0423
Software development for scientists

Félix Bertoni
contact@felix-bertoni.fr

26.10.2020
(gen. 11.01.2021)
version 1.0

Abstract

Software development can take an important place in a scientist's life, would it be to display data, perform simulations and calculations or develop tools for other scientists. Not all scientists had the opportunity to learn software development, and some may find themselves having insufficient knowledge and skills regarding their needs. Software development field is wide, and scientists don't forcibly have time to explore it by themselves in order to find knowledge they need.

This course aims to provide basic software development knowledge scientists may need, as well as an overview of software development field and strong basis to help them find more advanced knowledge in case they need it.

It starts by an introduction to software development, with code edition and development tools, as well as some advices on both conducting a software project and sharing it. Then it explores programming field, with an introduction to object-oriented programming, architecture, algorithmics, and performance concerns. After programming comes code quality, discussing the best way to make easily readable and understandable code, which is useful for reproducibility in scientific context. Finally, it presents testing concepts and approaches, as well as Test Driven Development method, meant to ensure reliability of software.

Acknowledgement

I want to address my deep thanks to all benevolent people and institutions that supported me while writing this course, and even more to those who gave some of their time or money to help its development. Contributors list can be found at 5.7.6, Contributors, as well as financial contributors list at 5.7.6, Funders. In the process of writing this course, a lot of free software were used, I think it is important to mention them. A non-exhaustive list of them can be found at 5.7.6,5.7.6. I want to address a special *thank you !* to Edward Andó, for pointing out the need for this course as well as his help in early thinking and design.

Contents

List of Figures

List of Tables

Listings

Chapter 1

About this course

1.1 Goals and prerequisites

This course's main target audience is scientists with a beginner level in the software development field who want to improve their programming and development skills, while not spending too much time on internet research and learning. This course, trying to be as concise as possible, carries on two goals. The first goal is to teach a set of software development knowledge and skills sufficient for everyday software development, as for example creating short programs for data analysis or simulation. The second objective is to provide sufficient fundamental knowledge, as well as an overview of software development field, for scientists to be able to get easily into larger development projects, and eventually gain specialized knowledge in case they need it.

This course assumes you are fairly familiar with Linux operating system. While it is not required to have prior software development and programming knowledge to take on this course, having extremely basic knowledge of programming, as knowing more or less what a *variable* and a *function* are, is better. In case you don't know about Linux nor programming, reading and understanding this course is still possible, however manipulations done as examples will be harder to carry on.

This course also assumes that scientists have specific needs regarding software they develop, that are different from what typical software developer would have. Therefore, it focuses mainly on three aspects of software development.

- Readability. For reproducibility and criticism of results, science software need to be easy to study.
- Reliability. A software used for science has to produce reliable results, to avoid additional effort due to errors.
- Efficient code and learning. A scientist may want to focus on its field and research rather than developing software and learning to develop them.

1.2 Reading guidelines

Remember that the best way to learn most of the knowledge in this course, as well as any software development related knowledge and skills, is to **practice**. It is advised to do all manipulations that are presented as examples in this course as practical work. Also, don't be afraid to explore while manipulating. Try changing things and see how they work.

This course contains some vulgarization. A lot of software development aspects will only seen briefly, to inform you of their existence in case you need to learn about them someday. A lot of software development practices are still discussed as for today. References are used to help you get deeper knowledge as well as contradictory point of view when it exists.

Larges examples that are in appendices of this course may have been given to you in plain code with this course. If not, you can get them on the GitLab page[1] of this course. If you can't reach the pages, please email the author[2] for help.

Whenever the narration is in first person, *I* ..., it likely refers to the author of the course, as for example for him to expose his point of view.

In case you have difficulties with a point in this course, you can eventually contact me for asking questions. I can't guarantee I'll be able to answer, but I'll do as much as I can for.

1.3 Known limits and incoming improvements

The current version of this course has been written in a fairly restricted time span and therefore there are known flaws that have not been corrected yet, but will be in a later iteration. This course it still sufficient to carry on initial goals, it simply could be better.

This course refers to abstract individuals, as for example *a developer*, as males. It has been done for saving some writing efforts and it will be improved in a later version of this course, either using neutral pronouns, an alternate of differently gendered pronouns or full inclusive style.

Development environment and Programming are as for now two separated chapters, leading Development environment to avoid referring to certain notions of programming since Programming chapter introduces them afterward. In a later version, those two chapters will be somehow merged into a continuous flow, as for example *Development basics* and *Advanced programming.*

This course lacks some additional chapters that will be added later. One is a larger practical example and the other is an *additional knowledge*, chapter, covering worth to mention points that have not been covered in the main body of the course, as for example ergonomics.

This course also lacks introduction to functional and other declarative programming, only discussing imperative programming. While only the Programming chapter is concerned, it may be better to show both paradigms in a later iteration, and it could be interesting for anyone reading this course to inform himself on what functional programming is. Even if you are a functional or declarative programmer, most concepts and practices shown in this course may be useful to you.

In a later iteration, this course will eventually be restructured with a project based approach, driven by a fully usable software project. Finding a project both small, interesting, entertaining, and suitable for any scientists from any field is a difficult matter. Suggestions are much welcome. Such approach would allow introducing notions in a much smoother way and provide a concrete example as well.

1.4 Licensing and usage

This course is licensed under Creative Commons CC BY-SA license[3]. Roughly said, you can reuse this course for both non-profit and commercial purpose, at the condition of citing author of the present course and publishing any derivative work under the same license. All code examples in this course are licensed under GNU-GPLv3 license[4].

This course should be available for free on the internet. If you liked it and want to support the work of author and contributors, you can :

- Leave a star on the GitLab[5] repository holding this course, it helps motivation.
- Contact author to tell you used the course and why, especially if you used it as a support for teaching.
- Share this course to your colleagues and friends if you think they may be interested.

[1] `https://courses.felix-bertoni.fr/0423.html`
[2] contact@felix-bertoni.fr
[3] `https://creativecommons.org/licenses/by-sa/4.0/`
[4] `https://www.gnu.org/licenses/licenses.html#GPL`
[5] `https://gitlab.com/feloxyde/courses/`

- Contribute to the course, see next section for more information.

Course services can be various, as requiring a full supervised course for several students or simply support to one student.

1.5 Error reporting and contribution

In case you find a major error in this course, as an incorrect statement, if you have suggestions for improvements, or want to contribute this course, please contact author.

You can also contribute to the course with financial support, by ordering a paid teaching service from the author. Services can be various, as for example supervised course or individual student support. They aren't forcibly limited to the scope of this course, please contact the author for more information.

Contributor list can be found at 5.7.6, Contributors, as well as a list of financial contributors at 5.7.6, Funders.

Chapter 2

Development environment

In this chapter, we will explore some notions of software development and tools we can use to enhance performance of developers. Software development includes programming as well as project management, goals definition and a lot of more. Programming is the actual software creation. If a software was a house, development would stretch from expressing the initial need to selling the house to someone. Programming would only be the actual building phase of the house. Please note that those two terms are often used one in place of the other.

2.1 About programming

2.1.1 Software, library, framework ?

In a software development context, we usually differentiate three entities that one can code.

A software is a set of instructions that a computer can understand and that is aimed to fulfill a specific need for a user. We often use software as a shortcut for *application software*. An application is a software meant for a user to carry a specific task, as for example image manipulation. A simulation script can be considered as an application as well. Figure 2.1 shows and example of interactions between a user and an application software.

When creating software, a programmer usually uses really common and abstracts treatments. For example, for a physics simulation, you may need vector operations or a numeric solver. Libraries[1] are a way for a developer to provide other developers with tool to develop software of a common field, as figure 2.2 shows. It is used to not reinvent the wheel too often. If a software was a house, libraries could provide bricks and cement.

A framework[2] is a special kind of library, or sometimes an aggregate of several libraries. The difference between a framework and a library is how they influence the structure of the code using them. A library can be seen as a collection of tools that the programmer can use as he wants. A framework is more like a pre-made software that the developer will enhance and configure to build an actual software, see figure 2.3. A physics simulation framework would feature an already working simulation in which we can add new kind of physics objects, or alter physics rules as for example by adding new constraints. If a software was a house, a framework would be a

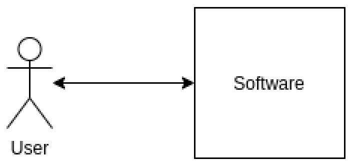


Figure 2.1: An application software and its user

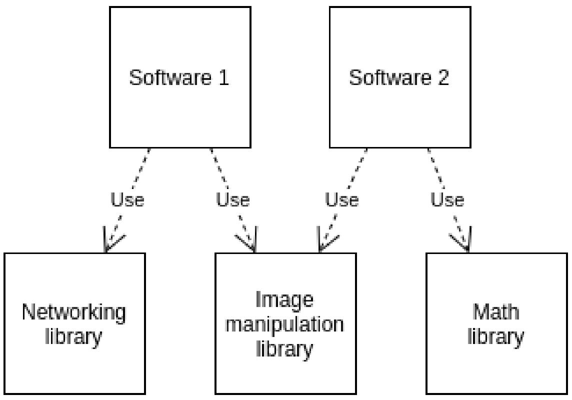


Figure 2.2: Libraries, bricks to build software

prefabricated house in which you can add window by yourself and for which you can change walls materials.

Libraries can be extremely simple to develop because they often don't require much coherence in the way the developer interacts with them, allowing to create all elements separately. Software is a bit harder to conceive, since they require some kind of coherence. However, as software have a fixed and clear goal, being coherent isn't difficult. Frameworks, on another hand, are really tricky to design, as they require both coherence and flexibility : you have to provide the developer as much as pre-made stuff as possible while allowing extension and modification of the behavior of the framework. Of course, this estimation isn't absolute, and some libraries or software can be as difficult as frameworks to implement and vice versa.

In the rest of the course, except if explicitly said, *software* and *program* will refer to both software, framework and library.

2.1.2 Language

Programming is done in most cases by writing *code*, also called *source code* or *sources*. Code is text written according a certain set of rules and describing what the computer shall do. Rules and meaning of code is expressed through a *programming language*, defining a syntax and a semantic[3]. Syntax defines what constructs are valid in the language. For example, we would imagine a language describing simple arithmetic operations as `number op number`, where `op` is either `+`, `-`, `*` or `/`. Then `10 + 10` would be valid, while `chat + 10` and `10 ++ 10` would not. Semantic gives each valid construct a meaning. As an example, we would decide that `20 + 70` means *add 20 and 70*.

There exists hundreds of programming languages, each having unique syntax and semantic. Usually, languages are designed with a goal in mind, and follow several *paradigms*[4], sort of philosophies on how to express what the computer should do. Some languages are much more specialized than other, as for example *Javascript* is meant mostly for web applications. We'll discuss further about languages in the next chapter, 3 Programming.

2.1.3 Compiler and interpreter

Once the code is written, we want to have a computer execute the instructions described.

There are two archetypes of a program execution[5]. The first one is to use a *compiler*. A compiler will translate our source code into *machine code*, forming an executable that can be executed straight by the processor, or *CPU* for *Central Processing Unit*. Compiled software tend to have high performance regarding execution speed and resource consumption, however, executables have a low portability. Production of machine code is dependent on hardware and operating system, and thus an executable made for a certain system is unlikely to be able to run

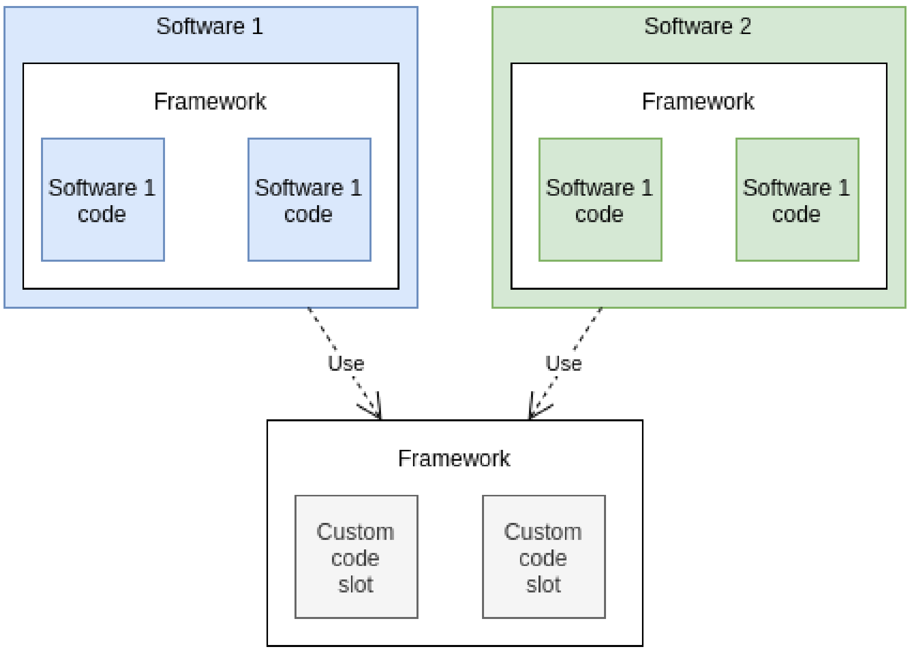


Figure 2.3: Frameworks, foundations for software

on other systems. If we want to run our software on different systems, we have to compile it for each system separately. In order to overcome this problem, we can use an *interpreter*. An interpreter is a software, usually compiled, that reads code of a certain language and executes its instructions. It acts as a medium between the code and the processor to help portability : as long as the interpreter can run on a system, then our software will run on it as well. As the interpreter acts as a medium between software and processor, interpreted language usually have lower performance regarding resource consumption and execution speed. Those two modalities are shown figure 2.4. Please note that languages are not compiler nor interpreted, as their use is simply to express what the computer should do. However, as languages are often associated with either compilation or interpretation, they are called either compiled or interpreted. Therefore, we will consider *C++* to be compiled language and *Python* to be interpreted, even if it is possible to interpret C++ code[6] or compile Python code[7]. This shortcut is acceptable, since it is much simpler to interpret Python and compile C++ than the opposite.

Some languages, as *Java* and *Golang* try to get the better of both compiler and interpretation, producing highly portable and performant executables. Java is both a compiled and interpreted language, as it is compiled in an intermediate form in between processor instructions and code called *bytecode*, and then run on a virtual machine offering much more performance than straightforward interpreters. Golang is compiled but allows to ship a lot of resources alongside the executables it produces, making them more portable than a majority of other executable compiled with from other languages.

Interpreted language's performances are improving drastically and the performance gap between compilation and interpretation is tight now. Some languages even provide mechanisms to allow enhancing critical parts of the software. One of them is *just in time* compilation, abbreviated *JIT*, where the interpreter will actually compile some parts of the program on the fly if it considers them to be critical. Another is to allow calling compiled libraries from compiled code, often written in *C*, to have the developer handle critical parts in a high performance language.

Interpreted languages offer also a lot more flexibility when it comes to allow user to extend the program, but this will not be discussed in this course as it is advanced feature. We will come back a bit on interpreted and compiled notions during chapter 3. Please note that in the rest of this course, we will not make any difference between compiler and interpreter, and use both words for

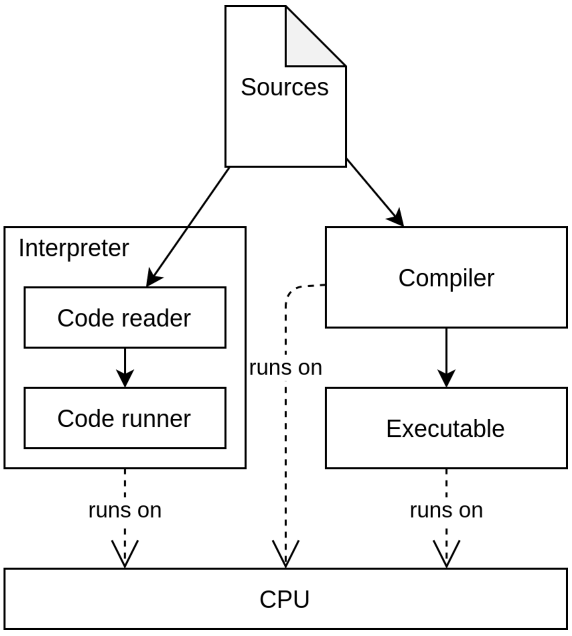


Figure 2.4: Compilation or interpretation

both meaning.

2.2 Programming tools

2.2.1 Editor

Source code is usually written in a plain text format, and therefore can be edited with any text editor. However, some editors specialize in code edition. They offer advanced features for coding. There are a lot of powerful open source code editors, as *Vim*[8], *Emacs*[9], or *VSCode*[10][11][12]. Those editors are highly customizable and offer a lot of possibilities to increase the speed at which you write code.

Syntax highlighting colors keywords, names and other elements according to their role.

Autocompletion provides you with suggestions when you start writing a word, demonstrated figure 2.5. This is extremely useful since in programming you need to have names being written always the same way : with autocompletion, you don't need to remember everything perfectly. It also saves you a lot of time when writing long names by allowing you to write the first few characters and simply select the right choice. Finally, it can expose you what you can write in a certain context.

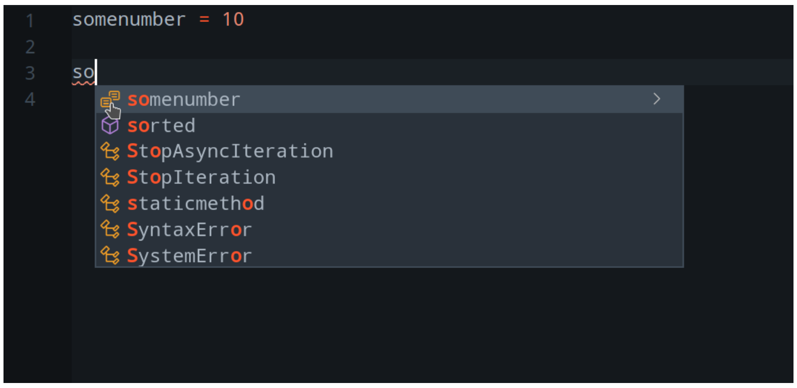
Auto-formatting will rearrange your code to have it look nicer. It will be discussed in section 4.3.1. It helps to keep code homogenous and readable with low efforts.

Code snippet management allows writing complicated or widely used code constructs with only a few keystrokes. It can be triggered either with shortcuts or alongside autocompletion, as in figure 2.6. A well configured code snippet manager can drastically increase the speed at which you write code. It is especially convenient when coding in a language you don't often use by acting as a syntax reminder.

Line wrapping allows to limit the width of a line of code on screen to a certain number of characters, while still writing on a single line in the file.

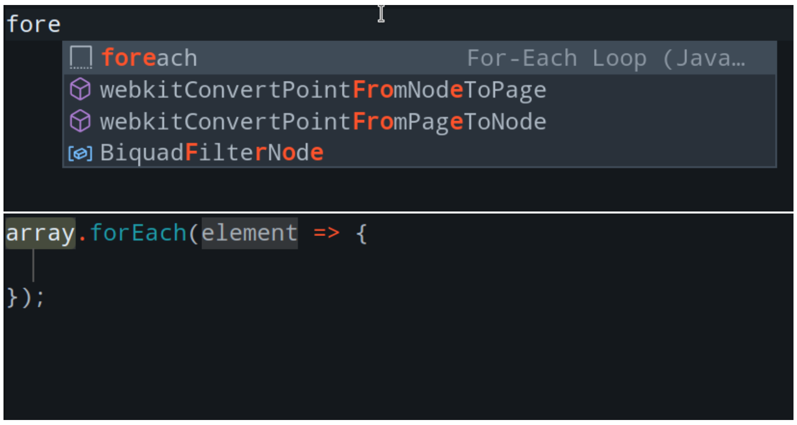
Block collapsing, or code collapsing, hides certain parts of the code and display only a one line summary, making it easier to read and explore. It is displayed figure 2.7.

Code navigation helps to find all mention of a certain names through multiple source files.



```
somenumber = 10

so
```

Figure 2.5: Autocompletion prompt

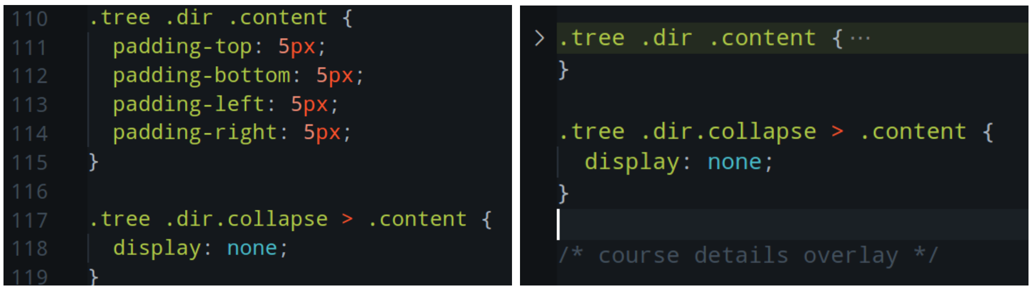


```
fore
```

foreach For-Each Loop (Java…
webkitConvertPointFromNodeToPage
webkitConvertPointFromPageToNode
BiquadFilterNode

```
array.forEach(element => {

});
```

Figure 2.6: Code snippet



```
.tree .dir .content {
  padding-top: 5px;
  padding-bottom: 5px;
  padding-left: 5px;
  padding-right: 5px;
}

.tree .dir.collapse > .content {
  display: none;
}
```

```
> .tree .dir .content {···
}

.tree .dir.collapse > .content {
  display: none;
}

/* course details overlay */
```

Figure 2.7: Code or block collapsing

Code refactoring allows you to change names of entities according to the structure of the code. It can be seen as an advanced *find and replace* feature.

Code editors pack in a lot more features, as file tree or integrated terminal. One really appreciable feature are *workspaces* and configurations alike. A workspace retains the configuration of your work environment, allowing you to quickly switch from a project to another and to define configuration local to a project.

2.2.2 Analyzers and debuggers

Getting information on a program, either during execution or by looking at code, can be extremely difficult for humans. A program of few lines of code can translate into thousands of machine instructions at runtime and manipulate gigabytes of data. Fortunately, there are software to assist us in this complicated task.

A *debugger* is a tool to execute a program step by step, as for example by stopping at each line of code, to observe the state of the program and its data, or even to change values on the fly during execution. It allows a developer to explore what happens in the program during runtime, and is really convenient for fixing bugs as it names implies. There exists at least one debugger for each popular languages, as *Javascript*, *C*, *C++*, or *Python*.

There are also automated tools that can gather data about your software. *Static analyzers* act by looking at your code without running it. They can for example detect security issues, bad practices, badly presented code, and suggest you changes. A good example of such tool are *linters*, discussed later in section 4.3.1, who ensures your code respects certain coding rules in addition to rules set by the language. *Runtime analyzers* observe the software at runtime, sometime using a modified version of it to improve observation abilities. For example, *profilers* will report information about performance, either speed or memory consumption, and *coverage tools*, covered in section 5.5.4, tells you which parts of the code ran during test.

2.2.3 Automation

Programming and software development both involve working with fairly complicated workflows. For example, in order to compile a program written in C, we usually need to compile each C file separately into *object files* and then link them together to form a software. The command line to compile a C file looks like `gcc -c file -o object --some-flag --some-other-flag --more-flag`. A C software can easily be composed of tens of files. It makes it nearly impossible to compile them by hand each time we want to run the program, because trust me, we want to run it often. Such problems are common in programming. Fortunately, there are a lot of *automation* tools available for us to never bother too much about that. They are a common practice and are both easy to use and widely used.

The first automation tools you can think of are *bash scripts.* This course assume you already know a bit about them. We will not discuss them further right now, as they are not the best suited tool for your everyday programming tasks.

Automation tools oriented on programming are often referred to as *build automation tools*, as they are designed to ease building and installing software. A really common build automation tool is *Make*[13]. Make allows running commands in a certain order by defining dependencies between commands. The `make` command will look in the current directory for a file named `Makefile` or `makefile` containing rules. A rule is a named set of commands with eventually dependencies, and is expressed as follows.

Listing 2.1: Make rule syntax

```
rulename : dependencyname1 dependencyname2 ...
    command1
    command2
    command3
    ...
```

The name of the rule is called a *target*. You can supply a target to the make command to build this specific target and its dependencies only. Let's have a quick example by manipulating text files using make. Let's say we have a directory containing two files, `file1.txt` and `file2.txt`, as well as a bash script, `create_file.sh`, which creates one more file, `created_file.txt`, when invoked. We want to be able to perform three actions. First is calling the script. Second is copying all files, including the one created by the script, in a newly made directory. Last is simply getting back to initial condition. We can write a tiny makefile for it.

```
call :
    ./create_file.sh

copy : call
    mkdir newdir
    cp file1.txt file2.txt created_file.txt newdir

clean :
    - rm newdir
    - rm created_file.txt
```

A - before a command tells make to not report if the command fails. `make call` will call the script. `make copy` will call the script if it hasn't already been done and then create `newdir` directory and copy files into it. `make clean` will get back into original state by deleting `newdir` and `created_file.` ↪ `txt`. Please note that invoking make command without any argument will select a default target, which is usually the first target declared in the file. In our case `make` is alike `make call`.

Make is capable of much more. Keep in mind that this quick presentation was an inaccurate vulgarization, and that it is highly advised to learn more about Make or other generalist automation tools as they are very useful, even outside pure programming projects.

Writing efficient makefiles can be fairly difficult when the project reaches large scale, or when a high portability is needed. In this case, using specialized tools may be better. Specialized build automation tools are usually tied to a specific language. As an example, *CMake* is specialized in the C family languages as C and C++. CMake actually generates makefiles, so it isn't a build automation tool strictly speaking. Yet, for the developer it makes no difference. When using multiple languages, one can use specialized automation tools for each and synchronize them using a generalist tool as Make.

Automation is a key factor to be efficient. Taking the time to set a fully automated development environment is extremely important and will save you a lot of time in the future. This matter will be discussed a bit more in chapter 5, Testing.

2.2.4 Version control

A version control software, also referred to as *source code manager* or *revision control*, is meant to help us developers manage the progression of our software. They allow tracing of all changes made in the software in form of a history, getting back on an older state, and coordinate the work of multiple people working on the same sources. There are plenty of version control software, as *Subversion*[14], *Git*[15], or *Mercurial*[16]. *In my opinion, working on any software development project, even the tiniest ones, without using version control is foolish. I won't simply recommend you to use version control, as I would recommend using a good editor : version control is primordial.* We will use Git to introduce version control, as it is widely used and extremely flexible.

Git usually works with a server storing files online. This way we can access and share them anytime. Having a server can also give more guarantees against failures than a local computer. Git works with local directory when it comes to editing code, and can also work perfectly fine without server. We will use this property to demonstrate its use. A project under Git is called a *repository*. Creating a repository is extremely simple. Just create a new directory, change into it and call `git init` command. You could also do that in an already created, non-empty, directory.

