Tools for Scientific Computing

Release 0.3

Gert-Ludwig Ingold

October 07, 2025

CONTENTS

CHAPTER ONE

INTRODUCTION

The daily routine in scientific work is characterized by careful checks and detailed documentation. Experimentalists calibrate their apparatuses and note all relevant aspects of an experiment in a lab book, either on paper or digitally. Theorists check their calculations and consider limiting cases to ensure the correctness of their results. On the other hand, it can frequently be observed that not the same standards are applied to scientific computational work, even though appropriate tools exist. Furthermore, knowledge of these tools can be an important asset when looking for a job outside of academia.

The present lecture notes cover a number of tools useful in scientific computing. In view of the aspects just discussed, we specifically mention the use of a version control system like Git introduced in Chapter 2, testing of code discussed in Chapter 3, and the documentation of code covered in Chapter 6. The discussion of the version control system Git is completely independent of the specific programming language used. On the other hand, the tools covered in the chapter on testing – doctests and the `pytest` package – are specific to the programming language Python. However, the basic ideas should be transferable to other programming languages. For the purpose of documentation, we introduce the Sphinx documentation generator. Despite its origin as a tool to generate the Python documentation, it is very flexible and can be used also for other programming languages. In fact, even the present lecture notes where produced with Sphinx.

The other chapters are concerned more with the performance of programs. This is an important issue when using Python as a programming language. Python has gained a significant popularity in scientific computing despite its reputation of not being the fastest language. However, there exist a variety of approaches to bring Python up to speed. One possibility is the use of scientific numerical libraries like NumPy and SciPy which are discussed in Chapter 4. This chapter is rather specific to Python apart from the aspect of illustrating the use of numerical libraries.

Chapter 5 is devoted to the run-time analysis of code with the aim of identifying the most time-consuming parts of a program. Here again, the tools are specific to Python but the concepts can be applied to other languages as well. Finally, Chapter 7 gives a brief introduction to aspects of parallel computing in Python. In view of the existence of the global interpreter lock, this discussion is rather specific to Python. In addition, possibilities offered by just-in-time compilers to increase the performance of programs are highlighted.

These lecture notes present material covered in a one-semester method course *Tools for scientific computing* taught at the Universität Augsburg consisting of a two-hour lecture and four-hour practical work per week. The material is thus intended for a total of 30 hours of lectures.

The sources of the lecture notes are publicly available on Github at https://github.com/gertingold/tools4scicomp. Suggestions for improvements through Github issues or pull request are welcome.

CHAPTER

TWO

VERSION CONTROL WITH GIT

2.1 Why version control?

A program is rarely written in one go but rather evolves through a number of stages where the code is improved, for example by fixing errors or improving its functionality. In this process, it is generally a good idea to keep old versions. Occasionally, one has an apparently good idea of how to improve a program, only to find out somewhat later that it was not such a good idea after all. Without having the original version available, one might have a hard time going back to it.

Often, old versions are kept in an informal way by inventing filenames to distinguish different versions. Unless one strictly abides by a naming convention, sooner or later one will be unable to identify the stage of development corresponding to a given file. Things become even more difficult if more than one developer is involved.

The potential loss of a working program is not the only motivation to keep a history of program versions. Suppose that a version of the program is used to compute scientific data and suppose that the program is further developed, e.g. by adding functionality. One might think that it is unnecessary to keep the old version. However, imagine that at some point it turns out that the program contains a mistake resulting in erroneous data. In such a situation, it may become essential to know whether the data obtained previously are affected by the mistake or not. Do the data have to be discarded or can continue to use them? If the version of the code used obtain the data is documented, this question can be decided. Otherwise, one probably could not trust the old data anymore.

Another reason of keeping the history of a program is to document its evolution. The motivation for design decisions can be made transparent and even bad decisions could be kept for further reference. In a scenario where code is developed by several or even a large number of people, it might be desirable to know who is to be praised or blamed for a certain piece of code. Version control systems often support collaborative development by providing tools to discuss code before accepting the associated changes and by the possibility of easily going back to an older version. In this way, trying out new ideas can be encouraged.

A version control system storing the history of a software project is clearly an invaluable tool. This insight is anything but new and indeed as early as in the 1970s, a first version control system, SCCS (short for source code control system), was developed. Later systems in wide use include RCS (revision control system) and CVS (concurrent versions system), both developed in the last century, Subversion developed around the turn of the century and more recent systems like Git, Mercurial and Bazaar.

Here, we will discuss the version control system Git created by Linus Torvalds in 2005. Its original purpose was to serve in the development of the Linux kernel. In order to make certain aspects of Git better understandable and to highlight some of its advantages, we will consider in the following section in some more detail different approaches to version control.

2.2 Centralized and distributed version control systems

Often software is developed by a team. For the sake of illustration let us think of a number of authors working jointly on a text. In fact, scientific manuscripts are often written in (La)TeX which can be viewed as a specialized programming language. Obviously, there exists a probability that persons working in parallel on the text will make incompatible changes. Inevitably, at some point the question arises which version should actually be accepted. We will encounter such situations later as so-called merge conflicts.

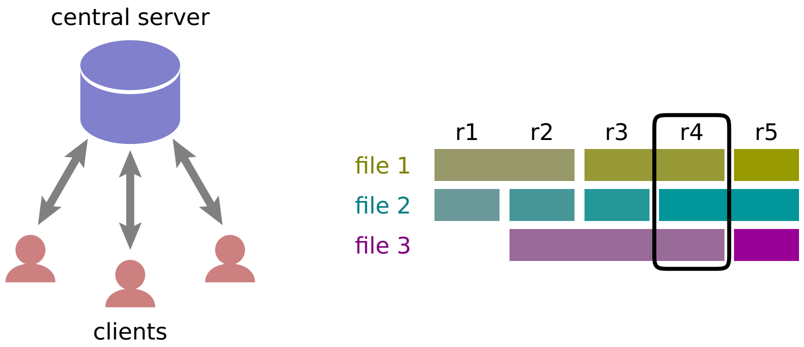


Figure 2.1: A centralized version control system contains a well defined set of files at any given moment in time which can be referred to by a sequential revision number.

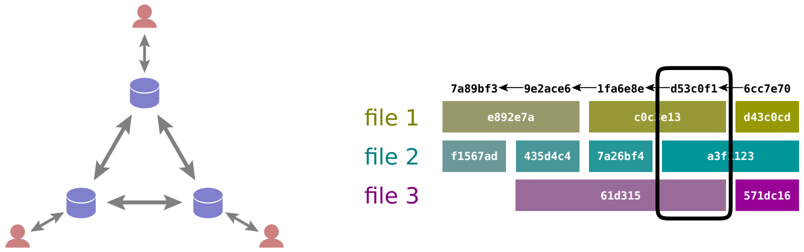


Figure 2.2: In a distributed version control system each user keeps file versions in a local repository and exchanges versions with other repositories when needed. As a consequence no global sequential history can be defined.

Early version control systems like RCS avoided such conflicts by a locking technique. In order to change the text or code, it was necessary to first lock the corresponding file, thus preventing other persons from modifying the same file at the same time. Unfortunately, this technique tends to impede parallel development. For our example of manuscript, it is perfectly fine if several persons work in parallel on different sections. Therefore, locking has been found not to be a good idea and it is not substitute for communication between team members about who is doing what.

More modern version control systems are designed to favor collaboration within a team. There exist two different approaches: centralized version control systems on the one hand and distributed version control systems on the other hand. The version control system Git, which we are going to discuss in more detail in this chapter, is a distributed version control system. In order to better understand some of its aspects, it is useful to contrast it with a centralized version control system like Subversion.

More modern version control systems are designed to favor collaboration within a team. There exist two different approaches: centralized version control systems on the one hand and distributed version control systems on the other hand. The version control system Git which we are going to discuss in more detail in this chapter is a distributed version control system. In order to better understand some of its aspects, it is useful to contrast it with a centralized version control system like Subversion.

The basic structure of a centralized version control system is depicted in the left part of Figure 2.1. One or more developers, referred to as clients here, exchange code versions via the internet with a central server. At any moment of time, the server contains a definite set of files, i.e. a revision which is numbered sequentially as indicated in the right part of Figure 2.1. From one revision to the next, files can change or remain unchanged and files can be added or removed. The price to pay for this simple sequential history is that an internet connection and a working server is needed in order to create a new revision. A developer cannot create new revisions of the code while working off-line, an important drawback of centralized version control systems.

As an alternative, one can use a distributed version control system which is schematically represented in Figure 2.2. In such a setup, each developer keeps his or her own versions in a local repository and exchanges files with other repositories when needed. Due to the local repository, one can create a new version at any time, even in the absence of an internet connection. On the other hand, there exist local version histories and the concept of a global sequential revision numbering scheme does not make sense anymore. Instead, Git uses hexadecimal hash values to identify versions of individual files and sets of files, so-called commits, which reflect changes in the code base. The main point to understand here is that the seemingly natural sequential numbering scheme cannot work in a distributed

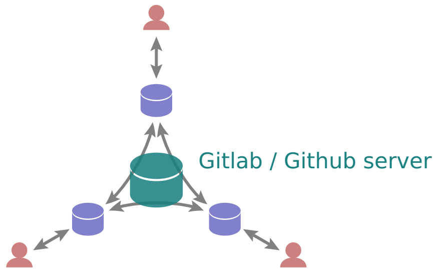


Figure 2.3: A typical setup for the distributed version control system Git uses a central server to exchange versions between local repositories.

version control system.

In most cases, a distributed version control system is not implemented precisely in the way presented in Figure 2.2 as it would require communication between potentially a large number of local repositories. A setup like the one shown in Figure 2.3 is typical instead. The important difference as compared to the centralized version control system displayed in Figure 2.1 consists in the existence of local repositories where individual developers can manage their code versions even if disconnected with the central server. The difference is most obvious in the case of a single developer. Then, a local repository is completely sufficient and there is no need to use another server.

A central server for the use with the version control system Git can be set up based on GitLab. Many institutions are running a GitLab instance[1]. In addition, there exists the GitHub service at github.com. GitHub is popular among developers of open software projects for which it provides repositories free of charge. Private repositories can be obtained at a monthly rate, but there exists also the possibility to apply for temporary free private repositories for academic use. In later sections, when discussing collaborative code development with Git, we will specifically address GitLab, but the differences to GitHub are usually minor.

In the following sections, we will start by explaining the use of Git in a single-user scenario with a local repository. This knowledge also forms the basis for work in a multi-developer environment using GitLab or GitHub.

2.3 Getting help

Before starting to explore the version control system Git, it is useful to know where one can get help. Generally, Git tries to be quite helpful even on the command line by adding useful hints to its output. As the general structure of a Git command starts with `git <command>`, one can ask for help as follows:

```
$ git help
usage: git [--version] [--help] [-C <path>] [-c <name>=<value>]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p | --paginate | -P | --no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           <command> [<args>]

These are common Git commands used in various situations:

start a working area (see also: git help tutorial)
   clone             Clone a repository into a new directory
   init              Create an empty Git repository or reinitialize an existing one

work on the current change (see also: git help everyday)
   add               Add file contents to the index
   mv                Move or rename a file, a directory, or a symlink
   restore           Restore working tree files
   rm                Remove files from the working tree and from the index
```

(continues on next page)

[1] The computing center of the University of Augsburg is running a GitLab server at `git.rz.uni-augsburg.de` which is accessible to anybody in possession of a valid user-ID of the computing center.

(continued from previous page)

```
   sparse-checkout   Initialize and modify the sparse-checkout

examine the history and state (see also: git help revisions)
   bisect            Use binary search to find the commit that introduced a bug
   diff              Show changes between commits, commit and working tree, etc
   grep              Print lines matching a pattern
   log               Show commit logs
   show              Show various types of objects
   status            Show the working tree status

grow, mark and tweak your common history
   branch            List, create, or delete branches
   commit            Record changes to the repository
   merge             Join two or more development histories together
   rebase            Reapply commits on top of another base tip
   reset             Reset current HEAD to the specified state
   switch            Switch branches
   tag               Create, list, delete or verify a tag object signed with GPG

collaborate (see also: git help workflows)
   fetch             Download objects and refs from another repository
   pull              Fetch from and integrate with another repository or a local branch
   push              Update remote refs along with associated objects

'git help -a' and 'git help -g' list available subcommands and some
concept guides. See 'git help <command>' or 'git help <concept>'
to read about a specific subcommand or concept.
See 'git help git' for an overview of the system.
```

Information on a specific command is obtained by means of `git help <command>`.

Furthermore, Git provides a number of guides which can be read in a terminal window. A list of available guides can easily be obtained:

```
$ git help -g

The common Git guides are:
   attributes          Defining attributes per path
   cli                 Git command-line interface and conventions
   core-tutorial       A Git core tutorial for developers
   cvs-migration       Git for CVS users
   diffcore            Tweaking diff output
   everyday            A useful minimum set of commands for Everyday Git
   glossary            A Git Glossary
   hooks               Hooks used by Git
   ignore              Specifies intentionally untracked files to ignore
   modules             Defining submodule properties
   namespaces          Git namespaces
   repository-layout   Git Repository Layout
   revisions           Specifying revisions and ranges for Git
   submodules          Mounting one repository inside another
   tutorial            A tutorial introduction to Git
   tutorial-2          A tutorial introduction to Git: part two
   workflows           An overview of recommended workflows with Git

'git help -a' and 'git help -g' list available subcommands and some
```

(continues on next page)

(continued from previous page)

```
concept guides. See 'git help <command>' or 'git help <concept>'
to read about a specific subcommand or concept.
See 'git help git' for an overview of the system.
```

For a detailed discussion of Git, the book *Pro Git* by Scott Chacon and Ben Straub is highly recommended. Its second edition is available in printed form online where also a PDF version can be downloaded freely. By the way, the book *Pro Git* as well as the present lecture notes have been written under version control with Git.

2.4 Setting up a local repository

The use of a version control system is not limited to large software projects but makes sense even for small individual projects. A prerequisite is the installation of the Git software which is freely available for Windows, MacOS and Unix systems from git-scm.com. This Git installation can be used for all projects to be put under version control and we assume in the following that Git is already installed on the computer. Even though some graphical user interfaces exist, we will mostly discuss the use of Git on the command line.

Putting a new project under version control with Git is easy. Once a directory exists in which the code will be developed, one initializes the repository by means of:

```
$ git init
```

Note that the dollar sign represents the command line prompt and should not be typed. Depending on your operating system setup, the dollar could be replaced by some other character(s). Initializing a new repository in this way will create a hidden subdirectory called `.git` in the directory where you executed the command. The directory is hidden to avoid that it is accidentally deleted.

> **Attention**
>
> Never delete the directory `.git` unless you really want to. You will lose the complete history of your project if you did not backup the project directory or synchronized your work with a GitLab server or GitHub. Removing the project directory will remove the subdirectory `.git` as well.

The newly created directory contains a number of files and subdirectories:

```
$ ls -a .git
.  ..  branches  config  description  HEAD  hooks  info  objects  refs
```

Refrain from modifying anything here as you might mess up files and in this way lose parts or all of your work.

After having initialized your project, you should let Git know about your name and your email address by using the following commands:

```
$ git config --global user.name <your name>
$ git config --global user.email <your email>
```

where the part in angle brackets has to be replaced by the corresponding information. Enclose the information, in particular your name, in double quotes if it contains one or more blanks like in the following example:

```
$ git config --global user.name "Gert-Ludwig Ingold"
```

This information will be used by Git when new or modified files are committed to the repository in order to document who has made the contribution.

If you have globally defined your name and email address as we did here, you do not need to repeat this step for each new repository. However, you can overwrite the global configuration locally. This might be useful if you intend to use a different email address for a specific project.

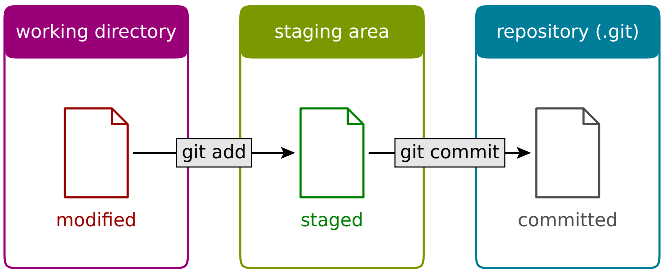


Figure 2.4: The transfer of a file to the repository is a two-step process. First one or more files are added to the staging area. In a second step, the files are committed to the repository.

There are more aspects of Git which can be configured and which are documented in Section 8.1 of the Git documentation. The presently active configuration can be inspected by means of:

```
$ git config --list
```

For example, you might consider setting `core.editor` to your preferred editor.

2.5 Basic workflow

A basic step in managing a project under version control is the transfer of one or more new or modified files to the repository where all versions together with metainformation about them is kept. What looks like a one-step process is actually done in Git in two steps. For beginners, this two-step process often gives rise to confusion. We therefore go through the process by means of an example and make reference to Figure 2.4 where the two-step process is illustrated. A convenient way to check the status of the project files is the command `git status`. When working with Git, you will use this command often to make sure that everything works as expected or to remind yourself of the status of the project files.

Suppose that we have just initialized our Git repository as explained in the previous section. Then, Git would report the following status:

```
$ git status
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

The output first tells us that we are on a branch called `master`[2]. Later, we will discuss the concept of branches and it will be useful to know this possibility of finding out the current branch. For the moment, we can ignore this line. Furthermore, Git informs us that we not committed anything yet so that the upcoming commit would be the initial one. However, since we have not created any files, there is nothing to commit. As promised earlier, Git tries to be helpful and adds some information about what we could do. Obviously, we first have to create a file in the project directory.

So let us go ahead and create a very simple Python file:

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

Now, the status reflects the fact that a new file `hello.py` exists:

```
$ git status
On branch master

No commits yet
```

(continues on next page)

[2] On Github, the default branch nowadays is called `main` instead of `master`.

(continued from previous page)

```
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        hello.py

nothing added to commit but untracked files present (use "git add" to track)
```

Git has detected the presence of a new file but it is an untracked file which will basically be ignored by Git. As we ultimately want to include our small script `hello.py` into our repository, we follow the advice and add the file. According to Figure 2.4 this corresponds to moving the file to the so-called staging area, a prerequisite to ultimately committing the file to the repository. Let us also check the status after adding the file:

```
$ git add hello.py
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   hello.py
```

Note that Git tells us how we could revert the step of adding a file in case of need. Having added a file to the staging area does not mean that this file has vanished from our working directory. As you can easily check, it is still there.

At this point it is worth emphasizing that we could collect several files in the staging area. We could then transfer all files to the repository in one single commit. Committing the file to the repository would be the next logical step. However, for the sake of illustration, we want to first modify our script. Our new script could read

```
for n in range(3):
    print("Hello world!")
```

The status now has changed to:

```
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   hello.py

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   hello.py
```

It reflects the fact that now there are two versions of our script `hello.py`. The section “Changes to be committed” lists the file or files in the staging area. In our example, Git refers to the version which we added, i.e. the script consisting of just a simple line. This version differs from the file present in our working directory. This two-line script is listed in the section “Changes not staged for commit”. We could move it to the staging area right away or at a later point in case we want to commit the two versions of the script separately. Note that the most recent version of the script is no longer listed as untracked file because a previous version had been added and the file is tracked now by Git.

Having a file in the staging are, we can now commit it by means of `git commit`. Doing so will open an editor allowing to define a commit message describing the purpose of the commit. The commit message should consist of a single line with preferably at most 50 characters. If necessary, one can add an empty line followed by a longer explanatory text. If a single-line commit message suffices, one can give the message as a command line argument:

```
$ git commit -m 'simple hello world script added'
[master (root-commit) a5b522b] simple hello world script added
 1 file changed, 1 insertion(+)
 create mode 100644 hello.py
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   hello.py

no changes added to commit (use "git add" and/or "git commit -a")
```

Checking the status, we see that our two-line script is still unstaged. We could add it to the staging area and then commit it. Since Git already tracks this file, we can carry out this procedure in one single step. However, this is only possible if we do not wish to commit more than one file:

```
$ git commit -a -m 'repetition of hello world implemented'
[master 011ce76] repetition of hello world implemented
 1 file changed, 2 insertions(+), 1 deletion(-)
(base) gli@gli-tp14-1:~/git_example$ git status
On branch master
nothing to commit, working tree clean
```

Now, we have committed two versions of our script as can easily be verified:

```
$ git log
commit 011ce76b848d6428e900373f177b1f6b2595a524 (HEAD -> master)
Author: Gert-Ludwig Ingold <gert.ingold@physik.uni-augsburg.de>
Date:   Wed Apr 27 15:31:01 2022 +0200

    repetition of hello world implemented

commit a5b522b125baa24f823df389b1b40f28b3a42bee
Author: Gert-Ludwig Ingold <gert.ingold@physik.uni-augsburg.de>
Date:   Wed Apr 27 15:28:41 2022 +0200

    simple hello world script added
```

As we had discussed in Section 2.2 the concept of distributed version control systems does not allow for sequential revision numbers. Our two commits can thus not be numbered as commit 1 and commit 2. Instead, commits in Git are identified by their SHA-1 checksum[3]. The output above lists the hashes consisting of 40 hexadecimal digits for the two commits. In practice, when referring to a commit, it is often sufficient to restrict oneself to the first 6 or 7 digits which typically characterize the commit in a unique way. To obtain idea of how sensitive the SHA-1 hash is with respect to small changes, consider the following examples:

```
$ echo Python|sha1sum
79c4e0b5abbd2f67a369ba6ee0b95438c38eb0cb  -
$ echo python|sha1sum
32886514c2621f81e01024aa84d0f829d2ce1fad  -
```

Now that we know how to commit one or more files, one can raise the question of how often files should be committed. Generally, the rule is to commit often. A good strategy is to combine changes in such a way that they form a logical unit. This approach is particularly helpful if one has to revert to a previous version. If a logical change affects several

[3] SHA-1 is a hash checksum which characterizes an object but does not allow to reconstruct it. Consisting of 160 bits, it allows for $2^{160} \approx 10^{48}$ different values.

files, it is easy to revert this change. If on the other hand, a big commit comprises many logically different changes, one will have to sort out which changes to revert and which ones to keep. Therefore, it makes sense to aim at so-called atomic commits where a commit collects all file changes associated with a minimal logical change[4]. On the other hand, in the initial versions of program development, it often does not make sense to do atomic commits. The situation may change though as the development of the code progresses.

At the end of this section on the basic workflow, we point out one issue which in a sense could already be addressed in the initial setting up of the repository, but which we can motivate only now. Having our previous versions safely stored in the repository, we might be brave enough to refactor our script by defining a function to repeatedly printing a given text. Doing so, we end up with two files

```python
# hello.py
from repeat import repeated_print

repeated_print("Hello world!", 3)
```

and

```python
# repeat.py
def repeated_print(text, repetitions):
    for n in range(repetitions):
        print(text)
```

We verify that the scripts do what they are supposed to do

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

Everything works fine so that we add the two files to the staging area and check the status before committing.

```
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   hello.py
        new file:   repeat.py

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        __pycache__/
```

Everything looks fine except for the fact that there is an untracked directory `__pycache__`. This directory and its content are created during the import of `repeat.py` and should not go into the repository. After all, they are automatically generated when needed. Here, it comes in handy to make use of a `.gitignore` file. Each line in this file contains one entry which defines files to be ignored by Git. For projects based on Python, Git proposes a `.gitignore` file starting with the following lines:

```
# Byte-compiled / optimized / DLL files
__pycache__/
*.py[cod]
*$py.class
```

Lines starting with # are interpreted as comments. The second line excludes the directory `__pycache__` as well as its content. The star in the last two lines can replace any number of characters. The third line will exclude all files ending with `.pyc`, `.pyo`, and `.pyc`. For more details see `git help ignore` and the collection of gitignore files,

[4] Occasionally, one has made several changes which should be separated into different atomic commits. In such a case `git add -p` might come in handy as it allows to select chunks of code while adding a file to the staging area.

in particular Python.gitignore. The .gitignore file should be put under version control as it might develop over time.

2.6 Working with branches

In the previous section, the result of the command git status contained in its first line the information On branch master. The existence of one branch strongly suggests that there could be more branches and this is actually the case. So far, we have been working on the branch which Git had created for us during initialization and which happens to be called master by default. As the use of branches can be very useful, we will discuss them in the following.

In the previous section, we had created a Git repository and made a few commits. Suppose that we have also committed the refactored version of our script as well as the .gitignore file. The history of our repository then looks as follows:

```
$ git log --oneline --graph --decorate --all
* aac6d17 (HEAD -> master) .gitignore for Python added
* 98628ce hello world script refactored
* 011ce76 repetition of hello world implemented
* a5b522b simple hello world script added
```

Before discussing the output, let us briefly comment on the options used in the git log command. Usually, this command will be more verbose, giving the full hash value of the commit, the name of the author and the date of the commit together with the commit message. Using the switch --oneline, this information can be reduced to a single line. Its content could be configured but we do not need to do this here. The options --graph and --all will have an effect once more than one branch is present. Then, we will obtain a graphical representation of the commit tree, i.e. the relation between the different branches. In addition, we will be shown information about all branches, not only the branch we are on. Finally, --decorate shows us references existing for certain commits. In our case, the commit aac6d17 is referred to as HEAD because that is the version we are presently seeing in our working directory. This is also where the branch master is positioned right now. The usefulness of this information will become clear once we have more than one branch or when even working with remote branches.

The history documented by the output of git log is linear with the most recent commit on top. As we have discussed earlier, Git is a distributed version control system. Therefore, we have to expect that other developers are doing work in parallel which at some time should connect to our work. Otherwise, we could simply ignore these developers. Consequently, in general we cannot expect the history of our repository to be as simple as it is up to now.

However, we do not need other developers to have several lines of development running in parallel for some time. Even for a single developer, it makes sense to keep different lines of development separated at least for some time. Suppose for the moment that you have a working program that is used to produce data, the production version of the program. At the same time, you want to develop this program further, e.g. in order to add functionality or to improve its speed. Such a development should be carried out separately from the production version so that the latter can easily be accessed in the repository at any time. Or you have a potentially good idea which you would like to try out, but you do not know whether this idea will make it into the main code. Again, it is useful to keep the exploration of your idea separate from the production version of your program. Of course, if the idea turns out to be a good one, it should be possible to merge the new code into the production version.

The solution to the needs occurring in these scenarios are branches. In a typical scenario, one would keep the production version in the master branch which in a sense forms the trunk of a tree. At a certain commit of the master branch, a new branch will take that commit as a parent on which further development of, e.g., a new aspect of the program is based. There could be different branches extending from various commits and a branch can have further branches. The picture of a tree thus seems quite appropriate. However, typically branches will not grow forever in their own direction. Ideally, the result of the development in a branch should ultimately flow back into the production code, a step referred to as merging.

Let us take a look at an example. As branches can become a bit confusing once you have several of them, it makes sense to make sure from time to time that you are still on the right branch. We have not created a new branch and therefore are on the master branch. This can be verified as follows:

```
$ git branch
* master
```

So far, we have only a single branch named `master`. The star in front indicates that we are indeed on that branch.

Now suppose that the idea came up not to greet the whole world but a single person instead. This implies a major change of the program and there is a risk that the program used in production might not always be working correctly if we do our work on the master branch. It is definitely time to create a new branch. We call the new branch `dev` for development but we could choose any other name. In general, it is a good idea to choose telling names, in particular as the number of branches grows.

The new branch can be created by means of

```
$ git branch dev
```

We can verify the existence of the new branch:

```
$ git branch
  dev
* master
```

As the star indicates, we are still on the master branch, but a new branch named `dev` exists. Switching back and forth between different branches is done by means of the `switch` command. With the following commands, we got to the development branch and back to the master branch while verifying where we are after each checkout:

```
$ git switch dev
Switched to branch 'dev'
$ git branch
* dev
  master
$ git switch master
Switched to branch 'master'
$ git branch
  dev
* master
```

In addition, we can check the history of our repository:

```
$ git log --oneline --graph --decorate --all
* aac6d17 (HEAD -> master, dev) .gitignore for Python added
* 98628ce hello world script refactored
* 011ce76 repetition of hello world implemented
* a5b522b simple hello world script added
```

Now, commit `aac6d17` is also part of the branch `dev`. For the moment, the new branch is not really visible as branch because we have not done any development.

Above, we have first created a new branch and then switched to the new branch. As one typically wants to switch to the new branch immediately after having created it, there exists a shortcut:

```
$ git switch -c dev
Switched to a new branch 'dev'
```

The option `-c` demands a new branch to be created.

Everything is set up now to work on the new idea. Let us suppose that at some point you arrive at the following script:

```python
# hello.py
from repeat import repeated_print

def hello(name="", repetitions=1):
    if name:
        repeated_print(f"Hello, {name}", repetitions)
```

(continues on next page)

(continued from previous page)

```
    else:
        repeated_print("Hello world!", repetitions)
```

After committing it, the commit log looks as follows:

```
$ git log --oneline --graph --decorate --all
* f113188 (HEAD -> dev) name as new argument implemented
* aac6d17 (master) .gitignore for Python added
* 98628ce hello world script refactored
* 011ce76 repetition of hello world implemented
* a5b522b simple hello world script added
```

The history is still linear, but clearly the master branch and the development branch are in different states now. The master branch is still at commit `aac6d17` while the development branch is at `f113188`. At this point, it is worth going back to the master branch and to check the content of `hello.py`. At first, it might appear that we have lost our recent work but this is not the case because we had committed the new version in the development branch. Switching back to `dev`, we indeed find the new version of the script.

During the development of the new script, we realized that it is a good idea to define a default value for the number of repetitions and we decide that it is a good idea to make a corresponding change in the master branch. Before continuing to work in the development branch, we perform the following steps:

1. check out the master branch

   ```
   $ git switch master
   ```

2. make modifications to `repeat.py`

   ```
   # repeat.py
   def repeated_print(text, repetitions=1):
       for n in range(repetitions):
           print(text)
   ```

3. commit the new version of the script

   ```
   $ git commit -a -m 'default value for number of repetitions defined'
   ```

4. check out the development branch

   ```
   $ git switch dev
   ```

The commit history is no longer linear but has clearly separated into two branches:

```
$ git log --oneline --graph --decorate --all
* 1d9a25f (master) default value for number of repetitions defined
| * f113188 (HEAD -> dev) name as new argument implemented
|/
* aac6d17 .gitignore for Python added
* 98628ce hello world script refactored
* 011ce76 repetition of hello world implemented
* a5b522b simple hello world script added
```

Now it is time to complete the script `hello.py` by adding an exclamation mark after the name and calling the new function `hello`:

```
# hello.py
from repeat import repeated_print

def hello(name="", repetitions=1):
```

(continues on next page)

(continued from previous page)

```
    if name:
        s = "Hello, " + name + "!"
        repeated_print(s, repetitions)
    else:
        repeated_print("Hello world!", repetitions)

if __name__ == "__main__":
    hello("Alice", 3)
```

Before committing the new version, we start thinking about atomic commits. Strictly speaking, we made two different kinds of changes. We have added the exclamation mark and added the function call. Instead of going back and making the changes one after the other, we can recall that the option `-p` allows to choose which changes to add to the staging area:

```
$ git add -p hello.py
diff --git a/hello.py b/hello.py
index b2ee076..c287658 100644
--- a/hello.py
+++ b/hello.py
@@ -3,6 +3,9 @@ from repeat import repeated_print

 def hello(name="", repetitions=1):
     if name:
-        repeated_print(f"Hello, {name}", repetitions)
+        repeated_print(f"Hello, {name}!", repetitions)
     else:
         repeated_print("Hello world!", repetitions)
+
+if __name__ == "__main__":
+    hello("Alice", 3)
(1/1) Stage this hunk [y,n,q,a,d,s,e,?]?
```

Answering the question with `s`, i.e. `split`, we are offered the possibility to add the two changes separately to the changing area. In this way, we can create two separate commits. After actually doing the commits, we arrive at the following history:

```
$ git log --oneline --graph --decorate --all
* a807c98 (HEAD -> dev) function call added
* d07dbda exclamation mark added
* f113188 name as new argument implemented
| * 1d9a25f (master) default value for number of repetitions defined
|/
* aac6d17 .gitignore for Python added
* 98628ce hello world script refactored
* 011ce76 repetition of hello world implemented
* a5b522b simple hello world script added
```

Now, it is time to make the new functionality available for production, i.e. to merge the commits from the development branch into the master branch. To this end, we switch to the master branch and merge the development branch:

```
$ git switch master
Switched to branch 'master'
$ git merge dev
Merge made by the 'recursive' strategy.
 hello.py | 9 ++++++++-
 1 file changed, 8 insertions(+), 1 deletion(-)
$ git log --oneline --graph --decorate --all
```

(continues on next page)

(continued from previous page)

```
*   53e12db (HEAD -> master) Merge branch 'dev'
|\
| * a807c98 (dev) function call added
| * d07dbda exclamation mark added
| * f113188 name as new argument implemented
* | 1d9a25f default value for number of repetitions defined
|/
* aac6d17 .gitignore for Python added
* 98628ce hello world script refactored
* 011ce76 repetition of hello world implemented
* a5b522b simple hello world script added
```

In this case, Git has made a so-called three-way merge based on the common ancestor of the two branches (`aac6d17`) and the current versions in the two branches (`1d9a25f`) and (`a807c98`). It is interesting to compare the script `repeat.py` in these three versions. The version in the common ancestor was:

```python
# repeat.py aac6d17
def repeated_print(text, repetitions):
    for n in range(repetitions):
        print(text)
```

In the master branch, we have

```python
# repeat.py 1d9a25f
def repeated_print(text, repetitions=1):
    for n in range(repetitions):
        print(text)
```

while in the development branch, the script reads

```python
# repeat.py a807c98
def repeated_print(text, repetitions):
    for n in range(repetitions):
        print(text)
```

Note that in `1d9a25f` a default value for the variable `repetitions` is present while it is not in `a807c98`. The common ancestor serves to resolve this discrepancy. Obviously, a change was made in the master branch while it was not done in the development branch. Therefore, the change is kept. The other modifications in the branches were not in contradiction, so that the merge could be done automatically and produced the desired result.

The life of the development branch does not necessarily end here if we decide to continue to work on it. In fact, the branch `dev` continues to exist until we decide to delete it. Since all work done in the development branch is now present in the master branch, we decide to delete the branch `dev`:

```
$ git branch -d dev
Deleted branch dev (was a807c98).
```

An attempt to delete a branch which was not fully merged, will be rejected. This could be the case if the idea developed in a branch turns out not to be a good idea after all. The deletion of the branch can be forced by replacing the option `-d` by `-D`.

In general, one cannot expect a merge to run as smoothly as in our example. Frequently, a so-called merge conflict arises. This is quite common if different developers work in the same part of the code and their results are incompatible. For the sake of example, let us assume that we add a doc string to the `repeated_print` function but choose a different text in the master branch and in the development branch. In the master branch we have

```python
# repeat.py in master
def repeated_print(text, repetitions=1):
```

(continues on next page)

(continued from previous page)

```
    """print text repeatedly

    """
    for n in range(repetitions):
        print(text)
```

while in the development branch we have chosen a different doc string

```
# repeat.py in dev
def repeated_print(text, repetitions):
    """print text several times"""
    for n in range(repetitions):
        print(text)
```

The commit history of which we only show the more recent part now becomes a bit more complex:

```
* c3ab8cb (HEAD -> dev) added a doc string
| * cc484da (master) doc string added
| *   53e12db Merge branch 'dev'
| |\
| |/
|/|
* | a807c98 function call added
* | d07dbda exclamation mark added
* | f113188 name as new argument implemented
| * 1d9a25f default value for number of repetitions defined
|/
* aac6d17 .gitignore for Python added
```

We switch to the master branch and try to merge once more the development branch:

```
$ git switch master
Switched to branch 'master'
$ git merge dev
Auto-merging repeat.py
CONFLICT (content): Merge conflict in repeat.py
Automatic merge failed; fix conflicts and then commit the result.
```

This time, the merge fails and Git informs us about a merge conflict. At this point, Git needs to be told which version of the doc string should be used in the master branch. Let us take a look at our script:

```
# repeat.py
<<<<<<< HEAD
def repeated_print(text, repetitions=1):
    """print text repeatedly

    """
=======
def repeated_print(text, repetitions):
    """print text several times"""
>>>>>>> dev
    for n in range(repetitions):
        print(text)
```

There are two blocks separated by `=======`. The first block starting with `<<<<<<< HEAD` is the present version in the master branch where we are right now. The second block terminated by `>>>>>>> dev` stems from the development branch. The reason for the conflict lies in the different doc strings. In such a situation, Git needs help. The script should now be brought into the desired form by using an editor or a tool to handle merge conflicts. We choose

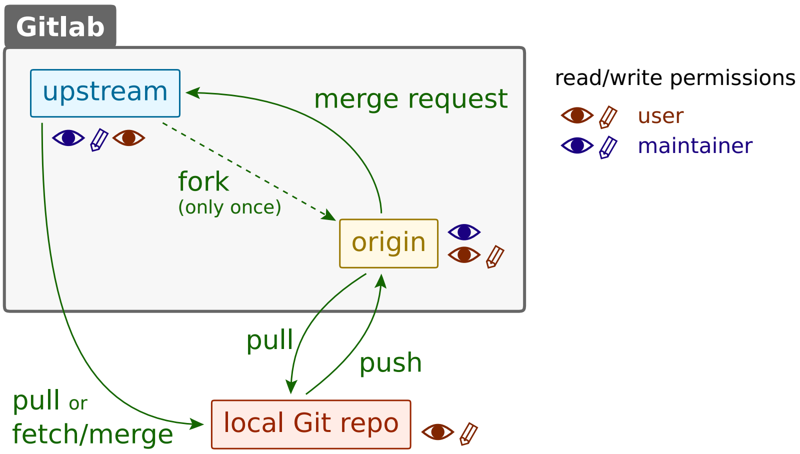


Figure 2.5: Workflow for collaborative development in a distributed version control system with a GitLab instance as central server.

