etc.). The function `fun` can be an anonymous lambda expression or a standard function. For anonymous lambda expressions review Subsections 8.18 - 8.26. The result returned by the `map` function is a Map Object which needs to be cast back to an iterable.

In the following example, the function `fun` is given as a lambda expression, and the iterable is a list of numbers [0, 1, ..., 9] created via `range(10)`:

```
numbers = range(10)
squares = map(lambda x: x**2, numbers)
print(list(squares))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Let's take a minute to do the same using list comprehension:

```
numbers = range(10)
squares = [x**2 for x in numbers]
print(squares)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

You can see that a cast to list is not needed in this case. As another example, let's show that map can use any function, not only lambda expressions:

```
def fn(a):
    return a**2

numbers = range(10)
squares = map(fn, numbers)
print(list(squares))
```
```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

And last, let's show that maps can be applied to a different iterable, for example to a text string:

```
text = 'I love maps!'
asciicodes = map(lambda c: ord(c), text)
print(list(asciicodes))
```
```
[73, 32, 108, 111, 118, 101, 32, 109, 97, 112, 115, 33]
```

19.3 Filters

The built-in function `filter(fun, iterable)` is somewhat similar to `map(fun, iterable)`. The difference is that the function `fun` should return `True` or `False`. It is applied to all items of the iterable, and all items for which it does not return `True` are left out. Analogously to `map`, the function `filter` returns a Filter Object which needs to be cast to an iterable. Here is an example where the filter picks from a list of text strings all items which begin with a capital letter:

```
words = ['My', 'name', 'is', 'Nobody']
result = filter(lambda w: w[0] == w[0].upper(), words)
print(list(result))
```
```
['My', 'Nobody']
```

Again, the same can be done using list comprehension, which moreover is a bit simpler because it does not involve the cast to list:

```
words = ['My', 'name', 'is', 'Nobody']
result = [w for w in words if w[0] == w[0].upper()]
print(result)
```
```
['My', 'Nobody']
```

19.4 Function `reduce()`

Function `reduce(fun, iterable)` from the `functools` module is very handy for cumulative operations with lists. By a *cumulative operation* we mean an operation which takes all list items, one by one, and does something with them. For example - adding all items, multiplying all items, concatenating all items (if they are text strings), etc. The following example adds all items in a list:

```
import functools as ft
L = [2, 4, 5, 7, 8, 9]
ft.reduce(lambda x, y: x+y, L)
```
```
35
```

The example is not self-explanatory, so let's go through it one step at a time:

1. The first x and y to go into the lambda are the first two items in the list: x=1, y=2, and x+y yields 3.
2. The next x and y to go into the lambda are the last result 3 and the next list item (also 3): x=3, y=3, and x+y = 6.
3. The next x and y to go into the lambda are the last result 6 and the next list item 4: x=6, y=4, and x+y = 10.
4. Last step: x=10, y=5, and x+y = 10.

Of course, the same can be done the hard way:

```
L = [2, 4, 5, 7, 8, 9]
s = 0
for n in L:
    s += n
print(s)
```
```
35
```

However, using `reduce` combined with an anonymous function shows that you have a different level of knowledge of Python. Here is one more example:

Example 2: Logical product of a Boolean list

Imagine that we have a list L of Boolean values (`True` or `False`). By a *logical product* of all items we mean `L[0] and L[1] and ... and L[n-1]`. The result will be `True` if all values in the list are `True`, and `False` otherwise. This can be nicely done with the help of `reduce`:

```python
import functools as ft
L = [True, True, True, False, True]
ft.reduce(lambda x, y: x and y, L)
```