Listing 2.2: Creating a new git repository

```
$ mkdir git_test
$ cd git_test
$ git init
Initialized empty Git repository in /home/felix/git_test/.git/
$ ls -la
drwxr-xr-x  3 felix felix 4096 24 sept. 13:11 .
drwx------ 78 felix felix 4096 24 sept. 13:12 ..
drwxr-xr-x  7 felix felix 4096 24 sept. 13:11 .git
```

We can see that it created a new hidden directory, `.git`, at the root of our repository. This directory is used by git to store information. Let's check the *status* of our repository to see if there are changes made, using `git status` command. Then let's create a file to see status change.

Listing 2.3: Git status

```
$ git status
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
$ touch file1.txt
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        file1.txt

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

We have now an untracked file. We can add it to the history by making a *commit*. A commit is simply a change in the repository. We first need to declare files we want to add to the commit using `git add` command. `git add -u` adds all files git already knows of, or *tracked files*, that had an update. `git add -A` adds all files, both tracked and untracked, that had an update. `git add filename` adds the file named `filename`. Please note that `git add -A` and `git add .` commands shall be used with care, as they are adding to git new files and can therefore introduce files you didn't want, as for example build files.

Listing 2.4: Git add

```
$ git add -A
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   file1.txt
```

Then we can add changes to history with `git commit` command. Without any argument, this command will open a text editor for you to write a commit message. You can directly supply a message using `git commit -m "message"` command. In a commit, try to only focus on a single subject. Add only relevant files and write precise commit messages.

Listing 2.5: Git commit

```
$ git commit -m "added file1.txt"
[master (root-commit) d92e102] added file1.txt
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 file1.txt
```

We can now consult history with command `git log`.

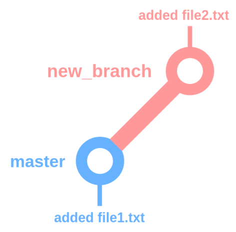


Figure 2.8: Git history after branching

Listing 2.6: Git log result

```
commit d92e102e179e12857cc51195bf9b21c830601e7b (HEAD -> master)
Author: Felix Bertoni <felix.bertoni987@gmail.com>
Date:   Thu Sep 24 13:30:34 2020 +0200

    added file1.txt
(END)
```

We can press `q` to exit log display. Our git commit has been registered and its message is also displayed.

You may have noticed that `branch master` or `master` is often mentioned in the result of previous git commands. This is because master is the default name of the main *branch* in Git. For simplicity, we will continue to call it master in this course, however, this term is a bit controversial[17][18][19][20], and for your projects it may be better to rename the branch as *main* or *stable*.

But wait, what is a branch ? We usually imagine a history as being a timeline. Git sees it as a graph, with multiple histories running on parallel, that can be joined together later on, or become completely independent. It can be used, for example, to maintain two versions of a project at the same time : one *stable*, guaranteed to work, and one *experimental*, where new changes are introduced and tested. Let's try branches on our example project.

We first need to create a new branch using `git checkout -b branchname` command. It will make a new history starting at the current repository state we are in.

Listing 2.7: Git checkout to a new branch

```
$ git checkout -b new_branch
Switched to a new branch 'new_branch'
```

Using `git log`, we can see that history is the same as the `master` branch. We can add some changes into it. Let's add some changes.

Listing 2.8: Git add new changes to new branch

```
$ touch file2.txt
$ echo "something" >> file2.txt
$ git add -A
$ git commit -m "added file2.txt with something inside"
[new_branch e9a0670] added file2.txt with something inside
1 file changed, 1 insertions(+)
create mode 100644 file2.txt
$ ls
file1.txt file2.txt
```

Figure 2.8 gives an overview of our git history at this point. We can navigate through branches using command `git checkout branch_name`. Let's see how `master` is going.

Listing 2.9: Git getting back to master

```
$ git checkout master
Switched to branch 'master'
$ ls
file1.txt
```

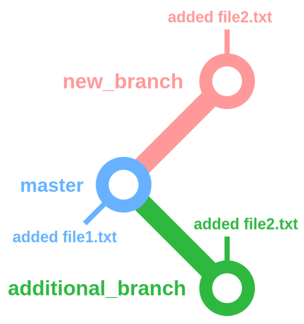


Figure 2.9: Git history after branching a second time

We can see that `master` kept its old state. To simulate concurrent working of two people, we can simply add a new branch that will create and work on the same `file2.txt` file as `new_branch`.

Listing 2.10: Git creating one more branch

```
$ git checkout -b additional_branch
Switched to a new branch 'additional_branch'
$ echo "smthelse" > file2.txt
$ git add -A
$ git commit -m "added file2.txt with something else inside"
[additional_branch aa76d01] added file2.txt with something else inside
1 file changed, 1 insertion(+)
create mode 100644 file2.txt
```

Our history now looks like figure ref 2.9. As we are done in `new_branch`, we want to propagate the changes into `master`. For this, we need to get back into `master` and initiate merging using `git merge` ↪ `branchname` command. This command will merge `branchname` with the branch we are currently in, called *active* branch.

Listing 2.11: Git merging new_branch into master

```
$ git checkout master
Switched to branch 'master'
$ git merge new_branch
Updating d92e102..e9a0670
Fast-forward
 file2.txt | 1
 1 file changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 file2.txt
```

Figure 2.10 shows how our history is at this point. We can also confirm with `git log` that `master` is now in the same state as `new_branch`. This happened because merging was *fast-forward*, meaning that `master` was simply a snapshot of the past of `new_branch`. But what happens if they actually differ ? For this, let's now try to merge `additional_branch` with `master`.

Listing 2.12: Git merging additional_branch into master

```
$ git merge additional_branch
CONFLICT (add/add): Merge conflict in file2.txt
Auto-merging file2.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Git is now signaling us that an unresolvable conflict appeared during merge. It comes from the fact that both our history from `new_branch` and `additional_branch` had `file2.txt` added with different content. Git is able to automatically solve most conflicts, but sometimes it can't and therefore asks us to do it manually. If we look at `file2.txt`, we can see how git signaled us the unresolvable conflict.

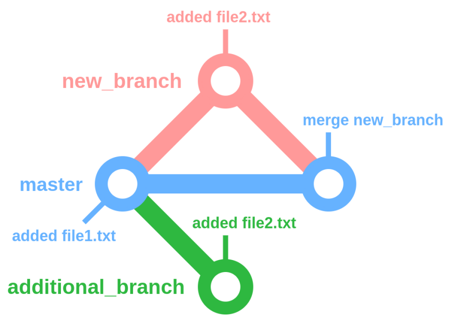


Figure 2.10: Git history after merging new_branch

Listing 2.13: Git merge conflict in file2.txt

```
$ vim file2.txt
1 <<<<<<< HEAD
2 something
3 =======
4 smthelse
5 >>>>>>> additional_branch
```

Git edited the file for signaling the conflict. Line 2 is current state, present in `master` and line 4 is incoming changes from `additional_branch`. We have to edit the file as we like and then commit.

Listing 2.14: Git conflict solution

```
$vim file2.txt
1 something, smthelse
```

Listing 2.15: Git finishing merge

```
$ git add -u
$ git status
On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:
        modified:   file2.txt
git commit -m "merged additional_branch into master [add reason here]"
```

Our history now looks like figure 2.11. There are two more commands to know that are interesting, and are used to communicate with the server. The first one is `git pull`, merging local active branch with corresponding branch on the server. The second one is `git push`, updating branch on the server, by merging it with corresponding local branch. Please note that most git configurations will prevent you to push without first pulling and solving conflicts.

This demonstration was only a picture of Git's most basic abilities, and you will learn more about them while working on projects and encountering problems.

A typical organization for a git repository is show on figure 2.12. The stable branch can only receive fast-forward commits and shall always be in a state at which the software is functional. The development branch is meant to merge and test working branches before pushing into stable branch. Working branches are created from development branch and are meant to edit code or bring new functionalities.

2.2.5 IDE : Integrated Development Environment

When a development software features both an editor and several other tools, as for example a debugger, build automation management, and version control, we call it an *Integrated Development*

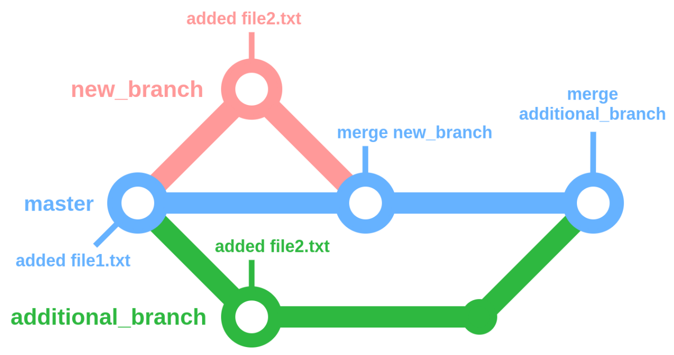


Figure 2.11: Git history after merging additional_branch

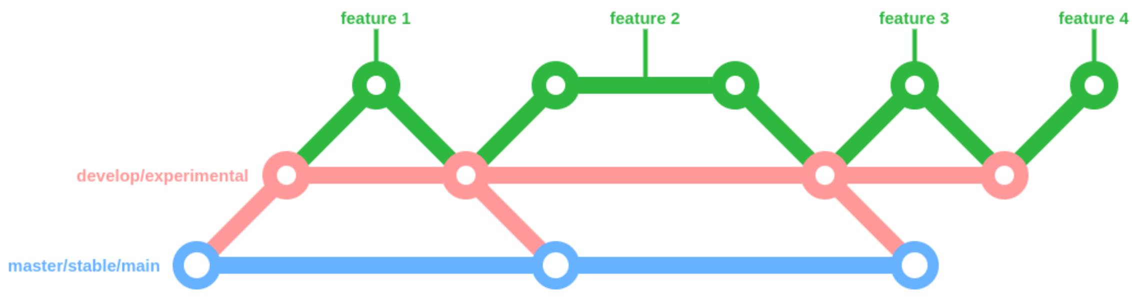


Figure 2.12: Typical look of a git history

Environment, or *IDE*. IDEs are often focused on a language or a set of languages, and even if they can often be adapted to other languages, they aren't as convenient as custom environments when dealing with unusual language combination in a project. As example, we can cite *CodeBlocks*[21] for C and C++, or *Pycharm*[22] for Python.

The choice between a custom environment or an IDE is not easy, as they both have strengths and weaknesses. Please note that following comparison is a *general* comparison and there exists exceptions.

IDEs are much simpler than custom environments to set up, as their default configuration is already ready for development. They also have better support for project-wide operations, for example refactoring that we will discuss later in section 4.3.3. It is because they provide a way to define a *project* and include files in it, which is rarely the case for standalone code editors that need to guess which files are to be treated by looking at build automation scripts. Because of this, you can expect an IDE features, as for example autocompletion, to be of higher quality compared to an editor. IDEs consume more resources than standalone editors, as they need to handle more tools at once, however this drastically varies and is rarely a bother.

As each IDE has its own way to manage projects, it can be bothersome to work on a project made with a certain IDE using another IDE or a standalone editor. Some editors, as VSCode, Emacs of Vim, can be customized to work alike an IDE and provide nearly all functionalities you can expect on one. A custom environment also has the advantage to teach you tools that you can easily reuse in other projects, even ones unrelated to software development. Finally, custom environment is easier to automate without using editor, as for example for building the project on a distant machine, or to allow someone else to work on the project with different tools.

2.2.6 Concerns when setting up your development environment

There are few points left to think of when setting up your development environment. They are listed in this section.

Exporting customization

Heavily customized environment, using an IDE or not, makes it difficult to set up again from scratch. Let's say for example you changed a lot of shortcuts in your editor. If you need to change your computer and then reinstall your editor, you will need to reconfigure them again, because you are used to them already. If you installed several plugins, you may also need to reinstall them.

There are some ways to export customization in both IDE and editors, for example in the form of configuration files. You can eventually include these config files in your project or store them elsewhere, as for example in another git repository. Of course, all automation scripts shall be included in your project as well. Finally, try to customize only what you can export or if the customization is easy to redo.

Operating system and desktop manager

Your operating system, as well as your computer setup are also part of your development environment, and when configured properly they can improve your development performance. For example, a tiling windows manager as *i3wm* is really convenient to display windows side by side, as your code editor and a documentation page. Setting up shortcuts in order to have quick access to a terminal or switching windows in an instant can save you a lot of effort later on.

Open source or Free software for environment

Open source and free development software have not much to envy to proprietary ones. A fully free or open source development environment has the advantage to be always available for anyone to use, and you will never run into license issues. However, it is important to note that some proprietary tools, as *IDEA Intellij*[23] IDE family have special discounts or free access for students[24], teachers and researchers, or even open source bases. Except when explicitly designated as proprietary, all tools presented in this course have open source or free software versions.

When setting up a project, it is advised to make sure it isn't tied to a specific software, especially proprietary software, in order to allow as much people as possible to use it and participate in the development.

Internet search and documentation

A lot of free resources are available online, as documentation, tutorials or programming and software development forums. When you encounter a problem, for example with a language syntax or an error from a library, it is highly likely that someone already encountered it, and thus that it has been solved and documented.

In case you ask for help on a forum, remember to read their conditions and guideline beforehand. Writing a help request by following the rules of the forum you post will help people understanding it better and answering quickly.

Use your keyboard

Each time you want to reach your mouse or pointing device, you loose some time and you reduce your focus. Most actions you can perform in a code editor, a terminal or any development tool do not require to use your mouse. Keyboard is much faster and efficient that mouse in a lot of cases.

- Keyboard keystrokes are binary, you press or you don't, while mouse needs a bit of precision. You can press keys extremely fast, however being both fast and precise with a mouse isn't really easy.
- Constantly reaching for your mouse is tiring on the long run. Left-handed people have an advantage to do this because if the mouse is on the left of the keyboard, it is closer to the position of the hand on the keyboard. Right-handed people can compensate by using a small factor keyboard or ten-key-less keyboard.

- A lot of important actions requiring multiple clicks and menu navigation can be done with a single shortcut in a well-configured software.

If you see that you perform an action often, remember to check if you can set a keyboard shortcut for it or simply perform it only with keyboard. Same goes for long to perform actions.

Physical environment

The last thing to remember is that your physical environment, your desktop, your screen, your mouse and keyboard, your chair, and the room you are in are part of your development environment. Inform yourself on ergonomics practices for working on a computer, which will depend on the time you spend on it each day, don't forget to take breaks fairly often.

2.3 Development

Programming is the activity of creating programs. Development is the activity of creating programs from need study to distribution, including project management. Development methodologies choice is an extremely complex matter, especially because of the client-developer relation management. We can assume that such relations are rare in a science and research context. Scientists are often their own customers when writing software, and when they are not, their development product is still targeted at other scientists. Because of that, requirements of the software are usually fairly clear. This course will later introduce a development method, `test-driven development`, in section 5.6. However, rather than presenting a set of development methods, this section will give some hints to conduct software development projects.

2.3.1 Incremental approach

In any project, software development or not, it is advised to divide large tasks and goals into tiny steps. However, some software development practices may seem a bit non-intuitive at first sight.

Let's consider two scenarios in which we develop the same application software for data visualization. It does roughly three things. First, loads data from several possible file format and convert them in an abstract representation. Second, analyze abstracted data with a set of algorithms selected by the user in a list. Finally, display results or save them in an image.

In the first scenario, shown figure 2.13, we plan the development of the software as we plan building a car or a furniture : one component after another. This approach is of course suitable but may pose some problems. The software is only functional the very end, so if for some reason we have to stop development, either because a step is impossible to achieve or because the project is dropped, we will leave an unusable software and time spent on it would have been wasted. Since project isn't functional during most of the development, progression is only shown through code and eventually tests. It may make it hard to see for developers and result in a loss of motivation. It will surely be even harder to show to people external to the project, as for example an administration that needs to decide on your funding.

In the second scenario, show figure 2.14, we chose our steps in order to always have a partially functional software at their end. This is called an *iterative approach.* It involves implementing all components at the same time, little by little. Functionality of the software increases with each step, and program can already be used for basic tasks at the end of the first step. Developers can clearly see achievements, and it is easier to share them with external people. In case the project stops, only the efforts put in the current step will be thrown away. Finally, it makes adding new functionalities during the project much more natural, because it is acknowledged from start that any component of the software can be modified at any time.

It isn't always possible nor good to achieve functionality at the end of each step, as sometimes some core components may require a lot of development before being usable. For example, a constraint based physics simulation engine will probably need few steps before it is even able to perform any simulation, even basic ones : vector math, geometry, numeric solver, and constraint

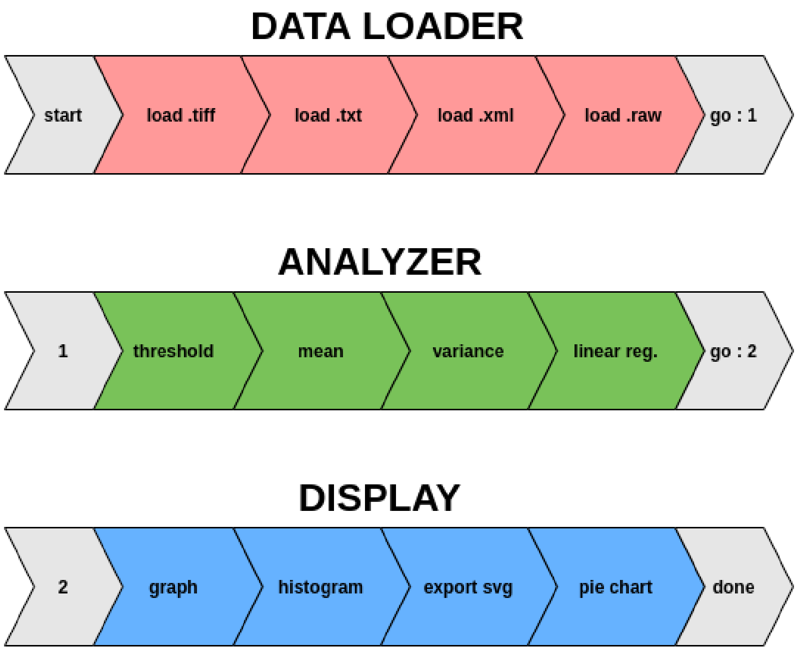


Figure 2.13: Linear plan. *Done* means software is functional.

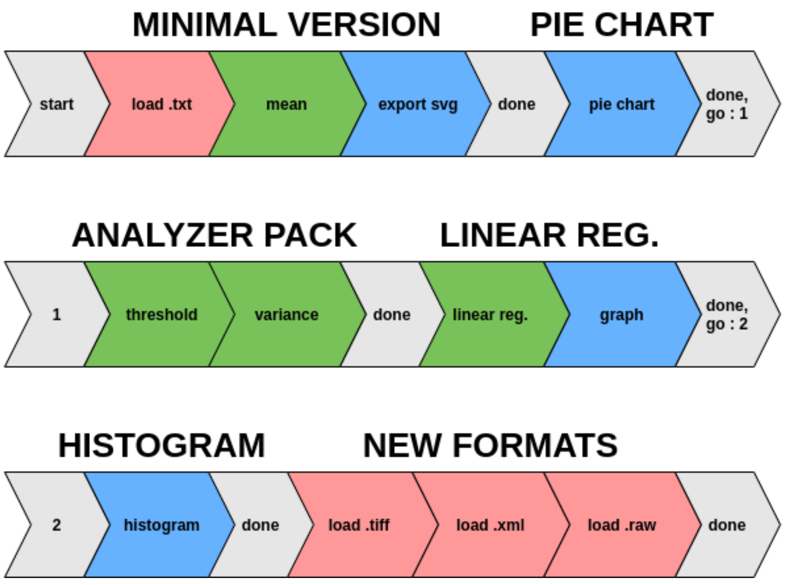


Figure 2.14: Iterative plan. *Done* means software is functional.

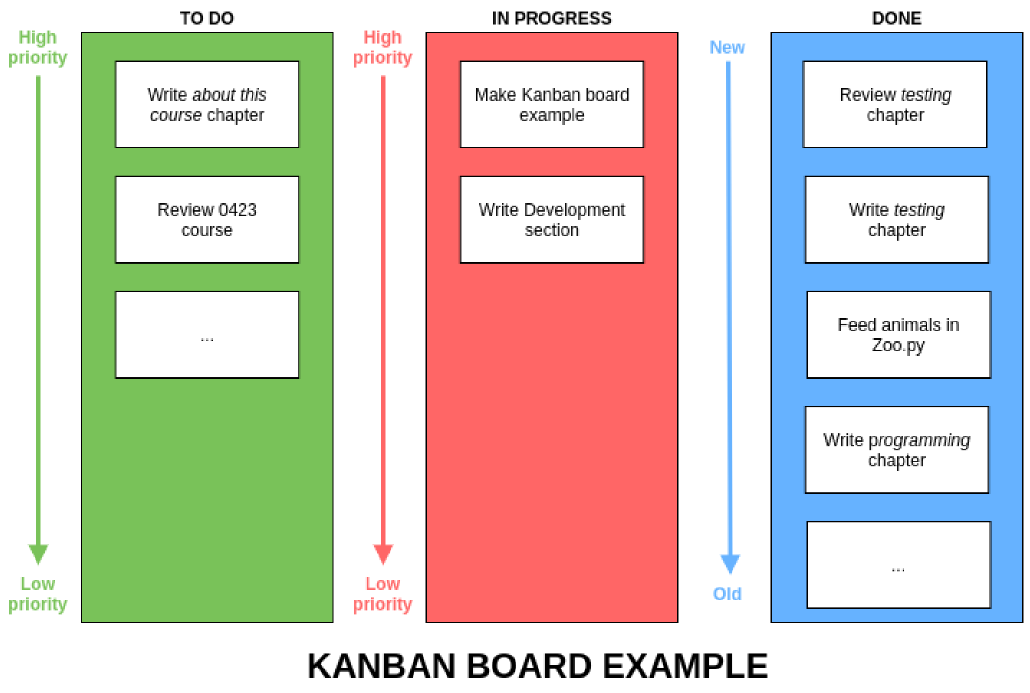


Figure 2.15: Kanban board example.

solver. Even code using the engine may need few steps before actually making use of it and displaying functionality. In that case, it could be advised to make such components, as the physics engine, a library or a framework. This way the notion of functionality is looked at directly from a code perspective.

We will, in the rest of this section, refer to steps as *iterations*.

2.3.2 Expressing and splitting goal

First, let's fix ourselves upper and lower bounds for an iteration duration. *I would propose one week to four weeks*, in order to both allow fairly wide atomic tasks but still guarantee a continuous functional improvement. Then, let's fix our target duration, of course in between bounds. Why not two weeks. Please keep in mind that those values examples are a bit arbitrary, and are meant to be adapted, we'll discuss that in section 2.3.4.

In order to split and express our goal, we can use a Kanban board, shown figure 2.15. It is a board made of several columns, each representing a state of the task, as for example *to do*, *in progress* and *done*. Tasks are represented by stickers, or *cards*. Columns are usually sorted chronologically, and most columns have an internal sticker ordering, the highest task in the columns having the higher priority. For Kanban boards, we can use a real board and stickers or a software tool for this. We'll do 5 columns, you can do more or less depending on the granularity you want.

- **Ideas**, for all ideas, expressed in a really general manner. These are simply reminders to think about stuff.
- **Features** where we will move interesting ideas after thinking on them and defining clear requirements for the feature to be considered as implemented and functional.
- **To do** where we register tasks corresponding each to an iteration. An item from Features can be moved here straight, or split in subtasks eventually.
- **In progress**, we move tasks here when we finally choose to implement them in the current iteration.

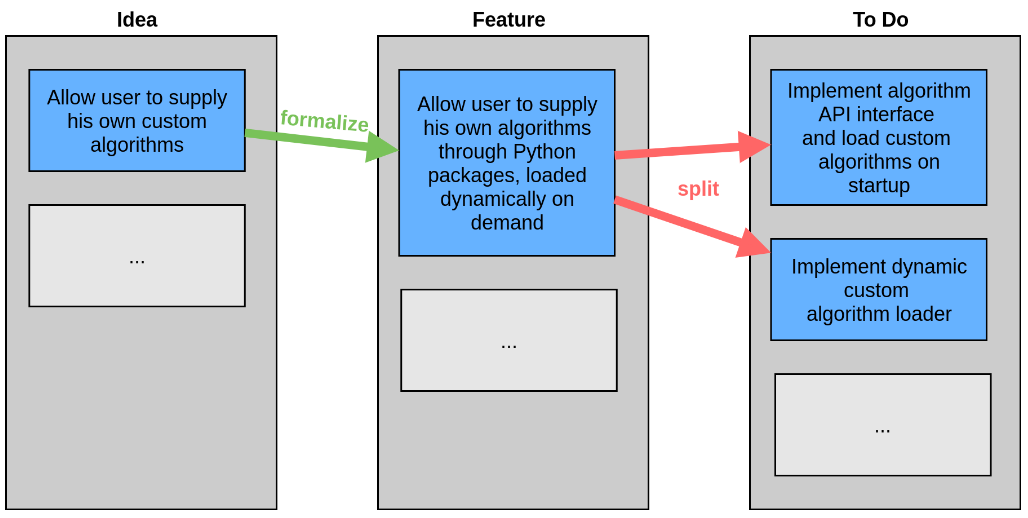

Idea
Feature
To Do
Allow user to supply his own custom algorithms
...
formalize
Allow user to supply his own algorithms through Python packages, loaded dynamically on demand
...
split
Implement algorithm API interface and load custom algorithms on startup
Implement dynamic custom algorithm loader
...

Figure 2.16: Example of card movement through the board.

- **Done**, we move tasks here once completed.

Please note there can be more than one task per iteration, if there are more than one developer working on the project or if some tasks are paused. One could eventually add a Paused column, or move paused tasks back in the To do column if it makes it clearer.

In the first part of a project, we will add a lot of stuff in both Ideas and Features as it is expressing the initial need and idea motivating our software, and then as the project advances, we will sometime add a bit more in them. Let's consider data visualization software from previous section, 2.3.1. At some point in the development, we have an idea : why not allowing people to supply their own analysis algorithms ? We therefore add a *Allow user to supply his own custom algorithms* card in the idea column to remember the idea. When we consider the development is sufficiently advanced for the idea to be implemented. We study possibilities and move the card to Features, after changing its description to *Allow user to supply custom analysis algorithms through python plugins, loading them dynamically on demand.* Sooner or later, it will become the top priority feature, and we will move it into To do, while splitting it into two tasks : *Implement algorithm API interface and load custom algorithms on startup* and *Implement dynamic custom algorithm loader.* When one becomes the top priority, we can simply move it into In progress and then Done once done. Figure 2.16 summarizes those manipulations.