```
# repeat.py
def repeated_print(text, repetitions=1):
    """print text repeatedly

    """
    for n in range(repetitions):
        print(text)
```

but the other version or a version with further modifications would have been possible as well. In order to tell Git that the version conflict has been resolved, we add it to the staging area and commit it as usual. The history now looks as follows:

```
*   d10bdbb (HEAD -> master) merge conflict resolved
|\
| * c3ab8cb (dev) added a doc string
* | cc484da doc string added
* | 53e12db Merge branch 'dev'
|\|
| * a807c98 function call added
| * d07dbda exclamation mark added
| * f113188 name as new argument implemented
* | 1d9a25f default value for number of repetitions defined
|/
* aac6d17 .gitignore for Python added
```

While the use of branches can be an extremely valuable technique even for a single developer, branches will inevitably appear in a multi-developer environment. A good understanding of branches will therefore be helpful in the following section.

2.7 Collaborative code development with GitLab

So far, we have only worked within a single developer scenario and a local Git repository was sufficient. However, scientific research is often carried out in teams with several persons working on the same project at the same time. While a distributed version control system like Git allows each person to work with her or his local repository for some time, it will become necessary at some point to share code. One way would be to grant all persons on the project read access to all local repositories. However, in general such an approach will result in a significant administrative load. It is much more common to exchange code via a central server, typically a GitLab server run by an institution or a service like GitHub.

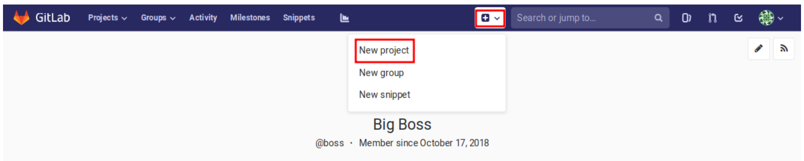


Figure 2.6: Creation of a new project in a GitLab repository.

Independently of whether one uses a GitLab server or GitHub, the typical setup looks like depicted in Figure 2.5 and consists of three repositories. In order to understand this setup, we introduce to roles. The user is representative of one of the individual developers while the maintainer controls the main project repository. As a consequence of their respective roles, the user has read and write access to her or his local repository while the maintainer has read and write access to the main project repository, often referred to as `upstream`. Within a project team, every member should be able to access the common code base and therefore should have read access to `upstream`. In order to avoid that the maintainer needs read access to the user's local repository, it is common to create a third repository often called `origin` to which the user has read and write access while the maintainer has read access. In order to facilitate the rights management, `origin` and `upstream` are usually hosted on the same central server. At same point in time, the user creates `origin` by a process called forking, thereby creating her or his own copy of `upstream`. This process needs only to be done once. Afterwards, the code can flow in counter-clockwise direction in Figure 2.5. The individual steps are as follows:

1. The user can always get the code from the `upstream` repository, e.g. to use it as basis for the future development. There are two options, namely `git pull` and the two-step process `git fetch` and `git merge` which we will discuss below.
2. Having read and write access both on the local repository and the `origin` repository, the user can `git push` to move code to the central server. With `git pull`, code can also be brought from the central server to a local repository. The latter is particularly useful if the user is working on several machines with individual local repositories.
3. As long as the user has no write access to `upstream`, only the maintainer can transfer code from the user's `origin` to `upstream`. Usually, the user will inform the maintainer by means of a merge request that code is being ready to be merged into the `upstream` repository[5]. After an optional discussion of the suitability of the code, the maintainer can merge the code into the `upstream` repository.

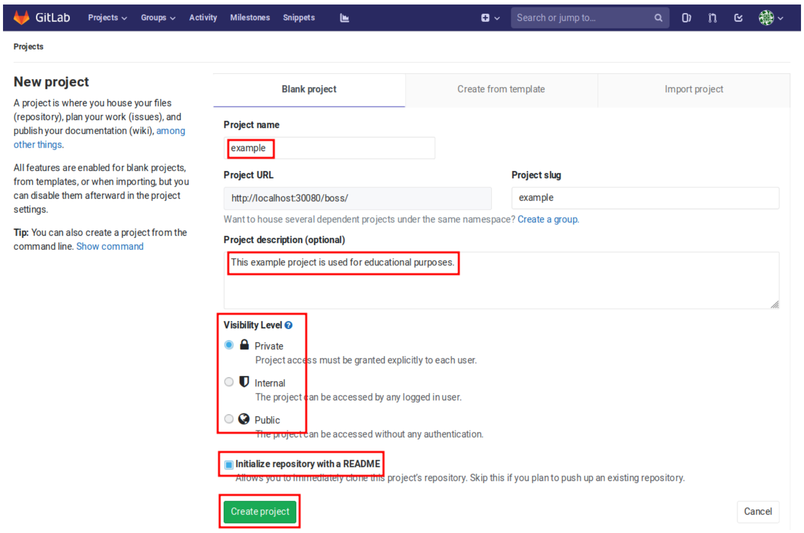
After these conceptual considerations, we discuss a more practical example. The maintainer of the project will be called Big Boss with username `boss` and she or he starts by creating a repository for a project named `example`. We will first go through the steps required to set up the project and then focus on how one remotely interacts with this repository either as an owner of the repository or a collaborator who contributes code via his or her repository.

After logging into a GitLab server, one finds in the dashboard on the top of the screen the possibility to create a new project as shown in Figure 2.6. In order to actually create a new project, some basic information is needed as shown in Figure 2.7. Mandatory are the name as well as the visibility level of the project. A private project will only be visible to the owner and members who were invited to join the project. Public projects, on the other hand, can be accessed without any authentication. It is recommended to add a short description of the project so that its purpose becomes apparent to visitors of the project page. In addition, it is useful to add at least a short README file. This README file initially will contain the name of the repository and the project description. It can be extended over time by adding information useful for visitors of the project page. Creating a README file also ensures that the repository contains at least one file.

Tip

Markup can be used to format the README page. Markup features include headers, lists, web links and more. GitLab and GitHub recognize markdown (file extension `.md`) and restructured text (file extension `.rst`). We recommend to take a look at the Markdown Style Guide of GitLab and to experiment with different formatting possibilities. This is also a good opportunity to exercise your version control skills. You can check the effect of the markup by taking a look at the project page.

[5] On GitHub, instead of "merge request" the term "pull request" is used, meaning the same.



New project

A project is where you house your files (repository), plan your work (issues), and publish your documentation (wiki), among other things.

All features are enabled for blank projects, from templates, or when importing, but you can disable them afterward in the project settings.

Tip: You can also create a project from the command line. Show command

Project name: example

Project URL: http://localhost:30080/boss/

Project slug: example

Want to house several dependent projects under the same namespace? Create a group.

Project description (optional): This example project is used for educational purposes.

Visibility Level

● Private
Project access must be granted explicitly to each user.

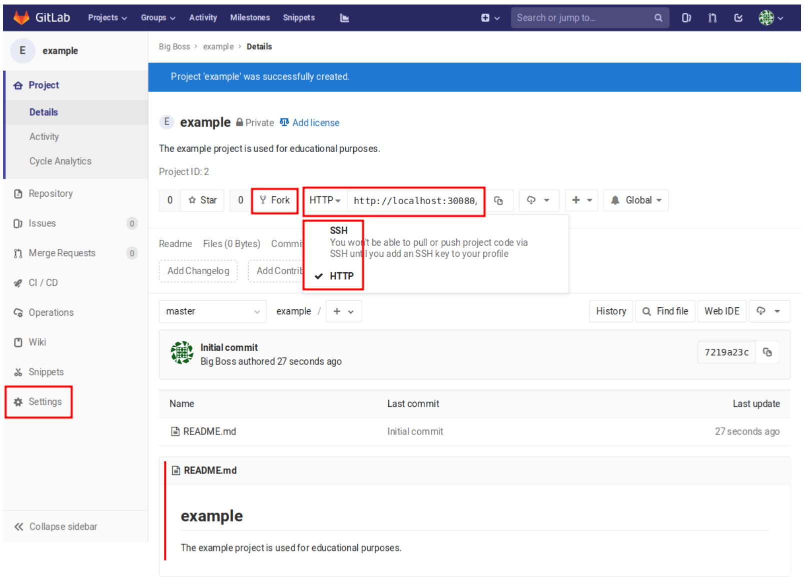
○ Internal
The project can be accessed by any logged in user.

○ Public
The project can be accessed without any authentication.

☑ Initialize repository with a README
Allows you to immediately clone this project's repository. Skip this if you plan to push up an existing repository.

Create project

Figure 2.7: During the creation of a project its name and its visibility level need to be defined. In addition, it makes sense to add a project description and to initialize the repository with a README file.



Project 'example' was successfully created.

example Private Add license

The example project is used for educational purposes.

Project ID: 2

http://localhost:30080,

SSH
You won't be able to pull or push project code via SSH until you add an SSH key to your profile

✓ HTTP

Initial commit
Big Boss authored 27 seconds ago
7219a23c

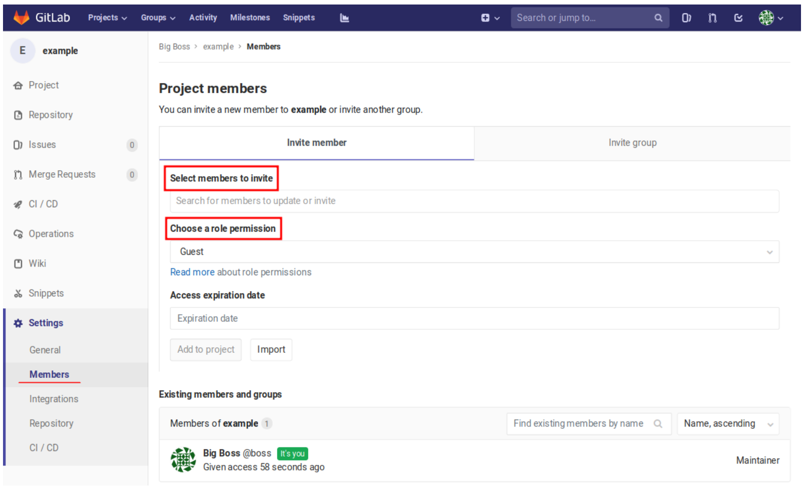
Name	Last commit	Last update
README.md	Initial commit	27 seconds ago

README.md

example

The example project is used for educational purposes.

Figure 2.8: The new repository can be accessed via the HTTP and SSH protocols. Users with access to the repository can also fork it.



Project members

You can invite a new member to **example** or invite another group.

Select members to invite: Search for members to update or invite

Choose a role permission: Guest

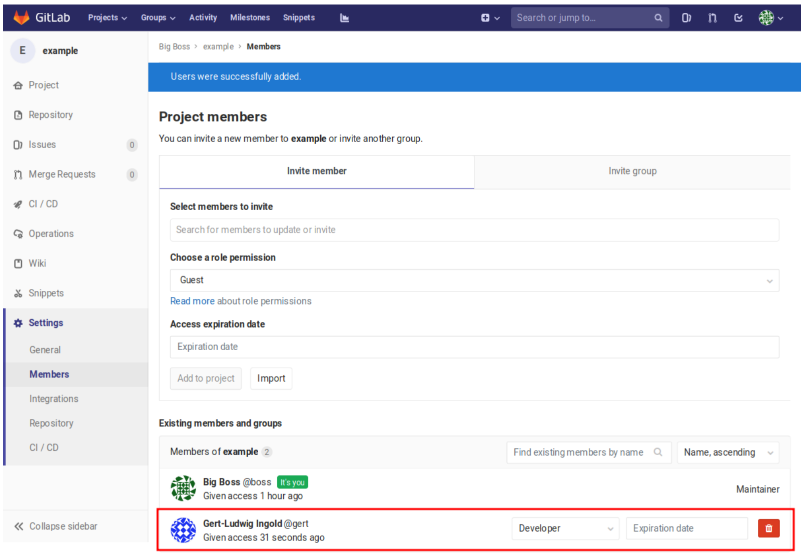
Access expiration date: Expiration date

Existing members and groups

Members of **example** 1

Big Boss @boss It's you — Given access 58 seconds ago — Maintainer

Figure 2.9: On the setting page, other users can be invited to join the project and their permissions can be defined.



Users were successfully added.

Project members

You can invite a new member to **example** or invite another group.

Select members to invite: Search for members to update or invite

Choose a role permission: Guest

Access expiration date: Expiration date

Existing members and groups

Members of **example** 2

Big Boss @boss It's you — Given access 1 hour ago — Maintainer

Gert-Ludwig Ingold @gert — Given access 31 seconds ago — Developer — Expiration date

Figure 2.10: A new team member has been added to the project as developer.

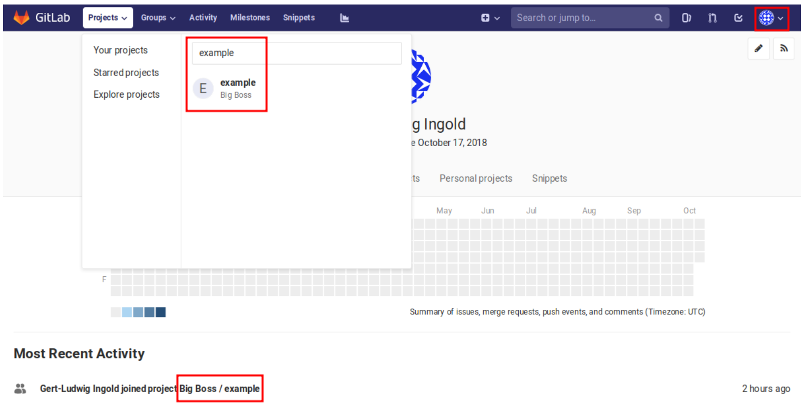
The project page shown in Figure 2.8 contains relevant elements for users collaborating on the project. There is the possibility to create a fork of the project. According to the workflow represented in Figure 2.5, forking a project creates a new repository usually referred to as `origin` which is based on the repository referred to as `upstream`. The key point in forking is to create a repository to which the user has write access, which need not be the case for the original project.

Furthermore, the screen depicted in Figure 2.8 contains information about the URL under which the repository can be accessed. We will need this information later on. As the figure shows, the repository can be accessed via the HTTP protocol which will ask for the username and password, if necessary. An alternative is the SSH protocol which requires that a public SSH key of the user is stored on the GitLab server. Finally, Figure 2.8 demonstrates how the information entered when setting up the project is used to create a minimal README file which is displayed in a formatted way at the bottom of the project page.

Tip

Information on how to create a SSH key can be found for example in the section GitLab and SSH keys of the GitLab documentation.

The previous discussion had already the idea of collaborative work on the project in mind. However, for the moment



Summary of issues, merge requests, push events, and comments (Timezone: UTC)

Most Recent Activity

Gert-Ludwig Ingold joined project Big Boss / example 2 hours ago

Figure 2.11: In order to navigate to a repository, one can for example search for it or use a direct link if one has joined the project recently. This page can be accessed by choosing "Profile" from the avatar menu in the upper right corner.

nobody has access to the project except the owner who had created the project. Additional team members can be invited in the settings menu by accessing the members page shown in Figure 2.9. Here, team members can be invited and their permissions can be defined. If a new team member should be able to contribute code to the project, he or she while typically take on the role of a developer. Figure 2.10 shows that a new team member has been successfully added in the role of a developer. The project maintainer can remove team members at any time by clicking on the red icon on the right.

We are now in a position to explore the collaborative workflow shown in Figure 2.5. There exists an alternative approach relying on protected branches which we do not cover here[6].

For the following discussion, we assume that user `boss` has created a project called `example` which can be accessed as indicated in Figure 2.8. In our case, the HTTP access would be via the address `http://localhost:30080/boss/example.git` and for SSH access we would use `ssh://git@localhost:30080/boss/example.git`. In a real application, be sure to replace these addresses by the addresses indicated on the project page. Maintainer `boss` has invited developer `gert` to the project team and the latter now has to set up his system to be able to contribute to project `example`. During the discussion, it might be useful to occasionally take a look at Figure 2.5 in order to connect the details to the overall picture.

In a first step, user `gert` logs into the GitLab server and goes to the project `example` of user `boss`. A possibility to do so consists in searching for the project name in the dashboard as shown in Figure 2.11. On the user's profile page, there might be alternative ways like in Figure 2.11 where the repository is listed because the user joined it recently. At a later stage, it would also be possible to go via the forked repository or the list of contributed projects. In any case, the user `gert` will see a page looking almost like the one displayed in Figure 2.8. In particular, there will be a fork button which initiates the creation of a fork of the original project as a project of user `gert`. In the notation of Figure 2.5, a repository `origin` has been created as a copy of the present state of the repository `upstream`.

According to Figure 2.5, the developer now needs to create a local repository for the project based on his or her own repository on the GitLab server, i.e. the repository referred to as `origin`. Using the URL shown in Figure 2.8, the repository is cloned into a local directory as follows:

```
$ git clone ssh://git@localhost:30022/gert/example.git
Cloning into 'example' ...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0)
Receiving objects: 100% (3/3), done.
$ ls -a example
.  ..  .git  README.md
```

In the third line, the passphrase for the SSH key needs to be given. If the HTTP protocol were used, username and

[6] More information on working with protected branches can be found at Protected Branches in the GitLab documentation.

password would have been requested. In the last line we see that the directory *.git* has been created without the need of initializing the repository. By default, *git clone* transfers the repository with its complete history, unless only part of the history is requested by means of the `--depth` argument.

In contrast to the previous sections, we are no longer only working with a local repository but also with the two remote repositories `origin` and `upstream` on the GitLab server. To find out which remote repositories are locally known, we go to the directory where the repository is located and use:

```
$ git remote -v
origin  ssh://git@localhost:30022/gert/example.git (fetch)
origin  ssh://git@localhost:30022/gert/example.git (push)
```

These lines tell us that the developer's repository `example` on the remote server is available for read and write under the name `origin`. However, we also need access to the repository usually referred to as `upstream`. This can be achieved by telling Git about this remote repository:

```
$ git remote add upstream ssh://git@localhost:30022/boss/example.git
$ git remote -v
origin  ssh://git@localhost:30022/gert/example.git (fetch)
origin  ssh://git@localhost:30022/gert/example.git (push)
upstream        ssh://git@localhost:30022/boss/example.git (fetch)
upstream        ssh://git@localhost:30022/boss/example.git (push)
```

Now we can refer to the original remote repository as `upstream`. The existence of a channel for pushing does not necessarily imply that we have the permission to actually write to `upstream`.

Being a developer on the `example` project, we want to contribute code to the project. Already in our discussion of the workflow within a purely local repository we have seen that it might be useful to do development work in dedicated branches. The same is true in a setup involving remote repositories. In the discussion of merge requests we will give an additional argument in favor of using dedicated branches for different aspects of development. While various approaches to the use of branches are possible, a judicious choice would be to attribute a special role to the master branch by keeping it in sync with the `upstream` repository. By branching off from the `master` repository, the development activities can be kept close to the code on `upstream`, thereby facilitating a later merge into the main code base.

The developer decides to contribute a "Hello world" script to the `example` project and first creates a new branch named `hello`:

```
$ git checkout -b hello
Switched to a new branch 'hello'
$ git branch
* hello
  master
```

We already know how to commit a script to the new branch. After doing so, the content of the main directory is:

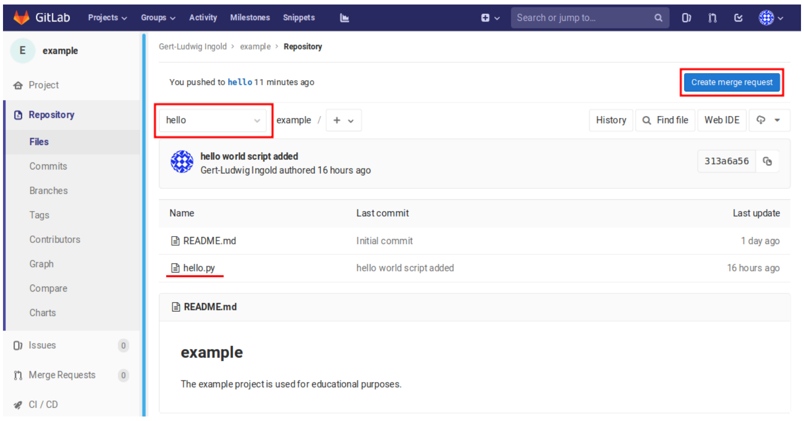
```
$ ls -a
.  ..  .git  hello.py  README.md
```

and the history reads:

```
$ git log --oneline --decorate
* 313a6a5 (HEAD -> hello) hello world script added
* 7219a23 (origin/master, origin/HEAD, master) Initial commit
```

The local branch `master` as well as the remote branch `origin/master` are still at the initial commit `7219a23` while the local branch `hello` is one commit ahead. The remote repository `origin` is not aware of the new branch yet. Furthermore, the local repository has not yet any information about the remote repository `upstream`.

In a next step, the developer pushes the new commit or several of them to the remote repository `origin` where he or she has write permission:



You pushed to hello 11 minutes ago

hello world script added
Gert-Ludwig Ingold authored 16 hours ago
313a6a56

Name	Last commit	Last update
README.md	Initial commit	1 day ago
hello.py	hello world script added	16 hours ago

README.md

example

The example project is used for educational purposes.

Figure 2.12: The script `hello.py` has been successfully pushed to the remote branch `origin/hello`. It can now be brought to the remote repository `upstream` by means of a merge request.

```
$ git push -u origin hello
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 328 bytes | 328.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
remote:
remote: To create a merge request for hello, visit:
remote:   http://localhost:30080/gert/example/merge_requests/new?merge_request
↪%5Bsource_branch%5D=hello
remote:
To ssh://localhost:30022/gert/example.git
 * [new branch]      hello -> hello
Branch 'hello' set up to track remote branch 'hello' from 'origin'.
```

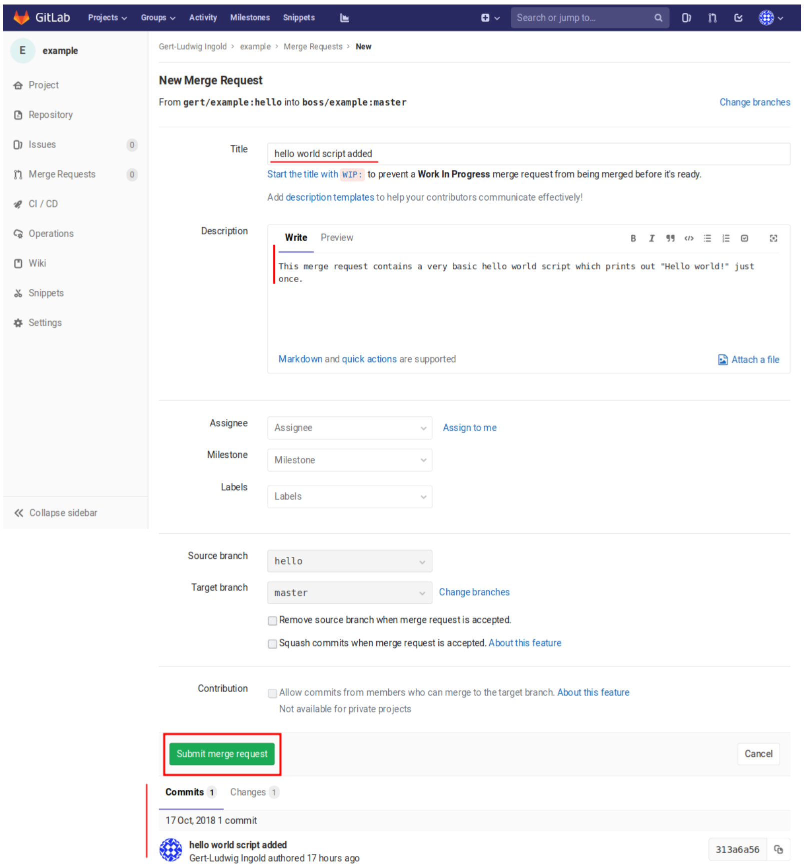
Actually, two things have happened here at the same time. The commit `313a6a5` was pushed to the branch `hello` on `origin`. Because of the option `-u`, the local branch was associated with the remote branch. From now on, if one wants to push commits from the local branch `hello` to the corresponding remote branch, it suffices to use `git push`. This is not only shorter to type but also avoids to accidentally push commits to the wrong branch. We can verify that the commit is now present on the remote server either by means of:

```
$ git log --oneline --decorate
313a6a5 (HEAD -> hello, origin/hello) hello world script added
7219a23 (origin/master, origin/HEAD, master) Initial commit
```

where commit `313a6a5` now also refers to `origin/hello`. Alternatively, one can take a look at the project page on the GitLab server which will look like Figure 2.12. Make sure that the branch has been changed from `master` to `hello` because that is where the script has been pushed to. It is not and should not be present in `origin/master` at this point.

Following the workflow displayed in Figure 2.5, the developer might now want to contribute the new script to the `upstream` repository. If the developer has no write access to this repository, he or she can make a merge request as we will explain now. If, on the other hand, the developer has write access to the `upstream` repository, he or she could push the script directly there. However, even with with write access it might be preferable to contribute code via a merge request and this could be the general policy applying even to maintainers. The advantage of merge requests is that other team members can automatically be informed about new contributions and have a chance to discuss them before they become part of the `upstream` repository. As long as the person merging the submitted code is different from the submitter, a second pair of eyes can take a look at the code and spot potential problems. In the end, the project team or the team leaders have to decide which policy to follow.

On the project page shown in Figure 2.12, there is a button in the upper right with the title “Create merge request”



New Merge Request

From `gert/example:hello` into `boss/example:master`

Title: hello world script added

Start the title with `WIP:` to prevent a **Work In Progress** merge request from being merged before it's ready.

Add description templates to help your contributors communicate effectively!

Description:

```
This merge request contains a very basic hello world script which prints out "Hello world!" just
once.
```

Markdown and quick actions are supported

Assignee: Assignee

Milestone: Milestone

Labels: Labels

Source branch: `hello`

Target branch: `master`

□ Remove source branch when merge request is accepted.

□ Squash commits when merge request is accepted. About this feature

Contribution: □ Allow commits from members who can merge to the target branch. About this feature

Not available for private projects

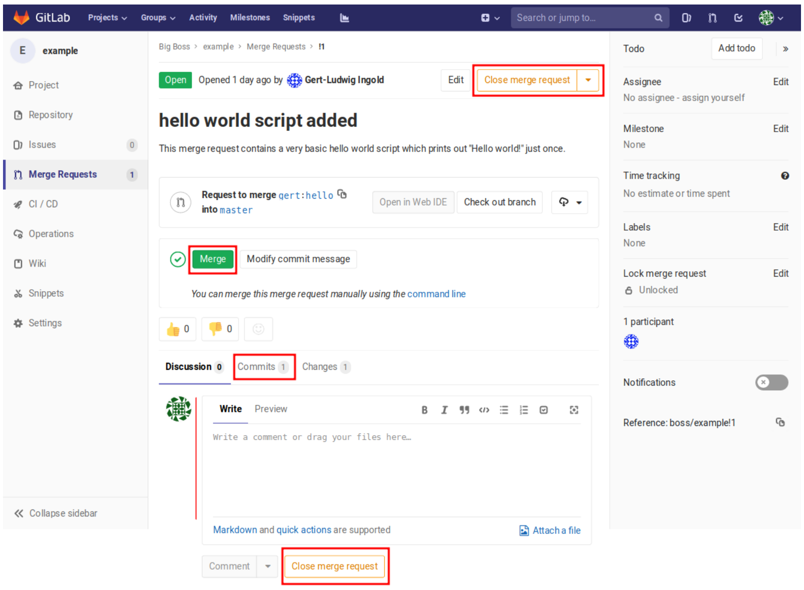
17 Oct, 2018 1 commit

hello world script added

Gert-Ludwig Ingold authored 17 hours ago

313a6a56

Figure 2.13: GitLab page for the preparation of a new merge request.



Opened 1 day ago by Gert-Ludwig Ingold

hello world script added

This merge request contains a very basic hello world script which prints out "Hello world!" just once.

Request to merge gert:hello into master

You can merge this merge request manually using the command line

Write a comment or drag your files here…

Markdown and quick actions are supported

Assignee: No assignee - assign yourself

Milestone: None

Time tracking: No estimate or time spent

Labels: None

Lock merge request: Unlocked

1 participant

Reference: boss/example!1

Figure 2.14: A merge request can be discussed. It can be merged and closed or even closed without merging if the code has been found to be unsuitable for the project. The page shown here assumes that the user logged in has write permission for the project.

which does precisely what this title says. Clicking this button will bring up a page like the one depicted in Figure 2.13. It is important to give a descriptive title as it will appear in a list of potentially many merge requests. In addition, the purpose of the merge request as well as additional relevant information like design considerations should be stated in the description field. Optionally, labels can be attributed to the merge request or merge requests can be assigned to milestones. As these possibilities are mostly of interest in larger projects, we will not discuss them any further here.

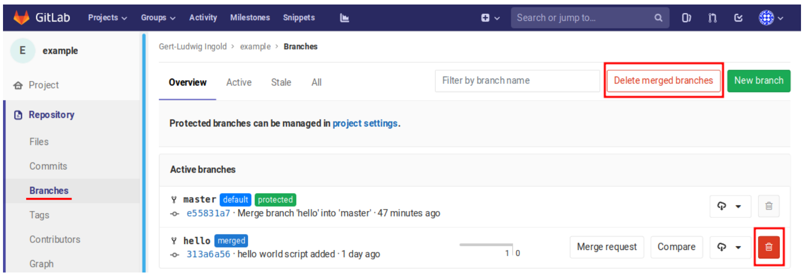
At this point, it is appropriate to give the use of branches a bit more consideration. Suppose that the merge request is not merged into `upstream` right away and that the developer is continuing development. After some time, he or she will commit the new work to the `hello` branch on `origin`. Then this new commit will automatically be part of the present merge request even though the new commit might not be logically related to the merge request. In such a situation, it is better to start a new branch, probably based on the local `master` branch.

Even though the merge request is based on code in the repository `origin`, it will appear in the list of merge requests for the repository `upstream` because that is where the code should be merged. The page of an open merge request looks similar to Figure 2.14. It offers the possibility to view the commits included in the merge request and to comment on them. Persons with write permission on `upstream` have the possibility to merge the commits contained in the merge request and to close it afterwards. If the code should not be included in `upstream`, the merge request can also be closed without merging. In this case, reasons should of course be given in the discussion section. Let us assume that the maintainer merges the commits in the merge request without further discussion and closes the merge request.

The developer's code has successfully found its way to the `upstream` repository. However, his or her local repository does not yet reflect this change. It is now time to complete the circle depicted in Figure 2.5 and to get the changes from the `upstream` repository into the local repository. We will assume that we organise our branches in such a way that the local `master` branch should be kept in sync with the `master` branch in the `upstream` repository. If we are still in the development branch `hello`, it is now time to go back to the `master` branch:

```
$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
```

Now, we have two options. With `git pull upstream master`, the present state of the remote branch `master` on `upstream` would be downloaded and merged into the present local branch. For a better control of the process, one can split it into two steps:



Protected branches can be managed in project settings.

Active branches

master default protected
e55831a7 · Merge branch 'hello' into 'master' · 47 minutes ago

hello merged
313a6a56 · hello world script added · 1 day ago

1 0

Figure 2.15: At the tab “Settings - Branches” individual branches or all merged branches can be removed.