```
False
```

And, let's show one last example:

Example 3: Area below the graph of a function

The area below the graph of a function $f(x)$ can be obtained by integrating $f(x)$ between the points a and b. While integration is an advanced topic in calculus, one does not need any calculus to do this in Python. One just needs to know how to calculate the area of a rectangle. Then one constructs thin columns under the graph of the function, and adds their areas together - that's it! BTW, the width of the individual columns is $h = (b - a)/n$ and the array of function values at the grid points is $f(X)$.
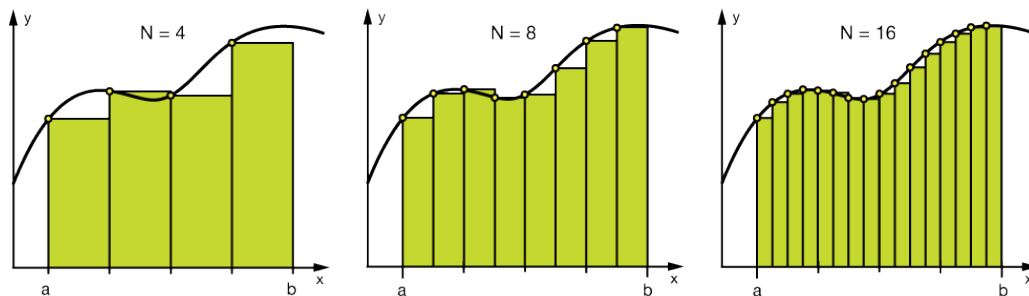

Fig. 80. Calculating the area below the graph of a function.

Here is the code for a sample function $f(x) = \sin(x)$ in the interval $(0, \pi)$:

```
# Import Numpy:
import numpy as np

# Sample function f(x) = sin(x):
f = np.sin

# End points on the X axis:
a = 0
b = np.pi

# Number of columns:
n = 10

# Equidistant grid between a and b:
X = np.linspace(a, b, n+1)

# Width of the columns:
h = (b - a) / n

# Add the areas of all columns:
import functools as ft
ft.reduce(lambda x, y: x + y*h, f(X))
```
```
1.9835235375094544
```

The exact value of the area is 2. So, obtaining 1.9835235375094544 with just 10 columns
is not bad at all. When one increases the number of columns, the result becomes more
accurate: With $n = 100$ one obtains 1.9998355038874436, and increasing n even further
to $n = 1000$ yields 1.9999983550656637.

19.5   Shallow and deep copying

Before we start talking about shallow and deep copies, recall that copying immutable
objects (numerical variables, text strings, tuples, ...) is easy because just assigning them
to a new variable creates a new copy automatically. (Mutability was discussed in Sub-
section 7.33.)

For illustration, in the following example we assign 5 to the variable a, then assign
a to b, and change a afterwards. Note that b is still 5:

```
a = 5
b = a
a = 4
print(b)
```
```
5
```

We will see the same for text strings which are also immutable:

```
a = 'Asterix'
b = a
a = 'Obelix'
print(b)
```
```
Asterix
```

But the situation is completely different when we work with a mutable object (list, dictionary, set, class, ...):

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

As you can see, changing object `a` caused the same change to occur in object `b`. The reason is that objects `a` and `b` are at the same place in the memory, so altering one will automatically alter the other (this was explained already in Subsection 7.13).

Shallow copying

The difference between shallow and deep copying is only relevant for compound objects (such as lists that contain other lists, lists which contain instances of classes, etc.). A *shallow copy* with create a new object, but it will not create recursive copies of the other objects which are embedded in it. In other words, shallow copy is only one level deep. This is best illustrated on an example.

Let us create a list of lists named `a`, make a shallow copy `b`, and then alter one of the lists contained in the original list `a`. As a result, the corresponding list contained in the shallow copy `b` will change as well:

```
a = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
b = list(a)   # make a shallow copy
a[0].append(-1)
print(b)
```
```
[[1, 2, 3, -1], [4, 5, 6], [7, 8, 9]]
```

Copying the list a via

```
b = a[:]
```

leads to the same result (it also creates a shallow copy). For completeness, a shallow copy of a dictionary a can be obtained via

```
b = dict(a)
```

and a shallow copy of a set a can be obtained by typing:

```
b = set(a)
```

Deep copying

In contrast to shallow copying, a deep copy constructs a new compound object and then, recursively, inserts into it of the copies of the objects found in the original. The best way to do this is to use the copy module in the Python standard library. Let's return to the previous example with the list of lists, replacing the shallow copy with a deep one:

```
import copy
a = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
b = copy.deepcopy(a)   # make a deep copy
a[0].append(-1)
print(b)
```
```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

As you can see, this time altering the object a did not cause any changes in the object b. Finally, let us remark that the copy module also provides a function copy to create shallow copies of objects.