Some tools allow extremely refined expression of relations between cards, as for example stating that two tasks are related to the same feature card or that one task is dependent on another task to be completed or chosen.

2.3.3 Forge software

A software forge is a tool offering both code management and project management possibilities. We will show *Gitlab*[25][26] as an example, which has two versions, one free software licensed, *Gitlab Community Edition*[27], the other proprietary and built upon community edition, *Gitlab Entreprise Edition*[28]. Please note that there are plenty of software with similar features, either free, as *Gitea*[29][30], or proprietary, as *GitHub*[31]. GitLab is a web service software and allows hosting projects online for free on their instance, or self-hosting an instance.

GitLab uses Git for version control, presented section 2.2.4, and adds a lot of convenient features around it.

Permission management

GitLab permits a really refined management of permissions related to the project[32]. For example, you can choose who can push to a given branch, see or interact with elements of the rest of this list.

Merge requests

Merging two branches may be a complicated task, requiring time and communication between developers of each branch to fix conflicts. Merge requests are a way to initiate and drive a merge[33]. A developer initiates a merge request in GitLab interface, stating source and destination branches. After that, a new discussion thread is opened on the page of the project, where developers can review code and changes, comment them, and ask for further modifications before allowing the merge. If automated verifications are configured for the project, the thread will display whether those verifications pass or fail. Finally, when merge request is accepted, a simple merge is performed.

Merge requests are useful if you want to have a protected branch, as for example the stable branch, where you want to disallow pushing and merging without review first. Merge requests are also extremely convenient for open source projects, allowing external people to propose improvements.

Issues and boards

Continuing into communication tools, issues are a way for any person to report bugs and ask for improvements[34]. They can be seen as Kanban cards somehow. For each issue submitted, a discussion thread is created, where developers and other people can exchange on the matter. Developers can choose to assign someone to treat the issue. A branch can be associated to an issue, and when the branch is merged, the issue is automatically marked as resolved.

Issues can then be managed by creating a kanban board for the project[35]. Boards associated to the project can also be displayed publicly as a roadmap.

Wiki

There is one wiki per project[36]. A wiki is really convenient to display information that are too wide to be put in a readme but too recent or special to be put directly in the doc, as for example a temporary fix for a bug.

It is possible to use the wiki as support for documentation. However, I would not recommend doing that in GitLab as wiki are stored in a separate repository, and therefore are not distributed with the code upon cloning a project.

Web editor

GitLab has an integrated web editor, and even a small web IDE[37]. It can be really useful to edit a file on the fly, in particular non-code files as readme. *In my opinion, it isn't suited for development in comparison to a real IDE or custom environment, especially because running the code you write is harder with the web IDE. However, for minor changes you are really confident with, it is a powerful tool.*

2.3.4 Evaluate and improve your method

Keep in mind that we didn't see a development method but only tips for development and project management. There are plenty of good and efficient development methods, iterative or not, and there is no consensus on the best way of conducting a software development project. To find a method that suits your needs better, *experiment* in several projects and systematically evaluate

Q number	Question
1	Am I following method properly ?
2	Am I happy with my current (short term past) actions ?
3	Is the project's progression satisfying ?

Q1	Q2	Q3	Direction
n	n	n	Try to follow the method before more investigation. If you can't, then change the method.
n	n	y	Your current actions are performant. Try to fix them as a method. However, beware of motivation loss.
n	y	n	Try to follow the method before more investigation.
n	y	y	Fix your current actions as the method.
y	n	n	You have to change your method.
y	n	y	Your current actions are performant. See if you can change details in the method to enjoy working with it.
y	y	n	First, see if your unsatisfying progression feeling doesn't come from a bias, as for example a difficult problem you face during current iterations. If not, try to change your method.
y	y	y	Everything is fine, keep up the good work !

Table 2.1: Questions to orient method evaluation

yourself. It is advised to start from an already existing development method, as Test Driven Development we'll see in section 5.6.

Then, write down the rules of your method. Try to be as precise and objective as possible. An objective rule could for example be : *Perform one commit per iteration. A commit shall express in first paragraph what has been achieved during iteration and difficulties encountered, in second paragraphs what shall be done for next iteration.* Such a rule is simple to verify. When writing a rule, write the motivation for it as well. Example : *With one commit per iteration we can track progression through git history. Expressing both achievement and difficulties helps compare work done to predictions. Having goals for next iterations means we only need to look at last commit to know where we are going in an iteration.*

With rules written down, it is easier to decide if the method is flawed or if you simply didn't follow it properly. Table 2.1 may help you start your evaluation. It can of course be applied to a team. Try to keep as much as possible history of your actions during a project, it can drastically simplify evaluation. Iterative approach is really convenient for method evaluation and improvement : after each iteration, or few iterations, you can perform a quick evaluation. Whatever method you choose, be sure to perform a deep evaluation after completion of each important goals, and as well at the success or failure of a project.

2.4 Sharing

You are likely to give access to your software to other people at some point, for them to use it or to help them criticize and reproduce your results. In case of a software solely attached to a paper, as a simulation script, not all the following points will apply.

2.4.1 Ease search

Try to make your software as simple as possible to find.

It all starts by properly picking a name for your software. An ideal name is not already used and easy to remember. If someone looks for it in a search engine, we want it to come in the firsts results. It will not forcibly help a discovery of your software, but it may help someone find your software again, for example after having heard of it in a conference. Try to find a name fitting following criterions.

- Not too short, not too long, 3-5 syllables is good. This way it is easy to remember
- Not solely related to applications field. For example, avoid *Asteroids* for an astronomy software.
- Not solely related to software field. Your software is a ... software.
- Simple spelling with none or few silent letters. *Asteroids* is better than *Ashteeeroids*. However, having an unusual spelling can be beneficial as long as it is heard when we pronounce the name or if it is an intuitive transformation as a simple wordplay. For example, *Azteroids* is simple to write and pronounced differently than *Asteroids*.
- Not solely made of widely used terms.
- If possible, not already used in any field.

Fitting these criterions isn't mandatory, there are simply advices. If you host your software on online resources, be sure that it is referenced by its name and not a derivative as an acronym. For example, if you host a software called *Dataminion Asteroids* on GitLab, you want the name of your repository to be something like *dataminion_asteroids* rather than *DA*.

If the mean of distribution you use can hold more detailed information about your software, as for example project description or readme on GitLab, provide a quick summary of its use. If your software is accessible to anybody, even non-scientists, it can be nice to have an accessible formulation. Remember that scientists are not the only ones who may be interested in using your software.

2.4.2 Ease installation and use

Finding a software is something, installing and using it is something else. Failing or struggling to install a software or a library is extremely frustrating. Fortunately, we can save some sweat to the user.

Write an installation guide, telling step by step the user how to install the software. *Step by step* is not to be taken lightly here. Try to be as precise as possible in your steps, and provide visuals of manipulations you go through. If you want user to enter a command line to install a package, show the command line, or even several versions if it is system dependent. If you want to skip some steps, as for example installing an external library, be sure to provide a link to an external tutorial on how to install it if possible.

Provide a list of dependencies, as needed libraries. If some dependencies are optional, meaning they are not required, or required for some secondary functionalities, state clearly what they are used for. Manually resolving dependencies can be extremely difficult depending on the context. For example, if the missing dependency is detected only after 3 hours of compilation.

Try to automate the installation process as much as you can. For example, if your software needs a configuration file somewhere in the system, its creation can be automated. Ideally, you can provide a fully automated installer that will completely install the software, eventually asking some options to the user. On Linux, such installation would be done through package managers. However, as there are a lot of different package managers, it can be tedious to provide packages to a large amount of Linux distributions. Another solution can be to use package manager from a language to ship your application, as for example *Pip* for Python.

Finally, document your software. This will be discussed in another chapter, section 4.2.8.

2.4.3 Contribution guides

In case your software is open to contributions, meaning other can submit modifications and improvements, or to discussion, as bug reporting or improvement suggestion, it can be good to document such processes.

A typical contribution guide mentions what kinds of changes are welcome and what are the requirements for them to be accepted.

- What parts of the software are open to modification or enhancement ? Example : *Physics code is not to be modified, however graphical interfaces improvements are welcome.*
- Are there contributions that are *wanted* ? Example : *We are looking for more data processing algorithms !*
- Are there technical rules to follow when contributing ? Example : *All C++ code shall be formatted using `.clang_format` configuration at the root of the project.*
- Are there soft rules to follow when contributing ? Example : *Any contribution shall be published under MIT license !*

It could also be advised to provide a development guide. A development guide provides flow of actions in order to set up development environment as well as performing important actions, as for example building the software or running tests.

A bug reporting or discussion guide is meant to help user to produce easy to process reports and suggestions. For example, if your software produces logs, a reporting guide may explain how to retrieve and share such logs. A suggestion guide can give a specific format for suggestions, or restrain suggestions to a certain set of functionalities. Example : *We are not accepting any demand of modification of Physics code.*

2.4.4 Licensing

This section is meant to give an overview of families of license you can put on a software and its sources. Especially, license will define under which conditions users can edit and share the software, and which warranty you offer to the user. We can roughly define three philosophies of licenses. All licenses have variants, and what is shown here is an approximation. Be sure to read licenses before using them with your software or using a software using them.

Proprietary

A license restraining software to use and forbidding modification and redistribution. Please note that a proprietary licensed software can still expose its sources. It simply is forbidden for user to modify them.

Permissive

A permissive license allows use, modification and redistribution of the software with only minimal restrictions. Such licenses tend to include a warranty disclaimer stating that developer can not be considered for bugs or faults of the software, neither damages it can cause. Permissive licenses do not require derivative work or modifications to be published under a particular license. It means that the modified version of a software under a permissive license, as *Apache License*[38], can be published under any license, even proprietary.

Copyleft

A copyleft license allows use, modification and redistribution of the software, but requires derivative work or modifications to be published under a particular license. Such licenses tend to include a warranty disclaimer as permissive licenses do. If a software uses a library under a copyleft license, it is likely that it has to be published under the same license as well. They ensure that all work done using the library or software is free software as well. One example of such licenses is the *GNU GPL*[39] family.

- **GNU GPL**[39][40][41] license requires any derivative work or modification, in case it is published, to be published under the same license.
- **GNU AGPL**[39][42][43] license requires any derivative work or modification, in case it is published or accessible through internet, to be published under the same license. It is well suited for a web application for example.
- **GNU LGPL**[39][44][44] Offers a bit more permissive definition, allowing a non LGPL-licensed work, as a library, to be used in a LGPL-licensed library under certain conditions. However, modifications of the library and derivative works still need to be published under LGPL or GPL licenses.

Chapter 3

Programming

3.1 Example-driven : Let's create animals !

This chapter presents programming and software architecture elements that can be useful for scientists. Programming and software architecture are both fields that may require years of learning and practice to be mastered, and what is shown here is only the surface of these two fields. It is sufficient for most of the small scale projects, and will provide you a basis for learning faster if you need specific skills to carry on larger projects. This chapter is conducted using an example : making a small program about creating a terminal-based chimera zoo.

3.1.1 Goal

Let's create a tiny zoo in our terminal. It will have three functionalities : managing the zoo, planning a route and finally taking a tour. This example will be done in Python since the language is simple to use and widely used in the scientific field.

Animals in the zoo will have four characteristics : a name, a noise, a movement and a cute action.

The route planning will be about the user selecting animals he wants to see. First we ask him about his preferences.

Listing 3.1: Route planning UI

```
Our zoo has a lot of animals : cat , tiger , albatros , whale
Which of them do you want to see ?
```

Then he answers.

Listing 3.2: Route planning UI with user wrong answer

```
Our zoo has a lot of animals : cat , tiger , albatros , whale
Which of them do you want to see ? cat , tiger , dragon
```

We check his answer : all animals specified shall be in the list.

Listing 3.3: Route planning UI with correction

```
Our zoo has a lot of animals : cat , tiger , albatros , whale
Which of them do you want to see ? cat , tiger , dragon
Answer incorrect , please choose animals from the list , separated by commas
Our zoo has a lot of animals : cat , tiger , albatros , whale
Which of them do you want to see ? tiger , cat , whale
Let 's go !
```

And then we take him to the touring step, which consists of each specified animal being shown[1] in the order specified by the user.

[1]Well, as nicely as a terminal allows us to show animals.

Listing 3.4: Touring UI

```
This animal is a tiger :
*roars*
*walks with grace and confidence*
*licks paws*

This animal is a cat :
*meows*
*runs and jumps with agility*
*licks paws*

This animal is a whale :
*inaudible ultrasonic noises*
*swims with ample and powerful movements*
*shoots water in the air*
```

The zoo management will occur before the user is invited to see the zoo, and will allow the user to create animals by mixing their characteristics. With no limits. Biology will be trampled here, hence the chimera zoo. We ask if user wants to create an animal, then offers to select each characteristic, and when done, user can either create a new animal or start planning its route.

Listing 3.5: Animal creation UI

```
Our zoo has a lot of animals : cat, tiger, albatros, whale
Do you want to add one more (y/n)? y
available noises are : meow, roar, ultrasonic, rifle, woof
your choice ? rifle
available movements are : walk, fast_run, agile_run, powerful_swimming,
    ↪ fast_swimming, teleportation
your choice ? teleportation
available cuteness are : paws_licking, water_shooting, hide, sarcastic_laughter
your choice ? sarcastic_laughter
please name your... thing : scary_boy

Our zoo has a lot of animals : cats, tiger, albatros, whale, scary_boy
Do you want to add one more (y/n) ? n
Let's go to route planning then !
```

You can note the ↪ at the start of one line in previous listing. This not actually part of the code shown in the listing and is used to denote a line wrapping. It means that the line was too long to be displayed in this document, and thus has to be split in several likes. What follows the arrow is the continuity of the previous line.

3.1.2 Order of implementation

In order to keep things simple, we will proceed by iterations, as presented in the previous chapter. There will be packed in three groups.

- Simple visit and route planning. The goal is to have a fixed set of animals from which the visitor can set up a route and then visit it.
  - Displaying one or two animals as a fixed route tour
  - Adding more animals
  - Getting route from the user
  - Touring following user's route
- Adding more animals in a simpler way. We want to be able to add more animals in a simple way.
  - Transferring animals from functions to classes
  - Grouping animals by their common behaviors using classes inheritance

- Making it possible to create animals. We want to be able to build animals.
  - Making our code easier to understand and splitting into specialized modules.
  - Making our modules independents from each other.
  - Allowing creating animals by combining characteristics.

Let's get started.

3.1.3 Getting started

Setting up a python project is pretty simple. We need to install Python first. Please note the rest of the course will be referring to Python 3[45].

Listing 3.6: Manjaro python install command

```
sudo pacman -S python
```

Then we create a folder where our code will be, and we create a Python file, named zoo.py.

Listing 3.7: Creation of the project directory

```
mkdir zoo_project
cd zoo_project
touch zoo.py
```

We can quickly test if our setup is working by doing a *hello world*, which is kind of a convention for programmers when learning new languages. Thus, let's open zoo.py file and write some code inside.

Listing 3.8: Python hello world

```
#This is our first Python script/program !

print("Hello, world !") #printing Hello, world ! to the terminal
```

Then we can simply execute our program from terminal by invoking Python interpreter and passing our file as an argument. It should output `Hello, world !` and terminate.

Listing 3.9: Running Python hello world

```
python zoo.py
Hello, world !
```

You may have noticed the first line in our python script, starting by a `#`. This is a comment in python[46]. Everything after this character on a line will be ignored by the Python interpreter, and thus we can use it to add notes in our program. Comments are really useful to help understand one's code. Their proper use will be discussed later in chapter 4, Code quality. In this chapter, they will be heavily used to explicit code. Please keep in mind that this is not exactly how they should be used in actual software development.

3.2 Basic programming : simply visiting zoo and route planning

3.2.1 Main code : fixed route

Let's erase all content of the zoo.py file first, and start off again. Our goal is simply to display one or two animals, as specified in listing 3.4. Our implementation will be really straightforward, and everything will be done in the *main code* of our program. In python, any instruction, or *statement*, written directly in a file is part of the main code[47][48]. This code is the one being first executed by the interpreter when passing it a program. Therefore, we can simply write our tour straight in the file :

Listing 3.10: Fixed route tour

```
print("This animal is a cat :")
print("*meows*")
print("*runs and jumps with agility*")
print("*licks paws*")
print("") #printing nothing to get simply a new line
print("This animal is a whale :")
print("*inaudible ultrasonic noises*")
print("*swims with ample and powerful movements*")
print("*shoots water in the air*")
```

Running this code should produce a satisfying output. However, doing so is often considered a bad practice[48], since it allows spreading code everywhere in the file, and it can get messy later, when our code is larger. Therefore, we will split this code in *functions*.

3.2.2 Functions : several animals

In programming field, a function is a set of instructions that can be used more than once, sometime called *procedures* or *subroutines*. They also are used to make code clearer and easier to maintain. Functions work with three steps. They need to be *declared* then *defined*, and finally *called*. Declaration is telling the interpreter that the function exists, it usually simply is giving it a name. Definition tells the interpreter what the function is, what are the instructions it relates to. In Python, declaration is done at the same time as definition, using the *keyword* `def`. You can see defining a function as teaching something to someone : you tell him how to do the thing, not to do it. We can express visiting an animal as a function. Let's erase our file and replace it by :

Listing 3.11: Animal as a function, function definition

```
def display_cat() :  #this is how we declare a function
    #and below is it's definition. It's called "function body"
    print("This animal is a cat :")
    print("*meows*")
    print("*runs and jumps with agility*")
    print("*licks paws*")

def display_whale() : #we declare another function
    print("This animal is a whale :")
    print("*inaudible ultrasonic noises*")
    print("*swims with ample and powerful movements*")
    print("*shoots water in the air*")
```

You can notice how the body of the function is indented using spaces. Python uses that to know when the body of a function stops.

Listing 3.12: Function body limitation

```
def a_function() :
    print("something")  #this is part of the function body

#indentation isn't anymore
print("something else") #this is not part of the body
```

Indented code pieces are called *code blocks*. In a lot of other languages, as C, they are delimited using braces `{body}`. We will talk more about them later in this chapter.

We successfully defined our functions. But if you run the file right now, nothing will happen. That's where the third step for using a function comes : we need to call them. Calling a function is the way we have to tell the interpreter to actually execute its body. At the bottom of our file, after the functions definitions, we can add the code for calling them :

Listing 3.13: Function call

```
display_cat() #displaying a cat
```

```
print("") #printing nothing to get a new line,
          #"print" is actually a function too !

display_whale() #displaying a whale

print("") #new line again

display_cat() #displaying a cat again
              #function can be called more than once !
```

Our code is now a bit clearer, because it is easier to see what it is made of :

Listing 3.14: Full animal code

```
#how we display a cat
def display_cat() :
    #and below is it's definition. It's called "function body"
    print("This animal is a cat :")
    print("*meows*")
    print("*runs and jumps with agility*")
    print("*licks paws*")

#how we display a whale
def display_whale() :
    print("This animal is a whale :")
    print("*inaudible ultrasonic noises*")
    print("*swims with ample and powerful movements*")
    print("*shoots water in the air*")

#what we do in our program ("main code")
display_cat()  #displaying a cat

print("")  #newline

display_whale() #diplaying a whale

print("")  #newline

display_cat() #displaying a cat again
```

Keep in mind that a function has to be declared first and then called. Therefore, its first call shall be lower in the file than its declaration. That's not true for every language, but it is for most. A function can also be called from a function, as long as it has been declared before.

Listing 3.15: Python function calling function

```
def hello() :  #first function
    print("Hello,")

def world() :  #second function
    print("world !")

def hello_world() : #third function
    hello()
    world()

#main code
hello_world()

#output should be :
#Hello,
#world !
```

3.2.3 Variables : getting user input

Now we want to allow visitor to set up a route. For that, we need to remember its answer somewhere. That's what variables are used for in a program. A variable is a box where we can put a value to access it later. Usually, a variable has a *type*. A type refers to the kind of value the variable holds, it can be a number, a character sequence, a boolean, and a lot more, as user defined types. We will see more about types a later section. A variable has to be declared and then have its value set before using it. In Python, we declare a variable by... assigning it a value. The type of the variable will be the type of the value we set, and it can change if we assign a value of different type.

Listing 3.16: Python variable declaration and initialization

```
a = 1 #we create a variable a and set it's value to 1
#a is now of type integer
a = 2 #we change the value of a to 2
#a is still an integer
a = "something" #we change the value of a to a string
#a is now a string (character sequence)
b = 1.0 #we create another variable b and set it to 1.0
#b is now a floating point number (float), e.g decimal
```

The type of variables defines which operations we can do on them. For example, we can add an integer to another integer or to a float, and we can split a string to form a *list*[49]. Lists are variables holding 0 to N values. In Python, a list can hold values of different types. We usually access each value independently using an index, an integer ranging from 0 to N-1. There are other ways of storing several values in a variable, as for example by using *dictionaries*[49], also called *map*, which allow indexing values with types different from integers, as for example a string. In this case, indexes are called *keys*.

Listing 3.17: Python lists and dictionnaries

```
a = [1, 2, 3] #creating a list
# a values are 1,2,3
a.append("22") #adding a value at the end of the list
# a values are 1,2,3,"22" (note how last value is a string and not integer)
a.pop(1) #removing second element in the list
# a values are 1,3,"22"
print(a[2]) #printing third value of the list, "22"
a[0] = 10 #changing value of the first element of the list
# a values are 10,3,"22"

d = {"key1":1, "key2" : "astring"} #creating a dictionnary
# d values are "key1":1,"key2":"astring"
d["key1"] = 2 #changing value of element
# d values are "key1":2,"key2":"astring"
print(d["key2"]) #printing "astring"
d["key3"] = True #adding a new element to dict, type is boolean
# d values are "key1":2,"key2":"astring","key3":True
```

We can also give variables to a function, and a function can produce variables as well. Variables produced by a function are called *return values*. A function needs to tell how many variables we can pass to it. Those declarations are called *parameters*. When passing variables to a function, those variables are called *arguments*. Some languages only allow functions to produce one variable. Python functions can produce more than one.

Listing 3.18: Passing variables to a function

```
def addsub(a, b): #declaring a function with two parameters, a and b
    add = a + b #new variable
    sub = a - b #new variable
    return add, sub #returning two values

#main code
#declaring some variables
d = 10
e = 5

#calling our function, passing d and e as argument,
#and assigning return values to f and g
f, g = addsub(d, e)
# now,
# f has a value of 15 (d + e)
# g has a value of (d - e)
```

We can notice that in the listing above, we created two variables in the body of the function. Those variables exist only in the function body, and can not be accessed from outside. Their value is also specific to the call. If we call the function twice, those variables will not be the same box each time[50]. We say those variables are *local* to the function. Usually, same goes for the parameters : when passing a variable as an argument, a new box is created and it is copied in it. However, this is not always the case, and this will be discussed later in this chapter.

Listing 3.19: Local variables of function

```
def a_function(param):
    ma_var = 10 #declaring local variable
    param = 5
    print(ma_var)
    print(param)

#main code
ma_var = 1 #same name as local one
a = 10
p = 20
#calling function
a_function(p)
print(ma_var) #prints 1
print(p) #prints 20

#output is :
#10
#5
#1
#20
```

For now, we will assume that our visitor will enter a valid route all the time, so we can add a new function in zoo.py.

Listing 3.20: Querying route to the user

```
def ask_route() :
    print("please list animals you want to see in order, separated by a comma")
    #getting user answer using "input(hint)" function returning a string
    answer = input("your answer : ")
    #then, we remove every whitespaces in the answer
    answer = answer.replace(" ","") #actually replacing spaces with nothing
    #finally, converting string as a list using "," as a delimiter
    #"str1,str2,str3" -> ["str1", "str2", "str3"]
    route = answer.split(",")
    return route #returning value
```

And to test it, we can simply display the returned value.

Listing 3.21: Displaying user’s answer

```
#before this are function definitions : display_cat(), display_whale(), ask_route()
#main code

answer = ask_route()
print(answer)

#touring code below
```

3.2.4 Flow controls : applying route

It is time to do real touring ! The solution presented here is NOT the best solution possible from a programming point of view as it is kept simple and as straightforward as possible. But before that we need to add animals and list them. First, add a bunch of animals with the same function construct we did.

Listing 3.22: Adding function animals

```python
#define some functions as we did before
def display_animal_name() :
    print("This animal is an animal name")
    print("animal noises")
    print("animal movement")
    print("animal cute action")
```

Then, create a function that lists names of all animals you have. I added a tiger and an alpine chough[2].

Listing 3.23: Listing animal names

```python
def list_animals() :
    animals = ["cat", "whale", "tiger", "alpinechough"] #simply listing their names
    return animals
```

Let's modify the `ask_route` function to ensure that the user gives a valid answer. In order to do this, we will use *control structures*. We will use two control structures called *iterations*. The first one is a *while loop*, that will run code as long as a condition is met, and the second one is a *foreach loop*, which will allow us to perform an operation on each element of a list. Loops have a body as well as function do, delimited by indentation. There can be variables local to the loop's body. They will also be local to each time the loop's body is executed.

Listing 3.24: Python while and foreach loop

```python
#this is a while loop :
a = 3
while a > 0 : #as long as a > 0,
    print(a) #we display a
    a = a -1 #we reduce a value by 1 #a is not local to the while

#this code produces an output as :
#3
#2
#1

#this is a foreach loop :
lst = ["THIS", "IS", "FOREACH!"]
for word in lst :
    print(word) #word is a variable local to the foreach's loop body

#this code produces output as :
#THIS
#IS
#FOREACH!
```

We will keep asking the user as long as he has not a valid route. A route is valid when every animal mentioned is listed by `list_animals()`. We can rewrite our `ask_route` now.

Listing 3.25: New ask_route function

```python
def ask_route(animals): #note that now we pass the animal list as a parameter

    valid = False #set to false to execute while loop once at least
    route = []
```

[2]A bird living at high altitude in mountains, often flying in and acrobatic manner.

```
    #a boolean value acts as a condition, not negates it (not False <=> True)
    while not valid :
        print("please list animals you want to see in order, separated by a comma")
        answer = input("your answer : ")
        answer = answer.replace(" ","")
        route = answer.split(",")
        #now let's check for validity
        valid = True #set to true because we assume that it is valid unless it
            ↪ fails
        for animal in route : #for each animal in route,
            #we check that it is present in animals,
            #and that it was present in each previous element of the route
            valid = valid and (animal in animals)

    #at this point, we know route is valid
    return route #return route
```

Conditions are expressed through boolean variables. They are values that can be converted to either `True` or `False`. There is roughly three ways to produce a boolean.