```
$ git fetch upstream
remote: Enumerating objects: 1, done.
remote: Counting objects: 100% (1/1), done.
remote: Total 1 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (1/1), done.
 * [new branch]      master      -> upstream/master
$ git merge upstream/master
Updating 7219a23..e55831a
Fast-forward
 hello.py | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 hello.py
```

`git fetch` gets new objects from the `master` branch and `git merge upstream/master` merges the objects from the remote branch `upstream/master`. The history of the local `master` repository looks as follows:

```
$ git log --oneline --graph --decorate --all
*   e55831a (HEAD -> master, upstream/master) Merge branch 'hello' into 'master'
|\
| * 313a6a5 (origin/hello, hello) hello world script added
|/
* 7219a23 (origin/master, origin/HEAD) Initial commit
```

As we can see, the local `master` branch and the remote `master` branch on the `upstream` repository are in sync while the `master` branch on the `origin` repository is still in its original state. This makes sense because the hello world script was pushed to the `hello` repository on the `origin` repository, but not its `master` branch. We can change this by pushing the local `master` branch to `origin`.

Before doing so, let us remove the `hello` branch which we do not need anymore:

```
$ git push origin --delete hello
To ssh://localhost:30022/gert/example.git
 - [deleted]         hello
$ git branch -d hello
Deleted branch hello (war 313a6a5).
```

The first command deleted the remote branch. As an alternative way, one can use the GitLab web interface as shown in Figure 2.15. There individual branches or all merged branches can be removed. However, the local references to the remote branches are not yet deleted. If one wants to remove references to branches on `origin` which do no longer exist, one can use `git remote prune origin`. The second command above deletes the local branch, provided that no unmerged commits are still present. One can force deletion of the branch with the option `-D` but may risk the loss of data. Using `-D` instead of `-d` should thus be done with care.

After pushing the local `master` branch to `origin`, the log looks as follows:

```
$ git push origin master
Counting objects: 1, done.
```

(continues on next page)

(continued from previous page)

```
Writing objects: 100% (1/1), 281 bytes | 281.00 KiB/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To ssh://localhost:30022/gert/example.git
   7219a23..e55831a  master -> master
$ git log --oneline --decorate --graph
*   e55831a (HEAD -> master, upstream/master, origin/master, origin/HEAD) Merge␣
↪branch 'hello' into 'master'
|\
| * 313a6a5 hello world script added
|/
* 7219a23 Initial commit
```

All three `master` branches are now in the same state and we have completed a basic development cycle.

2.8 Sundry topics

2.8.1 Stashing

In the previous sections, we have only discussed the basic workflows with Git and certainly did not even attempt to be complete. In the day-to-day work with a Git repository, certain problems occasionally arise. Some of them will be discussed in this section.

For the first scenario, let us assume that we have created a `dev` branch where we modified the `hello.py` script and committed the new version. We can then change between branches without any problem:

```
$ git checkout -b dev
Switched to a new branch 'dev'
$ cat hello.py
print("Hello world!")
print("Hello world!")
print("Hello world!")
$ git commit -a -m'repetitive output of message'
[dev 01dc5a1] repetitive output of message
 1 file changed, 2 insertions(+)
$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
$ git checkout dev
Switched to branch 'dev'
```

The situation is different if we do not commit the changes. In the following example, we have implemented the repetitive output by means of a for loop but did not commit the change. Git now does not allow us to change to the `master` branch because we might lose data:

```
$ cat hello.py
for _ in range(3):
    print("Hello world!")
$ git checkout master
error: Your local changes to the following files would be overwritten by checkout:
        hello.py
Please commit your changes or stash them before you switch branches.
Aborting
```

We could force Git to change branches by means of the option `-f` but probably it is a better idea to follow the advice given by Git and to commit or stash the changes. We know about committing but what does stashing mean? The idea is to pack away the uncommitted changes so that they can be retrieved when we return to the `dev` branch:

```
$ git stash
Saved working directory and index state WIP on dev: 01dc5a1 repetitive output of␣
↪message
$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
$ git checkout dev
Switched to branch 'dev'
$ cat hello.py
print("Hello world!")
print("Hello world!")
print("Hello world!")
```

After stashing the changes, Git allowed us to switch back and forth between the `master` and `dev` branch. However, after returning to the `dev` branch it looks as if the script with the for loop were lost. Fortunately, this is not the case as becomes clear from listing the content of the stash. One can retrieve the modified script by popping it from the stash:

```
$ git stash list
stash@{0}: WIP on dev: 01dc5a1 repetitive output of message
$ git stash pop
On branch dev
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:       hello.py

no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (049ca57b4dda40d0869129482e2d216f82186d75)
$ cat hello.py
for _ in range(3):
    print("Hello world!")
```

As the code example given above demonstrates, one can list the content of the stash. However, after some time it is easy to forget that one has stashed code in the first place. Therefore, stashing is most suited for brief interruptions where one needs to change branches for a short period of time. Otherwise, committing the changes might be a better solution.

2.8.2 Tagging

As we know, a specific revision of the code can be specified by means of its SHA1 value. Occasionally, it is useful to tag a revision with a name for easier reference. For example, one might want to introduce different versions of the code tagged by labels like `v1`, `v2` and so on.

The present revision can be tagged as follows:

```
$ git tag -a v1 -m "first production release"
```

Here, the option `-a` means that an annotated tag is created which will have additional information very similar to a commit. There can be e.g. a message, here given by means of the option `-m`, the name of the tagger and the date. This information and more can be displayed:

```
$ git show v1
tag v1
Tagger: Gert-Ludwig Ingold <gert.ingold@physik.uni-augsburg.de>
Date:   Wed Oct 24 14:23:44 2018 +0200

```

(continues on next page)

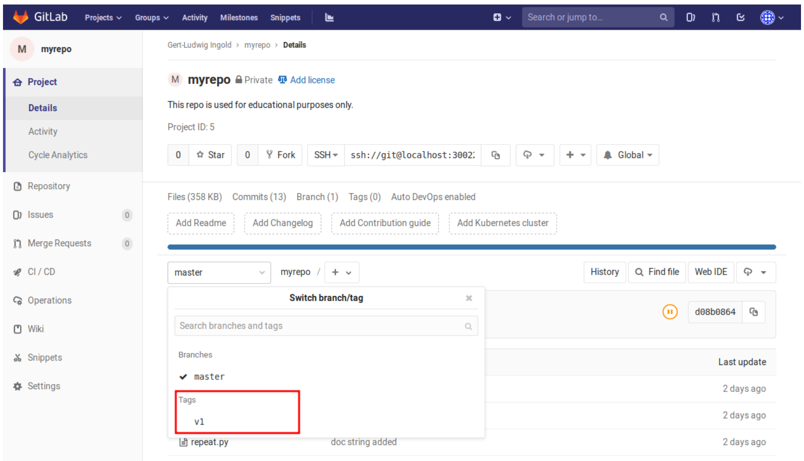


Figure 2.16: After a tag has been pushed to the remote repository, it can be used on the project's web page to navigate to the commit associated with the tag.

(continued from previous page)

```
first production release

commit d08b08646d933e0b7240cbbdbb194143ede1f29c (HEAD -> master, tag: v1)
Merge: a459aec 7a7b8f7
Author: Gert-Ludwig Ingold <gert.ingold@physik.uni-augsburg.de>
Date:   Mon Oct 22 09:28:03 2018 +0200

    Merge branch 'dev'
```

It is also possible to tag older revisions by referring to a specific commit like in the following example:

```
$ git tag -a v0.1 -m "prerelease version" a459aec
$ git tag
v0.1
v1
$ git log --oneline -n5
d08b086 (HEAD -> master, tag: v1) Merge branch 'dev'
7a7b8f7 added doc string
a459aec (tag: v0.1) doc string added
ac805d5 Merge branch 'dev'
41e9e21 function call added
```

The logs demonstrate that indeed the tags are connected with a certain commit. In addition to annotated tags, there are also so-called light-weight tags which cannot contain further attributes. Usually, light-weight tags are employed if they are only temporarily needed.

So far, the tag is only known to the local Git repository. In order for the tag to be known also on a remote repository like `origin`, one needs to push the information about the tag:

```
$ git push origin v1
Enumerating objects: 1, done.
Counting objects: 100% (1/1), done.
Writing objects: 100% (1/1), 187 bytes | 187.00 KiB/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To ssh://localhost:30022/gert/myrepo.git
 * [new tag]         v1 -> v1
```

The tag is now also visible on the project's web page as shown in Figure 2.16.

2.8.3 Detached head state

In Section 2.6, we have seen that we can move between the last commits in different branches. However, we may not only be interested in the most recent version of the code. After all, the whole point in keeping the history of a project is to be able to inspect older versions.

There is a number of different ways of specifying commits in Git and we will only mention a few ones. One possibility is to use the SHA1 value of the commit. In general, the seven first hex digits will be sufficient. If a commit has been tagged as described in the previous section, the tag can be used instead. It is also possible to use a relative notation. For example, the first ancestor of `HEAD` can be obtained by means of `HEAD^`. Note though that if the commit was generated by a merge, more than one ancestors can exist. For details of how in such situation to address a commit relative to another commit is explained in the git documentation, see e.g.,

```
$git help revisions
GITREVISIONS(7)                   Git Manual                   GITREVISIONS(7)

NAME
       gitrevisions - Specifying revisions and ranges for Git

SYNOPSIS
       gitrevisions

DESCRIPTION
       Many Git commands take revision parameters as arguments. Depending on
       the command, they denote a specific commit or, for commands which walk
       the revision graph (such as git-log(1)), all commits which are
       reachable from that commit. For commands that walk the revision graph
       one can also specify a range of revisions explicitly.

       In addition, some Git commands (such as git-show(1)) also take revision
       parameters which denote other objects than commits, e.g. blobs
       ("files") or trees ("directories of files").

SPECIFYING REVISIONS
       A revision parameter <rev> typically, but not necessarily, names a
       commit object. It uses what is called an extended SHA-1 syntax. Here
       [...]
```

Here, we reproduced only part of the help text.

Now let us suppose that the recent history of our repository looks as follows:

```
d08b086 (HEAD -> master, tag: v1) Merge branch 'dev'
7a7b8f7 added doc string
a459aec (tag: v0.1) doc string added
ac805d5 Merge branch 'dev'
41e9e21 function call added
1bac36d exclamation mark appended
d8d7313 default value for repetitions added
c95fa0e new argument 'name' added
```

For some reason, we want to take a look at commit `41e9e21` and decide to check this commit out:

```
$ git checkout 41e9e21
Note: checking out '41e9e21'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

```

(continues on next page)

(continued from previous page)

```
If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b <new-branch-name>

HEAD is now at 41e9e21 function call added
$ git branch
* (HEAD detached at 41e9e21)
  master
```

The important point here is that the branch is in a so-called “detached head state”. At first sight, this branch behaves like a usual branch where we can look around and even commit changes. However, once we leave the branch, there is no way to get back to these commits. As Git explains in the message reproduced above, one needs to check out the branch into a regular new branch if one wants to keep the commits generated in a branch in a “detached head state”. If one forgets to do so, Git will give the following warning:

```
$ git checkout master
Warning: you are leaving 1 commit behind, not connected to
any of your branches:

  4d252a9 'how are you' added

If you want to keep it by creating a new branch, this may be a good time
to do so with:

 git branch <new-branch-name> 4d252a9

Switched to branch 'master'
```

The new branch needs to be created before garbage collection destroys the commit `4d252a9`.

2.8.4 Manipulating history

Travelling back in time and changing the past can have strange effects on the future. What is well known to readers of science fiction also applies to some extent to users of Git. Occasionally, it is tempting to correct the history of the repository. Reasons can be for example typos in commit messages or stupid mistakes in the code. When code is concerned, it usually is preferable to simply correct mistakes in a new commit. On the other hand, it sometimes might make sense to remove a certain commit from the history. It also happens that right after committing code one realizes that there was a typo in the commit message. Correcting the message is still possible and usually is not harmful.

Generally speaking, one can get away with manipulations of the history of a repository as long as the part of the history affected by the manipulations is still completely local. Once the relevant commits have been pushed to a remote repository and others have pulled these commits into their own repositories, changing the history is a potentially great way to make fellow developers very unhappy, something which you definitely want to avoid.

Frequently it happens that one commits code and realizes immediately that the commit message contains a typo. It is rather straightforward to correct such a mistake locally. Suppose that “How are you?” has been added to the output of the script `hello.py` and that the recent history looks as follows:

```
$ git log --oneline -n5
c7be5c2 (HEAD -> master) 'Who are you' added
89f459f Merge branch 'dev'
7a7b8f7 added doc string
a459aec (tag: v0.1) doc string added
ac805d5 Merge branch 'dev'
```

Clearly, the commit message is wrong and even worse, it is misleading. The commit message of the last commit can be amended in the following way:

```
$ git commit --amend -m"'How are you?' added"
[master 3eec1a6] 'How are you?' added
 Date: Fri Oct 26 14:49:03 2018 +0200
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git log --oneline -n5
3eec1a6 (HEAD -> master) 'How are you?' added
89f459f Merge branch 'dev'
7a7b8f7 added doc string
a459aec (tag: v0.1) doc string added
ac805d5 Merge branch 'dev'
```

If the option `-m` is omitted, an editor will be opened to allow you to enter the new commit message.

If you have made a commit erroneously and want to get rid of it, `git reset` can be used to reset `HEAD` to another commit. For example, `HEAD^` denotes the first parent of `HEAD` so that the last commit can be removed by:

```
$ git reset --hard HEAD^
HEAD is now at 89f459f Merge branch 'dev'
$ git log --oneline -n5
89f459f (HEAD -> master) Merge branch 'dev'
7a7b8f7 added doc string
a459aec (tag: v0.1) doc string added
ac805d5 Merge branch 'dev'
41e9e21 function call added
```

As a result, commit `3eec1a6` is gone.

More general changes are possible by means of an interactive rebase. Rebase applies commits on top of a base tip and doing so interactively allows to decide which commits should actually be applied. While a rebase can be done within a single branch, we will directly proceed to the discussion of a rebase across two branches. Suppose that we have the following history:

```
$ git log --oneline --graph --all
* 06933ed (master) headline modified
| * e0ac1ba (HEAD -> dev) add __name__ to output
| * 8c167c1 Test output amended
|/
* 99091f2 Test script added
```

The test script `test.py` should output some headline and the content of the variable `__name__`. In the development branch, this headline has been modified and a print statement for the variable `__name__` was added. On the other hand, the headline has been modified in the master branch as well. For further development, the headline from the master branch should be used, so commit `8c167c1` should be replaced by `06933ed`. So solve this issue, an interactive rebase is done on `master`:

```
$ git rebase -i master
```

An editor opens and displays the following information:

```
pick 8c167c1 Test output amended
pick e0ac1ba add __name__ to output

# Rebase 06933ed..e0ac1ba onto 06933ed (2 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
```

(continues on next page)

(continued from previous page)

```
# f, fixup <commit> = like "squash", but discard this commit's log message
# x, exec <command> = run command (the rest of the line) using shell
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
# .       create a merge commit using the original merge commit's
# .       message (or the oneline, if no original merge commit was
# .       specified). Use -c <commit> to reword the commit message.
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
#       However, if you remove everything, the rebase will be aborted.
#
#
# Note that empty commits are commented out
```

Replacing `pick` by `drop` in front of commit `8c167c1` and leaving the editor, git answers

```
Auto-merging test.py
CONFLICT (content): Merge conflict in test.py
error: could not apply e0ac1ba... add __name__ to output

Resolve all conflicts manually, mark them as resolved with
"git add/rm <conflicted_files>", then run "git rebase --continue".
You can instead skip this commit: run "git rebase --skip".
To abort and get back to the state before "git rebase", run "git rebase --abort".

Could not apply e0ac1ba... add __name__ to output
```

The merge conflict needs to be resolved in the usual way so that the rebase can be continued:

```
$ git add test.py
$ git rebase --continue
[detached HEAD 7c9c8d1] add __name__ to output
 1 file changed, 2 insertions(+), 1 deletion(-)
Successfully rebased and updated refs/heads/dev.
```

The rebase operation was successfully carried out and the new history is:

```
$ git log --oneline --graph --all
* 7c9c8d1 (HEAD -> dev) add __name__ to output
* 06933ed (master) headline modified
* 99091f2 Test script added
```

Now, commit `7c9c8d1` is following directly after commit `06933ed`.

Two comments are in order here. The SHA1 hash of the commit with the commit message `add __name__ to output` has changed from `e0ac1ba` to `7c9c8d1`. Rebasing thus will create significant problems if the commits of the development branch have been pushed to a remote branch and pulled from there by other developers before the rebasing has been carried out. It is therefore strongly recommended that rebasing is done only if local commits are applied. On the other hand, as we have seen, a rebase gives us the opportunity to take into account what has happened in the master branch and to resolve merge conflicts. In this way it can be avoided that a merge request containing commits from the development branch will contain merge conflicts with the master branch.

CHAPTER

THREE

TESTING OF CODE

3.1 Why testing?

Scientific code usually aims at producing new insights which implies that typically the correctness of the result is difficult to assess. Occasionally, bugs in the code can be identified because the results do not make any sense. In general the situation may not be that clear. Therefore testing the code is crucial. However, it is insufficient to test code from time to time in an informal manner. Instead, one should aim at a comprehensive set of tests which can be applied to the code at any time. Ideally, all code committed to version control should successfully run the existing tests. Test can then also serve as documentation of which kind of functionality is guaranteed to work. Furthermore, tests constitute an important safety net when refactoring code, i.e. when rewriting the inner structure of the code without affecting its external behavior. Tests running correctly for the old code should do so also for the refactored code.

Whenever a bug has been discovered in the code, it is a good habit to write one or more tests capable of detecting this bug. While it is barely possible to detect all imaginable bugs with tests, one should ensure at least that bugs which appeared once do not have a chance to sneak back into the code. Furthermore, one should add tests for any new functionality implemented. One indicator for the quality of a test suite, i.e. a collection of tests, is code coverage which says which percentage of lines of code are run during the tests. In practice, one rarely will reach one hundred percent code coverage but one should nevertheless strive for a good code coverage. At the same time, code coverage is not the only aspect to look for. One should also make sure that tests are independent from each other and independent of the logic of the code, if possible. The meaning of this will become clear in some of the examples presented later. Corner cases deserve special attention in testing as they are frequently ignored when setting up the logic of a program.

Tests can be developed in parallel to the code or even after the code has been written. A typical example for the latter is when the presence of a bug is noticed. Then, the bug should be fixed and a test should be implemented which will detect the presence of the fixed bug in the future. Another approach is the so-called *test-driven development* where the tests are first written. In a second step, the code is developed until all test run successfully.

Testing a big piece of software is usually difficult to do and as mentioned in the beginning in the case of scientific software can be almost impossible because one cannot anticipate the result beforehand. Therefore, one often tests on a much finer level, an approach called *unit testing*. Here, typically relatively small functions are tested to make sure that they work as expected. An interesting side effect of unit testing is often a significant improvement in code structure. Often a function needs to be rewritten in order to be tested properly. It often needs to be better isolated from the rest of the code and its interface has to be defined more carefully, thereby improving the quality of the code.

In this chapter, we will be concerned with unit testing and mainly cover two approaches. The first one are doctests which are implemented in Python within the doc strings of a function or method. The second approach are unit tests based on asserts using `py.test`.

3.2 Doctests

The standard way to document a function in Python is a so-called docstring as shown in the following example.

```
# hello.py

def welcome(name):
```

(continues on next page)

(continued from previous page)

```
    """Print a greeting.

    name: name of the person to greet
    """
    return f'Hello {name}!'
```

In our example, the docstring is available as `welcome.__doc__` and can also be obtained by means of `help(welcome)`.

Even though we have not formulated any test, we can run the (non-existing) doctests:

```
$ python -m doctest hello.py
```

No news is good news, i.e. the fact that this command does not yield any output implied that no error occurred in running the tests. One can ask doctest to be more verbose by adding the option `-v`:

```
$ python -m doctest -v hello.py
1 items had no tests:
    hello.py
0 tests in 1 items.
0 passed and 0 failed.
Test passed.
```

This message states explicitly that no tests have been run and no tests have failed.

We now add our first doctest. Doing so is quite straightforward. One simply reproduces how a function call together with its output would look in the Python shell.

```
# hello.py

def welcome(name):
    """Print a greeting.

    name: name of the person to greet

    >>> welcome('Alice')
    'Hello Alice!'
    """
    return f'Hello {name}!'
```

Running the test with the option `-v` to obtain some output, we find:

```
$ python -m doctest -v hello.py
Trying:
    welcome('Alice')
Expecting:
    'Hello Alice!'
ok
1 items had no tests:
    hello
1 items passed all tests:
    1 tests in hello.welcome
1 tests in 2 items.
1 passed and 0 failed.
Test passed.
```

Our test passes as expected. It is worth noting that besides providing a test, the last two lines of the new doc string can also serve as a documentation of how to call the function `welcome`.

Now let us add a corner case. A special case occurs if no name is given. Even in this situation, the function should behave properly. However, an appropriate test will reveal in a second that we have not sufficiently considered this corner case when designing our function.

```
1  # hello.py
2  
3  def welcome(name):
4      """Print a greeting.
5  
6      name: name of the person to greet
7  
8      >>> welcome('')
9      'Hello!'
10     >>> welcome('Alice')
11     'Hello Alice!'
12     """
13     return f'Hello {name}!'
```

Running the doctests, we identify our first coding error by means of a test:

```
$ python -m doctest hello.py
**********************************************************************
File "hello.py", line 8, in hello.welcome
Failed example:
    welcome('')
Expected:
    'Hello!'
Got:
    'Hello !'
**********************************************************************
1 items had failures:
   1 of   2 in hello.welcome
***Test Failed*** 1 failures.
```

The call specified in line 8 of our script failed because we implicitly add a blank which should not be there. So let us modify our script to make the tests pass.

```
# hello.py

def welcome(name):
    """Print a greeting.

    name: name of the person to greet

    >>> welcome('')
    'Hello!'
    >>> welcome('Alice')
    'Hello Alice!'
    """
    if name:
        return f'Hello {name}!'
    else:
        return 'Hello!'
```

Now the tests pass successfully.

If now we decide to change our script, e.g. by giving a default value to the variable name, we can use the tests as a safety net. They should run for the modified script as well.

```
# hello.py

def welcome(name=''):
    """Print a greeting.

    name: name of the person to greet

    >>> welcome('')
    'Hello!'
    >>> welcome('Alice')
    'Hello Alice!'
    """
    if name:
        return f'Hello {name}!'
    else:
        return 'Hello!'
```

Both tests pass successfully. However, we have not yet tested the new default value for the variable `name`. So, let us add another test to make sure that everything works fine.

```
# hello.py

def welcome(name=''):
    """Print a greeting.

    name: name of the person to greet

    >>> welcome()
    'Hello!'
    >>> welcome('')
    'Hello!'
    >>> welcome('Alice')
    'Hello Alice!'
    """
    if name:
        return f'Hello {name}!'
    else:
        return 'Hello!'
```

All three tests pass successfully.

In a next step development step, we make the function `welcome` multilingual.

```
# hello.py

def welcome(name='', lang='en'):
    """Print a greeting.

    name: name of the person to greet

    >>> welcome()
    'Hello!'
    >>> welcome('')
    'Hello!'
    >>> welcome('Alice')
    'Hello Alice!'
    >>> welcome('Alice', lang='de')
    'Hallo Alice!'
```

(continues on next page)

(continued from previous page)

```
    """
    hellodict = {'en': 'Hello', 'de': 'Hallo'}
    hellostring = hellodict[lang]
    if name:
        return f'{hellostring} {name}!'
    else:
        return f'{hellostring}!'
```

It is interesting to consider the case where the value of `lang` is not a valid key. Calling the function with `lang` set to `fr`, one obtains:

```
$ python hello.py
Traceback (most recent call last):
  File "hello.py", line 25, in <module>
    welcome('Alice', 'fr')
  File "hello.py", line 18, in welcome
    hellostring = hellodict[lang]
KeyError: 'fr'
```

Typically, error messages related to exception can be quite complex and it is either cumbersome to reproduce them in a test or depending on the situation it might even by impossible. One might think that the complexity of an error message is irrelevant because error messages should not occur in the first place. However, there are two reasons to consider such a situation. First, it is not uncommon that an appropriate exception is raised and one should check in a test whether it is properly raised. Second, more complex outputs appear not only in the context of exceptions and one should know ways to handle such situations.

Let us assume that we handle the `KeyError` by raising a `ValueError` together with an appropriate error message.

```
# hello.py

def welcome(name='', lang='en'):
    """Print a greeting.

    name: name of the person to greet

    >>> welcome()
    'Hello!'
    >>> welcome('')
    'Hello!'
    >>> welcome('Alice')
    'Hello Alice!'
    >>> welcome('Alice', lang='de')
    'Hallo Alice!'
    >>> welcome('Alice', lang='fr')
    Traceback (most recent call last):
    ValueError: unknown language: fr
    """
    hellodict = {'en': 'Hello', 'de': 'Hallo'}
    try:
        hellostring = hellodict[lang]
    except KeyError:
        errmsg = f'unknown language: {lang}'
        raise ValueError(errmsg)
    if name:
        return f'{hellostring} {name}!'
    else:
        return f'{hellostring}!'
```

(continues on next page)

(continued from previous page)

```
30
31 if __name__ == '__main__':
32     welcome('Alice', 'fr')
```

All tests run successfully. Note that in lines 17 and 18 we did not reproduce the full traceback. It was sufficient to put line 17 which signals that the following traceback can be ignored. Line 18 is checked again to be consistent with the actual error message. If one does not need to verify the error message but just the type of exception raised, one can use a doctest directive. For example, one could replace lines 16 to 18 by the following code.

```
"""
>>> welcome('Alice', lang='fr') # doctest: +ELLIPSIS
Traceback (most recent call last):
ValueError: ...
"""
```

The directive is here specified by the comment "`# doctest: +ELLIPSIS`" and the ellipsis "`...`" in the last line will replace any output following the text "`ValueError:`".

Another useful directive is `+SKIP` which tells doctest to skip the test marked in this way. Sometimes, one has already written a test before the corresponding functionality has been implemented. Then it may make sense to temporarily deactivate the test to avoid getting distracted from seriously failing tests by tests which are known beforehand to fail. A complete list of directives can be found in the doctest documentation. For example, it is worth to check out the directive `+NORMALIZE_WHITESPACE` which helps avoiding trouble with different kinds of white spaces.

As we have seen, doctests are easy to write and in addition to testing code they are helpful in documenting the usage of functions or methods. On the other hand, they are particularly well suited for numerical tests where results have to agree only to a certain precision. For more complex test cases, it might also be helpful to choose the approach discussed in the next section instead of using doctests.

3.3 Testing with pytest

For more complex test cases, the Python standard library provides a framework called `unittest`. Another often used test framework is `nose`. Recently, `pytest` has become very popular which compared `unittest` requires less overhead when writing tests. In this section we will focus on `pytest` which is not part of the Python standard library but is included e.g. in the Anaconda distribution.

We illustrate the basic usage of `pytest` by testing a function generating a line of Pascal's triangle.

```
def pascal(n):
    """create the n-th line of Pascal's triangle

    The line numbers start with n=0 for the line
    containing only the entry 1. The elements of
    a line are generated successively.

    """
    x = 1
    yield x
    for k in range(n):
        x = x*(n-k)//(k+1)
        yield x

if __name__ == '__main__':
    for n in range(7):
        line = ' '.join(f'{x:2}' for x in pascal(n))
        print(str(n)+line.center(25))
```

Running this script returns the first seven lines of Pascal's triangle:

```
$ python pascal.py
0             1
1           1  1
2          1  2  1
3        1  3  3  1
4       1  4  6  4  1
5     1  5 10 10  5  1
6    1  6 15 20 15  6  1
```

We will now test the function `pascal(n)` which returns the elements of the n-th line of Pascal's triangle. The function is based on the fact that the elements of Pascal's triangle are binomial coefficients. While the output of the first seven lines looks fine, it make sense to test the function more thoroughly.

The first and most obvious test is to automate at least part of the test which we were just doing visually. It is always a good idea to check boundary cases. In our case this means that we make sure that `n=0` indeed corresponds to the first line. We also check the following line as well as a typical non-trivial line. We call the following script `test_pascal.py` because `pytest` will run scripts with names of the form `test_*.py` or `*_test.py` in the present directory or its subdirectories automatically. Here, the star stands for any other valid part of a filename. Within the script, the test functions should start with `test_` to distinguish them from other functions which may be present.

```python
from pascal import pascal

def test_n0():
    assert list(pascal(0)) == [1]

def test_n1():
    assert list(pascal(1)) == [1, 1]

def test_n5():
    expected = [1, 4, 6, 4, 1]
    assert list(pascal(5)) == expected
```

The tests contain an `assert` statement which raises an `AssertionError` in case the test should fail. In fact, this will happen for our test script, even though the implementation of the function `pascal` is not to blame. In this case, we have inserted a mistake into our test script to show the output of `pytest` in the case of errors. Can you find the mistake in the test script? If not, it suffices to run the script:

```
$ pytest
============================= test session starts ==============================
platform linux -- Python 3.6.6, pytest-3.8.0, py-1.6.0, pluggy-0.7.1
rootdir: /home/gert/pascal, inifile:
plugins: remotedata-0.3.0, openfiles-0.3.0, doctestplus-0.1.3, arraydiff-0.2
collected 3 items

test_pascal.py ..F                                                      [100%]

=================================== FAILURES ===================================
___________________________________ test_n5 ____________________________________

    def test_n5():
        expected = [1, 4, 6, 4, 1]
>       assert list(pascal(5)) == expected
E       assert [1, 5, 10, 10, 5, 1] == [1, 4, 6, 4, 1]
E         At index 1 diff: 5 != 4
E         Left contains more items, first extra item: 1
E         Use -v to get the full diff

test_pascal.py:11: AssertionError
```

(continues on next page)

(continued from previous page)

```
===================== 1 failed, 2 passed in 0.04 seconds ======================
```

The last line in the first part of the output, before the header entitled `FAILURES`, `pytest` gives a summary of the test run. It ran three tests present in the script `test_pascal.py` and the result is indicated by `..F`. The two dots represent two successful tests and the `F` marks test which failed and for which detailed information is given in the second part of the output. Clearly, the elements of line 5 in Pascal's triangle yielded by our function does not coincide with our expectation.

It occasionally happens that a test is known to fail in the present state of development. One still may want to keep the test in the test suite, but it should not be flagged as failure. In such a case, the test can be decorated with `pytest.mark.xfail`. Even though decorators can be used without knowing how they work, it can be useful to have an idea of this concept. A brief introduction to decorators is given in Section 8.1.

The relevant test then looks as follows

```
@pytest.mark.xfail
def test_n5():
    expected = [1, 4, 6, 4, 1]
    assert list(pascal(5)) == expected
```

In addition, the `pytest` module need to be imported. Now, the test is marked by an `x` for expected failure:

```
$ pytest
============================= test session starts ==============================
platform linux -- Python 3.6.6, pytest-3.8.0, py-1.6.0, pluggy-0.7.1
rootdir: /home/gert/pascal, inifile:
plugins: remotedata-0.3.0, openfiles-0.3.0, doctestplus-0.1.3, arraydiff-0.2
collected 3 items

test_pascal.py ..x                                                       [100%]

===================== 2 passed, 1 xfailed in 0.04 seconds =====================
```

The marker `x` is set in lowercase to distinguish it from serious failures like `F` for a failed test. If a test expected to fail actually passes, it will be marked by an uppercase `X` to indicate that corresponding test should not pass.

One can also skip tests by means of the decorator `pytest.mark.skip` which takes an optional variable `reason`.

```
@pytest.mark.skip(reason="just for demonstration")
def test_n5():
    expected = [1, 4, 6, 4, 1]
    assert list(pascal(5)) == expected
```

However, the reason will only be listed in the output, if the option `-r s` is applied:

```
$ pytest -r s
============================= test session starts ==============================
platform linux -- Python 3.6.6, pytest-3.8.0, py-1.6.0, pluggy-0.7.1
rootdir: /home/gert/pascal, inifile:
plugins: remotedata-0.3.0, openfiles-0.3.0, doctestplus-0.1.3, arraydiff-0.2
collected 3 items

test_pascal.py ..s                                                       [100%]
=========================== short test summary info ============================
SKIP [1] test_pascal.py:10: just for demonstration

===================== 2 passed, 1 skipped in 0.01 seconds =====================
```

In our case, it is of course better to correct the expected result in function `test_n5`. The we obtain the following output from `pytest`:

```
$ pytest
============================== test session starts =============================
platform linux -- Python 3.6.6, pytest-3.8.0, py-1.6.0, pluggy-0.7.1
rootdir: /home/gert/pascal, inifile:
plugins: remotedata-0.3.0, openfiles-0.3.0, doctestplus-0.1.3, arraydiff-0.2
collected 3 items

test_pascal.py ...                                                      [100%]

========================== 3 passed in 0.01 seconds ============================
```

Now, all tests pass just fine.

One might object that the tests so far only verify a few special cases and in particular are limited to very small values of `n`. How do we test line 10000 of Pascal's triangle without having to determine the expected result? We can test properties related to the fact that the elements of Pascal's triangle are binomial coefficients. The sum of the elements in the n-th line amounts to 2^n and if the sign is changed from element to element the sum vanishes. This kind of test is quite independent of the logic of the function `pascal` and therefore particularly significant. We can implement the two tests in the following way.

```
def test_sum():
    for n in (10, 100, 1000, 10000):
        assert sum(pascal(n)) == 2**n

def test_alternate_sum():
    for n in (10, 100, 1000, 10000):
        assert sum(alternate(pascal(n))) == 0

def alternate(g):
    sign = 1
    for elem in g:
        yield sign*elem
        sign = -sign
```

Here, the name of the function `alternate` does not start with the string `test` because this function is not intended to be executed as a test. Instead, it serves to alternate the sign of subsequent elements used in the test `test_alternate_sum`. One can verify that indeed five tests are run. For a change, we use the option `-v` for a verbose output listing the name of the test functions being executed.

```
$ pytest -v
============================ test session starts =============================
platform linux -- Python 3.6.6, pytest-3.8.0, py-1.6.0, pluggy-0.7.1 -- /home/gert/
↪anaconda3/bin/python
cachedir: .pytest_cache
rootdir: /home/gert/pascal, inifile:
plugins: remotedata-0.3.0, openfiles-0.3.0, doctestplus-0.1.3, arraydiff-0.2
collected 5 items

test_pascal.py::test_n0 PASSED                                        [ 20%]
test_pascal.py::test_n1 PASSED                                        [ 40%]
test_pascal.py::test_n5 PASSED                                        [ 60%]
test_pascal.py::test_sum PASSED                                       [ 80%]
test_pascal.py::test_alternate_sum PASSED                             [100%]

========================= 5 passed in 0.10 seconds ===========================
```

We could also check whether a line in Pascal's triangle can be constructed from the previous line by adding neighboring elements. This test is completely independent of the inner logic of the function to be tested. Furthermore, we can execute it for arbitrary line numbers, at least in principle. We add the test

```
def test_generate_next_line():
    for n in (10, 100, 1000, 10000):
        for left, right, new in zip(chain([0], pascal(n)),
                                    chain(pascal(n), [0]),
                                    pascal(n+1)):
            assert left+right == new
```

where we need to add `from itertools import chain` in the import section of our test script.

The last three of our tests contain loops, but they do not behave like several tests. As soon as an exception is raised, the test has failed. In contrast our first three tests for the lines in Pascal's triangle with numbers 0, 1, and 5 are individual tests which could be unified. How can we do this while the keeping the individuality of the test? The answer is the `parametrize` decorator which we use in the following new version of our test script.