Listing 3.26: Python boolean production

```
#First way : assign boolean value directly to a variable
bool_true = True
bool_false = False

#Second way : compare two values of other types
i = #here assign a value
j = #here assign a value
res = i == j # res is True if i equals j, false otherwise
res = i != j # res is True if i is different from j, fo
res = i < j # res is True if i is strictly inferior to j, fo
res = i <= j # res is True if i is inferior or equal to j, fo
res = i > j # res is True if i is strictly superior to j, fo
res = i >= j # res is True if i is superior or equal to j, fo

#third way : combine two boolean values
a = #boolean value assign here
b = #boolean value assign here
res = a and b #True if a and b are True, fo
res = a or b #True if a or b or both is True, fo
res = not a #True if a is False, fo

#those three ways can be combined into really complex expressions
res = a and b and not (10 < c or c == 19)
```

Now that we have a route correctly setup, we can finally take a real tour. It will be done in another function, `take_tour`. To implement our tour, we will use a foreach loop over the route and another type of control structure, a *conditional structure*, often referred as *if else-if else*. It allows us to execute code depending on a condition.

Listing 3.27: Python if elseif else

```
#this is a conditional structure in Python
a = ... #we don't tell value here, as it will be used to
        #look at the contitional flow

if a == 0: #if a equals 0
    print("a == 0")
elif a < 10: #if a < 10 AND a != 0
    print("a < 10")
elif a < 30: #if a < 30 and a >= 10
    print("a < 30")
else: #otherwise
    print("else")
```

```
#note that elif are executed only if their conditions are met
#and if the previous if and elifs conditions weren't met.
```

The `take_tour` function will therefore look like this.

Listing 3.28: take_tour function

```
def take_tour(route):
    for animal in route:
        ifanimal == "cat":
            display_cat()
        elif animal == "whale":
            display_whale()
        elif animal == "tiger":
            display_tiger()
        elif animal == "alpinechough":
            display_alpine_chough()
        else: #should never happen, case animal is not found
            print("animal ", animal, " not found")
```

Actually, such a construct of a lot of `if` and `elif` checking the value of a single variable could be done using a *selection structure*, also called *switch*[51][52][53], but as it isn't supported in Python, it will not be discussed here.

Finally, we can rewrite our main code for it to use our new wonderful functions !

Listing 3.29: simple tour main code

```
#functions are defined before this

#main code

animals = list_animals()
route = ask_route(animals)
take_tour(route)
```

The full code at this point can be found in appendix A.1. This implementation can cause problems if we want to add more animals, because we need to change code at three places : add a new function as well as change `list_animals()` and `take_tour()` functions. A more concise and flexible version using dictionaries to store animals instead of functions can be found in appendix A.2. It is not mandatory to look at it, its principles will be explained in the next section of this chapter. It was not demonstrated this way in this section to show conditional structures, and also since this construct is only valid with this exact example, if we were doing more than simply printing text for animal actions, it would not work anymore.

3.3 Object programming : more animals

3.3.1 Classes : more intuitive animals

You may have felt like representing animals with functions was a bit strange. And you definitely would be right. Because, through `display_xxx()` functions, animals were seen only as the effect they had on the environment : their show. Even if it was sufficient for our first steps, it would be inconvenient in case we need to define a lot of animals. A solution for that would be to simply do as in example found in appendix A.2, using dictionaries. In this example, animals characteristics are represented by data, and a function `display_animal()` has the responsibility of displaying an animal regarding its data.

Listing 3.30: Animals represented as dictionnaries

```
def create_animal(animals, name, noise, movement, cute_action):
    #adding a value to the animals dict and returning it
    dct = animals
    animal = {"name":name, "noise":noise, "move":movement, "cute":cute_action}
    dct[name] = animal
    return dct

def display_animal(animal): #accessing value using keys and displaying them
    print("This animal is a ", animal["name"])
    print("*",animal["noise"],"*")
    print("*",animal["move"],"*")
    print("*",animal["cute"],"*")
```

This allows us to easily create animals, however, if we suppose we want to give animals a more complex behavior, as for example for ecosystem simulation, representing all their characteristics as data can be hard. We also want to give them *actions*. Another problem of this dictionary representation is that it is a bit hard to see what data of an animal means, because we have to look at the code of a function.

To overcome this, we can create classes and objects. A class describes a *concept*. For example, the concept of an animal, in our case, is an entity having a name, a noise, a movement, and a cute action. An object, also called *instance* of a class, is a concrete thing corresponding to the class. For example, a cat is an object of class animal. Classes can hold data, called *fields*, but also functions, called *methods*. Usually, classes feature a special method, the *constructor*, which is used to create objects.

Listing 3.31: Python class

```
#this is how we define a class in python
class MyClass :
#note that class has a body indented as function and controle structures do
    static_field = 10 #defining static/class field

    @staticmethod  #defining static class method
    def static_method() :
        print("static_field=", MyClass.static_field)

    #below is an alternative way of defining a static method
    #the difference is that class is passed as the first parameter
    #which can be cool class name is long and you have to use it
    #a lot in the body
    @classmethod
    def static_method2(cls) :
        print("static_field=", cls.static_field)

    def member_method(self): #a method is member if it first argument is called "
        ↪ self"
        #self refers to the object
        print("member_field=", self.member_field, ", static_field=", MyClass.
            ↪ static_field)

    def __init__(self, value) : #defining constructor, named __init__, with an
        ↪ aditional "value" parameter
        self.member_field = value
```

You can note that we defined two different types of fields and methods : *class* or *static*, and *object* or *member*. Class fields are the same for the entire class, meaning that their values are shared between instances. Class methods can be called without an instance. On the other hand, member fields are local to each instance : their values differs from an object to another. Member methods need an object to be called. To access members or methods of a class or an instance, most languages, including python, uses the dot operator `MyClass.mymember`.

Listing 3.32: Python class usage

```
#instancing a class into an object
my_object = MyClass(20) #note that self does not need to be passed as a parameter

#calling a class method
MyClass.static_method() #prints static_field=10
MyClass.static_method2() #prints static_field=10

#accessing a class field
print(MyClass.static_field) #prints 10

#accessing a member field
print(my_object.member_field) #prints 20

#calling a member method
my_object.member_method() #again, self does not need to be passed as an argument,
                          #the dot operator is actually doing that for us
                          #prints member_field=20, static_field=10
```

At this point, we can change the way we represent animals in our code to use a class. It will heavily resemble the dictionary representations and yet will be a lot clearer.

Listing 3.33: Animals as a class

```
class Animal :
    #how we create an animal
    def __init__(self, name, noise, movement, cuteness) :
        self.name = name
        self.noise = noise
        self.movement = movement
        self.cuteness = cuteness

    #actions an animal can perform
    def introduce(self) :
        print("This animal is a ", self.name)

    def make_noise(self) :
        print("*", self.noise,"*")

    def move(self) :
        print("*", self.movement, "*")

    def be_cute(self) :
        print("*", self.cuteness, "*")

    #also, how we display an animal
    def display(self) :
        self.introduce()
        self.make_noise()
        self.move()
        self.be_cute()
```

We now will list animals into a dictionary, as the example in appendix A.2, and adapt our main code. We also need an intermediate step to extract animal names from the dictionary in order to display them to the visitor when planning route.

Listing 3.34: listing animals using classes

```
#listing animals
def list_animals() :
    animals = {}
    animals["cat"] = Animal("cat", "meows", "runs with agility", "licks paws")
    animals["whale"] = Animal("whale", "ultrasonic noising", "swims", "geysering")
    animals["tiger"] = Animal("tiger", "roars", "heavy walking", "licks paws")
    animals["alpinechough"] = Animal("alpinechough", "craws", "flies with elegance"
        ↪ , "shakes feathers")
    #adding an animal simply means adding one line here now
    return animals


#extracting animals names
def list_animals_names(animals) :
    names = []
    for key in animals : #iteration over a dictionnary is done using keys
        names.append(key) #simply listing keys, easy !
    return names

#touring
def take_tour(route, animals) :
    for animal  in route :
        animals[animal].display() #accessing the animal by its name and displaying
            ↪ it

#main code
animals = list_animals()
animals_names = list_animals_names(animals)
route = ask_route(animals_names) #this function did not change
take_tour(route, animals)
```

It surely feels a bit nicer, however, it can still be enhanced. When we look at listing 3.34, we can see two things : All animals have a name, and some animals share common behavior : tiger and cat shares their paws licking as a cute action. We can improve our representation using inheritance.

3.3.2 Inheritance : animal factorization

Before getting into inheritance, we need to imagine how it would be in a much, much larger software. Let's say for example an ecosystem simulation. Animals would have kind of an artificial intelligence, and some would even have complicated behavior, *hard to model simply with data*. Even if all animals would more or less share the same action set (getting food, mating, resting, protecting territory), it would become hard to describe such behavior with data: we need to write code for that. We already have done it when writing method for actions of our animals. We will still be simply printing stuff in the rest of this chapter, but please keep in mind that these printing are now representing potentially complex processing.

Listing 3.35: Animal complex behavior

```
class Animal :
    # ... (these ... show that the class has some non mentioned code)
    #actions an animal can perform
    def introduce(self) :
        #was print("This animal is a ", self.name)
        #but could be extremely complicated behavior, taking
        #environment and internal parameters as hunger in account
        #to make decisions
    # ...
```

Doing so, we may end up wanting to have a different method body for the same action of two animals, as cat and dog, and therefore declaring each animal type by a class on its own, yet we want to avoid writing the same code twice. Looking at the following listing, we can see that cat and tiger have one behavior in common.

Listing 3.36: One animal, one class

```
class Cat :
    # ...
    def introduce(self) :
        #behavior of a cat
    def be_cute(self) :
        #complex paws licking
    # ...

class Tiger :
    # ...
    def introduce(self) :
        #behavior of a tiger
    def be_cute(self) :
        #complex paws licking, same as cat
    # ...
```

We can either write complex paws licking code twice, once in Tiger, once in Cat, but if we want to change it later, we may forget to modify both of them. Inheritance is here to solve this problem. Inheritance involves two classes : a *base* class, also called *mother* or *parent* class, and a *derived* class, also called *daughter* class. The derived class *inherits* from the base class : it has all its fields and methods. In some languages, including Python, a class can inherit from more than one class, but we will only talk about it later, in 3.5. For now, consider it is not a good practice, since most problems solved by multiple inheritance can be solved by composition, that we will introduce in section 3.5.3. Using inheritance, we would have cat and tiger share behavior :

Listing 3.37: Inheritance to share behavior

```
#defining a base Feline class
class Feline :
    # ...
    def be_cute(self) :
        print("complex paws licking")

#this is the syntax to create a class Cat that derives (inherits) from Feline class
class Cat(Feline) :
    # ...
    def introduce(self) :
        print("This animal is a cat")

#this is the syntax to create a class Tiger that derives (inherits) from Feline
    ↪ class
class Tiger(Feline) :
    # ...
    def introduce(self) :
        print("This animal is a tiger")

# main code
cat = Cat()
tiger = Tiger()

tiger.be_cute() #prints complex paws licking
cat.be_cute() #prints complex paws licking

tiger.introduce() #prints This animal is a tiger
cat.introduce() #prints This animal is a cat
```

A derived class can redefine a method of its base class. This is useful to create default behavior in the base class and specific behavior in derived class.

Listing 3.38: Redefining inherited behavior

```
#defining a base Feline class
class Feline :
    # ...
    def be_cute(self) :
        print("complex paws licking")

#this is the syntax to create a class Cat that derives (inherits) from Feline class
class Cat(Feline) :
    # ...
    def introduce(self) :
        print("This animal is a cat")

#lets create a special cat class that does not licks paws
class AngryCat(Cat) : #it inherits from Cat, which inherits from Feline
    # ...
    #redefining cuteness
    def be_cute(self) :
        print("Angry paws licking")

#main code
angry_cat = AngryCat()
angry_cat.introduce() #prints this animal is a cat
angry_cat.be_cute() #prints Angry paws licking
```

When constructing a derived class, we may want to construct it's base class first. We can do that by calling the constructor of the base class withing the body of the constructor of the derived class.

Listing 3.39: Constructing base class from derived

```
#base class want's two parameters in its constructor
class Base :
    def __init__(self, param1, param2):
        p1 = param1;
        p2 = param2;
    def print(self):
        print(self.p1, ",", self.p2)

#derived class will take only one parameter to be constructed and
#will use it to create its base part :
class Derived(base)
    def __init__(self, param1) :
        #calling the constructor of the base class, do not forget to pass self !
        Base.__init__(self, param1/2, param1*2)

#main code
derived = Derived(10)
derived.print() #prints 5,20
```

It's time to rewrite animals.

Listing 3.40: Rewriting animals with inheritance

```
#defining a base class Animal, with default behaviors
class Animal() :
    def __init__(self) :
        pass #nothing to do
    def display(self) :
        self.introduce()
        self.make_noise()
        self.move()
        self.be_cute()
    def introduce(self) :
        print("This animal is a ", self.name)
    def name(self) :
        return "unknown specie"
    def be_cute(self) :
        print("*does nothing*")
    def move(self) :
        print("*does nothing*")
    def make_noise(self) :
        print("*stays silent*")

#feline class : feline licks their paws and that's cute
class Feline(Animal) :
    def __init__(self) :
        pass #nothing to
    def be_cute(self) :
        print("*licks paws*")

#specific feline
class Cat(Feline) :
    def __init__(self) :
        pass #nothing to do
    def name(self): #redefining name method to change name of the class
```

```
        return "cat"
    def make_noise(self) :
        print("*meows*")
    def move(self) :
        print("*Runs and jumps with agility*")

class Whale(Animal) :
    def __init__(self) :
        pass #nothing to do
    def name(self): #redefining name method to change name of the class
        return "whale"
    def make_noise(self) :
        print("*ultrasonic noises*")
    def move(self) :
        print("*Swimms with powerful noises*")
    def be_cute(self) :
        print("*expels water from its back*")

# ... (other classes)
```

We now only have one last adaptation to make, which is the way we build the list of animals

Listing 3.41: Inheritance animal listing

```
def list_animals() :
    animals = {} #dictionnary
    animals["cat"] = Cat() #simply creating the object corresponding to the animal
        ↪ we want
    animals["whale"] = Whale()
    animals["tiger"] = Tiger()
    animals["alpinechough"] = AlpineChough()
    return animals
```

The full code of our zoo at this point can be found in appendix A.3.

You may have noticed that before inheriting, our dictionary of animals was full of only one type, `Animal`, and after implementing animals with inheritance, we ended up with a dictionary filled with different animal types, `Cat`, `Whale`, `Tiger`, `AlpineChough`. Yet we are able to display them using method `display()` as if they were all `Animal`. This is because of Python's duck typing, which we will explain later in section 3.4.4, Polymorphism and genericity, at the same time we discuss types.

3.4 Types

We quickly mentioned types previously, in section 3.2.3, Variables : getting user input. A type, or data type, defines what data represents, what operations can be performed on it and how they shall be performed. It gives *meaning* to raw data.

Listing 3.42: Two types with same raw data and different meaning

```
#both these types are composed of two floating point number, yet they do not
   ↪ represent the same thing
class Vector2D : #this is a vector used for coordinates
    def __init__(self) :
        self.x = 1.0
        self.y = 2.0

    def dot_product(self) :
        # ...

class MeanMedian : #this is a pair of a mean and a median value
    def __init__(self) :
        self.mean = 1.0
        self.median = 2.0
    def mean_over_median :
        return mean > median

#despite these two types being composed of the same raw data, a pair of floats,
    ↪ they do not have the same meaning.
```

In this section, we will put aside our zoo project and take time to study how the implementation of types can vary from one language to another. We will illustrate that by comparing Python and C++. No strong knowledge of C++ is needed.

3.4.1 Typing parameters

The first characteristic of typing to look at in a language is whether it is *static* or *dynamic*. In static typing, the type of variables is processed and checked at compile time. In dynamic typing, the type of a variable is decided when a value is assigned to the variable, at runtime.

Listing 3.43: Python is a dynamically typed language

```python
a = 10 #here, a is an integer
b = 20.0 #here, b is a float
a = "astring" #here, a is now a string
```

Listing 3.44: C++ is a statically typed language

```cpp
//this is a C++ comment
int a = 1; //a is an integer, to declare it we precise its type
           // (int) before its name (a)
float b = 1.0; //b is a float
b = 10; //b is still a float
```

The second characteristic to look at whether the typing is *strong* or *weak*. This depends on how the language handles *casting*, which is the action of converting one type into another. We already performed a casting in the previous listing.

Listing 3.45: C++ type implicit casting

```cpp
float b = 1.0; //b is a float
b = 10; //b is a float but 10 is an integer value
        //therefore, C++ will cast 10 into a float (10.0)
        //before assigning it to b.
```

This casting is said to be *implicit*, because we didn't tell C++ to cast `10` from `integer` to `float` and yet the compiler decided by himself to perform the casting. The more a language will allow implicit casts, the weaker it's typing will be. On the other hand, the more a language will make *explicit* casts mandatory, the stronger it's typing will be. Most of modern languages, including C++ and Python, can be considered as strongly typed.

Listing 3.46: C++ type explicit casting

```cpp
float b = 1.0; //b is a float
b = static_cast<float>(10); //this is a way to perform an explicit cast in C++
```

Those two characteristics have a heavy impact on the performances and use of a language. Statically typed languages tends to lead to higher execution speed than dynamically typed ones. Strongly typed languages makes it easier to produce reliable software in comparison to weakly typed languages. However, dynamic typing and weak typing both help to develop faster, since their syntax is usually lighter.

3.4.2 Compile-time type safety

We can resume compile-time type safety by the ability of a language to detect programming errors at compile time, by relying solely on types used by the programmer. In other word, it is the ability of the language to forbid programmer from using operations on the wrong type *before* running the program. Python, being dynamically typed, isn't a good example for compile-time type safety.

Listing 3.47: Python compile-time type unsafe code

```
class A :
    #...

class B :
    #...
    def b_method(self) : #this method is not present in A definition
        #...

def do_something_on_b(b_object):
    b_object.b_method()

#main code
a = A()
do_something_on_b(a)

#the execution of this code produces an error at runtime, since
#a does not has a method called b_method()
```

Compile-time type unsafe code can lead to extremely hard to track errors, as they may occur randomly at runtime, and thus be detected months after having been introduced into the software.

Listing 3.48: C++ compile-time type safe code

```
class A { //this is how we define a class in C++
    //...
};

class B {
    //...
    void bMethod(){
        //...
    }//this is how we declare a method in C++
};

//please note that below, as we define the function, we need to state the type of
//the parameter "object"
void doSomethingOnB(B object){
    object.bMethod();
}

//main code,
A a = A();
doSomethingOnB(a); //this code does not compile, A type is incompatible with type
    ↪ of the parameter
                   //of the function (B), the error is therefore detected before
                       ↪ runtime.
```

Types and/or variables can also have *mutability* restrictions, which means we may forbid to change their value.

Listing 3.49: C++ immutability

```
//this function has an immutable parameter, wich means that in the body of the
    ↪ function,
//you won't be allowed to change it's value
void myFunction(const int a) //"const" means a is immutable
{
    a = 10; //this won't compile, as we are not allowed to change the value of a.
}
```

Compile time type safety helps to detect and fix a large amount of errors before even running the program. There are some manners to enforce type safety in dynamically typed languages, but

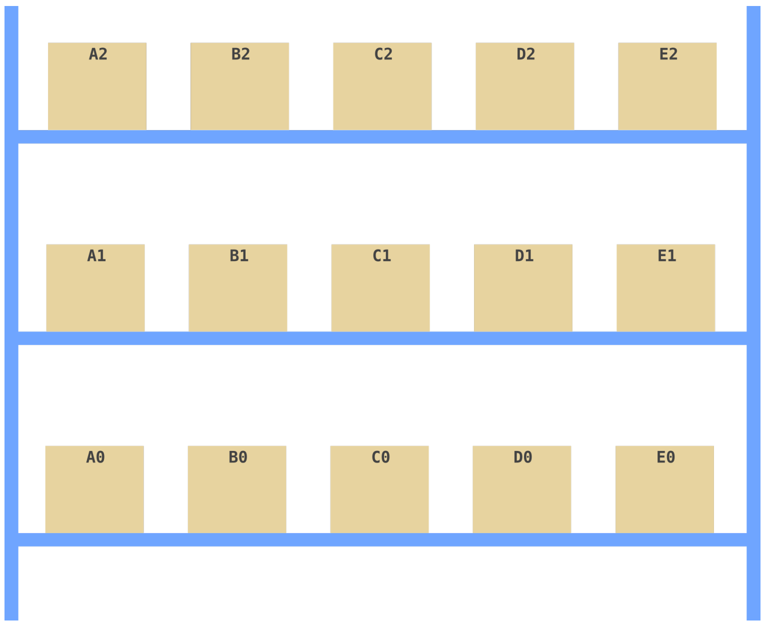


Figure 3.1: Memory seen as a shelf

they are out of the scope of this course. Please note that Python is *type safe* but not *compile-time type safe*, it simply enforces type safety during execution, raising an error if a type happens to be misused. Proving a language to be fully type-safe, either at compile time or at runtime, is close to impossible.

3.4.3 Pointers and references

An important notion in programming is the difference between a variable holding a *value* and a variable holding a *pointer*, or *reference*. Until now in this course, we only have seen every variable as holding a value : in this case, as variable was a named box in which we put the value in. To understand pointers, we can approach the way computer memory works by a box-themed metaphor : a shelf, shown in figure 3.1. The memory of our computer is like shelf on which there are boxes. Each box has a reference, representing the rows and column number. To make it simpler, let's say that columns are represented by a letter, and rows by a number, so `A2` represents first column, third row. It is called the memory *address*. When we create a variable, we associate a name with an address.

Listing 3.50: A variable is a name for an address

```
#when we do this,
a = 10
#the compiler finds an unused box on the shelf, (address B22 for example)
#puts 10 in it,
#and remembers that "a" now refers to the box at address (B22)
```

But actually, the address of a box is data as well. So how about we remember the address of a box in another box ? It can't be done in Python, but it would look like that :

Listing 3.51: Python storing address of a box (invalid code)

```
#when we do this,
a = 10  #the compiler finds an unused box on the shelf, (address B22 for example),
    ↪ ...
c = address(a) #the compiler puts address B22 in c
```

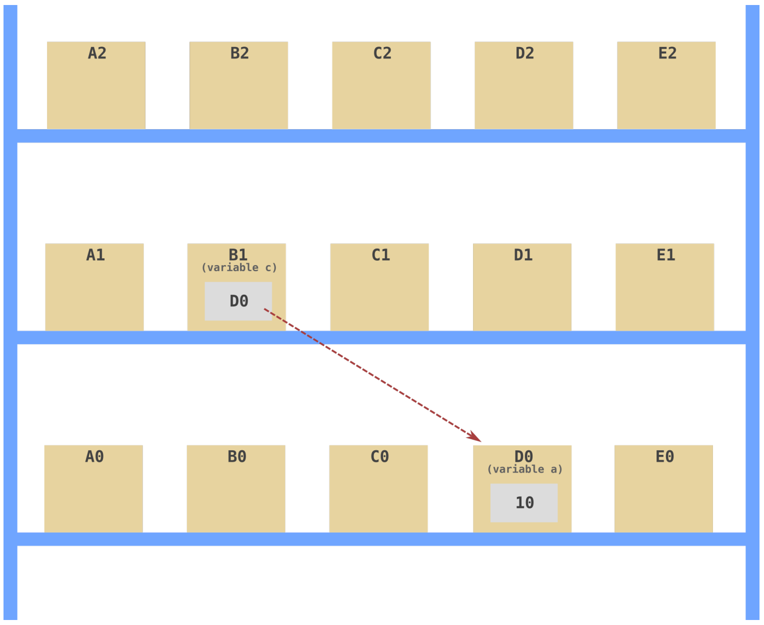


Figure 3.2: Box address in a box

In this case, `c` can be called a reference to `a`, or more accurately, a pointer to `a` : the value held by `c` is the address of another box, `a`, as shown in figure 3.2. Why would we actually do that ? It can be useful to save time actually, because boxes are not equal. Let's imagine we have a really, really heavy box, containing a lot of data. Passing it to a function would cost a lot of resources to copy it. Instead, we can tell the function the address of the box, which is light. This is called passing a variable by reference, instead of passing it by value, and it can't be done explicitly in Python, but it would look like following code.

Listing 3.52: Python passing by reference (invalid code)

```python
def my_function(data_address) :
    read_data(data_address, subdata) #we only access what we need using subdata

#main code
a = HeavyData()
my_function(address(a))
```

When a variable is passed by value to a function, it is copied, and copying heavy data can take time. Sending a pointer or reference can save that time. It is like if, instead of bringing this big box full of document to your colleague for him to read it, you would tell him that the box is at `A22` on the shelf. If it is faster, why don't Python allows us to do that ? Because actually, Python does that most of the time. In dynamically typed languages, most variables we create are actually references or pointers, and Python is no exception to the rule. Then, why don't every language do that ? There are two reasons for this.

First is side effects[54]. When copying a variable before passing it to a function, you ensure that changes happening inside the function will not affect value outside it. This is corrected in Python by setting some arguments of function as immutable.

Second is about performances. References are extremely fast to be passed, but before accessing the value we need to first read the address and then access the value box. This operation is called dereferencing. To be short, in a computer, looking at a tiny element in a box, no matter it's weight and content, takes always the same time. Let's compare two codes performing the same task, in both Python and C++.

Listing 3.53: Python heavy dereferencing

```
#we create classes encased in each other
class D :
def __init__(self):
    self.e = #something

class C :
def __init__(self):
    self.b = D()

class B :
def __init__(self):
        self.b = C()

class A :
    def __init__(self):
        self.b = B()

#main code
a = A()

g = a.b.c.d.e #this results in massive performance drawback !
```

In this code, when we create an object of type `A`, it's sub part `b` is actually a reference to a value of type `B`, having itself a sub part `c` and so on. So, each sub part is in a different box. Therefore, accessing `a.b.c.d.e`, even if written in one line, implies to first dereference `a.b`, then read the corresponding value to dereference `a.b.c` and so on and finally leads to three dereferencing. It is more or less equivalent to that code :

Listing 3.54: Python heavy dereferencing step by step

```
#this
g = a.b.c.d.e
#is somehow equivalent to doing this
a_b = a.b #dereferencing #1
a_b_c = a_b.c #dereferencing #2
g = a_b_c.d #dereferencing #3
```

In a language like C++ that allows handling variables by value, such an access would be done in a single operation.

Listing 3.55: C++ no dereferencing (invalid code)

```
//we create classes encased in each other
class D {
    int e; //that's how we declare a member field of the class
};

class C {
    D d;
};

class B{
    C c;
};

class A {
    B b;
};

//main code
A a = A();
int z = 10;
int g = a.b.c.d.e //this is exactly equivalent to
int y = z         //to this
```

The difference of performance comes from the fact that when encased by value, all sub parts of `A` object, `a`, are stored in the same box as `a`.

Choosing to use values instead of references isn't always possible, and performance drawback of references is often only noticeable in data intensive operations. To compensate that, Python offers really efficient libraries, usually implemented in C, to handle large amount of data, as NumPy and SciPy.

3.4.4 Polymorphism and genericity

In section 3.3.2, we saw that two different classes can have common behavior that can be factorized using inheritance. Sometimes, we want to be able to handle a set of variables with different types as if they were all the same type. This feature is called *polymorphism*, and can take place either at compile time or at runtime. Since python is dynamically typed, it only implements runtime polymorphism, *dynamic polymorphism*. The approach Python uses for polymorphism is called *duck typing*. It states that as long as a type supports an operation, then it can be handled with this operation like any other type having this operation.

Listing 3.56: Python duck typing

```python
#class A and B are unrelated classes defining the same operation
class A :
    def some_operation(self) :
        print("A")

class B :
    def some_operation(self) :
        print("B")

#main code
list = [A(), B()]
for elem in list :
    elem.some_operation()

#this program produces following output :
#A
#B
```

Duck typing is implemented in a lot of modern languages, even statically typed like Golang, because it is a bit more flexible and simple than inheritance-based polymorphism that we can find in C++. In C++, it is impossible[3] to create a list of mixed types like `list = [A(), B()]`. To do that in C++, we would need to have classes `A` and `B` have a common ancestor defining the operation we want to perform, and make a list of this said ancestor.

Listing 3.57: C++ inheritance based polymorphism (invalid code)

```cpp
//THIS CODE WOULD BE INVALID IN C++, IT IS SIMPLIFIED TO BE EASIER TO UNDERSTAND
class Ancestor {
    void do_something()
    {
        //...
    }
};

class A : Ancestor { //this is how we tell that A inherits from ancestor
    void do_something()
    {
        cout << "A" << endl; //this is equivalent to print("A") in Python
    }
}

class B : Ancestor {
```

[3]actually, it's possible, but that's really, really hard and most of the time not the best solution

```
    void do_something()
    {
        cout << "B" << endl;
    }
}

//main code
list<Ancestor> list = {A(), B()}; //this is a list of object of type ancestor,
    ↪ containig A and B instances

for (elem : list){ //equivalent of Python for elem in list
    elem.do_something()
}

//this code produces the following output
//A
//B
```

In the last C++ example, we manipulated a list of elements of type `Ancestor`. However, as `A` and `B` inherit from `Ancestor`, we can consider them as such : any operation supported by `Ancestor` will also be supported by `A` and `B`.

But C++, as a lot of statically typed languages, has more to offer through static polymorphism, taking the form of *overloading* and *genericity*. Overloading allows us to determine which operation to use regarding the type of the parameters we pass to it.

Listing 3.58: C++ overloading

```
//two functions, same name, not same parameters type
void do_something(int a){
    cout << "do_something(int)" << endl;
}

void do_something(float a) {
    cout << "do_something(float)" << endl;
}

//main code
int i = 1;
float f = 1.0;
do_something(i);
do_something(f);

//this code produces the following output :
//do_something(int)
//do_something(float)
```

Python forbids to declare two functions with the same name. C++ allows it, as long as their parameter list isn't the same : different number of parameters or at least one parameter with different type. At compile type, the compiler then selects the right function to use according to types of variables passed as arguments. Selecting different functions at compile time based on handled types is nice, but we can also do the opposite through genericity : implementing a function once for multiple types. In C++, such generic function is called a *templated* function.

Listing 3.59: C++ templated function

```
template<typename T> //we declare function as a template on a type, T
T add (T a, T b){
    return a + b
}

//main code
float a = 10;
float b = 11;
float c = add<float>(a, b); //here , "add" takes two floats, adds them
                            //and returns the result
```

```
int i = 10;
int j = 1;
int k = add(i, j); // The type to use can also be deduced by the compiler.
                   // here, "add" takes two ints, adds them and returns the result
```

This can also be done for classes, as we have seen in listing 3.58 : `list<Ancestor>`. C++'s templates are by far out of the scope of this course. It is only important to know that such thing exists in certain languages, as it allows writing less code while keeping the benefits of compile-time type safety. Also, duck typing in Python replaces efficiently C++ templates if we don't consider compile-time type safety.

Listing 3.60: Python duck typing vs C++ templates

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

a = 10.0
b = 11.0
c = add(a, b) //valid

i = 10
j = 1
k = add(i, j) //valid
```

3.5 Architecture : animal creation

Only one thing is missing in our chimera zoo : chimeras. Implementing chimeras will allow us to explore the surface of software architecture. Programming is about writing software. Architecture is about conceiving them. Most of the time, a software solution involves both programming and architecture. We already did a bit of architecture without even knowing all along the project : algorithmics as well as designing structure of a program are both a matter of architecture. Architecture is an extra wide field, probably wider than programming, and there is rarely perfect solution, it all depends on the context, which means that learning architecture is more a matter of experience than of a theoretical course. The goal of this section is not to teach architecture, only to give knowledge of some principles that can help design a software.

3.5.1 Modules and packages : clearer code structure

In this chapter, our code will eventually become bigger. *Much bigger.* For this reason we want to split our code between *modules.* Modules are group of functions, classes and other code stuff having a common purpose. You may have noticed that when writing our Python zoo code, some comments were acting as borders for implicit areas :

Listing 3.61: Comment delimited areas

```
#animals
#...

#touring
#...

#main code
#...
```

The idea stays the same, but we will do it in different files. We will then create three new files, alongside our `zoo.py` :

- `animals.py`, in which we move all animal related classes and the function `list_animals`.
- `route.py`, in which we move `ask_route` and `list_animals_names` functions.

- `touring.py`, in which we move `take_tour` function.

`zoo.py` file should now be left with only main code inside. Now we need to tell Python where to find all it needs to run main code.

Listing 3.62: Importing modules

```
#syntax for importing module is
#from module import names
from animals import list_animals
#we don't need to import animals (Cat etc) because they are used only by
    ↪ list_animals
from route import ask_route, list_animal_names
from touring import take_tour
#we need to specify names otherwise we would need to call them by module :
#import animals
#animals.list_animals()

#main code
animals = list_animals()
animals_names = list_animals_names(animals)
route = ask_route(animals_name)
take_tour(route, animals)
```

In the future, when a file becomes too large, we can simply split it between several submodules. To do so we need to create *packages*. Packages in Python are a group of modules. However, in some other language package and module can be synonyms. We can see packages as libraries. Let's split again our animals module into an animals package. First, we create a directory called `animals`, alongside `zoo.py`. Then we add one file per animal class : `animal.py`, `cat.py`, `whale.py`, `tiger.py`, `alpine_chough.py`. Then we move classes in their respective files. Don't forget to add `from animal import Animal` in all other four files. We can create a last file, `list.py`, importing all four animal classes and defining `list_animals` function. To finalize our package creation, we need to add a file `__init__.py`[55][56] in our directory to tell Python that this directory is containing a package, and then remove our `animals.py` file. We can now import `list_animals` from our package in `zoo.py`.

Listing 3.63: Importing from package

```
#we simply state the route to the module
#from package.module import names
from animals.list import list_animals
from route import ask_route, list_animal_names
from touring import take_tour

#main code
animals = list_animals()
animals_names = list_animals_names(animals)
route = ask_route(animals_name)
take_tour(route, animals)
```

Splitting like this can seem a bit overkill, considering our file was only a bit larger than 100 lines, but in the two next steps, we will write a lot of code.

3.5.2 Abstraction and interfaces

In section 3.4.4, 3.4.4, we saw that we can handle variables of different types as if they had all the same type, by defining a common way of interacting with them. This process is a form of *abstraction* and the *how to interact* is called an *interface*. Interfaces can take various forms, and we already encountered several of them. For example, when we declare a function, we actually create an interface to interact with the function, stating the number of parameters and eventually their types and the return type of the function. The `Animal` class we used as a base for concrete animal classes, as `Cat`, also acts as an interface, by defining operations like `introduce()` or `display()`. Types are interfaces to the data in a variable. Interfaces are useful from both a design and a programming point of view. They can either be explicit or implicit. We will consider an interface explicit if it can be expressed solely in code and be known and understood by both the programmer and the compiler.

In programming, explicit interfaces are useful to state what needs to be used with a component, as classes or functions, without needing to look into the component's code. The following example is in C++, because Python makes it hard to define explicit interfaces. Following code defines an interface but is not valid C++, it has been simplified for understandability.

Listing 3.64: An interface in C++ (invalid code)

```
//this is a way of defining an interface in C++ : a class without any field,
//having only methods. Please note methods are not implemented here.
//class TableInterface can NOT be instanciated into an object
class TableInterface {
    void putOn(Object o);
};

//Here, TableInterface is used as an interface : if you want to call
//setTheTable, you need to pass it an object that
//matches the TableInterface interface (eg inherits from it)
void setTheTable(TableInterface table){
    //do something with table
}
```

If we place ourselves in the point view of someone wanting to use `setTheTable`, we know that we need to give an object that allows to `Object` to be `putOn` it.

Listing 3.65: A C++ class matching an interface (invalid code)

```
//By making WoodenTable inherit from TableInterface, we tell that
//WoodenTable matches the interface TableInterface
class WoodenTable : TableInterface {
    //...
    void putOn(Object o){
        //do something with o
    }
};
```

Defining interfaces has a heavy consequence on the way we design software : we can define components without even implementing components it will use or act on. In the previous C++ example, we first defined the `TableInterface`, then `setTheTable`, and finally `WoodenTable`. If we didn't define an interface, we would have needed to first define WoodenTable, and then a function `setTheTable(WoodenTable object)`. Of course, it also allows `setTheTable` to work on any possible table in a polymorphic way. This abstraction is extremely convenient when developing libraries, where you can declare what your tools will need without having to define it, letting this responsibility to the user. Dependency between components is called *coupling*, and through interfaces we can reduce coupling, as shown figure 3.3. Loosely coupled code is easier to test. This will be discussed in chapter 5, Testing.

Even if Python does not allow creating fully explicit interfaces, it still helps to reduce coupling. More than that, because of the combination of dynamic typing and duck typing, Python ensures that all components we create are fairly loosely coupled. However, it makes harder to express requirements for the programmer, which has to be done through documentation.

Listing 3.66: Python interface through documentation

```
#a python documentation is done using specific comments,
#right after the first line of the code block of a component,
#in between """ """. It can be spread over multiple lines. These are called
    ↪ Docstrings
def set_the_table(table_like) :
    """set the table sets the table, it requires table_like to have a putOn(object)
        ↪  method."""
    #this line shall be empty
    #here do something with table_like.
```

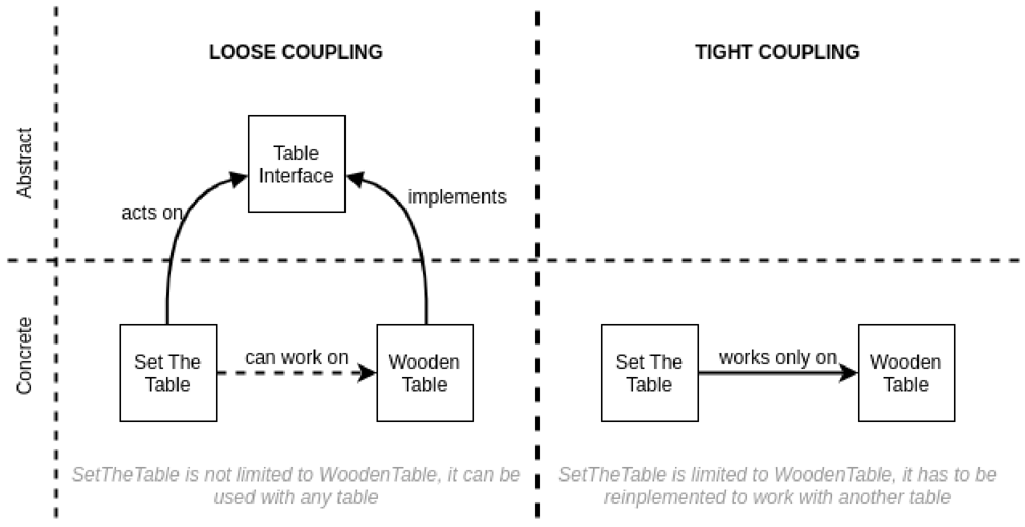

LOOSE COUPLING
TIGHT COUPLING
Abstract
Concrete
Table Interface
acts on
implements
Set The Table
can work on
Wooden Table
Set The Table
works only on
Wooden Table
SetTheTable is not limited to WoodenTable, it can be used with any table
SetTheTable is limited to WoodenTable, it has to be reinplemented to work with another table

Figure 3.3: Illustration of loose or tight coupling

Documentation and the implication of not having explicit interfaces will be discussed further in chapter 4.

3.5.3 Composition : animals better than with inheritance

It's time to put aside ethical concerns about mixing animals and focus on *how* we can mix animals. I meant mix animals *together*, which is as unethical as mixing an animal alone, but is much harder to do. To keep this course and example relatively simple, we will continue to consider that animals have four characteristics.

First, we need to observe what happens to our class structure when we drastically diversify animals in our zoo. For now, there were only few animals, and creating an ancestor tree to factorize their characteristics is fairly easy. But can we reach a moment where it will be impossible to factorize using inheritance ? Yes, and it can come much faster than we think. Let's add one more animal, the *fishing cat*, which is actually a real specie. Even if its name is fairly explicit, here are its characteristics :

- *name* : fishing cat.
- *movement* : powerful swimming.
- *noise* : purring. Actually, I'm not sure they can purr, but let's say they can because it's cute.
- *cute action* : paws licking.

Well, that's a cat who fishes. And now with inheritance we can't avoid redundancy anymore. As shown figure 3.4, we can either make the fishing cat a feline, sharing paws licking with others, but then we would have to write twice the powerful swimming code it has in common with whale, or create a *powerful swimmer* class that would be ancestor of both whale and fishing cat, and write twice the paws licking code. From a biology point of view, it may make more sense to consider that fishing cat is a feline. However, since we intend to create chimeras, biology does not make a lot of sense in our code. Stating fishing cat as a feline would only postpone the problem, and as we create more chimeras we will have to write again code we already wrote in another branch of our inheritance tree.

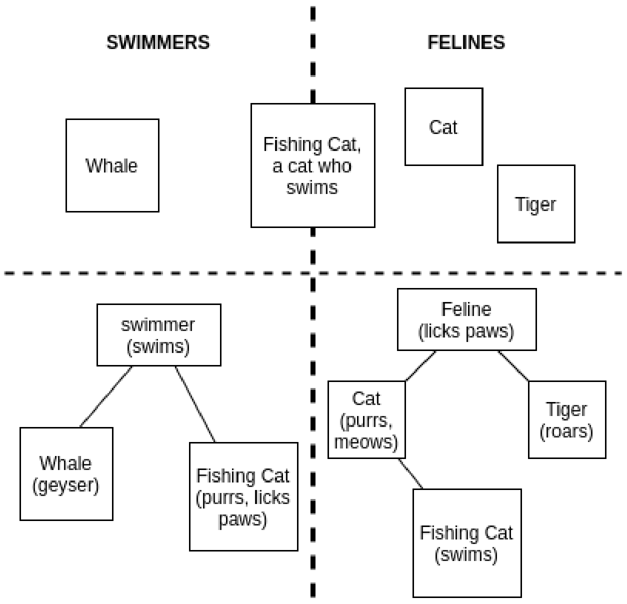


Figure 3.4: Fishing cat ancestors problem

Multiple inheritance could be a solution : we would create one base class for each characteristic, and each concrete animal class would inherit from three characteristics classes, name being common to all.

Listing 3.67: Python animals with multiple inheritance, name field is ommitted

```
class PawLicker :
    def be_cute(self):
        #...

class PowerfulSwimmer :
    def move(self) :
        #...

class PurringThing :
    def make_noise(self) :
        #...

class FishingCat(PawLicker, PowerfulSwimmer, PurringThing) :
    def introduce(self) :
        print("this animal is a fishing_cat)
        print("btw multiple inheritance is bad)
```

As we already said in section 3.3.2, multiple inheritance is, in most of the cases, the wrong solution. A lot of languages do not support multiple inheritance, and in most languages supporting multiple inheritance it is a mess to work with. Python is an exception, and multiple inheritance is *not so bad* with it. When someone feels an urge to use multiple inheritance, often have a misunderstood desire to use *composition*. We can model an animal as an assembly of parts. One part for movements, one part for noise and one part for cuteness, and that's what we call composition. Name is always a string so it will be kept a string, field of `Animal` class.

Listing 3.68: Animal by composition

```
class Animal :
    def __init__(self, name, noise, movement, cuteness):
        self.name = name
        self.noise = noise
        self.movement = movement
        self.cuteness = cuteness

    def introduce(self) :
        print("This animal is a ", self.name)

    def display(self) :
        self.introduce()
        self.noise.make_noise()
        self.movement.move()
        self.cuteness.be_cute()

#main code
#how we create a Cat, we suppose classes Meow, AgileRun and PawsLicking
#to be defined
cat = Animal("cat", Meow(), AgileRun(), PawsLicking())
```

This construction makes it really easy to create animals with a lot of diversity, while never having to write the same code twice. It also makes it possible to create animals at runtime, as composition of an `Animal` object is chosen at runtime. However, it makes creation of a specific animal a bit harder. We can compensate this deficit using either a class or a function to create an animal explicitly.

Listing 3.69: Easy composition animal creation

```
#cat explicit creation by function
def create_cat():
    return Animal("cat", Meow(), AgileRun(), PawsLicking())

#cat explicit creation by class, we pass components to base class constructor
#in constructor
class Cat : (Animal) :
    def __init__(self):
        Animal.__init__(self, "cat", Meow(), AgileRun(), PawsLicking())
```

We can now create three files in the package `animals`: `movement.py`, `noise.py` and `cuteness.py`. They will receive components. Each component has to define a name, unique within components of a same file. The file shall also define a `list_components` function that returns a dictionary of all components of the file, indexed by their name. Here is an example for `cuteness.py`.

Listing 3.70: Cuteness sub package

```
#defining components
class PawsLicking :
    def __init__(self) :
        pass
    def name(self) :
        return "paws_licking"
    def be_cute(self) :
        print("*licks paws*")

class Geysering :
    def __init__(self) :
        pass
    def name(self) :
        return "geysering"
    def be_cute(self) :
        print("*licks paws*")

#... other components

#listing them
def list_cutenesses() :
    ctns = {}
    ctns["paws_licking"] = PawsLicking()
    ctns["geysering"] = Geysering()
    #... and so on
    return ctns
```

Let's then remove files `cat.py`, `whale.py`, `tiger.py` and `Alpine_chough.py`, and then update `animal.py` with code shown in listing 3.68. We also can update function `list_animals` in file `list.py` to adapt to the new way of creating animals.

Now, we will put visitor related code in a new file called `chimera.py`, alongside `zoo.py` at the root of our project, and implement a function to create chimera there. Don't forget to import components and `Animal`.

Listing 3.71: Chimera creation

```
#an utility function for listing keys of a dictionnary
#and returns them as a string
def list_keys_str(dict) :
    list = []
    for key in dict :
        list.append(key)
    glu = ", "
    return glu.join(list)

#this code assumes the visitor does no errors when writing part names
def create_chimeras(animals)
    noises = list_noises()
    movements = list_movements()
    cutenesses = list_cutenesses()

    noise_query = "Available noises" + list_keys_str(noises)
    movement_query = "Available movements" + list_keys_str(movements)
    cutenes_query = "Available cuteness" + list_keys_str(cutenesses)
    print("animals : ", list_keys_str(animals))
    answer = input("Do you want to create a chimera (y/n)?")
    continue = (answer == y)
    while continue :
        print(noise_query)
        noise = input("your choice : ")
        print(movement_query)
        movement = input("you choice : ")
        print(cuteness_query)
        cuteness = input("your choice : ")
        name = input("please name your animal")
        #animals will also get added outside the function since it is passed by ref
        animals[name] = Animal(name, noises[noise], movements[movement], cutenesses
            ↪ [cuteness])

        answer = input("Do you want to create another chimera (y/n)?")
        continue = (answer == y)
```

After few modifications to `zoo.py`, to change imports and create chimera, our zoo will be ready.

Listing 3.72: Chimera creation zoo.py

```
from animals.list import list_animals
from route import ask_route, list_animal_names
from touring import take_tour
from chimera import create_chimeras

#main code
animals = list_animals()
create_chimeras(animals)
animals_names = list_animals_names(animals)
route = ask_route(animals_name)
take_tour(route, animals)
```

You can find the complete code of the zoo in appendix A.4, as well as a C++ version in appendix B. When using composition instead of inheritance, we somehow move behavior from method to a field, and a field is much simpler to change at both compile time and runtime. It helps to develop flexible software, but isn't the solutions for all cases where genericity is needed. Inheritance usually leads to writing less code, even if this difference might be small sometimes.

When choosing between the two solutions isn't easy to decide, it can be advised to choose the one preferred by the programmer, and to eventually change later, as it will be discussed in chapter 4, Code quality, section 4.3.3.

3.5.4 Design patterns : pre-made solutions

Software development often requires to solve architectural problems. Fortunately for us most problems archetypes have already been encountered and solved by other developers or architects. Therefore, when encountering a problem that is non-trivial the best reflex to have is to look in literature or on the internet if it hasn't already been solved. This can be bothersome if we don't know specific names of structure we want to achieve. For this matter, one name can be remembered : *design patterns*. They are a group of solutions that address nearly any architectural problem a developer can encounter, and by remembering this name, it is easy to find a list of them. Design patterns are only archetypes of solutions and have to be adapted to perfectly fit one's need. Design patterns are often expressed in terms of object-oriented programming but can easily be translated to other programming paradigms. Using them provides an already battle tested solution, that can be named for better design and documentation clarity. Let's observe some of them.

The composite pattern allows representing tree-like structures, as for example a file and directory structure.

Listing 3.73: Composite pattern

```
#we can use this pattern to represent a file structure
#defining a base class component, it is an interface
class Component :
    def display(self) : #declaring operation
        pass

#defining a composite class directory, which a component as well
class Directory (Component) :
    def __init__(self, name) :
        self.components = [] #holds a list of components
        self.name = name
    def display(self) :  #operation usually involves performing operation on
        ↪ components
        print("directory ", name, " :")
        for comp in self.components :
            comp.display()
    def add(self, component) : #obviously, component is supposed derived from
        ↪ Component
        self.components.append(component)

#defining another component class
class File (Component) :
    def __init__(self, name) :
        self.name = name
    def display(self) :
        print(self.name)
```

The observer pattern enables an object to be notified by another object. It can be useful for simulations, where objects need to react to a change of a specific object.

Listing 3.74: Observer pattern

```
#defining an Observable class, that is a rigid body
class RigidBody :
    def __init__(self) :
        self.observers = [] #remembers the list of observers
    def add(self, observer) :
        self.observers.append(observer) #registers observer to the list
    def on_collision(self) :
        #do some complicated physics stuffs
        #notify observers
        for obs in self.observers :
            obs.notify_collision(self)

#defining an observer that can be notified, it is an interface
class RigidBodyObserver() :
    def observe(self, body) :
        body.watch(self)
    def notify_collision(self, body) :
        pass

an observer could be for example an object counting collisions
class CollisionCounter(RigidBodyObserver) :
    def __init__(self) :
        self.count = 0
    def notify_collision(self, body) :
        self.count += 1 #increasing by 1
```

The iterator pattern abstracts a group of data to allow performing operations on it regardless of the structure of the group.

Listing 3.75: Iterator pattern

```
#interface declaring an object providing iterators
class Iterable :
    def begin(self) :
        pass

#inteface declaring an iterator
class Iterator() :
    def is_end(self) :  #are we done iterating ?
        pass
    def next(self) :  #go to next item
        pass
    def value(self) :  #get value of current item
        pass

#defining a list class with iterator
class ListIterator(Iterator) :
    def __init__(self, list, index) :
        self.list = list
        self.index = index
    def is_end(self) :
        return self.index == len(list.inner_list)
    def next(self) :
        self.index += 1
    def value(self) :
        return self.list.inner_list[self.index]

class List(Iterable) :
    def __init__(self)  :
        self.inner_list = [] #we use a list in our list, yep, that's cheating !
    def begin(self) :
        return ListIterator(self, 0)
    def append(elem) :

#defining a vector3D with class iterator
class Vector3DIterator(Iterator) :
    def __init__(self, vector, field) :
        self.vec = vector
        self.field = field
    def is_end(self) :
        return self.field == "end" #not at the 3rd field yet
    def next(self):
        if(self.field == "x") :
            self.field == "y"
        elsif(self.field == "y") :
            self.field == "z"
        else :
            self.field == "end"

    def value(self) :
        if(self.field == "x") :
            return self.vec.x
        elsif(self.field == "y") :
            return self.vec.y
        else :
            return self.vec.z

class Vector3D(Iterable) :
    def __init__(self, x, y, z) :
        self.x = x
        self.y = y
        self.z = z
    def begin(self) :
        return Vector3DIterator(self, "x")

# main code for iterating
vec = Vector3D(...)
iter = vec.begin()
while not iter.is_end() :
    print(iter.value())
    iter.next()

lst = List()
#... filling up list
#iteration code is absolutely the same !
iter = list.begin
while not iter.is_end() :
    print(iter.value())
    iter.next()
```

3.5.5 Justice for multiple inheritance

We said several times in this chapter that multiple inheritance is bad. Indeed, if used without care, multiple inheritance can later cause heavy and painful headaches, bringing confusion and despair on your project. However, if knowing its common caveats, and used with care, multiple inheritance can save a lot of work.

The first main problem of multiple inheritance occurs when redundant names appear in inheritance tree.

Listing 3.76: C++ multiple inheritance name issue

```
class A {
    void operation(){
        //...
    }
}

class B {
    void operation(){
        //...
    }
}

class C : A, B { //C inherits from both A and B
    //...
    //does not redefines operation
}

//main code
C c = C();
c.operation(); //is this operation method from A or B ?
```

In that case, C++ will force you either to reimplement `operation` in `C` definition, or to precise which base class you want to call `operation` from : `c.A::operation()`.

The second problem is a consequence of the first one : in case a class is inherited several times, shall the derived class have several copies of it in its definition ? This is referred as the *Diamond inheritance problem.*

Listing 3.77: C++ Diamond inheritance problem

```
class A {};

class B : A {}; //B is made of A

class C : A {}; //C is made of A

class D : B, C {}; //D is made of both B and C
//Does it has two A part or one ?
```

This may look like a trivial problem, but `B` or `C` can eventually require to maintain their base class `A` in a certain state, and having only one shared `A` part could cause a problem. On the other hand, not sharing `A` can lead to strange behavior is `A` is meant to be shared, as for example if it holds a counter that `C`, `B` and `D` will use.

This looks painful, so why would we use multiple inheritance ? If we need to express interfaces as we saw in section 3.5.2, multiple inheritance may be the solution.

Listing 3.78: C++ interfaces multiple inheritance, simplified code

```
class DoSomething {//this is an interface
    void something();
};

class DoSomethingElse { //another one
    void somethingElse();
}

//this is a concrete class matching both DoSomethingElse
// and DoSomething interfaces, and this is perfectly fine,
//this way, DoAll can be used in a polymorphic way as being
//both of them
class DoAll : DoSomethingElse, DoSomething {
    void something(){
        //...
    }
    void somethingElse(){
        //...
    }
}
```

Fortunately, C++ has specific rules for interfaces-like classes, containing only methods declarations and no fields nor method implementations, avoiding multiple inheritance related problems for interfaces[57]. Other languages may have other solutions, for example Java differentiates to entities, *interface* and *class*[58][59]. A class can only inherit from one class but implement multiple interface, allowing for flexible and yet simple polymorphism.

The other case where multiple inheritance shows its interest, is when we have a composition, as presented section 3.5.3, Composition : animals better than with inheritance, that is recursive. In such a case, the composite class also wants to match component's interface. This can easily be demonstrated through Python ducktyping.

Listing 3.79: Python recursive composition

```
Class Component1 :
    #...
    def method_one(self):
        #...
Class Component2 :
    #...
    def method_two(self):
        #...

class Composite :
    #...
    def __init__(self, component1, component2):
        self.component1 = component1
        self.component2 = component2

    def method_one(self):
        self.component1.method_one()

    def method_two(self):
        self.component2.method_two()
```

We can see that the composite implementation of methods simply calls for corresponding component's method. This is called *method forwarding.* This is additional work that would not have occurred with multiple inheritance.

Listing 3.80: Python recursive composition using multiple inheritance

```
Class Component1 :
    #...
    def method_one(self):
        #...
Class Component2 :
    #...
    def method_two(self):
        #...

class Composite(Component1, Component2) :
    #...
    # interfaces are already supported
```

To conclude on multiple inheritance, I'll give you my *opinion* on when it is always worth to consider.

When you want to have polymorphism through interfaces, multiple inheritance is worth, and is usually not a choice. Be careful in this case to have as few as possible *implementing* classes in the inheritance tree : only non-interface classes should be never multiply inherited.

When you have a need for composition, and you know that you will not have to increase the number of components nor often modify their inheritance tree, then multiple inheritance is maybe a better choice than composition.

Finally, when creating parts of your software that are not meant to change or be subject to any kind of flexibility, as the very core of your software, multiple inheritance can be good as well.

3.6 Algorithmics and performances

Desktop computers have enough processing power to allow developers to not focus too much on performances of the software and instead improve functionalities. However, some applications can require extreme processing capabilities, or some hardware may be really limited. Both cases are frequent in science field, and being aware of some notions of both low level programming and algorithmics can help improve performances of software.

3.6.1 Cost and complexity of algorithms

When manipulating a large amount of data, we want to use algorithms as efficient as possible. An algorithm is a sequence of instructions to solve a problem. To evaluate how efficient an algorithm is, we look at the number of operations it performs, *time* and how much memory *space* it needs to solve a problem regarding the size of the input. Algorithm 1 is performing multiplication of two 2D square matrices of size N. N is our input size here.