```
 1  import pytest
 2  from itertools import chain
 3  from pascal import pascal
 4
 5  @pytest.mark.parametrize("lineno, expected", [
 6      (0, [1]),
 7      (1, [1, 1]),
 8      (5, [1, 5, 10, 10, 5, 1])
 9  ])
10  def test_line(lineno, expected):
11      assert list(pascal(lineno)) == expected
12
13  powers_of_ten = pytest.mark.parametrize("lineno",
14                      [10, 100, 1000, 10000])
15
16  @powers_of_ten
17  def test_sum(lineno):
18      assert sum(pascal(lineno)) == 2**lineno
19
20  @powers_of_ten
21  def test_alternate_sum(lineno):
22      assert sum(alternate(pascal(lineno))) == 0
23
24  def alternate(g):
25      sign = 1
26      for elem in g:
27          yield sign*elem
28          sign = -sign
29
30  @powers_of_ten
31  def test_generate_next_line(lineno):
32      for left, right, new in zip(chain([0], pascal(lineno)),
33                                  chain(pascal(lineno), [0]),
34                                  pascal(lineno+1)):
35          assert left+right == new
```

The function `test_line` replaces the original first three tests. In order to do so, it takes two arguments which are provided by the decorator in lines 5 to 9. This decorator makes sure that the test function is run three times with different values of the line number in Pascal's triangle and the expected result. In the remaining three test functions, we have replaced the original loop by a `parametrize` decorator. In order to avoid repetitive code, we have defined a decorator `powers_of_ten` in line 13 and 14 which then is used in three tests. Our script now contains 15 tests.

When discussing doctests, we had seen how one can make sure that a certain exception is raised. Of course, this can also be achieved with `pytest`. At least in the present form, it does not make sense to call `pascal` with a negative value for the line number. In such a case, a `ValueError` should be raised, a behavior which can be tested with the following test.

```
def test_negative_int():
    with pytest.raises(ValueError):
        next(pascal(-1))
```

Here, `next` explicitly asks the generator to provide us with a value so that the function `pascal` gets a chance to check the validity of the line number. Of course, this test will only pass once we have adapted our function `pascal` accordingly.

In order to illustrate a problem frequently occurring when writing tests for scientific applications, we generalize our function `pascal` to floating point number arguments. As an example, let us choose the argument 1/3. We would then obtain the coefficients in the Taylor expansion

$$(1+x)^{1/3} = 1 + \frac{1}{3}x - \frac{1}{9}x^2 + \frac{5}{81}x^3 + \dots$$

Be aware that the generator will now provide us with an infinite number of return values so that we should take care not to let this happen. In the following script `pascal_float`, we do so by taking advantage of the fact that `zip` terminates whenever one of the generators is exhausted.

```
def taylor_power(power):
    """generate the Taylor coefficients of (1+x)**power

       This function is based on the function pascal().

    """
    coeff = 1
    yield coeff
    k = 0
    while power-k != 0:
        coeff = coeff*(power-k)/(k+1)
        k = k+1
        yield coeff

if __name__ == '__main__':
    for n, val in zip(range(5), taylor_power(1/3)):
        print(n, val)
```

We call this script `pascal_float.py` and obtain the following output by running it:

```
0 1
1 0.3333333333333333
2 -0.11111111111111112
3 0.0617283950617284
4 -0.0411522633744856
```

The first four lines match our expectations from the Taylor expansion of $(1+x)^{1/3}$.

We test our new function with the test script `test_taylor_power.py`.

```
import pytest
from pascal_float import taylor_power

def test_one_third():
    p = taylor_power(1/3)
    result = [next(p) for _ in range(4)]
    expected = [1, 1/3, -1/9, 5/81]
    assert result == expected
```

The failures section of the output of `pytest -v` shows where the problem lies:

```
______________________________ test_one_third _______________________________

    def test_one_third():
        p = taylor_power(1/3)
        result = [next(p) for _ in range(4)]
        expected = [1, 1/3, -1/9, 5/81]
>       assert result == expected
E       assert [1, 0.3333333...7283950617284] == [1, 0.33333333...2839506172839]
E         At index 2 diff: -0.11111111111111112 != -0.1111111111111111
E         Full diff:
E         - [1, 0.3333333333333333, -0.11111111111111112, 0.0617283950617284]
E         ?                                            -                    ^
E         + [1, 0.3333333333333333, -0.1111111111111111, 0.06172839506172839]
E         ?                                                               ^^

test_taylor_power.py:8: AssertionError
========================= 1 failed in 0.04 seconds ==========================
```

It looks like rounding errors spoil our test and this problem will get worse if we want to check further coefficients. We are thus left with two problems. First, one needs to have an idea of how well the actual and the expected result should agree. It is not straightforward to answer this, because the precision of a result may depend strongly on the numerical methods employed. For a numerical integration, a relative error of 10^{-8} might be perfectly acceptable while for a pure rounding error, this value would be too large. On a more practical side, how can we test in the presence of numerical errors?

There are actually a number of possibilities. The `math`-module of the Python standard library provides a function `isclose` which allows to check whether two numbers agree up to a given absolute or relative tolerance. However, one would have to compare each pair of numbers individually and then combine the Boolean results by means of `all`. When dealing with arrays, the NumPy library provides a number of useful functions in its `testing` module. Several of these functions can be useful when comparing floats. Finally, `pytest` itself provides a function `approx` which can test individual values or values collected in a list, a NumPy array, or even a dictionary. Using `pytest.approx`, our test could look as follows.

```python
import math
import pytest
from pascal_float import taylor_power

def test_one_third():
    p = taylor_power(1/3)
    result = [next(p) for _ in range(4)]
    expected = [1, 1/3, -1/9, 5/81]
    assert result == pytest.approx(expected, abs=0, rel=1e-15)
```

Here we test whether the relative tolerance between two values in a pair is at most 10^{-15}. By default, the absolute tolerance is set to 10^{-12} and the relative tolerance to 10^{-6} where in the end the larger value is taken. If we would not specify `abs=0`, a very small relative tolerance would be ignored in favor of the default absolute tolerance. On the other hand, if no relative tolerance is specified, the absolute tolerance is taken for the comparison.

`pytest.approx` and `math.isclose` differ when the relative tolerance is checked. While the first one takes the relative tolerance with respect to the argument of `pytest.approx`, the second one checks whether the relative tolerances are met with respect to both values.

In this section, we have discussed some of the more important aspects of `pytest` without being complete. More information can be found in the corresponding documentation. Of interest, in particular if more extensive tests are written, could be the possibility to group tests in classes. This can also be useful if a number of tests requires the same setup which then can be defined in a dedicated function.

CHAPTER

FOUR

SCIENTIFIC COMPUTING WITH NUMPY AND SCIPY

4.1 Python scientific ecosystem

Python comes with a rich variety of freely available third-party packages including quite a number of packages which are routinely used in scientific computing. Before developing code for a standard problem like an eigenvalue analysis or numerical quadrature to name just two examples, it is recommended to first check the functionality provided by the existing libraries. It is very likely that such libraries are more reliable and more efficient than self-developed code. This does not mean though that such libraries are guaranteed to be error-free and there may exist reasons to develop even basic numerical code oneself.

NumPy[1] constitutes the basis of the Python scientific ecosystem. The multi-dimensional array datatype defined in NumPy is pivotal for a huge number of scientific applications and is made use of in many ways in the other two core packages SciPy and matplotlib. SciPy provides submodules in many areas relevant for scientific applications like optimization, signal processing, linear algebra, statistics, special functions and several more[2]. Part of the code is written in C or Fortran resulting in fast execution speed. Matplotlib offers comprehensive support for graphical presentation of data[3].

In recent year, the Jupyter notebook, formerly known as IPython notebook, has become very popular, in particular among the data scientists[4]. The notebook can be used with Python and a number of other programming languages like Julia and R and allows to integrate code, text as well as images and other media in a single file.

In addition, there are a number of more dedicated packages of which we will name a few. The Python data analysis library pandas[5] offers high-performance, easy-to-use data structures and data analysis tools. Symbolic computation is possible in Python with the help of the sympy package[6]. Image-processing routines can be found in scikit-image[7]. Among the many features of this package, we mention image segmentation which can for example be used to analyse electron microscope images of heterogeneous surfaces. Machine learning has recently developed into a very active field which receives excellent support in Python through the scikit-learn package[8].

We emphasize that the list of packages briefly described here, is not exhaustive and there exist more interesting Python packages useful in scientific applications. A recommended source of information on the Python scientific ecosystem are the SciPy lecture notes.

4.2 NumPy

4.2.1 Python lists and matrices

It is rather typical in scientific applications to deal with homogeneous data, i.e. data of the same datatype, organized in arrays. An obvious example for one-dimensional arrays are vectors in their coordinate representation and matrices would naturally be stored in two-dimensional arrays. There also exist applications for even higher-dimensional arrays.

[1] For details see the NumPy Reference.

[2] For details see the SciPy API Reference.

[3] See the matplotlib gallery to obtain an idea of the possibilities offered by matplotlib.

[4] For details see the homepage of the Jupyter project.

[5] For details see the pandas homepage.

[6] For details see the sympy homepage.

[7] For details see the scikit-image homepage.

[8] For details see the scikit-learn homepage.

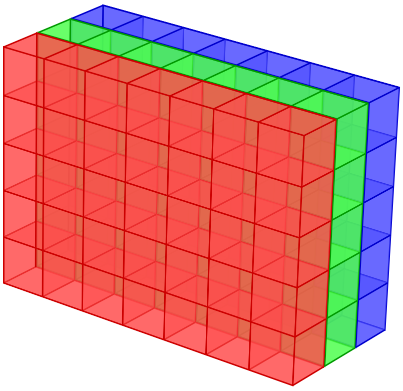


Figure 4.1: The data of a digital colour image composed of $N \times M$ pixels can be represented as a $N \times M \times 3$ array where the three planes correspond to the red, green, and blue channels.

The data representing a digital colour image composed of $N \times M$ pixels can be stored in an $N \times M \times 3$ array with three planes representing the three colour channels red, green, and blue as visualized in Figure 4.1.

The first question to address is how one can store such data structures in Python and how can one make sure that the data can be processed fast. Among the standard datatypes available in Python, a natural candidate would be lists. In Python, lists are very flexible objects which allow to store element of all kinds of datatypes including lists. While this offers us in principle the possibility to represent multi-dimensional data, the flexibility comes with a significant computational overhead. As we will see later, homogeneous data can be handled more efficiently. Leaving the question of efficiency aside for a moment, we can ask whether list are suited at all to represent matrices.

Let us consider a two-dimensional matrix

$$\mathsf{M} = \begin{pmatrix} 1.1 & 2.2 & 3.3 \\ 4.4 & 5.5 & 6.6 \\ 7.7 & 8.8 & 9.9 \end{pmatrix} .$$

It seems natural to store these data in a list of lists

```
>>> matrix = [[1.1, 2.2, 3.3], [4.4, 5.5, 6.6], [7.7, 8.8, 9.9]]
```

of which a single element can be accessed by first selecting the appropriate row and then the desired entry

```
>>> matrix[0]
[1.1, 2.2, 3.3]
>>> matrix[0][2]
3.3
```

The only difference with respect to the common mathematical notation is that the indices start at 0 and not at 1. In order to access a single row in a way which makes the two-dimensional character of the matrix more transparent, we could use

```
>>> matrix[0][:]
[1.1, 2.2, 3.3]
```

But does this also work for a column? Let us give it a try.

```
>>> matrix[:][0]
[1.1, 2.2, 3.3]
>>> matrix[:]
[[1.1, 2.2, 3.3], [4.4, 5.5, 6.6], [7.7, 8.8, 9.9]]
```

The result is rather disappointing because interchanging the two slices yields again the first row. The reason can be seen from the lower two lines. In the first step, we obtain again the full list and in the second step we access its first element, i.e. the first row, not the first column. Even though there are ways to extract a column from a list of lists, e.g. by means of a list comprehension, there is now consistent approach to extracting rows and columns from a list of lists. Our construction is certainly not a good one and we are in need of a new datatype.

4.2.2 NumPy arrays

The new datatype provided by NumPy is a multidimensional homogeneous array of fixed-size items called `ndarray`. Before starting to explore this datatype, we need to import the NumPy package. While there are different ways to do so, there is one recommended way. Let us take a look at the various alternatives:

```
from numpy import *                # don't do this!
from numpy import array, sin, cos  # not recommended
import numpy                       # ok, but the following line is better
import numpy as np                 # recommended way
```

Importing the complete namespace of NumPy as done in the first line is no good idea because the namespace is rather large. Therefore, there is a danger of name conflicts and loss of control. As an alternative, one could restrict the import to the functions actually needed as shown in the second line. However, as can be seen in our example, there exist functions like sine (`sin`) and cosine (`cos`) in NumPy. In the body of the code it might not always be evident whether these functions are taken from NumPy or rather the `math` or `cmath` module. It is better to more explicit. The import given in the third line is acceptable but it requires to put `numpy.` in front of each object taken from the NumPy namespace. The usual way to import NumPy is given in the fourth line. Virtually every user seeing `np.` in the code will assume that the corresponding object belongs to NumPy. It is always a good idea to stick to such conventions to render the code easily understandable.

As the next step, we need to create an array and fill it with data. Whenever we are simply referring to an array, we actually mean an object of datatype `ndarray`. Given certain similarities with Python lists, it is tempting to use the `append` method for that purpose as one often does with lists. In fact, NumPy provides an `append` method. However, because Python lists and NumPy arrays are conceptually quite different, there exist good reasons for avoiding this method if at all possible.

The objects contained in a Python list are typically scattered in memory and the position of each chunk of data is stored in a list of pointers. In contrast, the data of a NumPy array are stored in one contiguous piece of memory. As we will see later, this way of storing an array allows to determine by means of a simple calculation where a certain element can be found. Accessing elements therefore is very efficient.

When appending data to an array, there will generally be no place for the data in memory to guarantee the array to remain contiguous. Appending data in NumPy thus implies the creation of an entirely new array. As a consequence, the data constituting the original array have to be moved to a new place in memory. The time required for this process can become significant for larger arrays and ultimately is limited by the hardware. Using the `append` method can thus become a serious performance problem.

Generally, when working with NumPy arrays, it is a good idea to avoid the creation of new arrays as much as possible as this may drastically degrade performance. In particular, one should not count on changing the size of an array during the calculation. Already for the creation of the array one should decide how large it will need to be.

One way to find out how a NumPy array can be created it to search the NumPy documentation. This can be done even within Python:

```
>>> np.lookfor('create array')
Search results for 'create array'
---------------------------------
numpy.array
    Create an array.
numpy.memmap
    Create a memory-map to an array stored in a *binary* file on disk.
numpy.diagflat
    Create a two-dimensional array with the flattened input as a diagonal.
numpy.fromiter
    Create a new 1-dimensional array from an iterable object.
numpy.partition
    Return a partitioned copy of an array.
```

Here, we have only have reproduced a small part of the output. Furthermore, here and in the following, we assume that NumPy has been imported in the way recommended above so that its namespace can be accessed via the abbreviation np.

Already the first entry in the list of proposed methods is the one to use in our present situation. More information can be obtained as usual by means of help(np.array) or alternatively by

```
>>> np.info(np.array)
array(object, dtype=None, copy=True, order='K', subok=False, ndmin=0)

Create an array.

Parameters
----------
object : array_like
    An array, any object exposing the array interface, an object whose
    __array__ method returns an array, or any (nested) sequence.
dtype : data-type, optional
    The desired data-type for the array.  If not given, then the type will
    be determined as the minimum type required to hold the objects in the
    sequence.  This argument can only be used to 'upcast' the array.  For
    downcasting, use the .astype(t) method.
```

Again, only the first part of the output has been reproduced. It is recommended though to take a look at the rest of the help text as it provides a nice example how doctests can be used both for documentation purposes and for testing.

As can be seen from the help text, we need at least one argument object which should be an object with an __array__ method or a possibly nested sequence. Let us consider a first example:

```
>>> matrix = [[0, 1, 2],
...           [3, 4, 5],
...           [6, 7, 8]]
>>> myarray = np.array(matrix)
>>> myarray
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> type(myarray)
<class 'numpy.ndarray'>
```

We have started with a list of lists which is a valid argument for np.array. Printing out the result indicates indeed that we have obtained a NumPy array. A confirmation is obtained by asking for the type of myarray.

The data of an array are stored contiguously in memory but what does that really mean for the two-dimensional array which we have just created? Natural ways would be store the date columnwise or rowwise. The first variant is realized in the programming language C while the second variant is used by Fortran. Apart from the actual data, an array obviously needs a number of metadata in order to know how to interpret the content of the memory space attributed to the area. These metadata are a powerful concept because they make it possible to change the interpretation of the data without copying them, thereby contributing to the efficiency of NumPy arrays.

It is useful to get some basic insight into how a NumPy array works. In order to analyze the metadata, we use a short function enabling us to list the attributes of an array.

```
def array_attributes(a):
    for attr in ('ndim', 'size', 'itemsize', 'dtype', 'shape', 'strides'):
        print(f'{attr:8s}: {getattr(a, attr)}')
```

A convenient way of generating an array for test purposes is the arange function which works very much like the standard range iterator as far as its basic arguments start, stop, and step are concerned. In this way, we can easily construct a one-dimensional array with integer entries from 0 to 15 and inspect its properties:

```
>>> matrix = np.arange(16)
>>> matrix
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15])
>>> array_attributes(matrix)
ndim    : 1
size    : 16
itemsize: 8
dtype   : int64
shape   : (16,)
strides : (8,)
```

Let us take a look at the different attributes. The attribute `ndim` indicates the number of dimension of the array which in our example is one-dimensional and therefore `ndim` equals 1. The `size` of 16 means that the array contains a total of 16 items. Each item has an `itemsize` of 8 bytes or 64 bits, resulting in a total size of 128 bytes:

```
>>> matrix.nbytes
128
```

The attribute `dtype` represents the datatype which in our example is `int64`, i.e. an integer type of a length of 64 bits. Quite in contrast to the usual integer type in Python which can in principle handle integers of arbitrary size, the integer values in our array are clearly limited. An example using integers of only 8 bits length can serve to illustrate the problem of overflows:

```
>>> np.arange(1, 160, 10, dtype=np.int8)
array([   1,   11,   21,   31,   41,   51,   61,   71,   81,   91,  101,
        111,  121, -125, -115, -105], dtype=int8)
```

Take a look at the items in this array and try to understand what is going on.

```
>>> for k, v in np.core.numerictypes.sctypes.items():
...     print(k)
...     for elem in v:
...         print(f'    {elem}')
...
int
    <class 'numpy.int8'>
    <class 'numpy.int16'>
    <class 'numpy.int32'>
    <class 'numpy.int64'>
uint
    <class 'numpy.uint8'>
    <class 'numpy.uint16'>
    <class 'numpy.uint32'>
    <class 'numpy.uint64'>
float
    <class 'numpy.float16'>
    <class 'numpy.float32'>
    <class 'numpy.float64'>
    <class 'numpy.float128'>
complex
    <class 'numpy.complex64'>
    <class 'numpy.complex128'>
    <class 'numpy.complex256'>
others
    <class 'bool'>
    <class 'object'>
    <class 'bytes'>
    <class 'str'>
```

(continues on next page)

(continued from previous page)

```
    <class 'numpy.void'>
```

The first four groups of datatypes include integers, unsigned integers, floats and complex numbers of different sizes. Among the other types, booleans as well as strings are of some interest. Note, however, that the data in an array always should be homogeneous. If different datatypes are mixed in the assignment to an array, it may happen that a datatype is cast to a more flexible one. For strings, the size of each entry will be determined by the longest string.

Probably the most interesting attributes of an array are `shape` and `strides` because the allow us to reinterpret the data of the original one-dimensional array in different ways without the need to copy from memory to memory. Let us first try to understand the meaning of the tuples `(16,)` for `shape` and `(8,)` for `strides`. Both tuples have the same size which equals one because the considered array is one-dimensional. Therefore, `shape` does not contain any new information. It simply reflects the size of the array as does the attribute `size`. The value of `strides` means that in order to move from the beginning of an item in memory to the beginning of the next one, one needs to more eight bytes. This information is consistent with the `itemsize`. What seems like redundant information becomes more interesting when we go from a one-dimensional array to a multi-dimensional array. For simplicity we convert the our one-dimensional array `matrix` into a two-dimensional square array. To this purpose we make use of the `reshape` method:

```
>>> matrix = matrix.reshape(4, 4)
>>> matrix
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
>>> array_attributes(matrix)
ndim    : 2
size    : 16
itemsize: 8
dtype   : int64
shape   : (4, 4)
strides : (32, 8)
```

In the first line, we bring our one-dimensional array with 16 elements into a 4×4 array. Three attributes change their value in this process. `ndim` is now 2 because we created a two-dimensional array. The `shape` attribute with value `(4, 4)` reflects the fact that now we have 4 rows and 4 columns. Finally, the `strides` are given by the tuple `(32, 8)`. To go in memory from an item to the item in the next column and in the same row means that we should move by 8 bytes. The two items are neighbors in memory. However, if we stay within the same column and want to move to the next row, we have to jump by 32 bytes in memory.

To further illustrate the meaning of `shape` and `strides` we consider a second example. A linear arrangement of six data in memory can be interpreted in three different ways as depicted in Figure 4.2. In the uppermost example, `strides` is set to `(8,)`. The tuple `strides` tuple contains only one element and we are therefore dealing with a one-dimensional array. Assuming the datasize to be 8, the array consists of all six data elements. In the second case, `strides` are set to `(24, 8)`. Accordingly, the matrix consists of two rows and three columns. Finally, in the bottom example with `strides` equal to `(16, 8)`, the data are interpreted as a matrix consisting of two columns and three rows. Note that no rearrangement of data in memory is required in order to go from one matrix to another one. Only the way, how the position of a certain element in memory is obtained, changes when `strides` is modified.

A two-dimensional matrix can easily be transposed. Behind the scenes the values in the `strides` tuple are interchanged:

```
>>> a = np.arange(9).reshape(3, 3)
>>> a
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> a.strides
(24, 8)
```

(continues on next page)

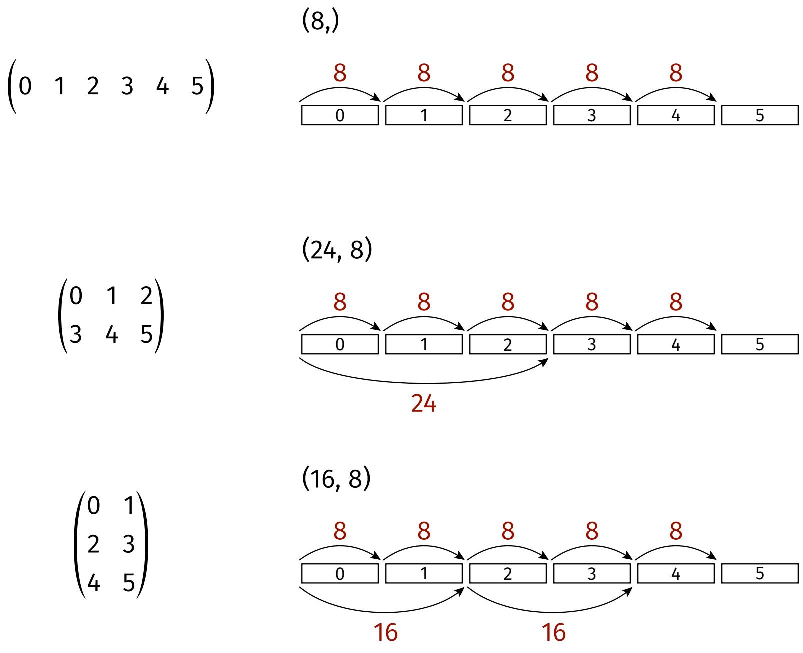


Figure 4.2: Linear data in memory can be interpreted in different ways by appropriately choosing the `strides` tuple.

(continued from previous page)

```
>>> a.T
array([[0, 3, 6],
       [1, 4, 7],
       [2, 5, 8]])
>>> a.T.strides
(8, 24)
```

Strides are a powerful concept. However, one should be careful not to violate the boundaries of the data because otherwise memory might be interpreted in a meaningless way. In the following two examples, the first demonstrates an interesting way to create a special pattern of data. The second example, where one of the strides is only half of the datasize, shows how useless results can be produced:

```
>>> a = np.arange(16).reshape(4, 4)
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
>>> a.strides = (8, 8)
>>> a
array([[0, 1, 2, 3],
       [1, 2, 3, 4],
       [2, 3, 4, 5],
       [3, 4, 5, 6]])
>>> a.strides = (8, 4)
>>> a
array([[          0,  4294967296,           1,  8589934592],
       [          1,  8589934592,           2, 12884901888],
       [          2, 12884901888,           3, 17179869184],
       [          3, 17179869184,           4, 21474836480]])
```

In the end, the user manipulating `strides` is responsible for all consequences which his or her action may have.

4.2.3 Creating arrays

We have seen in the previous section that an array can be created by providing `np.array` with an object possessing an `__array__` method or a nested sequence. However, this requires to create the object or nested sequence in the first place. Often, more convenient methods exist. As we have pointed out earlier, when creating an array, one should have an idea of the desired size and usually also of the datatype to be stored in the array. Given this information, there exists a variety of methods to create an array depending on the specific needs.

It is not unusual to start with an array filled with zeros. Let us create a 2×2 array:

```
>>> a = np.zeros((2, 2))
>>> a
array([[0., 0.],
       [0., 0.]])
>>> a.dtype
dtype('float64')
```

As we can see, the default type is `float64`. If we prefer an array of integers, we could specify the `dtype`:

```
>>> a = np.zeros((2, 2), dtype=int)
>>> a
array([[0, 0],
       [0, 0]])
>>> a.dtype
dtype('int64')
```

As an alternative, one can create an empty array which should however not be confused with an array filled with zeros. An empty array will just claim the necessary amount of memory without doing anything to the data present in that piece of memory. This is fine if one is going to specify the content of all array data subsequently before using the array. Otherwise, one will deal with random data:

```
>>> np.empty((3, 3))
array([[6.94870988e-310, 6.94870988e-310, 7.89614591e+150],
       [1.37038197e-013, 2.08399685e+064, 3.51988759e+016],
       [8.23900250e+015, 7.32845376e+025, 1.71130458e+059]])
```

An alternative to filling an array with zeros could be to fill it with ones or another value which can be obtained by multiplication:

```
>>> np.ones((2, 2))
array([[1., 1.],
       [1., 1.]])
>>> 10*np.ones((2, 2))
array([[10., 10.],
       [10., 10.]])
```

As one can see in this example, the multiplication by a number acts on all elements of the array. This behavior is probably what one would expect at this point. As we will see in Section 4.2.5, we are here making use of a more general concept referred to as broadcasting.

Often, one needs arrays with more structure than the one we have created so far. It is not uncommon, that the diagonal entries take a special form. An identity matrix can easily be created:

```
>>> np.identity(3)
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

The result will always be a square matrix. A more general method to fill the diagonal or a shifted diagonal is provided by `np.eye`:

```
>>> np.eye(2, 4)
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.]])
>>> np.eye(4, k=1)
array([[0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.],
       [0., 0., 0., 0.]])
>>> 2*np.eye(4)-np.eye(4, k=1)-np.eye(4, k=-1)
array([[ 2., -1.,  0.,  0.],
       [-1.,  2., -1.,  0.],
       [ 0., -1.,  2., -1.],
       [ 0.,  0., -1.,  2.]])
```

These examples show that `np.eye` does not expect a tuple specifying the shape. Instead, the first two arguments give the number of rows and columns. If the second argument is absent, the resulting matrix is a square matrix. In the second and third example, the missing second argument is the reason why we have to specify that the second argument is intended as the shift `k` of the diagonal. The third example gives an idea of how the Hamiltonian for the kinetic energy in a tight-binding model can be constructed.

It is also possible to generate diagonals or, by specifying `k`, shifted diagonals with different values:

```
>>> np.diag([1, 2, 3, 4])
array([[1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 0, 4]])
```

Using a two-dimensional array as argument, its diagonal elements can be extracted by means of the same function:

```
>>> matrix = np.arange(16).reshape(4, 4)
>>> matrix
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
>>> np.diag(matrix)
array([ 0,  5, 10, 15])
```

If the elements of an array can be expressed as a function of the indices, `fromfunction` can be used to generate the elements. As a simple example, we create a multiplication table:

```
>>> np.fromfunction(lambda i, j: (i+1)*(j+1), shape=(6, 6), dtype=int)
array([[ 1,  2,  3,  4,  5,  6],
       [ 2,  4,  6,  8, 10, 12],
       [ 3,  6,  9, 12, 15, 18],
       [ 4,  8, 12, 16, 20, 24],
       [ 5, 10, 15, 20, 25, 30],
       [ 6, 12, 18, 24, 30, 36]])
```

Even though we present a two-dimensional example, the latter approach can be used to create arrays of an arbitrary dimension.

The function used in the previous example was a very simple one. Occasionally, one might need more complicated functions like one of the trigonometric functions. In fact, NumPy provides a number of so-called universal functions which we will discuss in Section 4.2.6. Such functions accept an array as argument and return an array. Here, we will concentrate on creating arguments for universal functions.

A first function is `arange` which we have used before for integers. It is a generalization of the standard `range` which works even for floats:

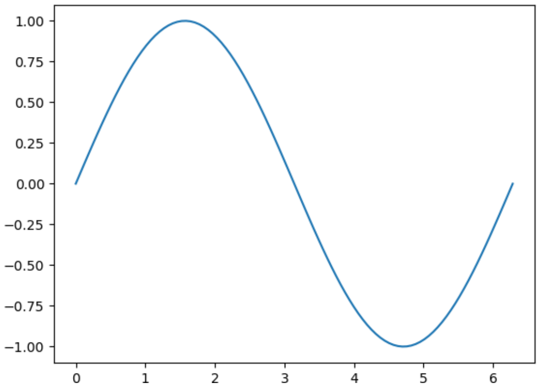


Figure 4.3: Simple example of a function graph generated by operating with a universal function on an array generated by `linspace`.

```
>>> np.arange(1, 2, 0.1)
array([1. , 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9])
```

As with `range`, the first argument is the start value while the second argument refers to the final value which is not included. Because of rounding errors, the last statement is not always true. Finally, the third argument is the stepwidth. An alternative is offered by the `linspace` function which by default will make sure that the start value and the final value are part of the array. Instead of the stepwidth, the number of points is specified:

```
>>> np.linspace(1, 2, 11)
array([1. , 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2. ])
```

A common mistake is to assume that the last argument gives the number of intervals which, however, is not the case. Thus, there is some danger that one is off by one in the last argument. Sometimes it is useful to ask for the stepwidth:

```
>>> np.linspace(1, 4, 7, retstep=True)
(array([1. , 1.5, 2. , 2.5, 3. , 3.5, 4. ]), 0.5)
```

Here, the stepwidth does not need to be determined by hand.

Occasionally, a logarithmic scale can be useful. In this case, the start value and the final value refer to the exponent. The base by default is ten but can be modified, if necessary:

```
>>> np.logspace(0, 3, 4)
array([   1.,   10.,  100., 1000.])
>>> np.logspace(0, 2, 5, base=2)
array([1.        , 1.41421356, 2.        , 2.82842712, 4.        ])
```

The following example illustrate the application of `linspace` in a universal function to produce a graphical representation of the function:

```
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(0, 2*np.pi, 100)
>>> y = np.sin(x)
>>> plt.plot(x, y)
[<matplotlib.lines.Line2D object at 0x7f22d619cc88>]
```

The generated graph is reproduced in Figure 4.3.

Arrays can also be filled with data taken from a file. This can for example be the case if data obtained from a measurement are first stored in a file before being processed or if numerical data are stored before a graphical representation is produced. Assume that we have a data file called `mydata.dat` with the following content:

```
# time position
   0.0    0.0
   0.1    0.1
   0.2    0.4
   0.3    0.9
```

Loading the data from the file, we obtain:

```
>>> np.loadtxt('mydata.dat')
array([[0. , 0. ],
       [0.1, 0.1],
       [0.2, 0.4],
       [0.3, 0.9]])
```

By default, lines starting with # will be considered as comments and are ignored. The function loadtxt offers a number of arguments to load data in a rather flexible way. Even more possibilities are offered by genfromtxt which is also able to deal with missing values. See the documentation of loadtxt and genfromtxt for more information.

In numerical simulations, it is often necessary to generate random numbers and if many of them are needed, it may be efficient to generate an array filled with random numbers. While NumPy offers many different distributions of random numbers, we concentrate on equally distributed random numbers in an interval from 0 to 1. An array of a given shape filled with such random numbers can be obtained as follows:

```
>>> np.random.rand(2, 5)
array([[0.76455979, 0.09264023, 0.47090143, 0.81327348, 0.42954314],
       [0.37729755, 0.20315983, 0.62982297, 0.0925838 , 0.37648008]])
>>> np.random.rand(2, 5)
array([[0.23714395, 0.22286043, 0.97736324, 0.19221663, 0.18420108],
       [0.14151036, 0.07817544, 0.4896872 , 0.90010128, 0.21834491]])
```

Clearly, the set of random numbers changes at each call to random.rand. Occasionally, one would like to have reproducible random numbers, for example during unit tests or to reproduce a particularly interesting scenario in a simulation. Then one can set a seed:

```
>>> np.random.seed(123456)
>>> np.random.rand(2, 5)
array([[0.12696983, 0.96671784, 0.26047601, 0.89723652, 0.37674972],
       [0.33622174, 0.45137647, 0.84025508, 0.12310214, 0.5430262 ]])
>>> np.random.seed(123456)
>>> np.random.rand(2, 5)
array([[0.12696983, 0.96671784, 0.26047601, 0.89723652, 0.37674972],
       [0.33622174, 0.45137647, 0.84025508, 0.12310214, 0.5430262 ]])
```

Sometimes, it is convenient to graphically represent the matrix elements. Figure 4.4 shows an example generated by the following code:

```
>>> import matplotlib.pyplot as plt
>>> np.random.seed(42)
>>> data = np.random.rand(20, 20)
>>> plt.imshow(data, cmap=plt.cm.hot, interpolation='none')
<matplotlib.image.AxesImage object at 0x7f39027afe48>
>>> plt.colorbar()
<matplotlib.colorbar.Colorbar object at 0x7f39027e58d0>
>>> plt.show()
```

Not that the argument interpolation of plt.imshow is set to 'none' to ensure that no interpolation is done which might blur the image.