```
Algorithm 1: Naive 2D square matrix multiplication
Data: $A$, $B$, $C$ : 2D square matrices of same size $N$
Result: $C$ is now equal to $A * B$
for $i$ from 0 to $N - 1$ do
    for $j$ from 0 to $N - 1$ do
        $s = 0$ ;
        for $k$ from 0 to $N - 1$ do
            $s = s + A[k][i] * B[j][k]$ ;
        $C[i][j] = s$ ;
```

This algorithm does 3 for loops of N iterations nested in each other, and therefore performs around N^3 operations before terminating. This is the complexity of the algorithm, usually expressed through the big O notation $O(n^3)$, as an asymptotic upper bound, which means it is only relevant with large values of N. Also, it only gives the magnitude of the growth of resource need regarding data, as all constants, even multiplicative, are neglected. This algorithm doesn't have additional memory needs, so its space complexity is $O(1)$: it is independent of the size of the input.

Understanding complexity is useful when selecting an algorithm from a library, as documentation usually specifies the complexity of each algorithm used, and it can give us a fairly easy way to compare their performances, as for example $O(n^3)$ is nearly guaranteed to be slower than $O(n^2)$ when input is large. Complexity of an algorithm may vary depending on the arrangement of the input. Because of this, complexity is often expressed as *best case*, *average* and *worst case*, which gives more precision on how the algorithm performs in certain conditions. Table 3.1 presents the time complexity of few sorting algorithms, reorganizing an array so elements are ordered.

In most cases, optimized bubble sort will be slower than any other sorting algorithm presented here. However, it is trivial to implement, and if the array is nearly sorted, as if elements are close to their real position, it can beat nearly any other sorting algorithm. Radix sort is extremely fast, but it can only operate on elements that can be converted into integers to be sorted. Merge sort and quicksort are well-balanced sort, as they have no constraining requirements and perform well in most cases.

It shows us that all there can be a lot of differently performing algorithms for a given task, and that there is no perfect algorithm : it all depends on context. For example, quicksort is most of the time the default sorting algorithm found in libraries, as it performs fairly fast and does not consume additional space. However, if we can guarantee that the array or list will always be nearly sorted, bubble sort is probably a much better choice. Such situation can occur in physics simulation with slowly moving objects, using a *sweep and prune* approach[60].

3.6.2 Data collection

As we may want to work on large amount of data, we usually want to store them in *collections*. Collections are abstract concepts of data properties and what operation we want to perform on

Name	**Best case**	**Average**	**Worst case**
Quicksort	$O(nlogn)$	$O(nlogn)$	$O(n^2)$
Merge sort	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$
Radix sort	$O(n)$	$O(n)$	$O(n)$
Optimized Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$

Table 3.1: Sorting algorithms time complexity

Operation	**array**	**forward linked**	**doubly linked**
insertion at end	$O(n)$, am. $O(1)$	$O(n)$	$O(1)$
insertion at beginning	$O(n)$	$O(1)$	$O(1)$
insertion at known position	$O(n)$	$O(1)$	$O(1)$
finding element	$O(n)$	$O(n)$	$O(n)$
finding element, list sorted	$O(logn)$	$O(n)$	$O(n)$
random access	$O(1)$	$O(n)$	$O(n)$
removing last	$O(n)$, am. $O(1)$	$O(n)$	$O(1)$
removing first	$O(n)$	$O(1)$	$O(1)$
removing at known position	$O(n)$	$O(1)$	$O(1)$

Table 3.2: List operations complexity regarding implementation, n is the number of elements, *known* means we have a pointer to the element. *am.*, for *amortized*, means that when performing a lot of the same operation, it will look like they have this complexity

them. They include :

- **List** : a sequence of elements keeping trace of their insertion order.
- **Map** : a collection of key-value pair where keys are unique.
- **Set** : a collection of unique elements, they can be seen as a map where values are key as well.
- **Stack** : a special list where we retrieve elements in the reverse order we added them.
- **Queue** : a special list where we retrieve elements in the same order we added them.
- **Priority queue** : a special list where we want to have access to max and min elements.

Their implementation are data *structures*. We already have seen two structures in Python in this chapter : list, which implements a list, and dictionary, implementing a map. Different methods can be chosen to implement a collection, leading to different performances regarding operations we can perform on them. For example, a list can be either represented as an array, which stores elements side by side in memory, as a forward linked list, where each element has a pointer to the next, or as a doubly linked list, where each element has a pointer to the next and the previous. Table 3.2 shows expected complexity of some common list operations regarding list implementations. We can see that linked implementations are overall better performing in insertion and removal, while array implementation is overall better for accessing elements. It comes from the fact that arrays, storing elements side by side in memory, allows to access an element : as we know the address of the first element, to access second element we simply add 1 to the address of the first element. In our shelf metaphor from section 3.4.3, it would mean that having an array starting at address A2, to access next element we need to access element at address A3.

Knowing the existence of different data structures in the languages and libraries/modules/-packages you work with is important, as it helps to select the one fitting your need the most. It can usually be found in documentation.

3.6.3 Recursion vs Iteration

Most programming languages allows some entities as classes or functions to refer to themselves, directly or indirectly, in their definition. This is called *recursion.* Recursion can sometimes be really intuitive to use, as seen in listing 3.73 in which composite pattern allowed us to recursively define a folder architecture. A lot of data structures and algorithms can be expressed recursively, as for example a list and a search algorithm.

Listing 3.81: Recursive list and search

```
#a list is an element and a sublist. The sublist is a list as well.
class List :
    def __init__(self) :
        self.value = 0
        self.sublist = None #none allows us to tell sublist has "no value"
    #appending is recursive as well, it self calls
    def append(self, value) :
        if self.sublist == None : #if no sublist (we are on last element)
            lst = List()
            lst.value = value
            self.sublist = lst #we create the new element
        else :
            self.sublist.append(value) #we append to sublist
    #search algorithm, recursive, returns List corresponding to element, starting
        ↪ from the sublist element
    #it starts from sublist element so we can use first element as an empty one, as
        ↪  if it had no value
    def find(self, value) :
        if self.sublist == None :
            return None #element has not been found
        else :
            if self.sublist.value == value : #found
                return self.sublist
            else :
                return self.sublist.find(value) #recursion occurs here

#Main code
#creating a list :
lst = List() #list is empty []
#appending element
lst.append(1) #list has value [1]
lst.append(2) #list has value [1, 2]
lst.append(3) #list has value [1, 2, 3]
lst.append(4) #list has value [1, 2, 3, 4]

res = lst.find(3) #res is a list containing [3, 4]
priny(res.value) #prints 3
```

Recursion usually leads to lower performance than iteration, because of call stack. In a computer, there are two types[4] of memory : *heap* and *call stack*. Stack is, as its name indicate, a stack data structure. Call stack is necessary to allow recursion. When we call a function, its parameters and local variable are piled on the stack, and when we return, they are unpiled, as seen on figure 3.5. It allows functions to be called multiple times recursively while guaranteeing that their variables are local. A recursive algorithm can rapidly consume all memory of the stack since it eventually makes thousands or millions function calls. This has to be kept in mind when using recursive approach. Some recursive algorithms can be optimized by the compiler or interpreter to not use stack, with methods as *tail call optimization*. Whether compilers or interpreters performs such optimizations is usually well documented for major languages.

3.6.4 Programming related performance issues

Some performance issues in software can come from how computers works, and from the fact that languages hides those underlying mechanisms. First, not all operations in a language are equal, and same operations can have different cost from one language to another. A function call is part of costly operations. It comes from the fact that when calling a function we have to move a lot of variables on the stack, as we have seen in previous section, Recursion vs Iteration. Even more costly operations are *system calls*. System calls are an abstraction made by the operating system to allow programs to interact with hardware and other programs within the system. Opening a file is a system call. Printing something on terminal is a system call. Using a networking library

[4]Actually, only one type used in two different manners.

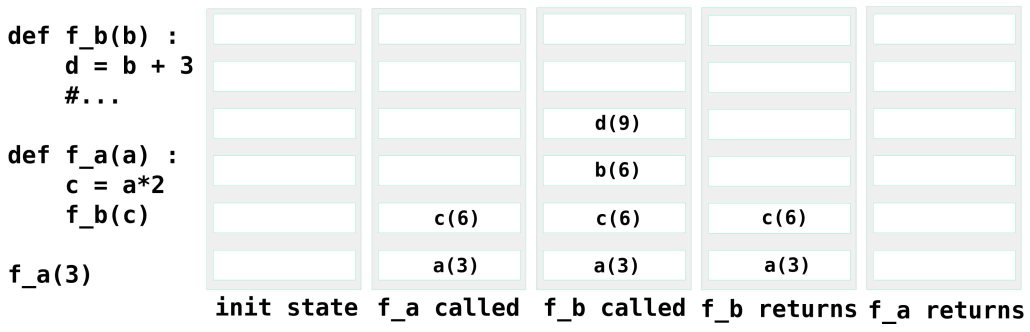


Figure 3.5: Changes on call stack during function call

to send data over internet leads to system calls. System calls can be really expensive because of a lot of security mechanisms, as well as the need for operating system to allow programs to access those functionalities at the same time. In order to improve performances, it is advised, *but not always true*, to reduce the number of system calls we perform. When reading from a file, it can mean reading the file once to load it in memory and then processing its data instead of loading data during processing.

Listing 3.82: Reducing system calls (invalid code)

```
#this syntax is pseudocode, not real python
#this function does a lot of system calls
def lot_of_system_calls() :
    file = #...
    file.open() #one syscall
    while(file.notFinished()) :
        line = file.getline() #one syscall per iteration
        process_line(line)
    file.close() #one more syscall

#this function does only few system calls
def lot_of_system_calls() :
    file = #...
    file.open() #one syscall
    data = file.read() #two syscalls
    file.close() #three syscalls
    while(file.notFinished()) :
        line = getline(data)
        process_line(line)
```

Memory allocation can also lead to system calls depending on language used. A memory allocation is done when reserving memory for use. It we reuse the shelf metaphor from section 3.4.3, we can picture the memory of the computer being an empty shelf, and allocation would be creation of a named box. Allocation can be done by two different ways : on stack and on *heap*. Allocations on stack are done by functions, for local variables, arguments and return values, they are usually really fast. Heap allocations are done when creating variables whose size can not be known at compile time. The size of a variable is the number of bytes it needs to encode its data. For example, an array of 7 integers will usually use 28 bytes, as integers typically use 4 bytes each. Languages offer either explicit or implicit choice between stack and heap allocations. C++ is explicit. Python is implicit, objects are usually allocated on heap, as well as lists and dictionaries, but some primitives types as integers may be allocated on stack.

Listing 3.83: C++ heap vs stack allocation

```
int function(int b) { //b will be allocated on stack, returned int as well
    int a; //a is allocated on stack
    int* c; //the * means that c is a pointer to an int type value.
            // c, the pointer, is allocated on stack
    c = new int(0); //creating a int type value on heap, using "new" operator. the
        ↪ returned value
                 //is a pointer to the value, which we store in c.
    delete c; //this is for memory management, see below in this section
    return 0; //return value is allocated on stack
}
```

Heap allocations are in most cases much slower than stack allocation, as a system call has to be performed. However, not using heap can prove to be really hard, as it is rare to know the size of all variables we need at compile time. Therefore, reusing already allocated objects instead of allocating new ones can be a wise choice, as for example by using the *object pool pattern*.

Listing 3.84: Python reuse versus reallocation

```
class Vector2D :
    def __init__(self, x, y) :
        self.x = x
        self.y = y

def realloc_object() :
    vec = Vector2D(1, 2)
    #using vector
    vec = Vector2D(2, 3) #by constructing object again, we perform a second
        ↪ allocation

def reuse_object() :
    vec = Vector2D(1, 2)
    #using vector
    vec.x = 2 #instead of reconstructing another object,
    vec.y = 3 #we simply change values of the fields
```

In order to use as few memory as possible, we need to be able to deallocate, or *free*, memory when it is not needed anymore. If memory was a shelf, it would mean removing a box from the shelf to tell this space can be used by another box. Stack memory is trivial to deallocate : it is deallocated when corresponding function call returns. For heap memory, two approach are possible. *Automatic* memory management, as in Python, keeps a track of whether memory allocated is still in use or not and deallocates it if needed. They are different algorithms and methods for that, depending on languages, compilers and interpreters. *Manual* memory management, as in C++, lets the programmer explicitly decide when he does not need memory anymore. Manual memory management can be extremely powerful when done the right way, as it will keep memory usage minimal. Automatic memory management is most times slower and leads to higher memory consumption, but offers a safety issue, as memory leaks aren't possible. A memory leak happens when an object is allocated, and every pointer and reference to it are lost : it cannot be used anymore but continues to exist, taking space for nothing. A memory leak on our shelf would be having a box with an unknown owner which we don't know whether we can remove it or not.

Listing 3.85: C++ memory leak

```
int memoryLeak() {
    int* c = new int(0); //allocating an int
    return 0;
    //when the function returns, c pointer will be removed from stack,
    //the int value it points to will be lost : memory leak occurs
}

int noMemoryLeak() {
    int* c = new int(0); //allocating an int
    delete c; //we deallocate int pointer by c : no memory leak
    return 0;
}
```

A lot of manual memory management languages gives way for programmer to avoid memory leak with mechanisms in between automatic and manual memory management. For example, C++ uses *smart pointers*, as *unique pointers*. A unique pointer is a pointer that automatically deallocates memory it points to when it is deallocated itself.

Listing 3.86: C++ unique pointer

```
int unique_pointer() {
    unique_ptr<int> pointer; //we declare an unique pointer to int, allocated on
        ↪ stack
    pointer = make_unique<int>(0); //we allocate memory on heap for int and give
        ↪ address to our pointer
    return 0;
    //when the function returns, pointer will be deallocated, and will
        ↪ automatically
    //deallocate the memory it points to : no memory leak occurs.
}
```

While automatic memory management offers a bit lower performances regarding speed and memory consumption, they offer a much simpler and straightforward way to program than manual memory management. As memory management differs from one language to another, it is wise to learn how the languages we use handle memory management, would it be automatic or manual, to adapt our programming style in order to get the best performance possible. The choice between automatic or manual memory management, as well as the level of optimization is a choice between performance versus clarity and simplicity, and will be discussed in the next chapter, Code quality.

Chapter 4

Code quality

Code quality is a subset of software quality. Software quality aims to produce software that is the best to perform its goal both in the present and in the future. It is joining multiple domains as testing, architecture, programming, ..., as well as the application domain the software is designated to. Software quality can be seen, in a certain way, as the ultimate goal of any developer. In the chapters before, we already had a glimpse of some concerns found in software quality, as for example the ability to add functionality to the program using nice architecture practices. In this course, we only discuss directly two aspects of software quality, code quality and testing. Some other aspects, as architecture, have been, or will be, introduced as secondary concerns of other subjects.

While software quality evaluates how a software fits its purpose on a global scale, code quality focuses on how code fits its purpose, which is to describe the software's behavior and structure in a way that is both understandable, efficient and reliable.

4.1 Motivations

Bringing or keeping a software at high code quality can feel like a hassle. Therefore, before discussing code quality, it is important to discuss *why* we want it to be as high as possible.

4.1.1 Understandability

The reason that rules them all is understandability. It is a combination between readability and clarity. Readability is inversely proportional to the amount of time and energy a computer or a human has to spend to formulate hypothesis on the meaning of the code. Clarity is inversely proportional the number of *reasonable* hypothesis we can formulate. Of course, we want both readability and clarity to be as high as possible.

Listing 4.1: Python, high readability, high clarity

```python
def divide_and_add(a, b):
    return a / b + a

#the code is fairly short and simple and therefore has a high readability
#only one hypothesis is reasonable (we divide a by b and then add a), so it is
    ↪ clear
```

Listing 4.2: Python, low readability, low clarity

```python
def special_operation(number1, number2_nonzero):
    return operation2(operation1(number1, number2_nonzero), number1)

#the code is hard to read, containing nested function calls
#and long identifiers as number2_nonzero, readability is low
```

```
#it is hard to understand straight what this code actually does without
#knowing what are operation2 and operation1, which are a bit abstract names,
#and therefore its clarity is low as well.
```

Understandability isn't exactly the same looking from the perspective of a computer or the one of a human.

4.1.2 Computer's perspective

The top reader of your code is likely to be a compiler or a static analysis tool. We summarize them by calling them a *computer* or *compiler*. A computer usually only see your code as a collection of directives expressed using the language's syntax and semantics. They can understand *what* you are doing, not *why*. Listing 4.3 shows what a computer sees. Please remember it is only a metaphor to picture how a computer sees the code, not how it is exactly processed and analyzed.

Listing 4.3: Python computer's reading example

```python
#comments shows what computer sees

students = ["Marc", "Remy", "Maxime", "Sophia", "Elodie"]
# create a list L and add 5 string values to it :
# "Marc", "Remy", "Maxime", "Sophia", and "Elodie"

# print a string whose value is "here are students"
print("here are students")
#iterate over the list L
for student in students :
    #print the each element of the list L
    print(student)
```

We can notice that variables and functions names have no meaning for a computer. The computer understands that we create a list, that we fill it with some string values, that we then print a string and that finally we display each element of the list. However, he doesn't understand we are working on a list of *students*. Any meaning passed through elements the compiler can not understand, as names or comments, are not taken in account by the compiler. It is therefore unable to detect and warn if those ship an incoherent meaning.

Listing 4.4: Python computer's incoherence acceptance

```python
def paint(wall):
    #paint stuff

#this triggers an error, the computer warns that "paint" only has one
#argument while you called it with two :
paint(Wall(), Wall())

#however, this does not triggers a warning during analysis :
#the computer does not understand that painting a student is a bit strange
paint(Student())
```

Computers only analyze your code based on non-ambiguous rules. Those rules are more or less the language's syntax and semantics definition. It is normal to proceed like this since we don't want the compiler to guess what to do, otherwise programming would be much harder and software probably less reliable.

Finally, computers have virtually infinite processing power and don't get tired. They can analyze syntactical constructions that would be nearly impossible for us humans to understand.

Listing 4.5: Python easy for computers to read

```python
#a computer do not struggles more to read this
a = funct1()
b = funct2()
c = funct3(a + b, b)
```

```
res = funct4(a + c)

#than this
res = funct4(funct1()+funct3(funct1()+funct2(),funct2()))
```

4.1.3 Human's perspective

While it is fine for a computer to only understand *what* to do when reading code, humans need to understand *why* doing it as well. Even worse, we need to have the *why* to better understand the *what*. This may come from the way our natural languages work, heavily using context to give meaning. The sentence *"If you are nasty I'll eat you tonight !"* carries a completely different meaning whether you say it with a malicious smile to your lover, or with appetite to an apple. Please note that if you talk to apples, how humans read code should maybe not be your main concern. Back to original topic, it is faster to understand some code if you know why it is doing things before wondering what it does.

Listing 4.6: Python contextual why

```
def hypothenuse(triangle):
    return sqrt( pow(dist(triangle.a, triangle.b)) + pow(dist(triangle.a, triangle.
       ↪ c)) )

# when reading the name of the function "hypothenuse", and the parameter, "triangle
    ↪ ",
# you already expect finding something of the form sqrt(AB + AC)
# when reading the body of the function, you only confirm or refute several
    ↪ hypothesis
# you made on what the code does

def something(param1, param2):
    return param2 / dist(param1.a, param1.b) * dist(param1.a, param1.c)

# when reading neutral names like something, param1 and param2,
# you have to decipher the formula and then try to understand what exactly it does
# to finally get the why. With the why first it becomes much simpler

def thales_ac(triangle1, ab):
    return ab / dist(triangle1.a, triangle1.b) * dist(triangle1.a, triangle1.c)
```

Like for natural languages, human *learn* how to read code. As a metaphor, we can consider that the syntax and semantics of a programming language is a natural language, while how each developer organize their code and name things is a dialect of this natural language. In any cases, This implies that the more a human reads code with a certain paradigm, language, and presentation, the faster he will read similar code.

In the end, humans have a limited processing power, they can only remember and analyze a tiny chunk of code and context at once. The more processing power they use, the faster they get tired.

Listing 4.7: Python hard for human to read

```
#a human do not have hard time reading this
a = funct1()
b = funct2()
c = funct3(a + b, b)
res = funct4(a + c)

#but they will struggle a bit reading this
res = funct4(funct1()+funct3(funct1()+funct2(),funct2()))
```

4.1.4 Efficiency

Code efficiency is about solving a problem with code as small and simple as possible. Size of the code is fairly simple to measure, it is the number of lines, keywords and identifiers, or instructions of our code. Code complexity, or more precisely code *cognitive complexity*, opposite of simplicity, *not to be confused with algorithmic complexity*, is much more complicated to evaluate. We could define it as being the minimum amount of processing power needed to analyze and understand the code. Efficient code is faster to write and read.

Listing 4.8: Python efficient code example

```python
# this code is fairly efficient :
list = [1,2,3,4,5]
for elem in list :
    print(elem)
print("end")

#this code does the same but is less efficient
endstr = "end"
list = [1,2,3,4,5]
i = 0 #endstr, list
while i < len(list) :
    print(list[i])
print(endstr)
```

Looking at the previous listing, we can try to measure its complexity. Let's consider that we read the code following the flow of the program, and then let's try to see how much we need to remember to understand the code. In order to do that, we read the code backward and each time something is used, we remember it until we find its definition.

Listing 4.9: Python code complexity calculation example

```python
# this code is fairly efficient :
list = [1,2,3,4,5]      #4 : nothing (found list !)
for elem in list :      #3 : list (found elem ! )
    print(elem)         #2 : elem
print("end")            #1 : nothing

#this code does the same but is less efficient
endstr = "end"          #7 : nothing (found endstr !)
list = [1,2,3,4,5]      #6 : endstr, (found list !)
i = 0 #endstr, list     #5 : endstr, list (found i !)
while i < len(list) :   #4 : endstr, list, i
    print(list[i])      #3 : endstr, list, i
    i += 1              #2 : endstr, i
print(endstr)           #1 : endstr
```

We can evaluate the complexity of the efficient code to be 1, and the one of the inefficient version to be 3. Of course, this is one way of measuring complexity. We could take also in account the distance between usage and definition, and therefore `endstr` would be counted as 6 or 7, or include functions calls, as they require to have knowledge of what the function does. Finally, we could have excluded usage of things that are really expressive, as `endstr` could be sufficiently self-explanatory to not have to see its definition to understand it. There are plenty more ways of measuring cognitive code complexity [61].

Code efficiency has a subjective aspect, and better than having a pure definition, the important matter is to understand the idea behind it and remember those complexity and size concerns.

4.1.5 Maintainability

We don't want our code to simply be easy to write and read, we also want it to be easy to edit. That's what maintainability is about. It depends on the quality of the code, architecture of the software, and tests. Edition can happen for a lot of reasons : enhancing functionality, extending the software, fixing a bug, ...

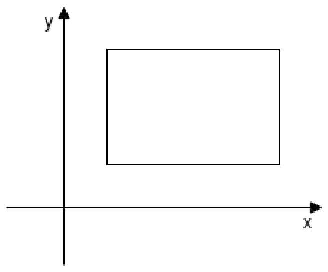


Figure 4.1: A 2D Axis Aligned Bounding Box is simply a rectangle with edges parallel to axes

We can evaluate maintainability by looking roughly at two things. First, how much edition I need to change or add something ? Second, when editing, to what extent I may introduce errors in my software ?

Let's have a quick look at two different Axis Aligned Bounding Box implementations, listings 4.10, 4.11 and how they react to getting from 2D to 3D modeling. An AABB example is shown figure 4.1. In both cases the AABB will be represented by an origin and an end.

Listing 4.10: Python maintainability example version 1