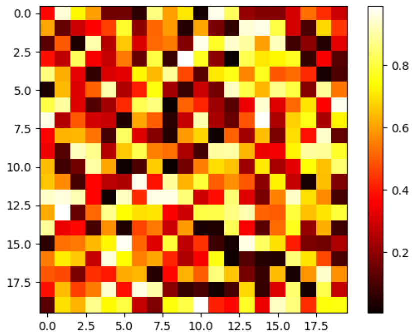


Figure 4.4: Graphical representation of an array filled with random numbers.

4.2.4 Indexing arrays

One way of accessing sets of elements of an array makes use of slices which we know from Python lists. A slice is characterized by a `start` index, a `stop` index whose corresponding element is excluded, and `step` which indicates the stepsize. Negative indices are counted from the end of the corresponding array dimension and a negative value of `step` implies walking in the direction of decreasing indices.

We start by a few examples of slicing for a one-dimensional array:

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[:]
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[1:4]
array([1, 2, 3])
>>> a[5:-2]
array([5, 6, 7])
>>> a[::2]
array([0, 2, 4, 6, 8])
>>> a[1::2]
array([1, 3, 5, 7, 9])
>>> a[::-1]
array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
```

The third input, i.e. `a[:]` leaves the `start` and `stop` values open so that all array elements are returned because by default `step` equals 1. In the next example, we recall that indices in an array as in a list start at 0. Therefore, we obtain the second up to the fourth element of the array. In the fifth input, the second element counted from the end of the array is not part of the result so that we obtain the numbers from 5 to 7. We could have used `a[5:8]` instead. In the sixth input, `start` and `stop` values are again left open, so that the resulting array starts with 0 but then proceeds in steps of 2 according to the value of `step` given. In the following example, `start` is set to 1 and we obtain the elements left out in the previous example. The last example inverts the sequence of array elements by specifying a `step` of -1.

The use of `a[:]` deserves a bit more attention. In the case of a list, it would yield a shallow copy of the original list. For an array, the behavior is somewhat different. Let us first consider an alias:

```
>>> b = a
>>> b
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> id(a), id(b)
(140493158678656, 140493158678656)
>>> b[0] = 42
```

(continues on next page)

(continued from previous page)

```
>>> a
array([42,  1,  2,  3,  4,  5,  6,  7,  8,  9])
```

In this case, `b` is simply an alias for `a` and refers to the same object. A modification of elements of `b` will also be visible in `a`. Now, let us consider a slice comprising all elements:

```
>>> a = np.arange(10)
>>> b = a[:]
>>> b
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> id(a), id(b)
(140493155003008, 140493155003168)
>>> b[0] = 42
>>> a
array([42,  1,  2,  3,  4,  5,  6,  7,  8,  9])
```

Now a new object is generated, but it refers to the same piece of memory. A modification of elements in `b` will still be visible in `a`. In order to really obtain a copy of an array, one applies the `copy` function:

```
>>> a = np.arange(10)
>>> b = np.copy(a)
>>> b[0] = 42
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> b
array([42,  1,  2,  3,  4,  5,  6,  7,  8,  9])
```

It is rather straightforward to extend the concept of slicing to higher dimensions and we again go through a number of examples to illustrate the idea. Note that in no case a new array is created in memory so that slicing is an efficient way of extracting a certain subset of array elements. Our base array is:

```
>>> a = np.arange(36).reshape(6, 6)
>>> a
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23],
       [24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35]])
```

In view of the two dimensions, we now need two slices separated by a comma, the first one for the rows and the second one for the columns. The full array is thus recovered by:

```
>>> a[:, :]
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23],
       [24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35]])
```

A sub-block can be extracted as follows:

```
>>> a[2:4, 3:6]
array([[15, 16, 17],
       [21, 22, 23]])
```

As already mentioned, the first slice pertains to the rows, so that we choose elements from the third and fourth row. The second slice refers to columns four to six so that we indeed end up with the output reproduced above.

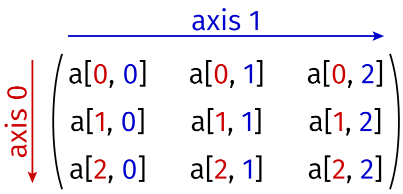


Figure 4.5: In a two-dimensional array, the first index corresponding to axes 0 denotes the row while the second index corresponding to axes 1 denotes the column. This convention is consistent with the one used in mathematics.

Sub-blocks do not need to be contiguous. We can even choose different values for `step` in different dimensions:

```
>>> a[::2, ::3]
array([[ 0,  3],
       [12, 15],
       [24, 27]])
```

In this example, we have selected every second row and every third column. If we want to start with the third row, we could write:

```
>>> a[2::2, ::3]
array([[12, 15],
       [24, 27]])
```

The following example illustrates a case where only one slice is specified:

```
>>> a[2:4]
array([[12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23]])
```

The first slice still applies to the row and the missing second slice is replaced by default by `::` representing all columns.

The interpretation of the last example requires to make connection between the axis number and its meaning in terms of the array elements. In a two-dimensional array, the position of the indices follows the convention used in mathematics as shown in Figure 4.5. This correctness of this interpretation can also be verified by means of operations which can act along a single axis as is the case for `sum`:

```
>>> a.sum(axis=0)
array([ 90,  96, 102, 108, 114, 120])
>>> a.sum(axis=1)
array([ 15,  51,  87, 123, 159, 195])
>>> a.sum()
630
```

In the first case, the summation is performed along the columns while in the second case the elements in a given row are added. If no axis is specified, all array elements are summed.

We illustrate the generalization to higher dimensions by considering a three-dimensional array:

```
>>> b = np.arange(24).reshape(2, 3, 4)
>>> b
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],

       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]]])
```

Interpreting the array in terms of nested lists, the outer level contains two two-dimensional arrays along axis 0 as

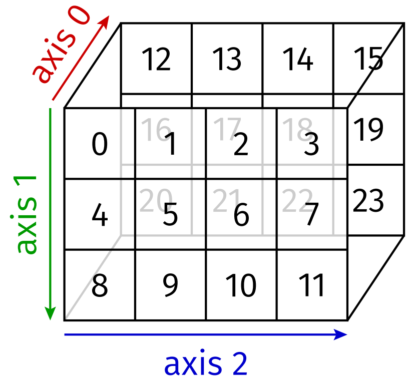


Figure 4.6: A three-dimensional array with its three axes.

displayed in Figure 4.6. Within the two-dimensional arrays, the outer level corresponds to axis 1 and the innermost level corresponds to axis 2.

Cutting along the three axes, we obtain the following two-dimensional arrays:

```
>>> b[0]
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> b[:, 0]
array([[ 0,  1,  2,  3],
       [12, 13, 14, 15]])
>>> b[:, :, 0]
array([[ 0,  4,  8],
       [12, 16, 20]])
```

These three arrays correspond to the front plane along axis 0, the upper plane along axis 1 and the left-most plane along axis 2, respectively. In the last example, an appropriate number of colons can simply be replaced by an ellipsis:

```
>>> b[..., 0]
array([[ 0,  4,  8],
       [12, 16, 20]])
```

In order to make the meaning of this notation unique, only one ellipsis is permitted, but it may appear even between indices like in the following example:

```
>>> c = np.arange(16).reshape(2, 2, 2, 2)
>>> c
array([[[[ 0,  1],
         [ 2,  3]],

        [[ 4,  5],
         [ 6,  7]]],


       [[[ 8,  9],
         [10, 11]],

        [[12, 13],
         [14, 15]]]])
>>> c[0, ..., 0]
array([[0, 2],
       [4, 6]])
```

When selecting a column in a two-dimensional array, one in principle has two ways to do so. However, they are leading to different results:

```
>>> a[:, 0:1]
array([[ 0],
       [ 6],
       [12],
       [18],
       [24],
       [30]])
>>> a[:, 0]
array([ 0,  6, 12, 18, 24, 30])
```

In the first case, a two-dimensional array is produced where the second dimension happens to be of length 1. In the second case, the first column is explicitly selected and one ends up with a one-dimensional array. This example may lead to the question whether there is a way to convert a one-dimensional array into a two-dimensional array containing one column or one row. Such a conversion may be necessary in the context of broadcasting which we will discuss in Section 4.2.5. The following example demonstrates how the dimension of an array can be increased by means of a `newaxis`:

```
>>> d = np.arange(4)
>>> d
array([0, 1, 2, 3])
>>> d[:, np.newaxis]
array([[0],
       [1],
       [2],
       [3]])
>>> d[:, np.newaxis].shape
(4, 1)
>>> d[np.newaxis, :]
array([[0, 1, 2, 3]])
>>> d[np.newaxis, :].shape
(1, 4)
```

So far, we have selected subsets of array elements by means of slicing. Another option is the so-called fancy indexing where elements are specified by lists or arrays of integers or Booleans for each dimension of the array. Let us consider a few examples:

```
>>> a
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23],
       [24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35]])
>>> a[[0, 2, 1], [1, 3, 5]]
array([ 1, 15, 11])
```

The lists for axes 0 and 1 combine to yield the index pairs of the elements to be selected. In our example, these are `a[0, 1]`, `a[2, 3]`, and `a[1, 5]`. As this example shows, the indices in one list do not need to increase. They rather have be to chosen as a function of the elements which shall be selected. In this example, there is no way how the two-dimensional form of the original array can be maintained and we simply obtain a one-dimensional array containing the three elements selected by the two lists.

A two-dimensional array can be obtained from an originally two-dimensional array if entire rows or columns are selected like in the following example:

```
>>> a[:, [0, 3, 5]]
array([[ 0,  3,  5],
       [ 6,  9, 11],
       [12, 15, 17],
       [18, 21, 23],
       [24, 27, 29],
       [30, 33, 35]])
```

Here, we have only specified a list for axis 1 and chosen entire columns. Note that the chosen columns are not equidistant and thus cannot be obtained by slicing.

Our last example uses fancy indexing with a boolean array. We create an array of random numbers and want to set all entries smaller than 0.5 to zero. After creating an array of random numbers from which we construct a Boolean area by comparing with 0.5. The resulting array is then used not to extract array elements but to set selected array elements to zero:

```
>>> randomarray = np.random.rand(10)
>>> randomarray
array([0.48644931, 0.13579493, 0.91986082, 0.38554513, 0.38398479,
       0.61285717, 0.60428045, 0.01715505, 0.44574082, 0.85642709])
>>> indexarray = randomarray < 0.5
>>> indexarray
array([ True,  True, False,  True,  True, False, False,  True,  True,
       False])
>>> randomarray[indexarray] = 0
>>> randomarray
array([0.        , 0.        , 0.91986082, 0.        , 0.        ,
       0.61285717, 0.60428045, 0.        , 0.        , 0.85642709])
```

If instead of setting values below 0.5 to zero, we would have wanted to set them to 0.5, we could have avoided fancy indexing by using `np.clip`.

As an application of slicing and fancy indexing, we consider a NumPy implementation of the sieve of Eratosthenes to determine prime numbers. The principle is illustrated in Figure 4.7 where the steps required to determine all prime numbers below 50 are depicted. We start out with a list of integers up to 49. It is convenient to include 0 to be consistent with the fact that indexing starts at 0 in NumPy. A corresponding array `is_prime` is initialized with the Boolean value `True`. In each iteration numbers found not be prime have their value set to `False`. Initially, we mark 0 and 1 as non-primes.

Now we iterate through the array and consider successively each prime number which we can find. The first one will be 2. Clearly, all multiples of 2 are not prime and we can cross them out. The next prime is 3, but now we can start crossing out multiples of 3 at 9. In general, for a prime number p, we start crossing out multiples of p at p^2 because all smaller multiples of p have been crossed out before. The maximum number to be considered as candidate is the largest integer smaller or equal to the maximum integer to be considered. In our example, we consider integers up to 49 and thus the largest candidate is 7 which happens to be prime.

This algorithm can be implemented in the following way where we have chosen to print not only the final result but also the intermediate steps.

```
import math
import numpy as np

nmax = 49
integers = np.arange(nmax+1)
is_prime = np.ones(nmax+1, dtype=bool)
is_prime[:2] = False
for j in range(2, math.isqrt(nmax)+1):
    if is_prime[j]:
        is_prime[j*j::j] = False
    print(integers[is_prime])
```

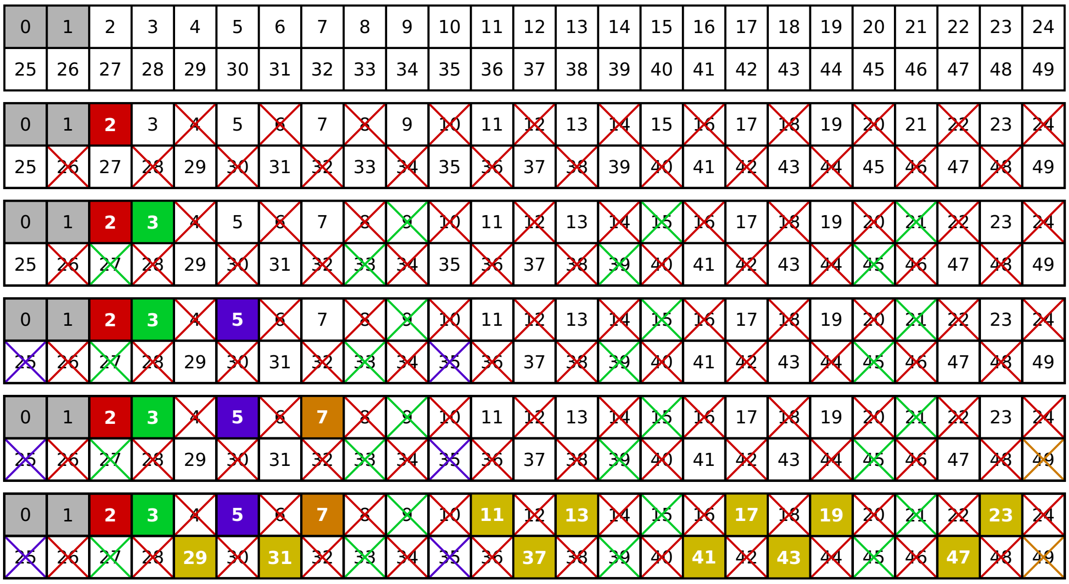


Figure 4.7: Iteration steps when the sieve of Eratosthenes is used to determine the prime numbers below 50. For details see the main text.

This script produces the following output:

```
[ 2  3  5  7  9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49]
[ 2  3  5  7 11 13 17 19 23 25 29 31 35 37 41 43 47 49]
[ 2  3  5  7 11 13 17 19 23 25 29 31 35 37 41 43 47 49]
[ 2  3  5  7 11 13 17 19 23 29 31 37 41 43 47 49]
[ 2  3  5  7 11 13 17 19 23 29 31 37 41 43 47 49]
[ 2  3  5  7 11 13 17 19 23 29 31 37 41 43 47]
```

In line 6 of the Python code, we use `np.ones` with type `bool` to mark all entries as potential primes. In line 10, slicing is used to mark all multiples of `j` starting at the square of `j` as non-primes. Finally, in line 11, fancy indexing is used. The Boolean array `is_prime` indicates through the value `True` which entries in the array `integers` should be printed.

4.2.5 Broadcasting

In the previous sections, we have seen examples where an operation involved a scalar value and an array. This was the case in Section 4.2.3 where we multiplied an array created by `np.ones` with a number. Another example appeared in Section 4.2.4 where in our discussion of fancy indexing we compared an array with a single number. Even though NumPy behaved in a perfectly natural way, these examples are special cases of a more general concept, the so-called broadcasting.

An array can be broadcast to a larger array provided the shapes satisfy certain conditions. In order to obtain the same dimension as the one of the target array, dimensions of size 1 are prepended. Then, each component of the shape of the original array has to be equal to the corresponding component of the shape of the target array or the component has to equal 1. In Figure 4.8, the target array has shape `(3, 4)`. The arrays with shapes `(1,)`, `(4, )`, and `(3, 1)` satisfy this conditions and can be broadcast as shown in the figure. In contrast, this is not possible for an array of shape `(3,)` as is demonstrated in the figure. We emphasize the difference between the arrays of shape `(3,)` and `(3, 1)`.

As the second image in Figure 4.8 shows, a scalar is broadcast to an array of the desired shape with all elements being equal. Multiplying an array with a scalar, we expect that each array element is multiplied by the scalar. As a consequence, the multiplication of two arrays is carried out element by element. In other words, a matrix multiplication cannot be done by means of the `*` operator:

```
>>> a = np.arange(4).reshape(2, 2)
```

(continues on next page)

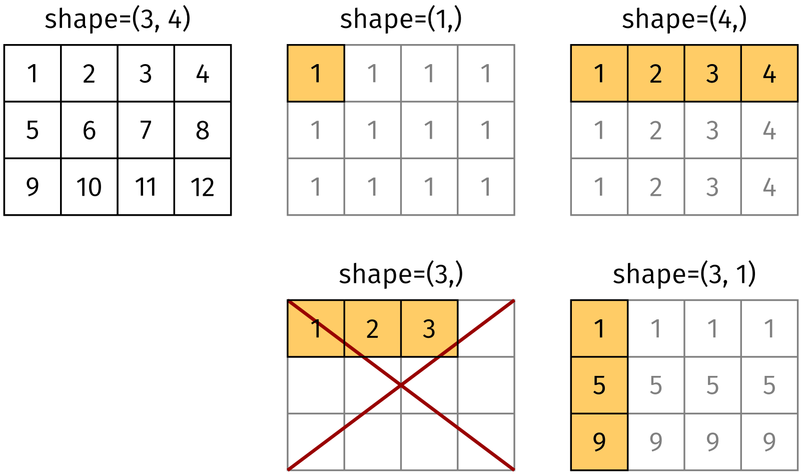


Figure 4.8: For appropriate shapes, the matrix elements in the highlighted cells can be broadcast to create the full shape `(3, 4)` in this example. An array of shape `(3,)` cannot be broadcast to shape `(3, 4)`

(continued from previous page)

```
>>> a
array([[0, 1],
       [2, 3]])
>>> b = np.arange(4, 8).reshape(2, 2)
>>> b
array([[4, 5],
       [6, 7]])
>>> a*b
array([[ 0,  5],
       [12, 21]])
```

The matrix multiplication can be achieved in a number of different ways:

```
>>> np.dot(a, b)
array([[ 6,  7],
       [26, 31]])
>>> a.dot(b)
array([[ 6,  7],
       [26, 31]])
>>> a @ b
array([[ 6,  7],
       [26, 31]])
```

The use of the `@` operator for the matrix multiplication requires at least Python 3.5 and NumPy 1.10.

4.2.6 Universal functions

The mathematical functions provided by the `math` and `cmath` modules from the Python standard library accept only single real or complex values but no arrays. For the latter purpose, NumPy and also the scientific library SciPy offer so-called universal functions:

```
>>> import math
>>> x = np.linspace(0, 2, 11)
>>> x
array([0. , 0.2, 0.4, 0.6, 0.8, 1. , 1.2, 1.4, 1.6, 1.8, 2. ])
>>> math.sin(x)
Traceback (most recent call last):
```

(continues on next page)

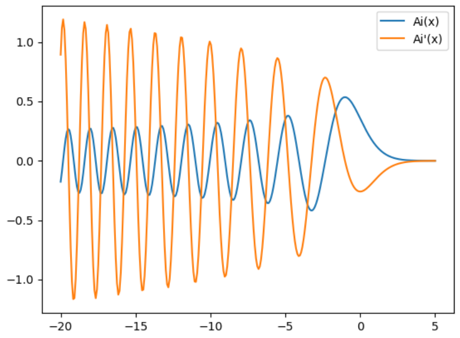


Figure 4.9: Airy function $\mathrm{Ai}(x)$ and its derivative.

(continued from previous page)

```
  File "<stdin>", line 1, in <module>
TypeError: only size-1 arrays can be converted to Python scalars
>>> np.sin(x)
array([0.        , 0.19866933, 0.38941834, 0.56464247, 0.71735609,
       0.84147098, 0.93203909, 0.98544973, 0.9995736 , 0.97384763,
       0.90929743])
```

Universal functions can handle multi-dimensional arrays as well:

```
>>> x = np.array([[0, np.pi/2], [np.pi, 3/2*np.pi]])
>>> x
array([[0.        , 1.57079633],
       [3.14159265, 4.71238898]])
>>> np.sin(x)
array([[ 0.0000000e+00,  1.0000000e+00],
       [ 1.2246468e-16, -1.0000000e+00]])
```

This example shows that the mathematical constant π is not only available from the math and cmath modules but also from the NumPy package. Many of the functions provided by the math module are available as universal functions in NumPy and NumPy offers a few universal functions not available as normal functions neither in math nor in cmath. Details on the functions provided by NumPy are given in the section on Mathematical functions in the NumPy reference guide.

While the universal functions in NumPy are mostly restricted to the common mathematical functions, special functions are available through the SciPy package. Often but not always, these functions are implemented as universal functions as well. As an example, we create a plot of the Airy function $\mathrm{Ai}(x)$ appearing e.g. in the theory of the rainbow or the quantum mechanics in a linear potential:

```
>>> from scipy.special import airy
>>> x = np.linspace(-20, 5, 300)
>>> ai, aip, bi, bip = airy(x)
>>> plt.plot(x, ai, label="Ai(x)")
[<matplotlib.lines.Line2D object at 0x7f62184e2278>]
>>> plt.plot(x, aip, label="Ai'(x)")
[<matplotlib.lines.Line2D object at 0x7f621bbb4518>]
>>> plt.legend()
>>> plt.show()
```

There exist two types of Airy functions $\mathrm{Ai}(x)$ and $\mathrm{Bi}(x)$ which together with their derivatives are calculated by the airy function in one go. The Airy function $\mathrm{Ai}(x)$ and its derivative are displayed in Figure 4.9.

Occasionally, one needs to create a two-dimensional plot of a function of two variables. In order to ensure that the

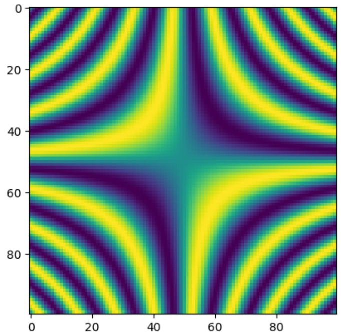


Figure 4.10: Application of the mesh grid in a two-dimensional representation of the function $\sin(xy)$.

resulting array is two-dimensional, the one-dimensional arrays for the two variables need to run along different axes. A convenient way to do so is the mesh grid. The function `mgrid` creates two two-dimensional arrays where in one array the values change along the column while in the other array they change along the rows:

```
>>> np.mgrid[0:2:0.5, 0:1:0.5]
array([[[0. , 0. ],
        [0.5, 0.5],
        [1. , 1. ],
        [1.5, 1.5]],

       [[0. , 0.5],
        [0. , 0.5],
        [0. , 0.5],
        [0. , 0.5]]])
```

The slicing syntax corresponds to what we are used from the `arange` function. The equivalence of the `linspace` function can be obtained by making the third argument imaginary:

```
>>> np.mgrid[0:2:3j, 0:2:5j]
array([[[0. , 0. , 0. , 0. , 0. ],
        [1. , 1. , 1. , 1. , 1. ],
        [2. , 2. , 2. , 2. , 2. ]],

       [[0. , 0.5, 1. , 1.5, 2. ],
        [0. , 0.5, 1. , 1.5, 2. ],
        [0. , 0.5, 1. , 1.5, 2. ]]])
```

A practical application produced by the following code is shown in Figure 4.10:

```
>>> x, y = np.mgrid[-5:5:100j, -5:5:100j]
>>> plt.imshow(np.sin(x*y))
<matplotlib.image.AxesImage object at 0x7fde9176ea90>
>>> plt.show()
```

Making use of broadcasting, one can reduce the memory requirement by creating an open mesh grid instead:

```
>>> np.ogrid[0:2:3j, 0:1:5j]
[array([[0.],
       [1.],
       [2.]]), array([[0.  , 0.25, 0.5 , 0.75, 1.  ]])]
```

The function `ogrid` returns two two-dimensional arrays where one dimension is of length 1. Figure 4.11 shows an application to Bessel functions obtained by means of the following code:

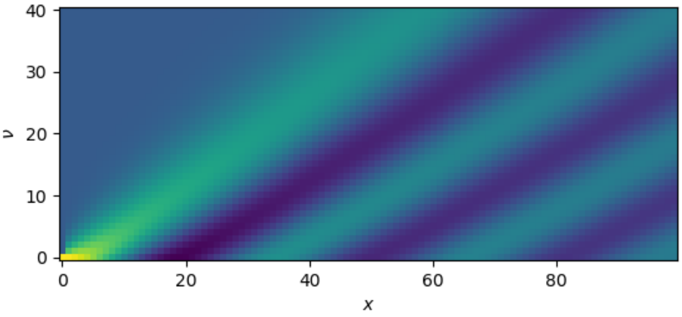


Figure 4.11: Two-dimensional plot of the family of Bessel functions $J_\nu(x)$ of order ν created by means of an open mesh grid created by `ogrid`.

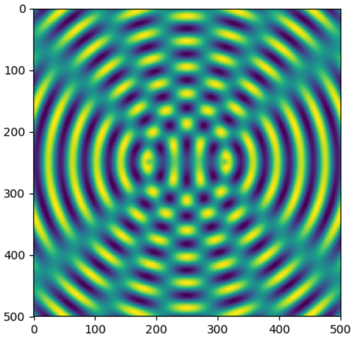


Figure 4.12: Interference pattern created with argument arrays obtained by means of `linspace` and by adding an additional axis in one of the two arrays. The function `hypot` determines the distance of the point given by the two argument coordinates from the origin.

```
>>> from scipy.special import jv
>>> nu, x = np.ogrid[0:10:41j, 0:20:100j]
>>> plt.imshow(jv(nu, x), origin='lower')
<matplotlib.image.AxesImage object at 0x7fde903736d8>
>>> plt.xlabel('$x$')
Text(0.5,0,'$x$')
>>> plt.ylabel(r'$\nu$')
Text(0,0.5,'$\\nu$')
>>> plt.show()
```

Instead of using the `ogrid` function, one can also construct the argument arrays by hand. In this case, one has to take care of adding an additional axis in one of the two arrays as in the following example which results in Figure 4.12:

```
>>> x = np.linspace(-40, 40, 500)
>>> y = x[:, np.newaxis]
>>> z = np.sin(np.hypot(x-10, y))+np.sin(np.hypot(x+10, y))
>>> plt.imshow(z)
<matplotlib.image.AxesImage object at 0x7fde92509278>
>>> plt.show()
```

So far, we have considered universal functions mostly as a convenient way to apply a function to an entire array. However, they can also make a significant contribution to speed up code. The following script compares the runtime between a for loop evaluating the sine function taken from the `math` module and a direct evaluation of the sine taken from NumPy for different array sizes:

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

def sin_math(nmax):
    xvals = np.linspace(0, 2*np.pi, nmax)
    start = time.time()
    for x in xvals:
        y = math.sin(x)
    return time.time()-start

def sin_numpy(nmax):
    xvals = np.linspace(0, 2*np.pi, nmax)
    start = time.time()
    yvals = np.sin(xvals)
    return time.time()-start

maxpower = 26
nvals = 2**np.arange(0, maxpower+1)
tvals = np.empty_like(nvals)
for nr, nmax in enumerate(nvals):
    tvals[nr] = sin_math(nmax)/sin_numpy(nmax)
plt.rc('text', usetex=True)
plt.xscale('log')
plt.yscale('log')
plt.xlabel('$n_\mathrm{max}$', fontsize=20)
plt.ylabel('$t_\mathrm{math}/t_\mathrm{numpy}$', fontsize=20)
plt.plot(nvals, tvals, 'o')
plt.show()
```

The results are presented in Figure 4.13 and depend on various factors including the hardware and details of the software environment. The data should therefore give a rough indication of the speedup and should not be taken too literally. The first point to note is that even for an array of size 1, NumPy is faster than the sine function taken from the `math` module. This seems to contradict our previous result on a scalar argument, but can be explained by the presence of the for loop in the `sin_math` function which results in an overhead even if the for loop is strictly speaking unnecessary. Then, for arrays of an intermediate size, a speed up of roughly a factor of 7 is observed. Interestingly, for array sizes beyond a few times 10^4, the speed up reaches values of around 100. This behavior can be explained by the use of the Anaconda distribution where NumPy is compiled to support Intel's math kernel library (MKL). Even without this effect, a speed up between 5 and 10 may be significant enough to seriously consider the use of universal functions.

4.2.7 Linear algebra

Scientific problems which can be formulated in terms of vectors or matrices often require tools of linear algebra. Therefore, we will discuss a few of the more important functions NumPy has to offer in that domain. For more details we recommend to take a look at the documentation of the numpy.linalg module.

As we have discussed earlier, the usual multiplication operator does element-wise multiplication and uses broadcasting where applicable. The multiplication of arrays can either be done by means of the `dot` method or the `@` operator:

```
>>> v1 = np.array([1, 2])
>>> v2 = np.array([3, 4])
>>> np.dot(v1, v2)
11
>>> m = np.array([[5, 6], [7, 8]])
>>> np.dot(m, v1)
array([17, 23])
```

(continues on next page)

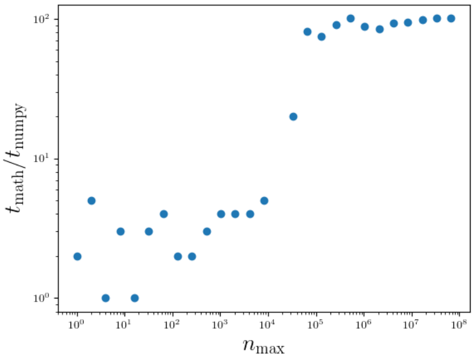


Figure 4.13: Runtime comparison between the sine function taken from the `math` module and from the NumPy package as a function of the array size. Larger values of the time ratio imply a larger speed up gained by means of NumPy. The data have been obtained by a version of NumPy with MKL support.

(continued from previous page)

```
>>> m @ v1
array([17, 23])
```

For the following, we will need to load the `numpy.linalg` module first:

```
>>> import numpy.linalg as LA
```

Here we have once more introduced a commonly used abbreviation. A vector can easily be normalized by means of the `norm` function:

```
>>> v = np.array([1, -2, 3])
>>> n = LA.norm(v)
>>> n**2
14.0
>>> v_normalized = v/n
>>> LA.norm(v_normalized)
1.0
```

Applying the `norm` function to a multi-dimensional array will return the Frobenius or Hilbert-Schmid norm, i.e. the square root of the sum over the squares of all matrix elements.

Some of the operations provided by the `numpy.linalg` module can be applied to a whole set of arrays. An example is the determinant which mathematically is defined only for two-dimensional arrays. For a three-dimensional array, determinants are calculated for each value of the index of axis 0:

```
>>> m = np.arange(12).reshape(3, 2, 2)
>>> m
array([[[ 0,  1],
        [ 2,  3]],

       [[ 4,  5],
        [ 6,  7]],

       [[ 8,  9],
        [10, 11]]])
>>> LA.det(m)
array([-2., -2., -2.])
```

It is also possible to determine the inverse for several matrices at the same time. Trying to invert a non-invertible matrix will result in a `numpy.linalg.linalg.LinAlgError` exception.

An inhomogeneous system of linear equations `ax=b` with a matrix `a` and a vector `b` can in principle be solved by inverting the matrix:

```
>>> a = np.array([[2, -1], [-3, 2]])
>>> b = np.array([1, 2])
>>> x = np.dot(LA.inv(a), b)
>>> x
array([4., 7.])
>>> np.dot(a, x)
array([1., 2.])
```

In the last command, we verified that the solution obtained one line above is indeed correct. Solving inhomogeneous systems of linear equations by inversion is however not very efficient and NumPy offers an alternative way based on an LU decomposition:

```
>>> LA.solve(a, b)
array([4., 7.])
```

As we have seen above for the determinant, the function `solve` also allows to solve several inhomogeneous systems of linear equations in one function call.

A frequent task in scientific applications is to solve an eigenvalue problem. The function `eig` determines the right eigenvectors and the associated eigenvalues for arbitrary square matrices:

```
>>> a = np.array([[1, 3], [4, -1]])
>>> evals, evecs = LA.eig(a)
>>> evals
array([ 3.60555128, -3.60555128])
>>> evecs
array([[ 0.75499722, -0.54580557],
       [ 0.65572799,  0.83791185]])
>>> for n in range(evecs.shape[0]):
...     print(np.dot(a, evecs[:, n]), evals[n]*evecs[:, n])
...
[2.72218119 2.36426089] [2.72218119 2.36426089]
[ 1.96792999 -3.02113415] [ 1.96792999 -3.02113415]
```

In the for loop, we compare the product of matrix and eigenvector with the corresponding product of eigenvalue and eigenvector and can verify that the results are indeed correct. In the matrix of eigenvectors, the eigenvectors are given by the columns.

Occasionally, it is sufficient to know the eigenvalues. In order to reduce the compute time, one can then replace `eig` by `eigvals`:

```
>>> LA.eigvals(a)
array([ 3.60555128, -3.60555128])
```

In many applications, the matrices appearing in eigenvalue problems are either symmetric of Hermitian. For these cases, NumPy provides the functions `eigh` and `eigvalsh`. One advantage is that it suffices to store only half of the matrix elements. More importantly, these specialized functions are much faster:

```
>>> import timeit
>>> a = np.random.random(250000).reshape(500, 500)
>>> a = a+a.T
>>> timeit.repeat('LA.eig(a)', number=100, globals=globals())
[13.307299479999529, 13.404196323999713, 13.798628489999828]
>>> timeit.repeat('LA.eigh(a)', number=100, globals=globals())
[1.8066274120001253, 1.7375857540000652, 1.739574907000133]
```

In the third line, we have made sure that the initial random matrix is turned into a symmetric matrix by adding its transpose. In this example, we observe a speedup of about a factor of 7.

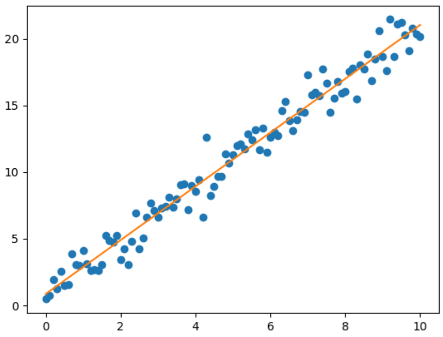


Figure 4.14: Noisy data (blue points) and result of the linear regression (orange line) obtained by means of `scipy.stats.linregress`.

4.3 SciPy

The functions offered through the SciPy package cover many tasks typically encountered in the numerical treatment of scientific problems. Here, we can only give an impression of the potential of SciPy by discussing a few examples. It is highly recommended to take a look at the SciPy API Reference.

As a first example, we consider the linear regression of noisy data. In a first step, we create data on a line with normally distributed noise added on top:

```
>>> x = np.linspace(0, 10, 101)
>>> y = 2*x + 1 + np.random.normal(0, 1, 101)
```

Now, we can use the `linregress` function from the statistical functions module of SciPy to do a least-squares regression of the noisy data:

```
>>> from scipy.stats import linregress
>>> slope, intercept, rvalue, pvalue, stderr = linregress(x, y)
>>> plt.plot(x, y, 'o')
>>> plt.plot(x, slope*x + intercept)
>>> plt.show()
>>> print(rvalue, stderr)
0.9853966954685487 0.0350427823008272
```

Here `rvalue` refers to the correlation coefficient and `stderr` is the standard error of the estimated gradient. The graph containing the noisy data and the linear fit is shown in Figure 4.14.

Fitting of data cannot always be reduced to linear regression. Then we can resort to the `curve_fit` function from the optimization module of SciPy:

```
>>> from scipy.optimize import curve_fit
>>> def fitfunc(x, a, b):
...     return a*np.sin(x+b)
...
>>> x = np.linspace(0, 10, 101)
>>> y = 2*np.sin(x+0.5) + np.random.normal(0, 1, 101)
>>> plt.plot(x, y, 'o')
>>> popt, pcov = curve_fit(fitfunc, x, y)
>>> popt
array([2.08496412, 0.43937489])
>>> plt.plot(x, popt[0]*np.sin(x+popt[1]))
>>> plt.show()
```

In order to fit to a general function, one needs to provide `curve_fit` with a function, called `fitfunc` here, which

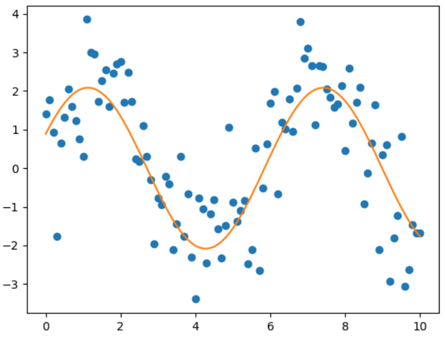


Figure 4.15: Fit of a noisy sine function by means of `scipy.optimize.curve_fit`.

depends on the variable as well as a set of parameters. In our example, we have chosen two parameters `a` and `b` but we are in principle not limited to this number. However, as the number of parameters increases, the fit tends to become less reliable. The fit values for the parameters are returned in the array `popt` together with the covariance matrix for the parameters `pcov`. The outcome of the fit is shown in Figure 4.15.

Occasionally, a root search is required. As an example, we consider the determination of the ground state energy in a finite potential well. The eigenvalue condition for a symmetric eigenstate reads

$$\sqrt{\epsilon}\cos(\alpha\sqrt{1-\epsilon}) - \sqrt{1-\epsilon}\sin(\alpha\sqrt{1-\epsilon}) = 0\,,$$

where ϵ is the energy in units of the well depth and α is a measure of the potential strength combining the well depth and its width. One way of solving this nonlinear equation for ϵ is by means of the `brentq` function, which needs at least the function of which the root should be determined as well as the bounds of an interval in which the function changes its sign. If the potential well is sufficiently shallow, i.e. if :math.``alpha`` is sufficiently small, the left-hand side contains only one root as can be seen from the blue line in Figure 4.16. In our example, the function requires an additional argument α which also needs to be given to `brentq`. Finally, in order to know how many iterations are need, we set `full_output` to `True`.

```
>>> from scipy.optimize import brentq
>>> def f(energy, alpha):
...     sqrt_1me = np.sqrt(1-energy)
...     return (np.sqrt(energy)*np.cos(alpha*sqrt_1me)
...             -sqrt_1me*np.sin(alpha*sqrt_1me))
...
>>> alpha = 1
>>> x0, r = brentq(f, a=0, b=1, args=alpha, full_output=True)
>>> x0
>>> 0.45375316586032827
>>> r
      converged: True
           flag: 'converged'
 function_calls: 7
     iterations: 6
           root: 0.45375316586032827)
>>> x = np.linspace(0, 1, 400)
>>> plt.plot(x, f(x, alpha))
>>> plt.plot(x0, 0, 'o')
>>> plt.show()
```

As the output indicates, the root is found within 6 iterations. The resulting root is depicted in Figure 4.16 as an orange dot.

Finally, we consider a more complex example which involves optimization and the solution of a coupled set of differential equations. The physical problem to be studied numerically is the fall of a chain where the equations of

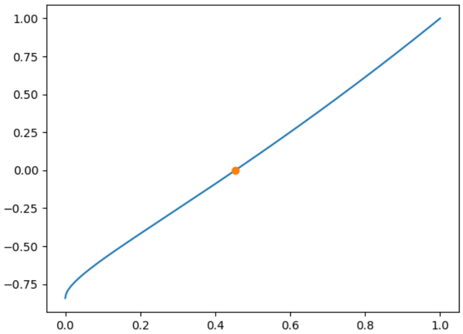


Figure 4.16: Determination of the ground state energy in a finite potential well of depth $\alpha = 1$ by means of `scipy.optimize.brentq`.

motion are derived in W. Tomaszweski, P. Pieranski, and J.-C. Geminard, Am. J. Phys. **74**, 776 (2006)[9] and account for damping inside the chain. For the initial configuration of the chain, we take a chain hanging in equilibrium where the two ends are at equal height and at a given horizontal distance. In the continuum limit, the chain follows a catenary, but in the discrete case treated here, we obtain the equilibrium configuration by optimizing the potential energy. This is done in the method `equilibrium` of our `Chain` class. The method essentially consists of a call to the `minimize` function taken from the `optimize` module of SciPy. It optimizes the potential energy defined in the `f_energy` method. In addition, we have to account for two constraints corresponding to the horizontal and vertical direction and implemented through the methods `x_constraint` and `y_constraint`. An example of the result of this optimization procedure is depicted in Figure 4.17 for a chain made of 9 links.

```
import numpy as np
import numpy.linalg as LA
import matplotlib.pyplot as plt
from scipy.optimize import minimize
from scipy.integrate import solve_ivp

class Chain:
    def __init__(self, nlinks, length, damping):
        if nlinks < length:
            raise ValueError('length requirement cannot be fulfilled with '
                             'the given number of links')
        self.nlinks = nlinks
        self.length = length
        self.m = self.matrix_m()
        self.a = self.vector_a()
        self.damping = (-2*np.identity(self.nlinks, dtype=np.float64)
                        +np.eye(self.nlinks, k=1)
                        +np.eye(self.nlinks, k=-1))
        self.damping[0, 0] = -1
        self.damping[self.nlinks-1, self.nlinks-1] = -1
        self.damping = damping*self.damping

    def x_constraint(self, phi):
        return np.sum(np.cos(phi))-self.length

    def y_constraint(self, phi):
        return np.sum(np.sin(phi))

    def f_energy(self, phi):
```

(continues on next page)

[9] doi:10.1119/1.2204074.

(continued from previous page)

```
        return np.sum(np.arange(self.nlinks, 0, -1)*np.sin(phi))

    def equilibrium(self):
        result = minimize(self.f_energy, np.linspace(-0.1, 0.1, self.nlinks),
                          method='SLSQP',
                          constraints=[{'type': 'eq', 'fun': self.x_constraint},
                                       {'type': 'eq', 'fun': self.y_constraint}])
        return result.x

    def plot_equilibrium(self):
        phis = chain.equilibrium()
        x = np.zeros(chain.nlinks+1)
        x[1:] = np.cumsum(np.cos(phis))
        y = np.zeros(chain.nlinks+1)
        y[1:] = np.cumsum(np.sin(phis))
        plt.plot(x, y)
        plt.plot(x, y, 'o')
        plt.axes().set_aspect('equal')
        plt.show()

    def matrix_m(self):
        m = np.fromfunction(lambda i, j: self.nlinks-np.maximum(i, j)-0.5,
                            (self.nlinks, self.nlinks), dtype=np.float64)
        m = m-np.identity(self.nlinks)/6
        return m

    def vector_a(self):
        a = np.arange(self.nlinks, 0, -1)-0.5
        return a

    def diff(self, t, y):
        momenta = y[:self.nlinks]
        angles = y[self.nlinks:]
        d_angles = momenta
        ci = np.cos(angles)
        cij = np.cos(angles[:,np.newaxis]-angles)
        sij = np.sin(angles[:,np.newaxis]-angles)
        mcinv = LA.inv(self.m*cij)
        d_momenta = -np.dot(self.m*sij, momenta*momenta)
        d_momenta = d_momenta+np.dot(self.damping, momenta)
        d_momenta = d_momenta-self.a*ci
        d_momenta = np.dot(mcinv, d_momenta)
        d = np.empty_like(y)
        d[:self.nlinks] = d_momenta
        d[self.nlinks:] = d_angles
        return d

    def solve_eq_of_motion(self, time_i, time_f, nt):
        y0 = np.zeros(2*self.nlinks, dtype=np.float64)
        y0[self.nlinks:] = self.equilibrium()
        self.solution = solve_ivp(self.diff, (time_i, time_f), y0, method='Radau',
                                  t_eval=np.linspace(time_i, time_f, nt))

    def plot_dynamics(self):
        for i in range(self.solution.y.shape[1]):
            phis = self.solution.y[:, i][self.nlinks:]
```

(continues on next page)

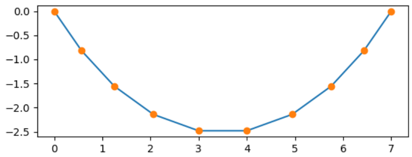


Figure 4.17: Chain consisting of 9 links hanging in its equilibrium position with a horizontal distance of the ends equivalent to the length of 7 links.

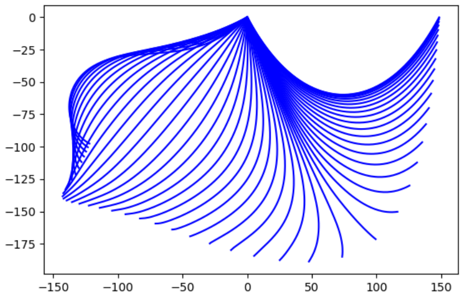


Figure 4.18: Stroboscopic image of a falling chain consisting of 200 elements starting out from its equilibrium state in the upper right during its first half period swinging to the left.

(continued from previous page)

```
            x = np.zeros(self.nlinks+1)
            x[1:] = np.cumsum(np.cos(phis))
            y = np.zeros(self.nlinks+1)
            y[1:] = np.cumsum(np.sin(phis))
            plt.plot(x, y, 'b')
        plt.axes().set_aspect('equal')
        plt.show()

chain = Chain(200, 150, 0.003)
chain.solve_eq_of_motion(0, 40, 50)
chain.plot_dynamics()
```

In a second step, the equation of motion for the chain links is solved in the `solve_eq_of_motion` method by means of `solve_ivp` taken from the `integrate` module of SciPy. We need to express the second order equations of motion in terms of first-order differential equations which can always be achieved by doubling the number of degrees of freedom by means of auxiliary variables. The main ingredient then is the function called `diff` in our example which for a given set of variables returns the time derivatives for these variables. Furthermore, `solve_ivp` needs to know the time interval on which the solution is to be determined together with the time values for which a solution is requested as well as the initial configuration. Finally, out of the various solvers, we choose `Radau` which implements an implicit Runge-Kutta method of Radau IIA family of order 5. Figure 4.18 displays a stroboscopic plot of the chain during is first half period swinging from the right to the left.

CHAPTER

FIVE

RUN-TIME ANALYSIS

5.1 General remarks

Frequently, the numerical treatment of scientific problems can lead to time-consuming code and the question arises how its performance can be improved. There exist a variety of different kinds of approaches. One could for example choose to make use of more powerful hardware or distribute the task on numerous compute nodes of a compute cluster. Even though today, human resources tend to be more costly than hardware resource, the latter should not be wasted by very inefficient code.

In certain cases, speed can be improved by making use of graphical processors (GPU) which are able to handle larger data sets in parallel. While writing code for GPUs may be cumbersome, there exists for example the CuPy library[1] which can serve as a drop-in replacement for most of the NumPy library and will run on NVIDIA GPUs.

On the software side, an option is to make use of optimized code provided by numerical libraries like NumPy and SciPy discussed in Section 4. Sometimes, it can make sense to implement particularly time-consuming parts in the programming language C for which highly optimizing compilers are available. This approach does not necessarily require to write proper C code. One can make use of Cython[2] instead, which will autogenerate C code from Python-like code.

Another option is the use of *just in time* (JIT) compilers as is done in PyPy[3] and Numba[4]. The latter is readily available through the Anaconda distribution and offers an easy approach to speeding up time-critical parts of the code by simply adding a decorator to a function. Generally, JIT compilers analyze the code before its first execution and create machine code allowing to run the code faster in subsequent calls.

While some of the methods just mentioned can easily be implemented, others may require a significant investment of time. One therefore needs to assess whether the gain in compute time really exceeds the cost in developer time. It is worth following the advice of the eminent computer scientist Donald E. Knuth[5] who wrote already 45 years ago[6]

> There is no doubt that the grail of efficiency leads to abuse. Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We *should* forget about small efficiencies, say about 97 % of the time: premature optimization is the root of all evil.
>
> Yet we should not pass up our opportunities in that critical 3 %. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified.

Before beginning to optimize code, it is important to identify the parts where most of the time is spent and the rest of this chapter will be devoted to techniques allowing to do so. At this point, it is worth emphasizing that a piece of code executed relatively fast can be more relevant than a piece of code executed slowly if the first one is executed very often while the second one is executed only once.

[1] For more information, see the CuPy homepage.

[2] For more information, see Cython – C-Extensions for Python.

[3] For more information, see the PyPy homepage.

[4] For more information, see the Numba homepage.

[5] Donald E. Knuth is well known far beyond the computer science community as the author of the typesetting system TeX.

[6] D.E. Knuth, Computing Surveys **6**, 261 (1974). The quote can be found on page 268.

Code optimization often entails a risk for bugs to enter the code. Obviously, fast-running code is not worth anything if it does not produce correct results. It can then be very reassuring if one can rely on a comprehensive test suite (Section 3) and if one has made use of a version control system (Section 2) during code development and code optimization.

Before discussing techniques for timing code execution, we will discuss a few potential pitfalls.

5.2 Some pitfalls in run-time analysis

A simple approach to performing a run-time analysis of code could consist in taking the time before and after the code is executed. The `time` module in the Python standard library offers the possibility to determine the current time:

```
>>> import time
>>> time.ctime()
'Thu Dec 27 11:13:33 2018'
```

While this result is nicely readable, it is not well suited to calculate time differences. For this purpose, the seconds passed since the beginning of the epoch are better suited. On Unix systems, the epoch starts on January 1, 1970 at 00:00:00 UTC:

```
>>> time.time()
1545905769.0189064
```

Now, it is straightforward to determine the time elapsed during the execution of a piece of code. The following code repeats the execution several times to convey an idea of the fluctuations to be expected.

```
import time

for _ in range(10):
    sum_of_ints = 0
    start = time.time()
    for n in range(1000000):
        sum_of_ints = sum_of_ints + 1
    end = time.time()
    print(f'{end-start:5.3f}s', end='  ')
```

Executing this code yields:

```
0.142s  0.100s  0.093s  0.093s  0.093s  0.093s  0.092s  0.091s  0.091s  0.091s
```

Doing a second run on the same hardware, we obtained:

```
0.131s  0.095s  0.085s  0.085s  0.088s  0.085s  0.084s  0.085s  0.085s  0.085s
```

While these numbers give an idea of the execution time, they should not be taken too literally. In particular, it makes sense to average over several loops. This is facilitated by the `timeit` module in the Python standard library which we will discuss in the following section.

When performing run-time analysis as just described, one should be aware that a computer may be occupied by other tasks as well. In general, the total elapsed time will thus differ from the time actually needed to execute a specific piece of code. The `time` module therefore provides two functions. In addition to the `time` function which records the wall clock time, there exist a `process_time` function which counts the time attributed to the specific process running our Python script. The following example demonstrates the difference by intentionally letting the program pause for a second once in a while. Note, that although the execution of `time.sleep` occurs within the process under consideration, the time needed is ignored by `process_time`. Therefore, we can use `time.sleep` to simulate other activities of the computer, even if it is done in a somewhat inappropriate way.

```
import time

```

(continues on next page)

(continued from previous page)

```
sum_of_ints = 0
start = time.time()
start_proc = time.process_time()
for n in range(10):
    for m in range(100000):
        sum_of_ints = sum_of_ints + 1
    time.sleep(1)
end = time.time()
end_proc = time.process_time()
print(f'total time:   {end-start:5.3f}s')
print(f'process time: {end_proc-start_proc:5.3f}s')
```

In a run on the same hardware as used before, we find the following result:

```
total time:   10.207s
process time: 0.197s
```

The difference basically consists of the ten seconds spent while the code was sleeping.

One should also be aware that enclosing the code in question in a function will lead to an additional contribution to the execution time. This particularly poses a problem if the execution of the code itself requires only little time. We compare the two scripts

```
import time

sum_of_ints = 0
start_proc = time.process_time()
for n in range(10000000):
    sum_of_ints = sum_of_ints + 1
end_proc = time.process_time()
print(f'process time: {end_proc-start_proc:5.3f}s')
```

and

```
import time

def increment_by_one(x):
    return x+1

sum_of_ints = 0
start_proc = time.process_time()
for n in range(10000000):
    increment_by_one(sum_of_ints)
end_proc = time.process_time()
print(f'process time: {end_proc-start_proc:5.3f}s')
```

Tht first script takes on average over 10 runs 0.9 seconds while the second script takes 1.1 seconds and thus runs about 20% slower.

Independently of the methods used and even if one of the methods discussed later is employed, a run-time analysis will always influence the execution of the code. The measured run time therefore will be larger than without doing any timing. However, we should still be able to identify the parts of the code which take most of the time.

A disadvantage of the methods discussed so far consists in the fact that they require a modification of the code. Usually, it is desirable to avoid such modifications as much as possible. In the following sections, we will present a few timing techniques which can be used according to the specific needs.

5.3 The `timeit` module

Short isolated pieces of code can conveniently be analyzed by functions provided by the `timeit` module. By default, the average code execution time will be determined on the basis of one million of runs. As a first example, let us determine the execution time for the evaluation of the square of 0.5:

```
>>> import timeit
>>> timeit.timeit('0.5**2')
0.021714384998631430
```

The result is given in seconds. In view of one million of code executions, we obtain an execution time of 22 nanoseconds. If we want to use an argument, we cannot define it in the outer scope:

```
>>> x = 0.5
>>> timeit.timeit('x**2')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/opt/anaconda3/lib/python3.6/timeit.py", line 233, in timeit
    return Timer(stmt, setup, timer, globals).timeit(number)
  File "/opt/anaconda3/lib/python3.6/timeit.py", line 178, in timeit
    timing = self.inner(it, self.timer)
  File "<timeit-src>", line 6, in inner
NameError: name 'x' is not defined
```

Instead, we can pass the global namespace through the `globals` argument:

```
>>> x = 0.5
>>> timeit.timeit('x**2', globals=globals())
0.103586286000791
```

As an alternative, one can explicitly assign the variable `x` in the second argument intended for setup code. Its execution time is not taken into account:

```
>>> timeit.timeit('x**2', 'x=0.5')
0.08539198899961775
```

If we want to compare with the `pow` function of the `math` module, we have to add the import statement to the setup code as well:

```
>>> timeit.timeit('math.pow(x, 2)', 'import math; x=0.5')
0.2346674630025518
```

A more complex example of the use of the `timeit` module compares the evaluation of a trigonometric function by means of a NumPy universal function with the use of the corresponding function of the `math` module:

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

def f_numpy(nmax):
    x = np.linspace(0, np.pi, nmax)
    result = np.sin(x)

def f_math(nmax):
    dx = math.pi/(nmax-1)
    result = [math.sin(n*dx) for n in range(nmax)]

x = []
```

(continues on next page)

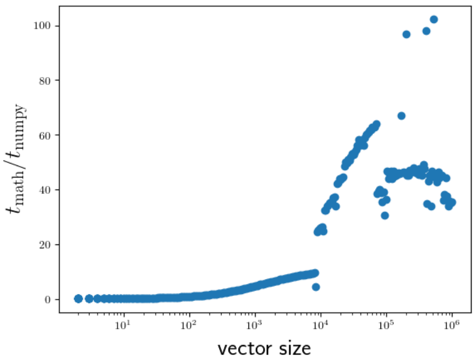


Figure 5.1: Comparison of execution times of the sine functions taken from the NumPy package and from the `math` module for a range of vector sizes.

(continued from previous page)

```
y = []
for n in np.logspace(0.31, 6, 300):
    nint = int(n)
    t_numpy = timeit.timeit('f_numpy(nint)', number=10, globals=globals())
    t_math = timeit.timeit('f_math(nint)', number=10, globals=globals())
    x.append(nint)
    y.append(t_math/t_numpy)

plt.rc('text', usetex=True)
plt.plot(x, y, 'o')
plt.xscale('log')
plt.xlabel('vector size', fontsize=20)
plt.ylabel(r'$t_\mathrm{math}/t_\mathrm{numpy}$', fontsize=20)
plt.show()
```

The result is displayed in Figure 5.1.

We close this section with two remarks. If one wants to assess the fluctuations of the measure execution times, one can replace the `timeit` function by the `repeat` function:

```
>>> x = 0.5
>>> timeit.repeat('x**2', repeat=10, globals=globals())
[0.1035151930009306, 0.07390781700087246, 0.06162133299949346,
 0.05376200799946673, 0.05260805999932927, 0.05276966699966579,
 0.05227632500100299, 0.052304120999906445, 0.0523306600007345,
 0.05286436900132685]
```

For users of the IPython shell or the Jupyter notebook, the magics `%timeit` and `%%timeit` provide a simple way to time the execution of a single line of code or a code cell, respectively. These magics choose a reasonable number of repetitions to obtain good statistics within a reasonable amount of time.

5.4 The `cProfile` module

The `timeit` module discussed in the previous section is useful to determine the execution time of one-liners or very short code segments. It is not very useful though to determine the compute-time intensive parts of a bigger program. If the program is nicely modularized in functions and methods, the `cProfile` module will be of help. It determines, how much time is spent in the individual functions and methods and thereby gives valuable information about which parts will benefit from code optimization.

We consider as a specific example the quantum mechanical time evolution of a narrow Gaussian wave packet initially

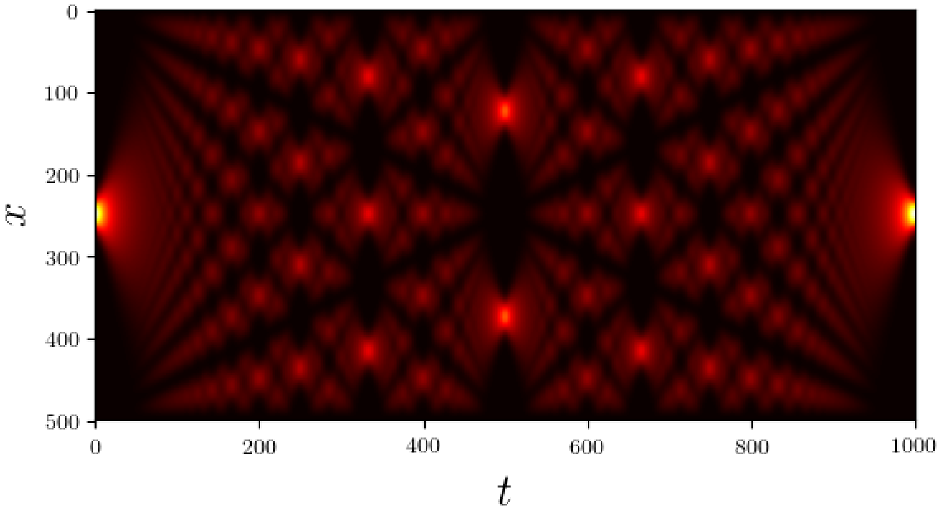


Figure 5.2: Time evolution of the probability density of an initial Gaussian wave packet positioned at the center of an infinite potential well. Brighter colors imply larger probability densities.

localized at the center of an infinite potential well[7]. The initial state is decomposed in the appropriately truncated eigenbasis of the potential well. Once the coefficients of the expansion are known, it is straightforward to determine the state at any later time. The time evolution of the probability density is shown in Figure 5.2.

This figure has been obtained by means of the following Python script called `carpet.py`.

```python
from math import cos, exp, pi, sin, sqrt
from cmath import exp as cexp
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm

class InfiniteWell:
    def __init__(self, psi0, width, nbase, nint):
        self.width = width
        self.nbase = nbase
        self.nint = nint
        self.coeffs = self.get_coeffs(psi0)

    def eigenfunction(self, n, x):
        if n % 2:
            return sqrt(2/self.width)*sin((n+1)*pi*x/self.width)
        return sqrt(2/self.width)*cos((n+1)*pi*x/self.width)

    def get_coeffs(self, psi):
        coeffs = []
        for n in range(self.nbase):
            f = lambda x: psi(x)*self.eigenfunction(n, x)
            c = trapezoidal(f, -0.5*self.width, 0.5*self.width, self.nint)
            coeffs.append(c)
        return coeffs

    def psi(self, x, t):
        psit = 0
        for n, c in enumerate(self.coeffs):
```

(continues on next page)

[7] For more details, see e.g. W. Kinzel, Bilder elementarer Quantenmechanik, Phys. Bl. **51**, 1190 (1995) and I. Marzoli, F. Saif, I. Bialynicki-Birula, O. M. Friesch, A. E. Kaplan, W. P. Schleich, Quantum carpets made simple, Acta Phys. Slov. **48**, 323 (1998).

(continued from previous page)

```
30                  psit = psit + c*cexp(-1j*(n+1)**2*t)*self.eigenfunction(n, x)
31              return psit
32  
33      def trapezoidal(func, a, b, nint):
34          delta = (b-a)/nint
35          integral = 0.5*(func(a)+func(b))
36          for k in range(1, nint):
37              integral = integral+func(a+k*delta)
38          return delta*integral
39  
40      def psi0(x):
41          sigma = 0.005
42          return exp(-x**2/(2*sigma))/(pi*sigma)**0.25
43  
44      w = InfiniteWell(psi0=psi0, width=2, nbase=100, nint=1000)
45      x = np.linspace(-0.5*w.width, 0.5*w.width, 500)
46      ntmax = 1000
47      z = np.zeros((500, ntmax))
48      for n in range(ntmax):
49          t = 0.25*pi*n/(ntmax-1)
50          y = np.array([abs(w.psi(x, t))**2 for x in x])
51          z[:, n] = y
52      z = z/np.max(z)
53      plt.rc('text', usetex=True)
54      plt.imshow(z, cmap=cm.hot)
55      plt.xlabel('$t$', fontsize=20)
56      plt.ylabel('$x$', fontsize=20)
57      plt.show()
```

This code is by no means optimal. After all, we want to discuss strategies to find out where most of the compute time is spent and what we can do to improve the situation. Before doing so, let us get a general idea of how the code works.

The initial wave function is the Gaussian defined in the function `psi0` in lines 40-42. Everything related to the basis functions is collected in the class `InfiniteWell`. During the instantiation, we define the initial wave function `psi0`, the total width of the well `width`, the number of basis states `nbase`, and the number of integration points `nint` to be used when determining the coefficients. The expansion coefficients of the initial state are determined in `get_coeffs` defined in lines 19-25 and called from line 12. The value of the eigenfunction corresponding to eigenvalue `n` at position `x` is obtained by means of the method `eigenfunction` defined in line 14-17. The integration is carried out very simply according to the trapezoidal rule as defined in function `trapezoidal` in lines 33-38. The wave function at a given point `x` and a given time `t` is calculated by method `psi` defined in lines 27-31. The code from line 44 to the end serves to calculate the time evolution and to render the image shown in Figure 5.2.

In this version of the code, we deliberately do not make use of NumPy except to obtain the image. Of course, NumPy would provide a significant speedup right away and one would probably never write the code in the way shown here. But it provides a good starting point to learn about run-time analysis. Where does the code spend most of its time?

To address this question, we make use of the `cProfile` module contained in the Python standard library. Among the various ways of using this module, we choose one which avoids having to change our script:

```
$ python -m cProfile -o carpet.prof carpet.py
```

This command runs the script `carpet.py` under the control of the `cProfile` module. The option `-o carpet.prof` indicates that the results of this profiling run are stored in the file `carpet.prof`. This binary file allows to analyze the obtained data in various ways by means of the `pstats` module. Let us try it out:

```
>>> import pstats
>>> p = pstats.Stats('carpet.prof')
```

(continues on next page)

(continued from previous page)

```
  >>> p.sort_stats('time').print_stats(15)
  Tue Jan 22 17:04:46 2019    carpet.prof

        202073424 function calls (202065686 primitive calls) in 633.175 seconds

  Ordered by: internal time
  List reduced from 3684 to 15 due to restriction <15>

  ncalls  tottime  percall  cumtime  percall filename:lineno(function)
50100100  232.867    0.000  354.468    0.000 carpet.py:14(eigenfunction)
  500000  184.931    0.000  623.191    0.001 carpet.py:27(psi)
50000000   84.417    0.000   84.417    0.000 {built-in method cmath.exp}
50100101   55.301    0.000   55.301    0.000 {built-in method math.sqrt}
25050064   33.170    0.000   33.170    0.000 {built-in method math.cos}
25050064   33.129    0.000   33.129    0.000 {built-in method math.sin}
       1    3.326    3.326    4.341    4.341 {built-in method exec_}
    1000    1.878    0.002  625.763    0.626 carpet.py:50(<listcomp>)
  503528    0.699    0.000    0.699    0.000 {built-in method builtins.abs}
    1430    0.372    0.000    0.372    0.000 {method 'read' of '_io.BufferedReader
↪' objects}
  100100    0.348    0.000    1.386    0.000 carpet.py:22(<lambda>)
       2    0.300    0.150    0.300    0.150 {built-in method statusBar}
  100100    0.294    0.000    0.413    0.000 carpet.py:40(psi0)
     100    0.154    0.002    1.540    0.015 carpet.py:33(trapezoidal)
  100101    0.119    0.000    0.119    0.000 {built-in method math.exp}
```

After having imported the `pstats` module, we load our profiling file `carpet.prof` to obtain a statistics object `p`. The data can then be sorted with the `sort_stats` method according to different criteria. Here, we have chosen the time spent in a function. Since the list is potentially very long, we have restricted the output to 15 entries by means of the `print_stats` method.

Let us take a look at the information provided by the run-time statistics. Each line corresponds to one of the 15 most time-consuming functions and methods out of a total of 3695. The total time of about 667 seconds is mostly spent in the function `psi` listed in the second line. There are actually two times given here. The total time (`tottime`) of 196 seconds counts only the time actually spent inside the function. The time required to execute functions called from `psi` are not counted. In contrast, these times count towards the cumulative time (`cumtime`). An important part of the difference can be explained by the evaluation of the eigenfunctions as listed in the first line.

Since we have sorted according to `time`, which actually corresponds to `tottime`, the first line lists the method consuming most of the time. Even though the time needed per call is so small that it is given as 0.000 seconds, this function is called very often. In the column `ncalls` the corresponding value is listed as 50100100. In such a situation, it makes sense to check whether this number can be understood.

In a first step, the expansion coefficients need to be determined. We use 100 basis functions as specified by `nbase` and 1001 nodes which is one more than the value of `nint`. This results in a total of 100100 evaluations of eigenfunctions, actually less than a percent of the total number of evaluations of eigenfunctions. In order to determine the data displayed in Figure 5.2, we evaluate 100 eigenfunctions on a grid of size 500×1000, resulting in a total of 50000000 evaluations.

These considerations show that we do not need to bother with improving the code determining the expansion coefficients. However, the situation might be quite different if we would not want to calculate data for a relatively large grid. Thinking a bit more about it, we realize that the number of 50000000 evaluations for the time evolution is much too big. After all, we are evaluating the eigenfunctions at 500 different positions and we are considering 100 eigenfunctions, resulting in only 50000 evaluations. For each of the 1000 time values, we are unnecessarily recalculating eigenfunctions for the same arguments. Avoiding this waste of compute time could speed up our script significantly.

There are basically two ways to do so. We could restructure our program in such a way that we evaluate the grid for constant position along the time direction. Then, we just need to keep the values of the 100 eigenfunctions at a given position. If we want to have the freedom to evaluate the wave function at a given position and time on a certain grid, we could also store the values of all eigenfunctions at all positions on the grid in a cache for later reuse. This is an example of trading compute time against memory. We will implement the latter idea in the next version of our script listed below.[8]

```
from math import cos, exp, pi, sin, sqrt
from cmath import exp as cexp
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm

class InfiniteWell:
    def __init__(self, psi0, width, nbase, nint):
        self.width = width
        self.nbase = nbase
        self.nint = nint
        self.coeffs = self.get_coeffs(psi0)
        self.eigenfunction_cache = {}

    def eigenfunction(self, n, x):
        if n % 2:
            return sqrt(2/self.width)*sin((n+1)*pi*x/self.width)
        return sqrt(2/self.width)*cos((n+1)*pi*x/self.width)

    def get_coeffs(self, psi):
        coeffs = []
        for n in range(self.nbase):
            f = lambda x: psi(x)*self.eigenfunction(n, x)
            c = trapezoidal(f, -0.5*self.width, 0.5*self.width, self.nint)
            coeffs.append(c)
        return coeffs

    def psi(self, x, t):
        if not x in self.eigenfunction_cache:
            self.eigenfunction_cache[x] = [self.eigenfunction(n, x)
                                           for n in range(self.nbase)]
        psit = 0
        for n, (c, ef) in enumerate(zip(self.coeffs, self.eigenfunction_cache[x])):
            psit = psit + c*ef*cexp(-1j*(n+1)**2*t)
        return psit

def trapezoidal(func, a, b, nint):
    delta = (b-a)/nint
    integral = 0.5*(func(a)+func(b))
    for k in range(1, nint):
        integral = integral+func(a+k*delta)
    return delta*integral

def psi0(x):
    sigma = 0.005
    return exp(-x**2/(2*sigma))/(pi*sigma)**0.25

w = InfiniteWell(psi0=psi0, width=2, nbase=100, nint=1000)
x = np.linspace(-0.5*w.width, 0.5*w.width, 500)
ntmax = 1000
z = np.zeros((500, ntmax))
```

(continues on next page)

[8] An alternative approach to caching would consist in using the `functools.lru_cache` decorator. However, in our specific case it turns out that the overhead introduced by the `lru_cache` has a significant impact on the performance. The situation might be different if it is more time consuming to obtain the results to be cached.

(continued from previous page)