```
#first version, remember AABB = axis aligned bounding box
class AABB():
    def __init__(self, begin, end) :
        self.begin = begin
        self.end = end

    def center(self) :
        return (begin + end) / 2

    def translate(move)  :
        self.begin += move
        self.end += move

#creation of an AABB
aabb = AABB(Vec2D(1,2), Vec2D(-2, 3))
```

As this first version represents its origin and end by a vector class and that all operations inside the class are related to the vector class, we could switch from 2D to 3D simply by passing compatible 3D vectors to the constructor : `AABB(Vec3D(1,2,3), Vec3D(-2, 3, -4))`. On the other hand, if we need to edit other parts of our code to change where the `AABB` class is used, like a `RigidBody` we may forget to change the vectors from 2D to 3D there and end up with bugs difficult to track. One possible fix would be to check that both vectors have the same dimension when building the box, or even to check for a fixed dimension (2 or 3).

Listing 4.11: Python maintainability example version 2

```
#second version remember AABB = axis aligned bounding box
class AABB():
    def __init(self, x0, y0, x1, y1) :
        self.x0 = x0
        self.y0 = y0
        self.x1 = x1
        self.y1 = y1

    def center_x(self) :
        return (x0 + x1) / 2

    def center_y(self) :
        return (y0 + y1) / 2

    def translate(x, y):
        self.x0 += x
        self.y0 += y
        self.x1 += x
        self.y1 += y

#creation of an AABB
aabb = AABB(1, 2, -2, 3)
```

This second version is a more straightforward implementation and encodes origin and end directly by coordinates values. It takes a lot of editing to go from 2D to 3D, as nearly all the methods need to be changed. However, because we will change the constructor as well, by adding `x2` and `y2` parameters, any code making usage of our `AABB` class and that we forget to convert from 2D to 3D will trigger an error at compile time. An in between solution would therefore be to mix both implementations, constructing the class using raw coordinates but representing the AABB using vectors.

Listing 4.12: Python maintainability example version 3

```
#last version, remember AABB = axis aligned bounding box
class AABB():
    def __init(self, x0, y0, x1, y1) :
        self.begin = Vec2D(x9, y0)
        self.end = Vec2D(x1, y1)

    def center(self) :
        return (begin + end) / 2

     def translate(move)  :
        self.begin += move
        self.end += move

#creation of an AABB
aabb = AABB(1, 2, -2, 3)
```

Maintainability has no perfect solution, as each case needs a specific study and depends on how you can anticipate changes. Code quality plays a really important role, since you will often have to get back into code you wrote months or even years ago with a minimum of wasted time and effort. A common example for such a change is adapting your code to new technologies or libraries, either to improve performances or to fix a newly occurring bug.

4.1.6 Reusability

Science advances because we don't have to rediscover and reinvent everything all the time. We reuse the work of other people as the foundations or our own. Software development is no different. Two different software usually share a lot of common parts, even more if they belong to the same application field. A physics simulation library is likely to need a constraint solver at some point. A software used for simulation and experimentation probably needs to plot data in a way or another. Better not implementing twice something you could implement only once. There are several reasons for us to prefer reusing over implementing again.

- Reusing the software piece as-is, or with few adaptations, saves a lot of conception and programming time.
- Reusing only algorithms or architecture is also an option, saving conception time.
- Code we reuse is, supposedly, already tested and functional.
- Code we reuse is, supposedly, already documented, and colleagues may be familiar with it.

Reusability simply measures how easy it is to take a part of a software into another software, and we can model this process by two steps.

In the prediction step , we try to evaluate how much the software piece fits our needs, decide to use it or not, and how we will use it. *Is there something interesting in this software for us to reuse ? How much we have to adapt the piece of code for it to fit in our software ? How much we have to adapt our software for the piece of code to fit into it ?* The prediction step is easier if the piece of code and its documentation are simple to understand.

In the execution step , we reuse the software in our code, given the prediction step produced a positive conclusion. This step heavily depends on how accurate and complete our understanding of the piece of code was in the prediction step. If for some reason we missed an important point in our prediction step, we may lose a lot of time, either in adaptation or because we have to give up the reuse midway.

The best case for reusability is to simply take a piece of code from one software and drop it in another software for reuse. This is one of the numerous reasons why people split their code into packages or modules and create libraries.

4.2 Guidelines

This section explores *good practices* that can enhance your code quality. It is by no means the absolute truth, only guidelines. Most programmers will agree on the majority of the points mentioned here, but they still are subject to discussion and adaptation. Each guideline presents a definition stating what it is, a rationale stating why you should follow it, and finally its limits. Methods and tools to enforce these guidelines will be discussed in section 4.3. In the rest of the section, we will have to describe code including comments, and therefore we will use *double comments.*

Listing 4.13: Python double comments

```
#this is a regular comment, noted by #, it shall be interpreted as being part of
    ↪ the code
##this is a double comment, noted by ##, it serves to describe what we do,
    ↪ including regular comments
```

Listing 4.14: C++ double comments

```
//this is a regular comment, noted by //, it shall be interpreted as being part of
    ↪ the code
////this is a double comment, noted by ////, it serves to describe what we do,
    ↪ including regular comments
```

This convention is only a convention internal to this course, don't expect it to be understood elsewhere.

4.2.1 Coherence

Definition

Choices you make in your code, regarding following guidelines, architecture, algorithmics, memory management, or whatever, shall be consistent with other choices you have made before. In a more general way, it is advised to follow the conventions and coding guidelines associated with the language you use. Depending on the language, it can be somehow strict, as Python's guidelines are[62], or permissive, as C++ guidelines are[63]. Please note this course's code doesn't follow any of them.

Additionally, if for some reason you have to step out of your coherence, make it as explicit as possible, either by using a comment or by code itself.

Listing 4.15: Code coherence example, good

```
##good, this code is compliant with official recommendations
```

```
class MyClass :
    def __init__(self):
        #do stuff
    def method_one(self) :
        #do stuff
    def method_two(self) :
        #do stuff


def function_one(argument1) :
    #do stuff


##good, we said that we are breaking the coherence and why with comment
#camelCase here to match XXX library's way
def functionTwo(argument2) :
    #do stuff
```

Listing 4.16: Code coherence example, bad

```
class my_class :  ##bad, not compliant with official recommendations
    def __init__(self)
    def method_one(self) :
        #do stuff

    def methodTwo(self) :  ##bad, inconsistent with method_one and methodTwo
        #do stuff

def function_one(argument1) :
    #do stuff
```

In other terms, it means that rules chosen to write you code shall be easy to understand by reading you code. Otherwise, they have to be explicitly stated somewhere.

Rationale

As mentioned in section 4.1.3, humans tend to learn as they read code. Having a code consistent helps people to get used to your code faster when they read it.

Following totally or partially official or standard guidelines helps people not related to your project understand your code. Such external readers can happen more than what we may think. They can be from a help forum, colleagues, or even other scientists that will read about your work.

Limits

This guideline does not have notable limits. Coherence may seem like it is encouraging keeping bad habits. It is not. If your existing code is bad and you want to change habits in new code you write, then you have to change your existing code base first by performing a *refactoring*, that we will describe in section 4.3.3. If refactoring is too costly and you still want to break coherence, it could be nice to have a clear separation between old and ugly, and new and beautiful codes, as for example through creation of libraries.

4.2.2 Expressiveness

Definition

Your code shall express what you *intend* to do, and this intent shall be expressed as much as possible solely in code and by using language's syntax and semantics. Intent is what you *want* to do, not how you do it. Language's syntax and semantics is everything that the compiler can check and understand before runtime. For convenience, we will refer to this as *syntax* in the rest of this chapter.

Listing 4.17: Pyton syntax intent example, good

```
##good, for counting from zero to 9 we use a for in range loop
##also, the compiler can see easily that we want to do 10 iterations,
##from 0 to 9
for(i in range(10)):
    print(i)
```

Listing 4.18: Pyton syntax intent example, bad

```
##bad, for counting from zero to 9 we a while loop, while a for in range loop
##would express intent better

#counting from 0 to 9 ##comment is bad as well, with for loop no need for it
i = 0
while i < 10 :
    print(i)
    i = i+1

##also, compiler will have a harder time to understand that we want
##to iterate over values from 0 to 9
```

When syntax is insufficient to express intent, naming can be used instead. Variables shall be named regarding what they are used for. Classes, functions and methods shall be named regarding what they are.

Listing 4.19: Python naming intent example, good

```
class Table : ##good, Table represents a table
    def __init__(self):
        self.foot = WoodenStick() ##good, our WoodenStick is used as a foot

    ##good, wash means we wash the table, and detergent clearly expresses that this
    ##argument (variable) will serve to clean the table
    def wash(self, detergent):
        #do stuff
```

Listing 4.20: Python naming intent example, bad

```
##bad, we are describing a table here, even if we can use it as a shelf
class Shelf :
    def __init__(self):
        self.wooden_stick = WoodenStick() ##bad, we have no idea that the wooden
            ↪ stick
                                          ##will be used as a foot
    ##bad, call_when_dirty does not tells us what the method does, and
    ## javel argument tells us a class we can pass, not what the object
    ## will be used for
    def call_when_dirty(self, javel) :
        #do stuff
```

Sometimes, naming and syntax can be insufficient to clearly express intent. Therefore, it shall be done using comments, to clarify the code. It can be especially useful when dealing with math formulas, when we want to keep things as they are presented in the formula, using abstract letters, and yet be explicit. It also is convenient when performing heavy optimization that prevents the code to be fully expressive.

Listing 4.21: Python comment intent example, good

```

u = Vector(...)
v = Vector(...)

##good, we say what we are doing as recognising a cross
##product onsight isn't something anyone can do
#w is cross product of u x v
w = ##complicated formula here
```

Listing 4.22: Python comment intent example, bad

```python
U = Vector(...)
V = Vector(...)

##bad, we do complicated stuff without even mentionning
##what we are doing
W = ##complicated formula here
```

Please note that comment intent can often be replaced by a function or another syntactical entity, and therefore be expressed through naming.

Listing 4.23: Python replaced comment intent example, good

```python

def cross_product(u, v):
    ##complex formula here
    ##good, we know that we are working on a cross product here,
    ##so no need for more precisions

u = Vector(...)
v = Vector(...)

##good, no need for comment, function call is already expressive enough
w = cross_product(u, v)
```

Rationale

An expressive code is easier to understand, both for humans and for compilers or various other tools. From a human perspective, expressive code is simpler to read, and allows detecting mistakes faster when looking at the code as it raises awareness if an ambiguous situation occurs.

Listing 4.24: Python intent mistake detection example

```python
##here, name of the function says we intent to define a cross product
def cross_product(u, v):
    ##however the action we perform clearly isn't a cross product
    return Vector(u.x + v.x, u.y + v.y, u.z + v.z)

##as the situation is ambiguous (is function name wrong or formula wrong ?), we can
    ↪ further investigate.
```

Using syntactical expression of the intent, we also allow the compiler and other tools to warn us in case a situation looks like a mistake, and we enable better and simpler optimization from it. Such syntactical expression can be done for example through types, explained section 3.4. We shall avoid comment when possible. Comments are an additional workload when doing modifications of our code and can make code harder to read when used too much. However, the right amount of comments can drastically increase expressiveness and thus understanding of our code.

Limits

Languages are limiting how much of intent we can express through sole syntax. This forces us to rely on the two intent expression ways (naming and comments) that are more subjective and ambiguous and can be only checked by humans.

Expressiveness is also dependent on the situation. A well-trained mathematician that is used to 3D vector geometry would have little to no problem recognizing a vector cross product formula in code, while this task would be extremely difficult for someone stranger to math. If the code is likely to be read by a beginner, explaining it thoroughly through comments can be a nice thing. If not, over explaining will simply get in the way of experts.

4.2.3 Locality

Definition

Things related to each other shall be put close by. A thing can be a lot of, well, things. As for example a piece of code, a module, a function, a class, ... Two things are related if one is the *context* of another, said differently, if one is needed to understand well the other.

Listing 4.25: Python locality by context example

```python
a = 100 ##this line is needed to fully understand what is a, it is the context
b = a/2 ##this line needs the previous line to be fully understood
##therefore, these two lines are related
```

Two things are related as well if they share a common *topic*, either meaning or use.

Listing 4.26: Python locality by topic example

```python
##dot product and cross product share a meaning (vector geometry), they are related
    ↪ ,
##even if they will maybe never be used for a common thing
def dot_product(...):
    ##...
def cross_product(...):
    ##...

##mesh and rigidbody are used for the same purpose here : a physics simulation.
##however they don't really share meaning as they are related to fairly different
    ↪ fields
mesh = geometry.create_mesh(...)
rigid_body =  physics.create_rigid_body(...)
```

The notion of distance to judge if two things are close by can be either in code, regarding number of lines/files to get through to reach one thing from another, or in scope, regarding how many functions, classes and packages you have to get through to reach one thing from another.

Listing 4.27: Python locality regarding scope

```python
class my_class :
    def __init__(self):
        ##a and b are close by, they are in the same scope
        a = 10
        b = 10

    def do_stuff(self):
        ## a is still fairly close to c since they are in the same class scope
        c = self.a
        d = 10
        return c * d - c

e = 200 ##e is far away from all, as it is out of class scope
```

In regard of this principle, global variables shall therefore be avoided.

Listing 4.28: Python locality example, good

```python
##good, list is passed as an argument, therefore its Guideline is close to us
##and follows the flow of the program
##to check where "list" originates from, we simply have to look where the function
##has been called

def register_student(student, list):
    list.append(student)

## more code here

#main code
list = List()
```

```
register_student("student1", list)
register_student("student2", list)
register_student("student3", list)
register_student("student4", list)
```

Listing 4.29: Python locality example, bad

```
##bad, list is global here
list = List()

## more code here

##bad, since list is global, to know where it comes from we can't simply
##follow back the flow of the program, as "list" never appears in main code
def register_student(student):
    list.append(student)

## more code here

#main code
register_student("student1")
register_student("student2")
register_student("student3")
register_student("student4")
```

Rationale

It is easier to read code as you read a book, going straight forward. There is also a fair limit of how much and how long a human can remember. Keeping related things close to each other helps these points by straightening the reading path and by reducing the time you need to remember things.

Having things packed by topic also helps to find them when you are in the need.

Limits

Locality is a complex matter and can be sometimes really subjective. For example, if we create a tiny graphics library, we can do two packages. `geometry`, containing all 2D and 3D primitives and `rendering`, containing all shaders for rendering. However, another valid approach would be to do two packages, `2D` and `3D`, both containing geometry and shaders, respectively for 2D and 3D.

4.2.4 Concision

Definition

Keep your code short and get straight to the point, use context to give meaning to things. If things have a specific name in the field the code's model refers to, use it. Prefer naming things with plain names rather than abbreviations, except if the plain name is long and abbreviation is easily understood and widely used.

Listing 4.30: Python naming concision example, good

```
## this code supposedly refers to 2D physics simulation geometry

## good, AABB is a faily common abbr. for AxisAlignedBoundingBox
## inheriting from BoundingBox also helps us confirming what it is
class AABB (BoundingBox) :
    ## good, overlapping is a common operation to do with two AABBs,
    ## no need to precise more
    def overlaps(self, other):
        ##good, we can deduce from context that other is another AABB.
        ##even if there is a bit of questionning whether it is a BoundingBox or
            ↪ AABB
```

Listing 4.31: Python naming concision example, bad

```
## bad, AxisAlignedBoundingBox is far too long and has a common abbrv.
## however, still not that bad since we use a "standard"/common name
class AxisAlignedBoundingBox (BoundingBox) :
    ## bad, too many words here, overlaps, overlapTest or such could
    ## have been sufficient
    def checkIfOverlapsWithOtherAABB(self, otherAABB)
        ## bad, the name already contains "AABB", so no need to
        ## tell it again in "otherAABB".
```

Listing 4.32: Python naming concision example, worse

```
## bad, well, horrible. The name is too long and we don't use
## a common name at all
class RectangularAlignedBoundingBox (BoundingBox) :
    ##more code here
```

Use the most efficient syntax construct your language provides, except if the construct is meant to be mostly read by unaware people.

Listing 4.33: C++ syntaxic concision example, good

```
int min(int a, int b){
    //// good, here we use ternary operator (condition ? iftrue : iffalse)
    return a < b ? a : b;
}
```

Listing 4.34: C++ syntaxic concision example, bad

```
int min(int a, int b){
    ////bad, this construct is fairly heavy for no reason, while the
    ////ternary operator could have been used instead
    if(a < b){
        return a;
    } else {
        return b;
    }
}
```

Classes, *namespaces*, packages and other scopes can help provide a clear context to concise code.

Listing 4.35: Python package concision example, good

```
##in package "physics"

## good, as we are in physics package, we know that Body is meant in terms of
    ↪ physics
class Body :
## good, as we know we are in a Body in term of physics, we don't need much
    ↪ precisions within
## the class scope
    def vertices(self):
        # code here
    def rotate(self, quat):
        ## good, we know that quat is a quaternion since we are in a rotation
            ↪ function
        ## that is in a physics body
        # code here
    def translate(self, vec):
        ##good, same for vec
        #code here
    def apply_impulse(self, imp):
        ## good, we know imp is an impulse since we said it in the function name
        #code here
```

Finally, be sure to split your code between classes, functions, modules, etc when it becomes too long.

Listing 4.36: Python code splitting concision example, good

```
## good, code is split in small parts that are fairly short
def average(array):
    sum = 0
    for elem in array :
        sum += elem
    return sum / len(array)

def min(array):
    ## some code here

## other functions here ...

def print_stats(array):
    print("avg = ", average(array))
    print("min = ", min(array))
    print("max = ", max(array))
```

Listing 4.37: Python code splitting concision example, bad

```
## bad, the function is far too long and could be split

def print_stats(array):
    sum = 0
    for elem in array :
        sum += elem
    avg = sum/len(array)
    print("avg = ", average)
    ## here compute min ...
    print("min = ", min)
    ##here compute max ...
    print("max = ", max)
```

Rationale

Having shorter code and names helps the reader read faster and save energy. People reading your code are likely to be aware of its context, so there is no reason to overload the code by reminding the context.

Syntactical shortcuts as ternary operator may be harder to understand for a beginner, yet they are easy and convenient to read by a developer used to the language.

Using standard and common words corresponding to the code's target domain enhances the understanding of people of said domain, which are most likely to be the ones reading your code. Even if they may decrease understanding of unaware readers at first, they will prove to be convenient for them to find more information.

Limits

Concision is excellent to make code more straightforward to understand. However, making code too short and concise can make it really hard to understand, as some people can misinterpret the context or abbreviations you choose. When in doubt, better be not concise that too much, as a long name is only tiring to read and not to understand, while an abbreviation or a short name will be impossible to understand in the worst case.

Concision also makes nearly mandatory to provide context when taking a piece of code out of your project, as for example when asking for help on a forum.

4.2.5 Nonredundancy

Definition

Use the syntactic tools your language provides to prevent writing the same instruction sequence or construct twice. It is often called *factorization*. Most trivial cases involves using functions.

Listing 4.38: Python function nonreduncancy example, good

```
## good, the code for the average will never be written twice.
def average(array):
    sum = 0
    for elem in array :
        sum += elem
    return sum/len(array)

def function1(array):
    avg = average(array)
    ## more code here

def function2(array):
    avg = average(array)
    ## more code here
```

Listing 4.39: Python function nonreduncancy example, good

```
##bad, the same code has been written twice
def function1()
    sum = 0
    for elem in array :
        sum += elem
    avg = sum/len(array)
    ## more code here

def function2()
    sum = 0
    for elem in array :
        sum += elem
    avg = sum/len(array)
    ## more code here
```

Using polymorphism, as templates in C++ or ducktyping in Python, we can ensure even more factorization.

Listing 4.40: Python polymorphic factorization example, good

```
##good, this can work on any element type defining value() method
def average(array):
    sum = 0
    for elem in array :
        sum += elem.value()
    return sum/len(array)
```

Listing 4.41: C++ polymorphic factorization example, good

```
//// please note syntax has been a bit lightened here.
//// good, we wrote the function so it can work on a list holding any type T
//// having "+" and "/" operators defined
template <typename T>
T average(list<T> l) {
    T sum = 0;
    for(auto elem : l){
        sum += elem;
    }
    return sum/l.size();
}
//// note that list<T> class is a template, so it factorized code !
```

By composition, classes can be factorized as well.

Listing 4.42: Python factorization by composition example, good

```python
##Image analyzer class, composed of a blur filter and an interpolation algorithm
class ImageAnalyzer() :
    def __init__(self, blur, interpolation):
        self.blur = blur
        self.interpolation = interpolation

    def analyse(self, image):
        ##do stuff here using blur and interpolation

##now we can simply use the ImageAnalyzer with any blur/interpolation combination :
#main code
analyzer1 = ImageAnalyzer(Gaussian(), Bilinear())
analyzer2 = ImageAnalyzer(Bokeh(), Cubic())
```

Rationale

If you have an algorithm or structure written only at one point, it is easier to fix errors and change your code. It can be seen by comparing listings 4.38 and 4.39. Our average calculation isn't perfect, since `return sum/len(array)` will result in an unwanted behavior if our array is empty, because of dividing by zero. No matter how we want to handle this problem, would it be by giving a specific average value for an empty array or reporting an error, in listing 4.38, we have to change the code at only one place, while in listing 4.39 we have to edit the body of two functions.

Factorization also helps you write code faster, as effort put into it allows you to never again write that sequence of instructions.

Limits

Factorization can sometime, through the abstraction it requires, remove part of the *intent* of your code and have the opposite effect on code flexibility : introducing changes into a generalist piece of code may break some components using it. Let's say we want to perform two operations on arrays. The first one is *ordering* elements regarding a certain hierarchy. The second one is *packing* elements, grouping them by close properties. Both can be done using a sorting algorithm. Ordering is trivial, while packing requires sorting elements according to a specially constructed value representing their properties.

Listing 4.43: Python factorization limits example, bad

```python
##here compare will give us the rule, it is an object with a method comp(a, b)
##that returns true if a < b
def sort(array, compare):
    ##sorting code here

def use_ordering(array):
    ##this method uses ordering on the array at some point
    sort(array, AscendingOrder())
    ## more code here

def use_packing(array):
    ##this method uses packing on the array at some point
    sort(array, PackingRule())
    ## more code here

##more functions using either ordering or packing below
```

Now, let's admit we found a better algorithm for packing than a simple sort. For example a hash based algorithm that would get the complexity from $O(nlogn)$ to $O(n)$. We either have to create a `pack` function and then change the code of every method using packing, or change the code of sorting at the risk or breaking every function using ordering. This dilemma happened because while factorizing *ordering* and *packing* by abstracting them into *sorting*, we lost trace of our *intent*. This is the reason why this guideline is called *nonredundancy* rather than *factorization*.

The problem can be simply solved by creating an *alias*, a function simply calling sort, but still expressing the intent behind our call.

Listing 4.44: Python factorization limits example, good

```python
##here the compare will give us the rule, it is an object with a method comp(a, b)
##that returns true if a < b
def sort(array, compare):
    ##sorting code here

##those two functions below acts as an alias for sort, expressing better
##the intent behind the sorting.
def order(array):
    sort(array, AscendingOrder())

def packing(array):
    sort(array, PackingRule())

def use_ordering(array):
    ##this method uses ordering on the array at some points
    order(array)
    ## more code here

def use_packing(array):
    ##this method uses packing on the array at some point
    pack(array)
    ## more code here

##more functions using either ordering or packing below
```

Finding the right balance between factorization, aliases and non factorized code can be tricky, and often no perfect solution exists. The choice lies on how the developer anticipates the evolution of his code and how much he wants to do that.

Factorization may also result in performances issues, as unspecialized code may have lower performance than specialized code on specific cases. A generalist sort algorithm will probably use either quicksort or merge sort, presented in table 3.1, having an average complexity of $O(nlogn)$. When handling integer values, radix sort also presented in table 3.1, would perform better with an average complexity of $O(n)$.

4.2.6 Clear errors

Definition

Treat errors as early as you know about them. An error is a situation preventing you from continuing the normal course of processing. For example, the definition of an array's average value don't forcibly takes in account arrays that contains no elements. This can lead to an error. In the ideal case, the error is recovered, which makes the error a special behavior case instead of an error. It is advised to handle errors as soon as we can check for them in the code.

Listing 4.45: Python early error recovery example, good

```python
def average(array):
    #good, error is handled as soon as we know it can exist
    if len(array) == 0 :
        return 0 ##we recover by returning an arbitrary value
    sum = 0
    for elem in array :
        sum += elem
    return sum/len(array)
```

Listing 4.46: Python early error recovery example, bad

```
def average(array):
    sum = 0
    for elem in array :
            sum += elem
    #bad, error code is handled only now while it could have been handled earlier
    if len(array) == 0 :
        return 0
    return sum/len(array)
```

Listing 4.47: Python early error recovery example, worse

```
def average(array):
    sum = 0
    for elem in array :
        sum += elem
    #horrible, error code isn't handled at all
    return sum/len(array)
```

If an error can not be recovered directly, we have to inform the developer using our code and let him recover the error as he can.

Listing 4.48: Python error recovery delegation example, good

```
def average(array):
    #good, we inform the caller that something went wrong
    if len(array) == 0 :
        return False, 0 ##by returning False, we tell that we were not able to
            ↪ perform
    sum = 0
    for elem in array :
        sum += elem
    return True, sum/len(array) ##by returning True, we tell that all went right

##note that True or False meaning isn't a convention and thus has to be documented
```

When the probability of an error occurring is really low, we can eventually delegate the checking to the developer, using documentation. In that case, it is mandatory to ensure, if possible, to give a clear message error if the requirements are not met. In Python, it is done through *assertions*, that allow to exit the program if a condition is not met while printing a message and displaying file and line of the assertion.

Listing 4.49: Python error checking delegation example, good

```
##Good, we mention in doc that it accepts only array > 0
def average(array):
    """This function returns average of the values in the array with len(array) > 0
        ↪ """
    ##Good, we ensure that the condition is met and inform the developer if not
    assert len(array) > 0 "array shall have at least one element to compute average
        ↪ "
    sum = 0
    for elem in array :
        sum += elem
    return sum/len(array)
```

Python assertions, as in many languages, state to the compiler that this code is only a safety check, and thus that if we require performance, as in the production version, it can ignore it. Certain languages, Python included, support more mechanisms, as *exceptions*, and it is up to the developer to learn about them and use them properly while learning a language.

Assertions and exceptions allow *clean termination* in the worst case : if the error occurs and isn't handled, the program crashes but at least we know *why* it crashed.

Rationale

Handling errors as soon as possible gives a clear separation between specific code and main purpose, and prevents doing useless computing that would be rendered invalid by the error anyway.

Delegating error handling to the developer gives more flexibility by replacing the error handling in the context it occurs at.

Delegating error checking to the developer lightens our code and allows developer to factorize error checking.

Listing 4.50: Python factorized error checking