```
for n in range(ntmax):
    t = 0.25*pi*n/(ntmax-1)
    y = np.array([abs(w.psi(x, t))**2 for x in x])
    z[:, n] = y
z = z/np.max(z)
plt.rc('text', usetex=True)
plt.imshow(z, cmap=cm.hot)
plt.xlabel('$t$', fontsize=20)
plt.ylabel('$x$', fontsize=20)
plt.show()
```

Now, we check in method `psi` whether the eigenfunction cache already contains data for a given position `x`. If this is not the case, the required values are calculated and the cache is updated.

As a result of this modification of the code, the profiling data change considerably:

```
Tue Jan 22 17:10:33 2019    carpet.prof

         52205250 function calls (52197669 primitive calls) in 185.581 seconds

   Ordered by: internal time
   List reduced from 3670 to 15 due to restriction <15>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   500000   97.612    0.000  177.453    0.000 carpet.py:28(psi)
 50000000   79.415    0.000   79.415    0.000 {built-in method cmath.exp}
     1000    2.162    0.002  180.342    0.180 carpet.py:54(<listcomp>)
        1    1.176    1.176    2.028    2.028 {built-in method exec_}
   503528    0.732    0.000    0.732    0.000 {built-in method builtins.abs}
   150100    0.596    0.000    0.954    0.000 carpet.py:15(eigenfunction)
   100100    0.353    0.000    1.349    0.000 carpet.py:23(<lambda>)
     1430    0.323    0.000    0.323    0.000 {method 'read' of '_io.BufferedReader
↪' objects}
        2    0.301    0.151    0.301    0.151 {built-in method statusBar}
   100100    0.278    0.000    0.396    0.000 carpet.py:44(psi0)
   150101    0.171    0.000    0.171    0.000 {built-in method math.sqrt}
      100    0.140    0.001    1.489    0.015 carpet.py:37(trapezoidal)
   100101    0.119    0.000    0.119    0.000 {built-in method math.exp}
    75064    0.095    0.000    0.095    0.000 {built-in method math.sin}
    75064    0.093    0.000    0.093    0.000 {built-in method math.cos}
```

We observe a speed-up of a factor of 3.6 by investing about 500×100×8 bytes of memory, i.e. roughly 400 kB. The exact value will be slightly different because we have stored the data in a dictionary and not in an array, but clearly we are not talking about a huge amount of memory. The time needed to evaluate the eigenfunctions has dropped so much that it can be neglected compared to the time required by the method `psi` and the evaluation of the complex exponential function.

The compute could be reduced further by caching the values of the complex exponential functions. In fact, we unnecessarily recalculate each value 500 times. However, there are still almost 96 seconds left which are spent in the rest of the `psi` method. We will see in the following section how one can find out which line of the code is responsible for this important contribution to the total run time.

Before doing so, we want to present a version of the code designed to profit from NumPy from the very beginning

```
from math import sqrt
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
```

(continues on next page)

(continued from previous page)

```
class InfiniteWell:
    def __init__(self, psi0, width, nbase, nint):
        self.width = width
        self.nbase = nbase
        self.nint = nint
        self.coeffs = trapezoidal(lambda x: psi0(x)*self.eigenfunction(x),
                                  -0.5*self.width, 0.5*self.width, self.nint)

    def eigenfunction(self, x):
        assert x.ndim == 1
        normalization = sqrt(2/self.width)
        args = (np.arange(self.nbase)[:, np.newaxis]+1)*np.pi*x/self.width
        result = np.empty((self.nbase, x.size))
        result[0::2, :] = normalization*np.cos(args[0::2])
        result[1::2, :] = normalization*np.sin(args[1::2])
        return result

    def psi(self, x, t):
        coeffs = self.coeffs[:, np.newaxis]
        eigenvals = np.arange(self.nbase)[:, np.newaxis]
        tvals = t[:, np.newaxis, np.newaxis]
        psit = np.sum(coeffs * self.eigenfunction(x)
                      * np.exp(-1j*(eigenvals+1)**2*tvals), axis= -2)
        return psit

def trapezoidal(func, a, b, nint):
    delta = (b-a)/nint
    x = np.linspace(a, b, nint+1)
    integrand = func(x)
    integrand[..., 0] = 0.5*integrand[..., 0]
    integrand[..., -1] = 0.5*integrand[..., -1]
    return delta*np.sum(integrand, axis=-1)

def psi0(x):
    sigma = 0.005
    return np.exp(-x**2/(2*sigma))/(np.pi*sigma)**0.25

w = InfiniteWell(psi0=psi0, width=2, nbase=100, nint=1000)
x = np.linspace(-0.5*w.width, 0.5*w.width, 500)
t = np.linspace(0, np.pi/4, 1000)
z = np.abs(w.psi(x, t))**2
z = z/np.max(z)
plt.rc('text', usetex=True)
plt.imshow(z.T, cmap=cm.hot)
plt.xlabel('$t$', fontsize=20)
plt.ylabel('$x$', fontsize=20)
plt.show()
```

The structure of the code is essentially unchanged, but we are making use of universal functions in several places. In the method `psi`, a three-dimensional array is used with axis 0 to 2 given by time, eigenvalue, and position. A run-time analysis yields the following result:

```
Tue Jan 22 17:12:41 2019    carpet.prof

         457912 function calls (450291 primitive calls) in 4.644 seconds
```

(continues on next page)

(continued from previous page)

```
   Ordered by: internal time
   List reduced from 3682 to 15 due to restriction <15>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    1.707    1.707    2.543    2.543 {built-in method exec_}
        1    0.410    0.410    0.481    0.481 carpet.py:23(psi)
     1430    0.284    0.000    0.284    0.000 {method 'read' of '_io.BufferedReader
↪' objects}
        2    0.276    0.138    0.276    0.138 {built-in method statusBar}
      407    0.074    0.000    0.074    0.000 {method 'reduce' of 'numpy.ufunc'␣
↪objects}
    48/46    0.056    0.001    0.061    0.001 {built-in method _imp.create_dynamic}
    35469    0.049    0.000    0.053    0.000 {built-in method builtins.isinstance}
      284    0.048    0.000    0.048    0.000 {built-in method marshal.loads}
        1    0.039    0.039    0.040    0.040 {built-in method show}
        2    0.036    0.018    0.210    0.105 /opt/anaconda3/lib/python3.7/site-
↪packages/matplotlib/font_manager.py:1198(_findfont_cached)
  418/185    0.028    0.000    0.099    0.001 /opt/anaconda3/lib/python3.7/sre_
↪parse.py:475(_parse)
    23838    0.028    0.000    0.028    0.000 {method 'lower' of 'str' objects}
       63    0.027    0.000    0.027    0.000 {built-in method io.open}
    21637    0.025    0.000    0.025    0.000 {method 'startswith' of 'str'␣
↪objects}
1169/1101    0.025    0.000    0.200    0.000 {built-in method builtins.__build_
↪class__}
```

Since the run time obtained by profiling is longer than the actual run time, we have measured the latter for the first version of the script and the NumPy version, resulting in an speed up by a factor of 90. Even though this improvement is impressive, the code given above could be improved even further. However, as this is not the main purpose of the chapter, we rather turn our attention to how one can do profiling on a line-by-line basis.

5.5 Line oriented run-time analysis

In the second version of the script discussed in the previous section, we had seen that by far most of the time was spent in the method `psi`. Almost half of the time was spent in the complex exponential function so that a significant amount of time must be spent elsewhere in the function. At this point, the `cProfile` module is not of much help as it only works on the function level.

Fortunately, there is a line profiling tool available. However, it is not part of the Anaconda distribution and needs to be installed separately. The package is called `line_profiler` and can be found on the Python package index (PyPI). It can be installed either into a virtual environment or in a conda environment.

Line profiling adds some overhead to the code execution and it makes sense to limit it to the most important function or a few of them. This can easily be done by decorating the function in question with `@profile`. Since we know that the `psi` method constitutes the bottleneck of our calculation, we only decorate that method. Running the line profiler on our script called carpet.py is done by[9]:

```
$ kernprof -l -v carpet.py
```

Here, the option `-l` requests the line-by-line profiler and `-v` allows to immediately view the results in addition to storing them in a file with extension `lprof`. We obtain the following result:

```
Wrote profile results to carpet.py.lprof
Timer unit: 1e-06 s
```

(continues on next page)

[9] The command name `kernprof` makes reference to the author of the package Robert Kern.

(continued from previous page)

```

Total time: 306.266 s
File: carpet.py
Function: psi at line 27

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    27                                               @profile
    28                                               def psi(self, x, t):
    29    500000    1171076.0      2.3      0.4          if not self.coeffs:
    30         1     348711.0 348711.0      0.1              self.get_coeffs(psi0)
    31    500000    1435067.0      2.9      0.5          if not x in self.
↪eigenfunction_cache:
    32       500       1256.0      2.5      0.0              self.eigenfunction_
↪cache[x] = [self.eigenfunction(n, x)
    33       500     105208.0    210.4      0.0                                    ␣
↪          for n in range(self.nbase)]
    34    500000    1190160.0      2.4      0.4          psit = 0
    35  50500000  132196091.0      2.6     43.2          for n, (c, ef) in␣
↪enumerate(zip(self.coeffs, self.eigenfunction_cache[x])):
    36  50000000  168606042.0      3.4     55.1              psit = psit +␣
↪c*ef*cexp(-1j*(n+1)**2*t)
    37    500000    1212643.0      2.4      0.4          return psit
```

The timing information only refers to the function on which the line profiler is run. We can see here that the for loop is responsible for a significant portion of the execution time. Making use of NumPy arrays can improve the performance of the code dramatically as we have seen at the end of the previous section.

By using the option `-v`, we were able to immediately see the result of the profiling run. In addition, a file `carpet.py.lprof` has been created. It is possible to obtain the profiling result from it later by means of:

```
python -m line_profiler carpet.py.lprof
```

CHAPTER

SIX

DOCUMENTATION OF CODE

Besides writing code and testing it, documenting the code is also an important task which should not be neglected. In Python, it is a good habit to provide each function or method with a docstring which might even contain doctests as we have discussed in Section 3.2. For more complex programs, modules or even packages, it will not be sufficient to limit the documentation to the doctests. This chapter will be devoted to the discussion of the documentation tool *Sphinx* which is commonly employed to document Python project but which can be used also to document projects written in other programming languages. Even the present lecture notes are making use of *Sphinx*.

Sphinx is based on the markup language *reStructuredText*. Due to its unobtrusive syntax the original text can easily be read. At the same time, the markup is powerful enough to produce nicely laid out output in different formats, in particular in HTML and LaTeX. The latter can directly be used to produce the documentation in PDF format.

The value of *Sphinx* for the documentation of software projects relies to a large extent on its capability to make use of docstrings for inclusion in the documentation. *Sphinx* thus provides another good reason to supply functions and methods with docstrings.

In the following, we will first give an introduction to the markup language *reStructuredText* and then explain some of the more important aspects of how *Sphinx* can be used to document code. For further information, we refer to the documentation of *reStructuredText*[1] and *Sphinx*[2].

6.1 Markup with reStructuredText

Markup languages are used to annotate text for electronic text processing, for example in order to specify its meaning. A text could be marked as representing a section title and a computer program could then represent it accordingly, e.g. as larger text set in boldface. A widespread markup language is the HyperText Markup Language HTML used for markup of webpages. A pair of tags `<LI>` and `</LI>` would indicate in HTML that the enclosed text represents an item in a list. An example of a markup language commonly found in a scientific context is LaTeX. Here, `x` and `$x$` will be typeset differently because the dollar signs in the second case indicate that the character `x` is meant to be a mathematical variable which usually is typeset in an italic font.

The markup in HTML and LaTeX helps computer programs to interpret the meaning of the text and to represent it correctly. However, text written in these markup languages often lacks a good human readability. This is particularly true for the very flexible extensible markup language XML.

On the other hand, there exist so-called lightweight markup languages like reStructuredText or Markdown where the latter may come in different variants. These markup languages are designed in such a way that the meaning of the markup appears rather natural to a human reader. From the following example written in reStructuredText

```
Markup of lists
===============

The following is a bullet-point list:

* first item
* second item
```

[1] More information on reStructuredText can be found in the documentation of the docutils project at http://docutils.sourceforge.net/rst.html.

[2] The *Sphinx* project page can be found at https://www.sphinx-doc.org/.

it is pretty clear that the first two lines represent a header title and the last two lines represent a list. Due to the simplicity of its markup, Markdown or one of its variants is commonly used in Wikis. Both, texts written in Markdown or in reStructuredText are frequently used for documentation files in software projects like README.md or README.rst, respectively, which usually specify the purpose of the software and give further useful information. In version control systems like Gitlab, they can be represented in a nice form in the browser.

The documentation generator *Sphinx* is based on reStructuredText. Therefore, we will now discuss some of the more important aspects of this markup language.

Within a text, parts can be emphasized or even strongly emphasized by enclosing them in one or two stars, respectively. Inline literals are enclosed in a pairs of back-quotes. It is important that these constructs should be delimited by characters which could also be used otherwise to delimit words like a whitespace or a punctuation character. If a whitespace is used but should not appear in the output, it needs to be escaped by means of a backslash. The text to which the markup is applied may not start or end with a whitespace. The following example provides an illustration.

```
Text can be *emphasized*, usually as italics, or even **strongly emphasized**,
usually as boldface. It is also possible to insert ``inline literals`` which
will usually be represented as monospaced text.

This is another paragraph showing how to embed an inline literal while
suppressing the surrounding blanks: re\ ``structured``\ Text.
```

will be represented as[3]

> Text can be *emphasized*, usually as italics, or even **strongly emphasized**, usually as boldface. It is also possible to insert `inline literals` which will usually be represented as monospaced text.
>
> This is another paragraph showing how to embed an inline literal while suppressing the surrounding blanks: re`structured`Text.

This example also shows that paragraphs are separated by a blank line. On a higher level, text is sectioned into parts, chapters, sections etc. A hierarchy is established by adorning titles in a systematic way. To this end, an underline or an underline together with an overline is added to the corresponding title. An underline or overline is at least as long as the title and contains only identical non-alphanumeric printable ASCII characters. It is recommended to choose among the characters `= - ` : . ' " ~ ^ _ * + #`. Note that even though in this way one can define a large number of different sectioning levels, in practice this number may be limited. For example, in HTML the number of different headings is limited to six. An example of sectioning of a text could look as follows:

```
============
Introduction
============

A first section
===============

Here comes some text ...

A second section
================
More text...

A subsection
------------
And so on...
```

As this example indicates, an empty line can be put after a title but this is not mandatory.

Lists, either as bullet-point lists or as enumerated lists, can easily be obtained in reStructuredText. In a bullet-point list, the items are indicated by a few characters including `* + - •`. If the text if an item runs over several lines,

[3] Note that the representation given here and in following examples is generated by the LaTeX builder of *Sphinx*. It may look differently if the representation is generated otherwise, e.g. with tools like `rst2html` or `rst2latex` provided by `docutils` or `rst2pdf`.

it needs to be consistently indented. Sublists need to be separated from the surrounding list by empty lines. The following example illustrates the use of bullet-point lists:

```
* This is the text for the first item which runs over several lines. Make
  sure that the text is consistently indented.

  Further paragraphs in an item can be added provided the indentation
  is consistent.
* second item

  * a subitem

* third item
```

This code results in

- This is the text for the first item which runs over several lines. Make sure that the text is consistently indented.

  Further paragraphs in an item can be added provided the indentation is consistent.

- second item

  - A subitem is obtained by indenting the corresponding entry.

- third item

An enumerated list can be numbered explicitly by numbers, alphabet characters in uppercase or lowercase, or Roman numerals. It is also possible to autonumber a list by means of `#.`. The sublist of the second item demonstrates two aspects. The first subitem specifies that lowercase letters should be used for enumeration and, in addition, the sublist should start with a letter other than "a". Once autonumbering has started, fixing a subsequent label would lead to the start of a new list.

The following code

```
#. first item with automatic numbering
#. second item

   p. subitem
   #. another subitem

#. another item
```

results in

1. first item with automatic numbering

2. second item

   p. subitem

   q. another subitem

3. another item

We have already seen how to produce inline literals which may be useful to mark for example keywords. To display multiline code, the *code* directive is appropriate. The following example makes use of the possibility to add linenumbers.

```
.. code:: python

   nmax = 10
   sum = 0
   for n in range(1, nmax+1):
       sum = sum+n**2
   print(nmax, sum)
```

Since it is indicated that the code is written in Python, the syntax of the code can be highlighted.

```python
nmax = 10
sum = 0
for n in range(1, nmax+1):
    sum = sum+n**2
print(nmax, sum)
```

Another possibility to typeset code is the use of two colons. If the colons follow the preceding text immediately, a single colon will be displayed at the end of the text:

```
The following script displays "hello world" three times::

   for _ in range(3):
       print('Hello world!')
```

Note the indentation of the code block which indicates which part of the text should be considered as code. The output is as follows:

> The following script displays “hello world” three times:
>
> ```python
> for _ in range(3):
>     print('Hello world!')
> ```

The colon in the output can be avoided if the colons are separated from the text by a blank:

```
The following script displays "hello world" three times. ::

   for _ in range(3):
       print('Hello world!')
```

Now, the output looks as follows:

> The following script displays “hello world” three times.
>
> ```python
> for _ in range(3):
>     print('Hello world!')
> ```

For scientific applications, one might want to include mathematical expressions. This can be done by means of the *math* role (`:math:`) for inline mathematical expressions and the *math* directive (`math::`) for displayed mathematical expressions. In both cases, the mathematical expression is entered in LaTeX format. The following code

```
Einstein found the famous formula :math:`E=mc^2` which describes the
equivalence of energy and mass.

.. math::

   \int_{-\infty}^\infty \mathrm{d}x \mathrm{e}^{-x^2} = \sqrt{\pi}
```

will result in the output:

> Einstein found the famous formula $E = mc^2$ which describes the equivalence of energy and mass.
>
> $$\int_{-\infty}^\infty \mathrm{d}x \mathrm{e}^{-x^2} = \sqrt{\pi}$$

There exists also a directive to include images:

```
.. image:: img/example.png
   :width: 100
   :height: 100
   :align: center
```

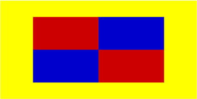
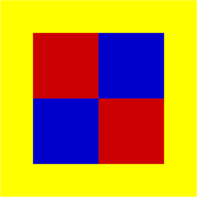


Figure 6.1: A graphics can be distorted by specifying `height` and `width`.

The name of the image file to be included needs to be specified. Here, the file happens to reside in a subdirectory `img` of the present directory. We have also specified the size and the alignment of the figure, resulting in the following output:



The `figure` directive can be used to add a figure caption. The caption text needs to be indented to indicate that it belongs to the figure directive.

```
.. figure:: img/example.png
   :height: 50
   :width: 100

   A graphics can be distorted by specifying ``height`` and ``width``.
```

This code results in Figure 6.1, which by means of the *Sphinx* LaTeX builder is created as a floating object.

Occasionally, one may want to include a link to a web resource. In a documentation, this might be desirable to refer to a publication where an algorithm or the theoretical basis of the code has been described. As an example, we present various ways to link to the seminal paper by Cooley and Tukey on the fast Fourier transformation. The numbering allows us to refer more easily to the three different versions and plays no role with respect to the links.

```
#. J. W. Cooley and J. W. Tukey, *An algorithm for the machine calculation
   of complex Fourier series*,
   `Math. Comput. 19, 297-301 (1965) <https://doi.org/10.2307/2003354>`_

#. J. W. Cooley and J. W. Tukey, *An algorithm for the machine calculation
   of complex Fourier series*,
   Math. Comput. **19**, 297-301 (1965) `<https://doi.org/10.2307/2003354>`_

#. J. W. Cooley and J. W. Tukey, *An algorithm for the machine calculation
   of complex Fourier series*,
   Math. Comput. **19**, 297-301 (1965) https://doi.org/10.2307/2003354
```

results in the output

1. J. W. Cooley and J. W. Tukey, *An algorithm for the machine calculation of complex Fourier series*, Math. Comput. 19, 297–301 (1965)
2. J. W. Cooley and J. W. Tukey, *An algorithm for the machine calculation of complex Fourier series*, Math. Comput. **19**, 297–301 (1965) https://doi.org/10.2307/2003354
3. J. W. Cooley and J. W. Tukey, *An algorithm for the machine calculation of complex Fourier series*, Math. Comput. **19**, 297–301 (1965) https://doi.org/10.2307/2003354

The first case represents the most comprehensive way to represent a link. The pair of back apostrophes encloses the text and the link delimited by a less-than and greater-than sign. The text will be shown in the output with the link associated with it. The underscore at the very end indicates that this is an outgoing link. In contrast to the two other variants, the volume number (19) can be set as boldface as nesting of the markup is not possible.

The second alternative explicitly displays the URL since no text is given. The same effect is obtained in the third variant by simply putting a URL which can be recognized as such.

In addition to external links, reStructuredText also allows to create internal links. An example are footnotes like in the following example.

```
This is some text. [#myfootnote]_ And more text...

.. [#myfootnote] Some remarks.
```

It is also possible to refer to titles of chapters or sections. The following example gives an illustration.

```
Introduction
============

This is an introductory chapter.

Chapter 1
=========

As discussed in the `Introduction`_ ...
```

Here, the text of the link has to agree with the text of the chapter or section.

The discussion of reStructuredText in this section did not attempt to cover all possibilities provided by this markup language. For more details, it is recommended to consult the documentation.

6.2 Sphinx documentation generator

The *Sphinx* documentation generator was initially created to produce the documentation for Python. However, it is very flexible and can be employed for many other use cases. In fact, the present lecture notes were also generated by means of *Sphinx*. As a documentation generator, *Sphinx* accepts documents written in reStructuredText including a number of extension and provides builders to convert the input into a variety of output formats, among them HTML and PDF, where the latter is obtained through LaTeX as an intermediate format. *Sphinx* offers the interesting possibility to autogenerate the documentation or part of it on the basis of the docstrings provided by the code being documented.

6.2.1 Setting up a Sphinx project

There is not a unique way to set up a *Sphinx* documentation project. For a unexperienced user of *Sphinx*, the probably simplest way is to invoke[4]

```
$ sphinx-quickstart
```

Remember that the dollar sign represents the command line prompt and should not be typed. The user will then be asked a number of questions and the answers will allow *Sphinx* to create the basic setup. For the documentation of a software project, it makes sense to store all documentation related material in a subdirectory `doc`. Then, `sphinx-quickstart` should either be run in this directory or the path to this directory should be given as an argument.

The dialog starts with a question about where to place the build directory relative to the source directory. The latter would for example be the directory `doc` and typically contains a configuration file, reStructuredText files, and possibly images. For a larger documentation, these files can be organized in a subdirectory structure. These source files will usually be put under version control. When creating the documentation in an output format, *Sphinx* puts intermediate files and the output in a special directory to avoid mixing these files with the source files. There are two ways to do so. A directory named `build` can be put in parallel to the `doc` directory or the it can be kept within the `doc` directory. Then it will be called `_build` where the underscore indicates its special role. It is not necessary to rerun `sphinx-quickstart` if you change your mind. One can instead modify the file `Makefile` and/or `make.bat`

[4] The following discussion is based on version 1.8.2 of *Sphinx* but should mostly apply to all recent versions of *Sphinx*.

which will be discussed below. It may be useful to add the build directory to the `.gitignore` file, provided a Git repository is used.

Sphinx now asks the user to choose between the two alternatives. `(y/n)` in the last line indicates the possible valid answers. `[n]` indicates the default value which can also be chosen by simply hitting the return key. Per default, *Sphinx* thus chooses to place build files into a directory `_build` within the source directory.

```
You have two options for placing the build directory for Sphinx output.
Either, you use a directory "_build" within the root path, or you separate
"source" and "build" directories within the root path.
> Separate source and build directories (y/n) [n]:
```

Often, it makes sense to follow the recommendations of *Sphinx*. Two pieces of information are however mandatory: the name of the project and the author name(s). The default language is English, but for example by choosing `de` it can be switched to German. This information is relevant when converting to LaTeX in order to choose the correct hyphenation patterns.

`sphinx-quickstart` offers also to enable a number of extensions. It is possible to change one's mind later by adapting the configuration file `conf.py`.

```
> autodoc: automatically insert docstrings from modules (y/n) [n]:
> doctest: automatically test code snippets in doctest blocks (y/n) [n]:
> intersphinx: link between Sphinx documentation of different projects (y/n) [n]:
> todo: write "todo" entries that can be shown or hidden on build (y/n) [n]:
> coverage: checks for documentation coverage (y/n) [n]:
> imgmath: include math, rendered as PNG or SVG images (y/n) [n]:
> mathjax: include math, rendered in the browser by MathJax (y/n) [n]:
> ifconfig: conditional inclusion of content based on config values (y/n) [n]:
> viewcode: include links to the source code of documented Python objects (y/n)␣
↪[n]:
> githubpages: create .nojekyll file to publish the document on GitHub pages (y/n)␣
↪[n]:
```

We briefly comment on a few of the more important extensions. `autodoc` should be enabled if one wants to generate documentation from the docstrings provided in the code being documented by the *Sphinx* project. `intersphinx` is useful if one wants to provide links to other projects. It is for example possible to refer to the NumPy documentation. `MathJax` is a Javascript package[5] which allows for high-quality typesetting of mathematical material in HTML.

Depending on the operating system(s) on which output is generated for the *Sphinx* project, one typically chooses either the `Makefile` for Un*x operating systems or a Windows command file for Windows operating systems or even both if more than one operating system is being used.

```
> Create Makefile? (y/n) [y]:
> Create Windows command file? (y/n) [y]:
```

While the conversion to an output format can always be done by means of `sphinx-build`, the task is facilitated by a Makefile or command file. On a Un*x system, running one of the commands

```
$ make html
$ make latexpdf
```

in the directory where the Makefile resides is sufficient to obtain HTML output of PDF output, respectively.

Accepting the default values proposed by *Sphinx*, the content of the source directory on a Un*x system will typically look as follows:

```
doc
+-- _build
+-- _static
```

(continues on next page)

[5] For more information see https://www.mathjax.org/.

(continued from previous page)

```
+-- _templates
+-- conf.py
+-- index.rst
+-- Makefile
```

As is indicated by the extension, `conf.py` is a Python file which defines the configuration of the *Sphinx* project. This file can be modified according to the user's need as long as the Python syntax is respected. `index.rst` is the main source file from which reference to other reStructuredText files can be made. Finally, `Makefile` defines what should be done when invoking `make` with one of the targets `html` or `latexpdf` or any other valid target specified by `make help`.

6.2.2 Sphinx configuration

As already mentioned, the file `conf.py` offers the possibility to adapt *Sphinx* to the needs of the project. Basic information includes the name of the project and of the author(s) as well as copyright information and version numbers. It makes sense to create a corresponding version tag in the project repository.

We have seen that `sphinx-quickstart` proposes the use of a number of extensions which, if selected, will appear in the list `extensions`. Here, other extensions may be added. When generating documentation from docstrings, the `napoleon` extensions is of particular interest. Its usefulness will be discussed in Section 6.2.3. This extension can be enabled by adding `sphinx.ext.napoleon` to the list of extensions.

The configuration file contains section for different output builders. We restrict ourselves here to HTML output and LaTeX output which can serve to produce a PDF document. Among the options for the HTML output, probably the most interesting variable is `html_theme`. https://www.sphinx-doc.org/en/stable/theming.html lists a few builtin themes which represent a simple way to change the look and feel of the HTML output. Third-party themes can be found at https://sphinx-themes.org/ and there is also the possibility to create one's own customized theme.

If the LaTeX output needs to be customized, the dictionary `latex_elements` is the place to look for. The configuration file created by `sphinx-quickstart` provides a structure which is commented out, but might be useful if one needs to customize certain aspects. For example, if one wants to typeset for A4 paper and would like to have floats preferentially placed on the top of the page, one might set the dictionary as follows:

```
latex_elements = {
     # The paper size ('letterpaper' or 'a4paper').
     #
     'papersize': 'a4paper',

     # The font size ('10pt', '11pt' or '12pt').
     #
     # 'pointsize': '10pt',

     # Additional stuff for the LaTeX preamble.
     #
     # 'preamble': '',

     # Latex figure (float) alignment
     #
     'figure_align': 'tbp',
}
```

An interesting entry is also `'preamble'` where all information can be specified which would normally go in the preamble of a LaTeX document, i.e. between the `documentclass` statement and the `\begin{document}` line. Note, however, that the use of backslashes common in LaTeX documents might require to specify the string as raw string by placing an `r` in front of it. A number of parameters specified by the Sphinx style file can be modified in an entry with key `'sphinxsetup'`. If the outline of the box enclosing code should be removed and a background color different from white should be used, one might specify:

```
'sphinxsetup': '''verbatimwithframe=false,
                  VerbatimColor={named}{AliceBlue},
               '''
```

This definition is used for the present document. Details about which parameters can be changed in this way and about other entries which can be added to the dictionary `latex_elements` can be found in the section LaTeX customization of the Sphinx documentation. Finally, it may be useful to know that in the list `latex_documents` several properties of the document are specified like the title and author appearing on the cover page of the documentation.

6.2.3 Autogeneration of a documentation

With *Sphinx* it is possible to autogenerate documentation from docstrings. Before discussing how docstrings can be formatted for use by *Sphinx*, we make a few general remarks on docstrings. Recommendations about how a docstring should look like are given in **PEP 257**.[8] We here focus on the keypoints pertinent to docstrings of methods and functions. The first line of the docstring should be a phrase ending in a period. It should be written as a command like "do something" instead of a description like "does something". This phrase should not exceed one line and it should be separated from the rest of the docstring by one empty line. Then, in particular the arguments of the function or method and the return value(s) should be explained. Or course, further information deemed useful can be given in addition.

In order to demonstrate the autogeneration of documentation from docstrings, we take as an example the last script for the quantum carpet discussed in Section 5. We have partially supplied the code with docstrings which now looks as follows:

```
from math import sqrt
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm

class InfiniteWell:
    """Quantum carpet for infinitely deep potential well.

    This class allows to determine the time evolution of an
    initial state in an infinitely deep potential well.

    :param func psi0: initial wave function
    :param float width: width of the potential well
    :param int nbase: number of basis states
    :param int nint: number of intervals used in the integration routine
    """

    def __init__(self, psi0, width, nbase, nint):
        self.width = width
        self.nbase = nbase
        self.nint = nint
        self.coeffs = trapezoidal(lambda x: psi0(x)*self.eigenfunction(x),
                                  -0.5*self.width, 0.5*self.width, self.nint)

    def eigenfunction(self, x):
        """Determine set of eigenfunction values at position ``x``.

        The basis set is limited by the number of eigenstates given by
        ``self.nbase``.

        :param x: position at which the eigenfunctions are to be determined
        :type x: float or numpy.ndarray
```

(continues on next page)

[8] PEP is short for Python Enhancement Proposal.

(continued from previous page)

```
        :return: array of eigenfunction values
        :rtype: numpy.ndarray
        :raises AssertionError: if the dimension of ``x`` does not equal 1
        """
        assert x.ndim == 1
        normalization = sqrt(2/self.width)
        args = (np.arange(self.nbase)[:, np.newaxis]+1)*np.pi*x/self.width
        result = np.empty((self.nbase, x.size))
        result[0::2, :] = normalization*np.cos(args[0::2])
        result[1::2, :] = normalization*np.sin(args[1::2])
        return result

    def psi(self, x, t):
        coeffs = self.coeffs[:, np.newaxis]
        eigenvals = np.arange(self.nbase)[:, np.newaxis]
        tvals = t[:, np.newaxis, np.newaxis]
        psit = np.sum(coeffs * self.eigenfunction(x)
                      * np.exp(-1j*(eigenvals+1)**2*tvals), axis= -2)
        return psit

def trapezoidal(func, a, b, nint):
    delta = (b-a)/nint
    x = np.linspace(a, b, nint+1)
    integrand = func(x)
    integrand[..., 0] = 0.5*integrand[..., 0]
    integrand[..., -1] = 0.5*integrand[..., -1]
    return delta*np.sum(integrand, axis=-1)

def psi0(x):
    """Determine Gaussian wave function.

    :param float x: position at which the wave function is determined
    :return: value of wave function at position ``x``
    :rtype: float
    """
    sigma = 0.005
    return np.exp(-x**2/(2*sigma))/(np.pi*sigma)**0.25

if __name__ == '__main__':
    w = InfiniteWell(psi0=psi0, width=2, nbase=100, nint=1000)
    x = np.linspace(-0.5*w.width, 0.5*w.width, 500)
    t = np.linspace(0, np.pi/4, 1000)
    z = np.abs(w.psi(x, t))**2
    z = z/np.max(z)
    plt.rc('text', usetex=True)
    plt.imshow(z.T, cmap=cm.hot)
    plt.xlabel('$t$', fontsize=20)
    plt.ylabel('$x$', fontsize=20)
    plt.show()
```

In addition to the docstrings, this code makes sure that the last part is not executed when this script is imported, because in order to access the docstrings, *Sphinx* will import the script. Execution of the last part in our case will simply cost time but in general can have more serious side effects. Note that the script can only be imported, if it is in the search path defined in the *Sphinx* configuration file. This usually requires to uncomment the three lines

```
import os
```

(continues on next page)

(continued from previous page)

```
import sys
sys.path.insert(0, os.path.abspath('.'))
```

and to adjust the argument of `os.path.abspath` according to the place where the script to be imported can be found.

The docstrings are written in reStructuredText and admittedly are not easy to read. We will discuss a solution to this problem shortly. For the moment, however, we will keep this form of the docstrings.

Now let us add the following code in one of our reStructuredText files which are part of the documentation:

```
.. automodule:: carpet
   :members:
   :undoc-members:
```

This code will only function correctly, if the `autodoc` extension is loaded, i.e. the list `extensions` in the *Sphinx* configuration file contains the entry `'sphinx.ext.autodoc'`. The argument `carpet` of the `automodule` directive implies that the file `carpet.py` will be imported. The autogenerated documentation will list all documented as well as all undocumented members. The generated output will look as follows:

> class carpet.InfiniteWell(psi0, width, nbase, nint)
>
> > Quantum carpet for infinitely deep potential well.
> >
> > This class allows to determine the time evolution of an initial state in an infinitely deep potential well.
> >
> > **Parameters:**
> >
> > - **psi0** (*func*) – initial wave function
> > - **width** (*float*) – width of the potential well
> > - **nbase** (*int*) – number of basis states
> > - **nint** (*int*) – number of intervals used in the integration routine
> >
> > eigenfunction(x)
> >
> > > Determine set of eigenfunction values at position *x*.
> > >
> > > The basis set is limited by the number of eigenstates given by `self.nbase`.
> > >
> > > **Parameters:**
> > > **x** (*float or numpy.ndarray*) – position at which the eigenfunctions are to be determined
> > >
> > > **Returns:**
> > > array of eigenfunction values
> > >
> > > **Return type:**
> > > numpy.ndarray
> > >
> > > **Raises:**
> > > `AssertionError` – if the dimension of *x* does not equal 1
> >
> > psi(x, t)
>
> carpet.psi0(x)
>
> > Determine Gaussian wave function.
> >
> > **Parameters:**
> > **x** (*float*) – position at which the wave function is determined
> >
> > **Returns:**
> > value of wave function at position `x`

Return type:
float

carpet.trapezoidal(func, a, b, nint)

If the line containing `:undoc-members:` were left out in the `automodule` directive, the output would contain only the documented class and methods. The listed methods could be restricted by giving the appropriate names after `:members:`.

As already mentioned, the docstrings given above are not particularly easy to read. There are two standards for docstrings which are handled by *Sphinx*, provided the extension `napoleon` is loaded. The list `extensions` in the *Sphinx* configuration file then should contain the string `'sphinx.ext.napoleon'`. The supported standards for docstrings are the Google style docstring[6] and NumPy style docstring[7]. We will focus our discussion of these two standards to the method `eigenfunction` in the quantum carpet script.

Applying the NumPy style to the method `eigenfunction` would result in

```python
def eigenfunction(self, x):
    """Determine set of eigenfunction values at position `x`.

    The basis set is limited by the number of eigenstates given by
    ``self.nbase``.

    Parameters
    ----------
    x : float or numpy.ndarray
        position at which the eigenfunctions are to be determined

    Returns
    -------
    numpy.ndarray
        array of eigenfunction values

    Raises
    ------
    AssertionError
        if the dimension of `x` does not equal 1

    """
```

where we did not repeat the code of the eigenfunction method. This docstring is nicely formatted in sections. The possible sections are not restricted to the ones used in this example. A complete list is given in the documentation of the napoleon preprocessor. The output obtained from this docstring corresponds to the one given before, possibly with minor differences, so that we do not reproduce it here.

The Google style for docstrings resembles the NumPy style in the sectioning of the docstring even though the details of the format differ. Our example would take the following form where we again leave out the code of the method:

```python
def eigenfunction(self, x):
    """Determine set of eigenfunction values at position `x`.

    The basis set is limited by the number of eigenstates given by
    ``self.nbase``.

    Args:
        x (float or numpy.ndarray): Position at which the eigenfunctions
            are to be determined.
```

(continues on next page)

[6] For a complete description see https://google.github.io/styleguide/pyguide.html#functions-and-methods and https://google.github.io/styleguide/pyguide.html#comments-in-classes.

[7] For details see https://numpydoc.readthedocs.io/en/latest/format.html.

(continued from previous page)

```
    Returns:
        numpy.ndarray: Array of eigenfunction values.

    Raises:
        AssertionError: The dimension of `x` does not equal 1.

    """
```

These examples may serve to get the basic idea of how a documentation can be autogenerated from docstrings. When working with the *Sphinx* documentation generator, questions are likely to come up which have not been addressed in this chapter. A complete description can be found in the extensive online documentation which should be consulted in the case of need.

CHAPTER

SEVEN

ASPECTS OF PARALLEL COMPUTING

Today even consumer computers are equipped with multi-core processors which allow to run programs truly in parallel. In numerical calculations, algorithms can often be parallelized so that the execution time can be reduced by running the code on several compute cores. A program could thus profit from the use of several cores within a single processor or from a potentially large number of cores in a computer cluster accommodating a large number of processors.

Parallel execution of a program can result in problems if the individual computations are not well synchronized among each other. The final result might then depend on how fast each of the computations is carried out. Such racing conditions may lead to problems which are sometimes difficult to debug. CPython, the most popular implementation of Python, therefore implements the so-called *Global Interpreter Lock* (GIL) which prevents actual parallel execution within a single Python process. This aspect will be discussed further in the following section.

Despite the GIL, parallel processing is possible in Python if several processes is started. In Section 7.2, we will demonstrate how this can be done by considering the calculation of the Mandelbrot set as an example. This problem is particularly simple because it allows to decompose the full problem into smaller problems without requiring any exchange of data between them. This kind of problems is referred to as *embarrassingly parallel*. Here, we will restrict ourselves to this type of problems. The communication between processes running in parallel raises a number of difficulties which are beyond the scope of the present lecture notes. Readers interested more deeply in this topic might want to read more on the *message passing interface* (MPI) and take a look at the `mpi4py` package.

In the last section of this chapter, we will address the possibilities offered by Numba, a so-called *Just in Time Compiler* (JIT Compiler). The use of Numba can lead to significant improvements of the run time of a program. Furthermore, Numba can support the parallel handling of Python code.

7.1 Threads, processes and the GIL

Modern operating systems can seemingly run several tasks in parallel even on a simple compute core. In practice, this is achieved by in turn allotting compute time to different tasks so that a single task usually cannot block other tasks from execution over a longer period of time.

It is important to distinguish two different kinds of tasks: processes and threads. Processes have at their disposal a reserved range of memory and their own access to other system resources. As a consequence, starting a new process comes with a certain overhead in time. A single process will start first one and subsequently possibly further threads in order to handle different tasks. Threads differ from processes by working on the same range of memory and by accessing the same system resources. Starting a thread is thus less demanding than starting a process.

Since threads share a common range of memory, they can access the same data and easily exchange data among each other. Communication between different threads thus leads to very little overhead. However, the access to common data is not without risks. If one does not take care that reading and writing data by different threads is done in the intended order, it may happen that a thread does not obtain the data it needs. As the occurrence of such mistakes depends on details of which thread executes which tasks at a given time, such problems are not easily reproducible and sometimes quite difficult to identify. There exist techniques to cope with the difficulties involved in the communication between different threads, making multithreading, i.e. the parallel treatment in several threads, possible. We will, however, not cover these techniques in the present lecture notes.

As already mentioned in the introduction, the most popular implementation of Python, CPython implemented in C, makes use of the GIL, the global interpreter lock. The GIL prevents a single Python process to execute more than one

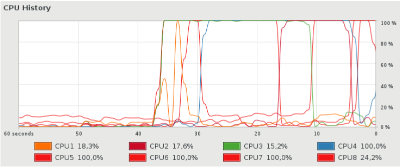


Figure 7.1: In this example, the graph of the system load shows that during the solution of the eigenvalue problem for a large matrix by means of NumPy compiled with the Intel® MKL four cores are used at the same time.

thread in parallel. While it is possible to make use of multithreading in Python[1], the GIL will ensure that individual threads will never run in parallel but in turn are allotted their slots of compute time. In this way, only an illusion of parallel processing is created.

If the time of execution of a Python script is limited by the compute time, multithreading will not result an in improvement. To the contrary, the overhead arising from the necessity to change between different threads will lead to a slow-down of the script. However, there exist scripts which are I/O-bound. An example could be a script processing data which need to be downloaded from the internet. While a thread is waiting for new data to arrive, another thread might use the time to process data already available. For I/O-bounded scripts, multithreading thus may be a good strategy even in Python.

However, numerical programs are usually not I/O-bound but limited by the compute time. Therefore, we will not consider multithreading any further but rather concentrate on multiprocessing, i.e. parallel treatment by means of several Python processes. It is worth mentioning though that multithreading may play a role even in numerical applications written in Python when numerical libraries are used. Such libraries are often based on C code which is not subject to the restrictions imposed by the GIL. Linear algebra routines provided by an appropriately compiled version of NumPy may serve as an example. This includes the NumPy library available through the Anaconda distribution which is compiled with the Intel® Math Kernel Library (MKL). Figure 7.1 demonstrates an example where four cores are used when determining the eigenvectors and eigenvalues of a large matrix. Another option to circumvent the GIL is offered by Cython[2] which allows to generate C extensions from Python code. Those parts of the code not accessing Python objects can then be executed in a `nogil` context outside the control of the GIL.

7.2 Parallel computing in Python

We will illustrate the use of parallel processes in Python by considering a specific example, namely the calculation of the Mandelbrot set. Mathematically, the Mandelbrot set is defined as the set of complex numbers c for which the series generated by the iteration

$$z_{n+1} = z_n^2 + c$$

with the initial value $z_0 = 0$ remains bounded. It is known that the series is not bound once $|z| > 2$ has been reached so that it suffices to perform the iteration until this threshold has been reached. The iterations for different values of c can be performed completely independently of each other so that it is straightforward to distribute different values of c to different processes. The problem is thus *embarrassingly parallel*. Once all individual calculations are finished, it suffices to collect all data and to represent them graphically.

We start out with the following initial version of a Python script to determine the Mandelbrot set.

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

def mandelbrot_iteration(cx, cy, nitermax):
    x = 0
    y = 0
    for n in range(nitermax):
```

(continues on next page)

[1] When referring to Python, we always mean CPython. An example of an implementation of Python without a GIL is Jython written in Java.

[2] Cython should not be confused with CPython, the C implementation of Python. More information on Cython can be found at https://cython.org/.

(continued from previous page)

```
        x2 = x*x
        y2 = y*y
        if x2+y2 > 4:
            return n
        x, y = x2-y2+cx, 2*x*y+cy
    return nitermax

def mandelbrot(xmin, xmax, ymin, ymax, npts, nitermax):
    data = np.empty(shape=(npts, npts), dtype=int)
    dx = (xmax-xmin)/(npts-1)
    dy = (ymax-ymin)/(npts-1)
    for nx in range(npts):
        x = xmin+nx*dx
        for ny in range(npts):
            y = ymin+ny*dy
            data[ny, nx] = mandelbrot_iteration(x, y, nitermax)
    return data

def plot(data):
    plt.imshow(data, extent=(xmin, xmax, ymin, ymax),
               cmap='jet', origin='lower', interpolation='none')
    plt.show()

nitermax = 2000
npts = 1024
xmin = -2
xmax = 1
ymin = -1.5
ymax = 1.5
start = time.time()
data = mandelbrot(xmin, xmax, ymin, ymax, npts, nitermax)
ende = time.time()
print(ende-start)
plot(data)
```

Here, the iteration prescription is carried out in the function `mandelbrot_iteration` up to maximum number of iterations given by `nitermax`. We handle real and imaginary parts separately instead of performing the iteration with complex numbers. It turns out that our choice is slightly faster, but more importantly, this approach can also be employed for the NumPy version which we are going to discuss next.

The purpose of the function `mandelbrot` is to walk through a grid of complex values c and to collect the results in the array `data`. For simple testing purposes, it is useful to graphically represent the results by means of the function `plot`. We also have added code to determine the time spent in the functions `mandelbrot` and `mandelbrot_iteration`. On an i7-6700HQ CPU, we measured an execution time of 81.1 seconds.

Before parallelizing code, it often makes sense to consider other possible improvements. In our case, it is natural to take a look at a version making use of NumPy. Here, we list only the code replacing the functions `mandelbrot` and `mandelbrot_iteration` in our first version.

```
def mandelbrot(xmin, xmax, ymin, ymax, npts, nitermax):
    cy, cx = np.mgrid[ymin:ymax:npts*1j, xmin:xmax:npts*1j]
    x = np.zeros_like(cx)
    y = np.zeros_like(cx)
    data = np.zeros(cx.shape, dtype=int)
    for n in range(nitermax):
        x2 = x*x
        y2 = y*y
```

(continues on next page)

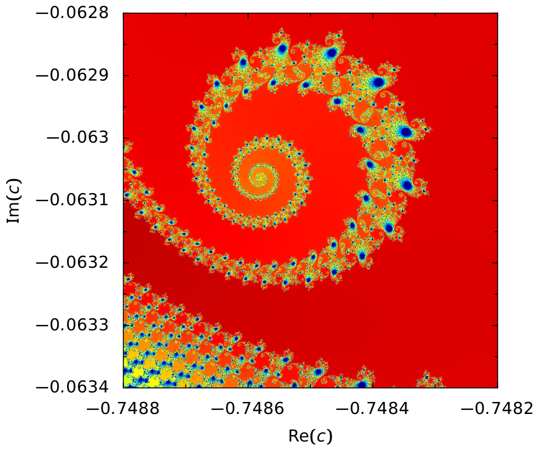


Figure 7.2: Detail of the Mandelbrot set where the color represents the number of iterations needed until the threshold of 2 for the absolute value of z is reached.

(continued from previous page)

```
        notdone = x2+y2 < 4
        data[notdone] = n
        x[notdone], y[notdone] = (x2[notdone]-y2[notdone]+cx[notdone],
                                  2*x[notdone]*y[notdone]+cy[notdone])
    return data
```

For an appealing graphical representation of the Mandelbrot set, we need to keep track of the number of iterations required to reach the threshold for the absolute value of z. We achieve this by fancy indexing with the array `notdone`. An entry of `True` means that the threshold has not been reached yet. An example of the graphical output generated by the NumPy version of the program is shown in Figure 7.2.

For the NumPy version, we have measured an execution time of 22.8s, i.e. almost a factor of 3.6 faster than our initial version.

Now, we will further accelerate the computation by splitting the task into several parts which will be attributed to a number of processes for processing. For this purpose, we will make use of the module `concurrent.futures` available from the Python standard library. The name `concurrent` indicates that several tasks are carried out at the same time while `futures` refers to objects which will provide the desired results at a later time.

For a parallel computation of the Mandelbrot set, we decompose the area in the complex plane covering the relevant values of c into tiles, which will be treated separately by the different processes. Figure 7.3 displays a distribution of 64 tiles on four processes indicated by different colors. Since the processing time for the tiles differs, there is one process which has treated only 15 tiles while another process has treated 17.

The following code demonstrates how the NumPy based version can be adapted to a parallel treatment. Again we concentrate on the Mandelbrot specific parts.

```
from concurrent import futures
from itertools import product
from functools import partial

import numpy as np

def mandelbrot_tile(nitermax, nx, ny, cx, cy):
    x = np.zeros_like(cx)
    y = np.zeros_like(cx)
    data = np.zeros(cx.shape, dtype=int)
    for n in range(nitermax):
        x2 = x*x
        y2 = y*y
```

(continues on next page)

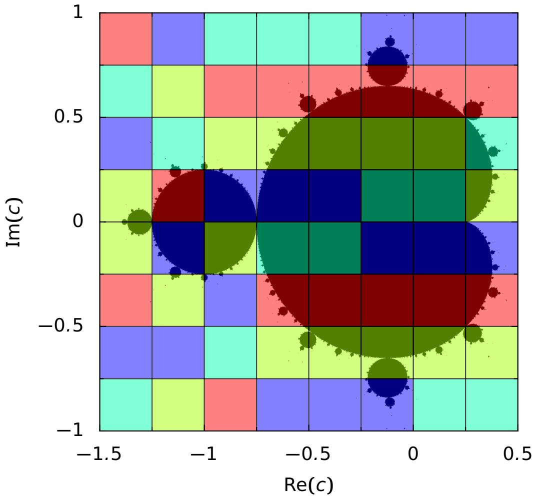


Figure 7.3: The four different colors indicate which one out of four processes has carried out the computation for the corresponding tile. Note that the number of tiles per process does not necessarily equal 16.

(continued from previous page)

```
14            notdone = x2+y2 < 4
15            data[notdone] = n
16            x[notdone], y[notdone] = (x2[notdone]-y2[notdone]+cx[notdone],
17                                      2*x[notdone]*y[notdone]+cy[notdone])
18        return (nx, ny, data)
19
20    def mandelbrot(xmin, xmax, ymin, ymax, npts, nitermax, ndiv, max_workers=4):
21        cy, cx = np.mgrid[ymin:ymax:npts*1j, xmin:xmax:npts*1j]
22        nlen = npts//ndiv
23        paramlist = [(nx, ny,
24                      cx[nx*nlen:(nx+1)*nlen, ny*nlen:(ny+1)*nlen],
25                      cy[nx*nlen:(nx+1)*nlen, ny*nlen:(ny+1)*nlen])
26                     for nx, ny in product(range(ndiv), repeat=2)]
27        with futures.ProcessPoolExecutor(max_workers=max_workers) as executors:
28            wait_for = [executors.submit(partial(mandelbrot_tile, nitermax),
29                                         nx, ny, cx, cy)
30                        for (nx, ny, cx, cy) in paramlist]
31            results = [f.result() for f in futures.as_completed(wait_for)]
32        data = np.zeros(cx.shape, dtype=int)
33        for nx, ny, result in results:
34            data[nx*nlen:(nx+1)*nlen, ny*nlen:(ny+1)*nlen] = result
35        return data
```

The main changes have occurred in the function `mandelbrot`. In addition to the arguments present already in earlier versions, two arguments have been added: `ndiv` and `max_workers`. `ndiv` defines the number of divisions in each dimension of the complex plane. In the example of Figure 7.3, `ndiv` was set to 8, resulting in 64 tiles. The argument `max_workers` defines the maximal number of processes which will run under the control of our script. The choice for this argument will depend on the number of cores available to the script.

In lines 23-26, we define a list of parameters characterizing the individual tiles. Each entry contains the coordinates (`nx`, `ny`) of the tile which will later be needed to collect all data. In addition, the section of the real and imaginary parts of c corresponding to the tile become part of the parameter list. The double loop required in the list comprehension is simplified by making use of the `product` method available from the `itertools` module of the Python standard library imported in line 2.

The main part responsible for the distribution of tasks to the different workers can be found in lines 27-31. This code runs under the control of a context manager which allocates a pool of `max_workers` executors. The method `ProcessPoolExecutor` is available from the `concurrent.futures` module.

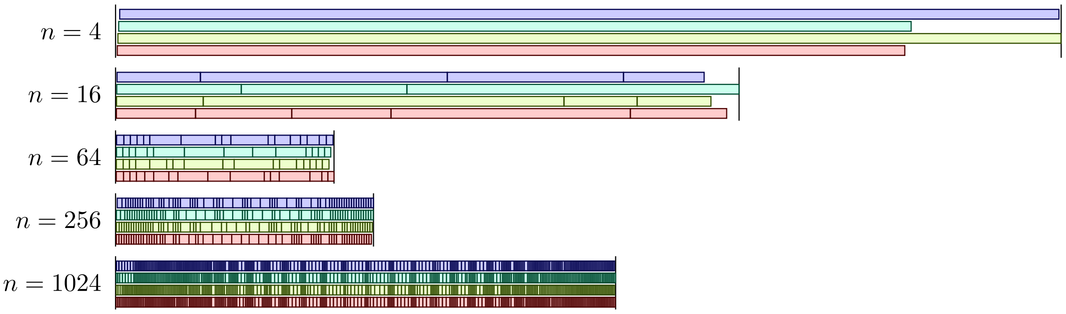


Figure 7.4: Distribution of tasks to determine the Mandelbrot set over four processes as a function of the number of tiles.

In lines 28-30 a list of tasks is submitted to the executors. Each submission consists of a function, in our case `mandelbrot_tile`, and the corresponding parameters. The function `mandelbrot_tile` possesses one argument `nitermax` which is the same for all tasks and the parameters listed in `paramlist` which differ from task to task. Therefore, we construct a partial function object which fixes `nitermax` and requires only `nx`, `ny`, `cx`, and `cy` as arguments. The `partial` method is imported from the `functools` module in line 3.

In line 31, the results are collected in a list comprehension. Once all tasks have been completed, the list `results` contains entries consisting of the coordinates (`nx`, `ny`) of the tile and the corresponding data as defined in line 18. In lines 33-34, the data are brought into order to fill the final array `data` which subsequently can be used to produce graphical output.

It is interesting to study how the total time to determine the Mandelbrot set depends on the number of tiles. The corresponding data are shown in Figure 7.4 for four parallel processes. In the case of four tiles, we see that the different tiles require different times so that we have to wait for the slowest process. For four tiles, where the memory requirement per process is relatively large, we also can see a significant time needed to start a process. Increasing the number of tiles leads to a reduction of the execution time. However, even for 16 tiles, one has to wait for the last process. The optimum for four processes is reached for 64 tiles. Increasing the number of tiles further will lead to an increasing overhead when switching from one task to the next.

Figure 7.5 depicts the acceleration for four processes as a function of the number of divisions per axis. The points connected by the dotted line are obtained by dividing the time required by a single process without subdividing the task through the time required by four processes with the subdivision indicated in the figure. In agreement with Figure 7.4 we find the largest acceleration for 8 divisions per axis, i.e. 64 tiles. Interestingly, the acceleration can reach values slightly exceeding a factor six. This effect may result from a more effective use of caches for smaller problems as compared to the full problem with $n = 1$. The effect of caches can be excluded by taking ratio of the execution times for one and four processes for the same number of tiles. As Figure 7.5 demonstrates, a factor of nearly four is reached beyond $n = 8$.

7.3 Numba

In the previous section we have seen how a program can be accelerated by means of NumPy and parallelization. For our example of the Mandelbrot set, this could be achieved in a rather straightforward manner because the use of arrays came quite naturally and parallelization did not require any communication between the different tasks. Besides the use of NumPy and parallelization of the code, there exist other options to accelerate Python scripts, some of them being very actively developed at present. Therefore, we do not attempt a complete description but rather highlight some ways to accelerate a Python script.

We will specifically discuss Numba[3] because it is designed to work with NumPy and also supports parallelization. Numba makes use of *just in time* (JIT) compilation. While Python scripts usually are interpreted, Numba will produce executable code for a function when it is called first. The compilation step implies a certain investment of time but the function can be executed faster during subsequent calls. Python allows to call functions with different signatures, i.e. the data types of the arguments are not fixed. Compiled code, on the other hand, depends on the signature. Therefore, additional compilation steps may become necessary.

[3] Up-to-date information on Numba can be found at https://numba.pydata.org/.

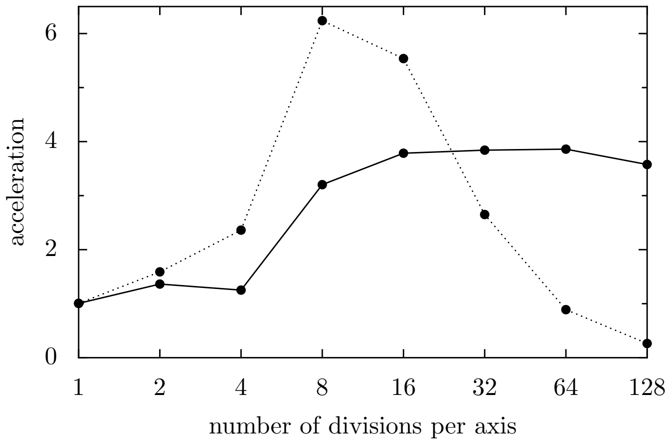


Figure 7.5: Acceleration by parallelization in the computation of the Mandelbrot set with four processes as a function of the number of divisions per axis. The points connected by the dotted line represent the acceleration of the parallelized version with respect to the unparallelized version without subdivision. The points connected by the full line represent the acceleration of the parallelized version with respect to the unparallelized version for the same number of divisions.

We will demonstrate just in time compilation and the effect of different signatures by approximately determining the Riemann zeta function

$$\zeta(s) = \sum_{n=1}^{\infty}\frac{1}{n^s}\,.$$

The following implementation of the code is not particularly well suited to efficiently determine the zeta function but this is not relevant for our discussion. Without using Numba, a direct implementation of the sum looks as follows:

```
def zeta(x, nmax):
    zetasum = 0
    for n in range(1, nmax+1):
        zetasum = zetasum+1/(n**x)
    return zetasum

print(zeta(2, 100000000))
```

We can now simply make use of Numba by importing it in line 1 and adding a decorator `numba.jit` to the function `zeta`:

```
1  import numba
2  
3  @numba.jit
4  def zeta(x, nmax):
5      zetasum = 0
6      for n in range(1, nmax+1):
7          zetasum = zetasum+1/(n**x)
8      return zetasum
9  
10 print(zeta(2, 100000000))
```

Running the two pieces of code, we find an execution time for the first version of 34.1 seconds while the second version takes only 0.85 seconds. After running the code, we can print out the signatures for which the function `zeta` was compiled by Numba:

```
print(zeta.signatures)
```

Because we called the function with two integers as arguments, we obtain not unexpectedly:

```
[(int64, int64)]
```

Like in NumPy and in contrast to Python, integers cannot become arbitrarily large. In our example, they have a length of eight bytes. Accordingly, one has to beware of overflows. For example, if we set `x` to 3, we will encounter a division by zero.

To demonstrate that Numba compiles the function for each signature anew, we call `zeta` with an integer, a float, and a complex number:

```
import time
import numba

@numba.jit
def zeta(x, nmax):
    zetasum = 0
    for n in range(1, nmax+1):
        zetasum = zetasum+1/(n**x)
    return zetasum

nmax = 100000000
for x in (2, 2.5, 2+1j):
    start = time.time()
    print(f'ζ({x}) = {zeta(x, nmax)}')
    print(f'execution time: {time.time()-start:5.2f}s\n')

print(zeta.signatures)
```

The resulting output demonstrates that the execution time depends on the type of variable `x`` and that Numba has indeed compiled the function for three different signatures:

```
ζ(2) = 1.644934057834575
execution time:  0.59s

ζ(2.5) = 1.341487257103954
execution time:  5.52s

ζ((2+1j)) = (1.1503556987382961-0.43753086346605924j)
execution time: 13.41s

[(int64, int64), (float64, int64), (complex128, int64)]
```

Numba also allows us to transform functions into universal functions or *ufuncs* which we have introduced in Section 4.2.6. Besides scalar arguments, universal functions are capable of handling array arguments. This is achieved already by using the decorator `jit`. By means of the decorator `vectorize`, the evaluation of the function with an array argument can even by performed in several threads in parallel.

In the following code example, we specify the signature for which the function `zeta` should be compiled as argument of the decorator `vectorize`. The argument `x` is a `float64` and can also be a corresponding array while `n` is an `int64`. The result is again a `float64` and is listed as first argument before the pair of parentheses enclosing the arguments' data type. The argument `target` is given the value `'parallel'` so that in the case of an array argument the use of several threads is possible. If a parallel processing is not desired, for example because for a small task starting a thread would cost too much time, one can set `target='cpu'` instead. If an appropriate graphics processor is available, one might consider setting `target='cuda'`.

```
import numpy as np
from numba import vectorize, float64, int64

@vectorize([float64(float64, int64)], target='parallel')
def zeta(x, nmax):
```

(continues on next page)

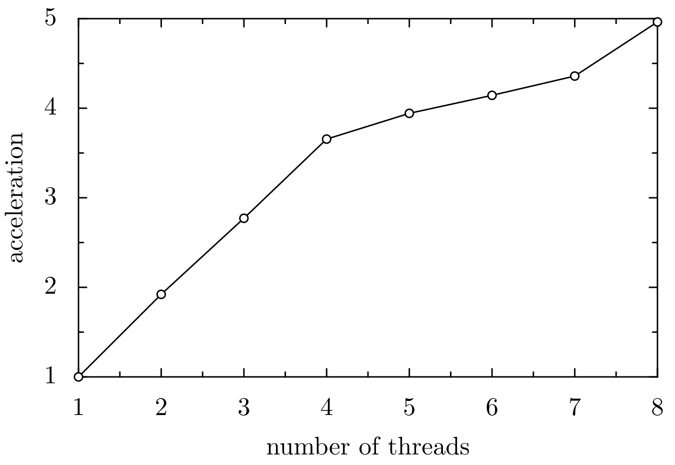


Figure 7.6: Acceleration of the computation of the Riemann zeta function as a function of the number of threads on a CPU with four cores and hyperthreading.

(continued from previous page)

```
    zetasum = 0.
    for n in range(nmax):
        zetasum = zetasum+1./((n+1)**x)
    return zetasum

x = np.linspace(2, 10, 200, dtype=np.float64)
y = zeta(x, 10000000)
```

Figure 7.6 shows how the execution time for the Riemann zeta function can be reduced by using more than one thread. The number of threads can be set by means of an environment variable. The following command sets the number of threads to four:

```
$ export NUMBA_NUM_THREADS=4; python zeta.py
```

The timing in Figure 7.6 was done on an i7-6700HQ processor with four cores and hyperthreading which allows to run eight threads in parallel. Up to four threads, the execution time decrease almost inversely proportional to the number of threads. Increasing the number of threads beyond the number of cores will further accelerate the execution but by a much smaller amount. The reason is that threads need to wait for free resources more often.

With Numba, universal functions can be further generalized by means of the decorator `guvectorize` so that not only scalars but also arrays can be employed in the inner loop. We will illustrate this by applying Numba to our Mandelbrot example.

```
from numba import jit, guvectorize, complex128, int64
import matplotlib.pyplot as plt
import numpy as np

@jit
def mandelbrot_iteration(c, maxiter):
    z = 0
    for n in range(maxiter):
        z = z**2+c
        if z.real*z.real+z.imag*z.imag > 4:
            return n
    return maxiter

@guvectorize([(complex128[:], int64[:], int64[:])], '(n), () -> (n)',
             target='parallel')
```

(continues on next page)

(continued from previous page)

```
16 def mandelbrot(c, itermax, output):
17     nitermax = itermax[0]
18     for i in range(c.shape[0]):
19         output[i] = mandelbrot_iteration(c[i], nitermax)
20
21 def mandelbrot_set(xmin, xmax, ymin, ymax, npts, nitermax):
22     cy, cx = np.ogrid[ymin:ymax:npts*1j, xmin:xmax:npts*1j]
23     c = cx+cy*1j
24     return mandelbrot(c, nitermax)
25
26 def plot(data, xmin, xmax, ymin, ymax):
27     plt.imshow(data, extent=(xmin, xmax, ymin, ymax),
28                cmap='jet', origin='lower', interpolation='none')
29     plt.show()
30
31 nitermax = 2000
32 npts = 1024
33 xmin = -2
34 xmax = 1
35 ymin = -1.5
36 ymax = 1.5
37 data = mandelbrot_set(xmin, xmax, ymin, ymax, npts, nitermax)
```

Let us take a closer look at the function `mandelbrot` decorated by `guvectorize` which has a special set of arguments. The function `mandelbrot` possesses three arguments. However, only two of them are intended as input: `c` and `itermax`. The third argument `output` will contain the data returned by the function. This can be inferred from the second argument of the decorator, the so-called layout. The present layout indicates that the returned array `output` has the same shape as the input array `c`. Because `c` is a two-dimensional array, the argument `c[i]` of the function `mandelbrot_iteration` is again an array which can be handled by several threads. While `maxiter` in the function `mandelbrot_iteration` has to be a scalar, the array `itermax` is converted in line 17 into a scalar.

On the same processor on which we timed earlier version of the Mandelbrot program and which through hyperthreads supports up to eight threads, we find an execution time of 0.56 seconds. Compared to our fastest parallelized program, we thus observe an acceleration by more than a factor of six and compared to our very first version the present version is faster by a factor of almost 150.

CHAPTER
EIGHT

APPENDIX

8.1 Decorators

In this appendix, we give a short introduction to decorators so that we have an idea of what they are about when making use of them. Decorators are a way to modify the behavior of functions, methods, or classes. We will restrict our discussion to functions. The effect of a decorator is then to replace the function by a modified version of the original function. Some examples will demonstrate how this works.

For the first example we assume that our script defines a couple of functions which we would like to register. To keep things simple, registering a function shall simply mean that an appropriate message is printed.

```
1  def register(func):
2      print(f'{func.__name__} registered')
3      return func
4
5  @register
6  def myfunc():
7      print('executing myfunc')
8
9  @register
10 def myotherfunc():
11     print('executing myotherfunc')
12
13 print('-'*40)
14 myfunc()
15 myotherfunc()
```

In lines 1-3 we define a decorator called `register` which is then applied in lines 5 and 9 to two functions `myfunc` and `myotherfunc`. Running the script produces the following output:

```
myfunc registered
myotherfunc registered
----------------------------------------
executing myfunc
executing myotherfunc
```

What has happened? Before running the code in lines 13-15, Python has defined the functions in the lines 1-11. When getting to `myfunc`, the decorator `register` will come into action. Its argument `func` will be the function `myfunc`, i.e. the argument of a decorator is implicitly given by the function following the decorator statement. Then, `register` is executed, first printing a message which contains the name of the decorated function. Then it returns the function `myfunc` unmodified. The effect of the decorator thus is simply to print a message that the function `myfunc` has been registered. As the output reproduced above shows, this is only done once, namely when Python processes the function code. Later, the function is executed which was returned by the decorator. In our case, this is simply the original function.

The decorator can also be used to modify the function so that a desired effect occurs each time the function is executed. In the following example, we define a somewhat more complex decorator named `logging` which prints a message when the function is starting execution and another message indicating the time of execution just before the function finishes execution. The interest of using the `logging` decorator is to analyse how the execution of a recursive function works.

```
1  import time
2  from itertools import chain
3  
4  def logging(func):
5      def func_with_log(*args, **kwargs):
6          arguments = ', '.join(map(repr, chain(args, kwargs.items())))
7          print(f'calling {func.__name__}({arguments})')
8          start = time.time()
9          result = func(*args, **kwargs)
10         elapsed = time.time()-start
11         print(f'got {func.__name__}({arguments}) = {result} '
12               f'in {elapsed*1000:5.3f} ms')
13         return result
14     return func_with_log
15 
16 @logging
17 def factorial(n):
18     if n == 1:
19         return 1
20     else:
21         return n*factorial(n-1)
22 
23 factorial(5)
```

The main difference to our first example consists in the fact that the decorator in lines 5-14 defines a new function. As a consequence, the modifications apply whenever the decorated function is run. Then new function `func_with_log` is written in a rather general way to allow its use for arbitrary functions. In particular, the decorated function can take an arbitrary number of arguments including keyword arguments. Whenever the decorated function is executed, it will print a message including the arguments with which the function was called. In addition, the starting time is stored. Then, the original function, in our case `factorial`, is run and when it returns, the elapsed time is determined. Before quitting, the result together with the elapsed time are printed.

Running the script, we obtain the following output:

```
calling factorial(5)
calling factorial(4)
calling factorial(3)
calling factorial(2)
calling factorial(1)
got factorial(1) = 1 in 0.001 ms
got factorial(2) = 2 in 0.042 ms
got factorial(3) = 6 in 0.069 ms
got factorial(4) = 24 in 0.094 ms
got factorial(5) = 120 in 0.127 ms
```

It nicely demonstrates how the function `factorial` is called recursively until the recursion comes to an end when the argument equals 1.

A decorator could even go as far as not running the decorated function at all and possibly returning a result nevertheless. A situation where such a decorator could make sense is during testing. Suppose that we want to test a program which relies on obtaining data from a measuring device. If we are not interested in testing the connection to the device but only how received data are handled, an appropriate decorator would allows us to test the program even without connection to the measuring device as long as the decorator provides us with appropriate data.

INDEX