```python
def array_transformations(array)
    if len(array) == 0 :
        ##error here

    ## it already has been checked for length so no reason to recheck
    avg = average(array)
```

Each of these three approaches shall be chosen independently for each error that has to be handled in your code.

Limits

Handling errors is mandatory, and limits of this guideline are only secondary. Choosing between one of the three approach can lead to drawbacks.

Handling errors directly may result in a lack of flexibility leading to re checking the error at some point.

Listing 4.51: Python direct error handling drawback

```python
def average(array):
    if len(array) == 0 :
        return 0 ##returned value defaults to 0
    sum = 0
    for elem in array :
        sum += elem
    return sum/len(array)

def my_function(array):
    ##here we don't want default to 0, so we recheck anyway
    avg = average(array)
    if(len(array) == 0) :
        #...
```

Delegating error handling may result in redundancy and a heavier code due to error handling code.

Listing 4.52: Python error handling delegation drawback

```python
def average(array):
    if len(array) == 0 :
        return True, 0 ##by returning True, we tell that an error occurred
    sum = 0
    for elem in array :
        sum += elem
    return False, sum/len(array) ##by returning False, we tell that all went right

def my_function(array) :
    err, avg = average(array)
    if ok :
        ##error handling
    ##other stuff

def my__other_function(array) :
    ##we write kind of the same code twice here !
    err, avg = average(array)
    if err :
        ##error handling
    ##other stuff
```

Finally, delegating error checking may result in unhandled errors if developers forget to perform checking, and thus it makes the code harder to check.

Listing 4.53: Python error checking delegation drawbacks

```
##Good, we mention in doc that it accepts only array > 0
def average(array):
    """This function returns average of the values in the array with len(array) > 0
        ↪ """
    ##Good, we ensure that the condition is met and inform the developer if not
    assert len(array) > 0 "array shall have at least one element to compute average
        ↪ "
    sum = 0
    for elem in array :
        sum += elem
    return sum/len(array)

def my_function(array):
    avg = average(array) ##OOPS, forgot to check !
```

4.2.7 Optimization is an alternative

Definition

When coding, prefer doing trivial and nice code over performant code. If non-trivial code is required for performances issues, try to provide this code as an alternative of an existing trivial code.

Listing 4.54: Python optimized version example, good

```
## good, we have a bubble sort here, trivial implementation
def bubble_sort_trivial(array):
    for i in range(0, len(array) -1) :
        for j in range(0, len(array) -1) :
            if(array[j] < array[j + 1]) :
                tmp = array[j]
                array[j] = array[j+1]
                array[j+1] = tmp


## good, we have the optimized version, non trivial, in a different function.
def bubble_sort_optimized(array):
## we suppose the code is ugly (well, it could be worse)
    permutated = True
    i = 0
    while i < len(array) and permutated :
        permutated = False
        for(j in range(i, len(array) -1)):
            if(array[j] < array[j + 1]) :
                tmp = array[j]
                array[j] = array[j+1]
                array[j+1] = tmp
                permutated = True
```

Listing 4.55: Python optimized version example, bad

```
## bad, the optimized version is the main version
def bubble_sort(array):
## we suppose the code is ugly (well, it could be worse)
    permutated = True
    i = 0
    while i < len(array) and permutated :
        permutated = False
        for(j in range(i, len(array) -1)):
            if(array[j] < array[j + 1]) :
                tmp = array[j]
                array[j] = array[j+1]
                array[j+1] = tmp
                permutated = True
```

Then, using an alias function or class allows us to choose which version we want.

Listing 4.56: Python alternative selector example, good

```
def bubble_sort(array):
    bubble_sort_trivial(array)
    ##or
    #bubble_sort_optimized(array)
```

Rationale

Keeping a trivial version of the code gives a functional reference. When creating software, you first want to do a reliable software first and then make it run fast.

The more complicated a piece of software is, the more it is exposed to a mistake that may result in a bug. Therefore, it is advised to always work on a trivial code version first when applying changes to a piece of code, and then propagate the changes to the optimized(s) version(s) once the trivial version has come to a satisfying point.

Keeping a trivial version also helps to give a nice overview of what is done in optimized versions.

Finally, on short pieces of code, the presence of a trivial version allows to fully rewrite optimized versions in case of changes, as you always have at least a functional version.

Limits

Having two or more pieces of code fulfilling the same purpose means more code to maintain, and the risk of neglecting a version over the others. Optimized alternative shall therefore be used very wisely, when the code can absolutely not be expressed in a clean and trivial way, and that it is meant to evolve a lot in the future. In this matter, bubble sort was an awful example : bubble sort algorithm is fairly trivial even in its optimized form, and we can suppose it would not be modified often.

Also, this principle works well for small pieces of codes but may prove to be harder or even dangerous to set up when dealing with larger code portions. Abstraction can help there, and especially decoupling, as presented in section 3.5.2, Abstraction and interfaces.

4.2.8 Documentation

Definition

Give additional understanding of your code by providing documentation alongside it. Documentation gives additional knowledge about both how to use the software and how it works. In the ideal configuration, your documentation's source shall be close to the code it refers to. Python docstring[64] is an example of such documentation.

Listing 4.57: Python docstring example, good

```
def my_function(myparam):
    """this is a docstring, it can span on several lines and is used
    to document functions and classes in python"""
    ##Stuff here
```

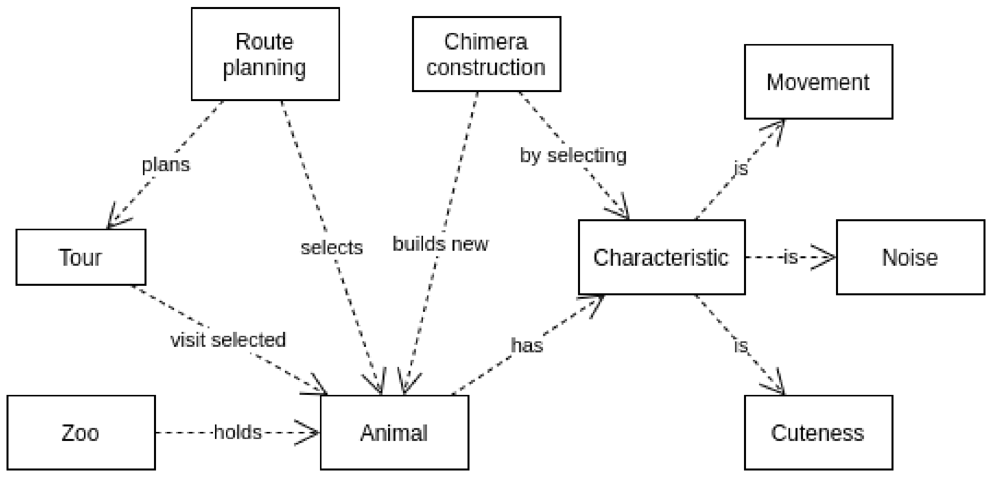


Figure 4.2: Example of a graphical summary of the zoo made at the end of chapter 3

C++, like most languages, do not have such documentation specific comments, and thus developers use conventions of documentation tools, as Doxygen[65][66].

Listing 4.58: C++ documentation comment example, good

```
/* this is a c++ multiline
comment */

/** this is Doxygen style comment to document functions and
other entities, allowing to later generate a documentation
website or PDF */
int myFunction(int a){
}
```

Documentation shall also provide information that are complicated or impossible to guess just by looking at the code. It can be for example giving an overview of the program's structure or principles. When writing a documentation for your software, keep in mind that a nice diagram is often much clearer than a load of text. You can eventually use a standard representation, as UML[67], or simply do a diagram out of the box, as show in figure 4.2.

When documenting something, try to provide examples. Examples are ideally really easy to try by oneself, providing all the needed setup with them.

Listing 4.59: Python quick setup example example, good

```
##good, this example can be copy-pasted into a .py file and run straight out of the
    ↪ box

#this example demonstrates how iterate over a list in Python using foreach
lst = []
lst.append(1)
lst.append(2)
lst.append(3)
lst.append(4)
lst.append(6)
lst.append(5)

# iterating over a list using foreach
for item in lst :
    print(item, " ")

#prints 1 2 3 4 6 5
```

Listing 4.60: Python quick setup example example, bad

```
##bad, this example needs additional setup before being usable.

#this example demonstrates how iterate over a list in Python using foreach
for item in lst :
    print(item, " ")
```

Finally, in your documentation, try to be as precise as possible and therefore use the specific terms and names if you can, as for example *bubble sort* or *visitor pattern*.

Rationale

Documenting architecture and other macro aspects of your software drastically helps people getting into it quickly. An overview of the software shortens the time to find a particular element, given that locality guideline is respected. Using a standard representation makes less ambiguous diagrams, however they can be harder to understand for a beginner. Non-standard diagrams can also be more flexible. Using a standard and then extending it is possible, but shall be done with care since it can introduce vicious ambiguity.

Documenting the code can compensate for the lack of syntactical expressiveness when it happens. Most modern code editors and IDEs are able to recognize formatted documentation, as docstrings and Doxygen comments, and then to display them in various moments, as for example alongside autocompletion suggestions.

Keeping the documentation's source close to the code it refers to eases maintaining the documentation and reduces the risk of losing it when sharing your code. It is fairly hard to forget to transmit a documentation that is *included* withing your code, or at least in the same folder.

Examples make your documentation more expressive. They are also lighter to both read and write than a large chunk of explanatory text. This is unlikely that your examples will always fit perfectly the expectations of the person reading your documentation. By making them standalone, executable out of the box, we allow readers to set up an environment for experimentation without any effort.

Using specific terms and names helps the reader investigate further the notions if he needs to.

Limits

A wide, detailed documentation takes time to write and to keep up to date. An outdated documentation can create confusion in regard to how the code is working.

When short on time, prefer focusing on code than on documentation, except for essential documentation.

4.2.9 Elegance

Definition

Code is better being elegant. What elegance actually means ? According to Wikipedia[68], *Elegance is beauty that shows unusual effectiveness and simplicity. Elegance is frequently used as a standard of tastefulness, particularly in visual design, decorative arts, literature, science, and the aesthetics of mathematics. Elegant things often exhibit refined grace and suggest maturity, and in the case of mathematics, a deep understanding into the subject matter.* And yes, that is extremely subjective.

Always question yourself whether you like writing and reading your code, and question others on that matter as well. An elegant code is, *in my opinion*, a code that is *efficient and pleasant to write and to read, while reflecting the developer's personality and his approach on the problem the software addresses.*

Elegance is bought by the combination of rigorous practices, like these guidelines, testing, algorithmics and others common or standard rules followed by the developer, and by his ability to take some freedom on points that will not bother himself or any other person working on or with his code. Such free points can for example be purely aesthetic, as braces.

Listing 4.61: C++ freedom on braces example

```
////some developers perfer this
int myFunction() {
////...
}

////others prefers this
int myfunction()
{
////...
}

////neither of these constructions changes significantly the understanding of the
    ↪ code.
```

It can also pass though mastery of tools, using them to their highest potential, or by some humoristic comments in your code, or writing comments in a dialogue-style with your reader, as long as it does not reduce readability significantly.

Listing 4.62: Python freedom on comments example

```
#encodes behavior of a swan.
#polymorphic with Animal interface using DUCK typing. Lol.
class Swan():
    ##...
```

Rationale

Having a code that *you* like, can be proud of, and that other people like motivates yourself and others into working on it, testing it and making an overall high quality software in the end.

A code that is original will stand out. Would it be for the better or the worse.

Limits

Elegance is really subjective. Giving the code some personality without being careful can result in lower understandability of the code. And understandability is one of the pillars of elegance. Don't force yourself into making elegant code. Train and learn, experiment, and the elegance will come by itself as your mastery increases.

4.3 Enforcement

Producing high quality code requires a lot of discipline to remain consistent as much as to follow rules. Evaluating code isn't an easy task either, as complexity and other metrics can be fairly hard to compute by hand. Fortunately, there are tools and methods to helps us in this task. In this section, we will discuss how guidelines before and more generally a high quality of code can be enforced.

4.3.1 Automation

Setting up a good, tool-assisted environment makes you lose time at first when starting you project. On the long run it saves you an *insane* amount of time and effort. The most obvious tools that can assist you are static analyzers. We can consider a compiler or interpreter as a static analyzer, as it performs checks on your code without running it first. Static analyzers are often called *linters*, and can be found as a standalone tool, integrated into an editor or IDE, or even through and online service acting on a git repository. They will usually work by producing a report and point out possible weaknesses of your code, as for example a lack of documentation or an inconsistency of your code with some guidelines.

Listing 4.63: Python linting example

```python
def function1():
    a = 10
    b = 20
    print(a)
    print(b)
    print(a + b)
```

A common linter for Python is Pylint[69], that we can run, after installing it, with command `pylint [filetocheck]`. Running Pylint on the previous code example produces the following output, telling us that docstrings are missing and that our variables do not follow Python standard for naming.

Listing 4.64: Python linting result

```
************* Module filetocheck
testpylint.py:1:0: C0114: Missing module docstring (missing-module-docstring)
testpylint.py:1:0: C0116: Missing function or method docstring (missing-function-
    ↪ docstring)
testpylint.py:2:4: C0103: Variable name "a" doesn't conform to snake_case naming
    ↪ style (invalid-name)
testpylint.py:3:4: C0103: Variable name "b" doesn't conform to snake_case naming
    ↪ style (invalid-name)

-----------------------------------------------------------------
Your code has been rated at 3.33/10
```

As previously said, some linters and analyzers are online and analyze a git, GitHub or GitLab repository on demand, for example when pushing a commit. A lot of them, like proprietary tool CodeClimate's Quality[70], provide free services for open source projects[71].

Some linters are more advanced than others, or more specialized, like focusing on security or performance. As linters usually don't edit your code, you can use more than once without worrying about conflicts.

Another exciting code quality tool are code formatters, also called beautifiers. Code formatters will parse your code and rearrange it to match certain rules. They help maintain a code with a visual strong coherence without having to pay too much attention to it while writing. C++ can be formatted, for example, with clang-format[72], using a configuration file or a default style.

Listing 4.65: C++ clang-format formatting

```cpp
//this code can be auto formated
int function(){int a=1;int b=2;int c=3;cout<<a+b+c<<endl;}

//into this code
int function(int a ) {
    int a = 1;
    int b = 2;
    int c = 3;
    cout << a + b + c << endl;
}
```

Code formatters don't change the syntactical construct of the program, only it's appearance. It is not advised to use more than one formatter software at once for a file since the file will get edited. Also, because of this edition, formatters can create a lot of unwanted conflicts in git versioning if they aren't used frequently, therefore it is strongly advised to configure the editor or IDE to run the auto formatter on the code automatically when saving, It is also not advised to run them on the full code base at once unless nobody else is working on it.

Using automation tool and keeping config files local to the project is a good practice, as described in chapter 2, Development environment : it will ensure people working on the project always work with the same tools as you.

Setting up such tools helps to solve part of the subjective aspects of code quality by delegating the decisions to the computer and objective rules. However, some rules can hardly be automated. Those are meant to be mentioned into a set of guidelines such as shown in this chapter section 4.2, Guidelines. Referring to an already existing and widely used guideline set is even better.

4.3.2 Code reviews

Code review is a static analysis of the code, but performed by humans. In this section we will discuss how to conduct code reviews. The general idea behind a code review is to verify properties of the code that are too complicated or ambiguous to be enforced by automatic processes. Such reviews can still be assisted by automated tools. For example, a static analyzer can look for possible non-terminating recursion and then reviewers will decide which are non-terminating or not.

Listing 4.66: Python non terminating recursion example

```
#this function is a non-terminating recursion that will indefinitely call for
    ↪ itself
#it can cause a stack overflow
def non_terminating(number)
    non_terminating(number + 1) # self calling here
    print(number) #this statement will never be executed

#this function is a terminating recursion, as it calls itself only certain
    ↪ conditions
#that are guaranteed to be met at some point
def terminating(number)
    print(number)
    if number > 0 :
        terminating(number  -1) # at some point number will reach 0
```

Code review can take a lot of shapes. Let's see what parameters we can modulate to have the perfect one fitting for our needs.

Reviewer's awareness : Reviewer awareness is how much a reviewer knows about the code before starting to review it. A highly aware reviewer can for example be the writer of the code itself, or a close by colleague. A highly unaware reviewer can be a random person found on the internet. A highly aware reviewer is likely to be biased toward to code as his mind is already oriented toward a certain idea of what the code should be. On the other side, a highly unaware reviewer may miss details related to the context of the code, as architecture flaws.

Temporality : Reviews can occur at various temporality. The three main possibilities are :

- When writing the code, as for example with pair programming.
- When code changes, for example when a new commit is pushed on the git repository.
- At an arbitrary moment, for example each three weeks, or at the end of each cycle.

Rigorous or freestyle : A rigorous style review focuses on a certain set of points that will methodically be examined, while a freestyle review will look at the code freely without looking for something in particular. Rigorous review offers reliability if conducted with discipline, while a freestyle review is more entertaining and flexible, as it can detect non-predicted errors. Of course, those two are archetypes and an in between is possible : ensuring a set of point is always checked, while still allowing for freestyle remarks.

Support : A code review can be held with a meeting room and a computer, by displaying the code and discussing it. It can also take place on internet, for example in the thread of a merge request, with all reviewers being remote, or use dedicated reviewing tools. There are pros and cons to all configurations, but the most important actor in the decision is you and colleagues. If you need flexibility, doing remove and deferred reviews is a nice idea. If you prefer discussion, then having a direct contact between reviewers and eventually devs may be more interesting.

Code reviews are much more important that they may seem, especially if they are done regularly and early. Regularity allows reviewing only small piece code each time, reducing tiredness and boredom that may loosen your proficiency as the review advances. Reviewing the code as quickly as possible after it is declared to be a valid version help detect defects before the code is widely used, and thus minimize the need for changes later on.

Like in any other development practice, code reviews have to be adapted to specific needs of the team and software, and the chosen method shall be open to changes if need be.

4.3.3 Design and Refactoring in the cycle

When writing a software or working on technologies, sometimes one may *voluntarily* ignore flaws instead of fixing them, because at that time it costs less to adapt new code than to fix the existing code. It is understandable as some minor flaws can require to edit the entire code base, as for example if it affects the core of the software. Let's say it explicitly, this is a horrible, evil practice. Over the time, minor flaws accumulate and end up being a real handicap in the development of the software. This is a vicious circle : as flaws fixing is avoided and code adapted to those flaws is added, the effort required to fix them increases, and thus the temptation to ignore those flaws strengthens. This is the hell of the technical debt. *Dark choirs play in the background*.

Note that a really local flaw, for example within the body of a function with no side effect, may not create technical debt since it will not affect the rest of the code.

Listing 4.67: Python code flaw example

```python
#this function has a local flaw, as the while loop could be replaced
#by a foreach loop for elem in array :
def sum(array):
    i = 0
    sum = 0
    while i < len(array) :
        sum += array[i]
    return sum

#changing that will not change the behavior or the way to call the function at all,
    ↪ so
# this flaw will not create technical debt
```

Fight against technical debt happens by two ways that are complementary. The first one is prevention. We want to have design phases in which technical debt can not happen before getting into a new objective. For that, we consider everything done in this phase to be fully disposable, and we mentally prepare ourselves for that. Restarting the phase from zero shall not make you afraid. While doing so, we explore different designs, study problems we encounter and try to predict other problems we may encounter in the future. We only get out of thinking phase and start implementing our solution when we have a clear and fairly detailed overview of what we need to do to complete our objective, in other words, when the implementation will be trivial for us. It isn't forbidden to code during this phase, but we need to be prepared to fully delete our code and start again if needed. If the design seems too complicated to achieve, divide your objective in smaller ones as independent as possible and work on them once at a time. It may be tempting to skip this phase and get straight into coding. It can be partly explained because when we design it is harder to see progression in comparison to when we code, as design doesn't produce a functionality when it is done. To address that, you can do *design reviews* where you

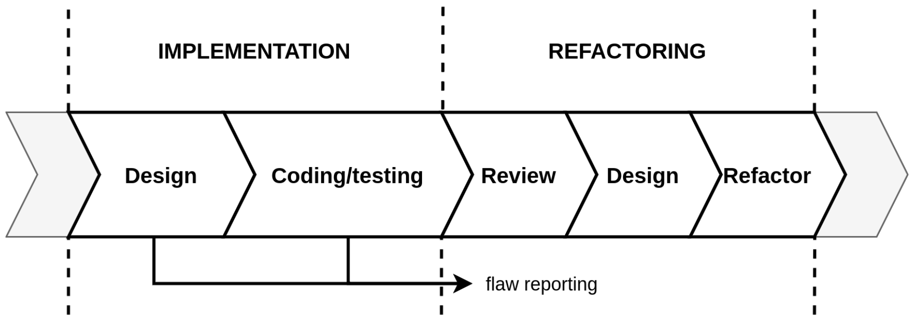


Figure 4.3: Example of a two-phases development cycle

share thoughts on you design with other people as in a code review. It will both give you a feeling of accomplishment and also help to fix flaws of your design.

The second way is to fix remaining flaws short after they are detected, through a refactoring. Its purpose is only to enhance software and code quality, and thus it has to be differentiated from functionalities implementation and bug fixes. The behavior of the software do not change during a refactoring, only it's appearance and expression. It is advised to have a dedicated time for refactoring, during which the target code is locked and no behavior change is allowed, in order to avoid conflicts that can happen. You can even consider locking your entire code base while refactoring in case you may underestimate the extent of changes. It is important for refactoring to happen as soon as possible upon detection of the flaw to minimize the amount of changes needed to fix the flaw. A refactoring phase can be opened by a design phase if the solution to the flaw isn't trivial. Testing also plays a critical role in refactoring as it reduce the risks that we break the code without noticing. And as luck would have it, next chapter is related to testing. Code reviews can be part of the refactoring.

Some flaws will appear naturally in your software, as introducing new functionalities or changing behavior can require design changes withing already existing code. In these cases, it is extremely important to fix those flaws before starting producing new code.

Finally, a development cycle will have two main phases, one of implementation, and one of refactoring, both starting by a design step. Usually, when detecting a flaw during an implementation phase, we simply wait for the next refactoring phase to fix it, unless it is critical and stops the current implementation phase, as shown in figure 4.3.

Chapter 5

Testing

5.1 Motivations

5.1.1 Functional correctness

Functional correctness is met if the software fulfills its functional requirements. Such requirements can be features, a task the program can perform, stability, the program being able to deal with heavy load or unexpected situations as wrong inputs, program's performances, the program being able to perform a task while using a limited amount of resources, or usability, the program meeting certain criterions of ergonomics. Table 5.1 shows a way of expressing requirements for a small image conversion software.

Usability requirements may be really hard to test if they involve subjective notions as intuitiveness or ease of use. Fortunately, in science we will rarely focus on usability since user interactions tend to be limited. In most cases, features are verifiable by simply looking at the output of a program given a set of inputs, while performances and stability require to keep an eye on the program during runtime to do measurements.

A program not fulfilling feature requirements simply does not work. A program being unstable, slow or consuming too many resources can be even more frustrating, as problems will eventually occur long after you are using it. Imagine your 7 days simulation crashing because nobody checked how the program performs after 7 days, or because of a tiny memory leak. Nobody wants that, right ?

5.1.2 Maintainability

We already mentioned maintainability in section 4.1.5. When changing the behavior of a program, either for adding or updating features, or for fixing a bug, we want to avoid breaking something elsewhere. This is called non-regression and is ensured by non-regression tests. They are a fantastic guardian that will warn you whenever you broke something in your code while editing it. Tests are your last and strongest rampart. If a bug or error isn't detected by tests, it is likely that it will not get fixed before affecting user. Remember that 7 days simulation crash ?

5.2 Composition of a test

The notion carried by the word *test* is vague. This sections aims to clarify a bit what can be included in the *test* concept. To summarize, a test aims to check that a program produces the right output, given set context, precondition and input.

R1	feat.	Software is invoked with `sname dest form file[ file]*`.
R2	feat.	dest is a destination folder, expressed either with relative or absolute path
R3	feat.	form is the destination format, either *png*, *jpg* or *tiff*
R4	feat.	file is an input file, either *png* or *jpg* format, path relative or absolute.
R5	feat.	upon invocation, all file are copied to dest in the specified form
R6	perf.	the program shall not use significantly more memory than what is needed to store twice the largest image in a batch
R7	stab.	the program shall ignore file that are not in format specified at R4
R8	stab.	if there is not enough memory to process a file, program shall ignore it and continue
R9	usab.	files ignored at R7 and R8 shall be signaled by an error message printed in the invocation terminal

Table 5.1: Functional requirements example

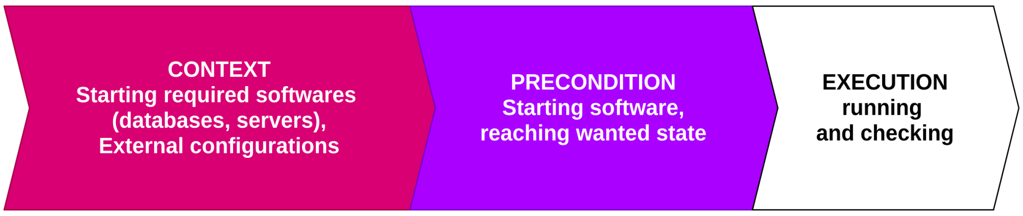

CONTEXT
Starting required softwares (databases, servers), External configurations
PRECONDITION
Starting software, reaching wanted state
EXECUTION
running and checking

Figure 5.1: Test context and precondition

5.2.1 Context and precondition

A test will always be executed in a certain context, which consists of everything external to the program that can have influence on it. For example, it can be the operating system, or a distant web server with which our program will interact. The context can be set explicitly, which would probably be the case for a server, or implicitly, as it will be most of the case for the operating system.

Precondition is the state in which the program is right before executing the test. For example, if we want to verify that user can save a file, we could want as precondition for the test that a file is already loaded in the software and ready for saving. This is illustrated in figure 5.1.

5.2.2 Execution and verification

Running a test involves both sending inputs to the program and checking that the program reacts correctly to it.

If verification only checks for the effect the program has on the environment, its output, as shown in figure 5.2, we will call it a *behavioral verification*, or *black box test*. It has the advantage to offer a really high level of abstraction, and therefore to be fairly independent of the program's structure, allowing to change the program without needing to edit the tests.

If verification checks whether the program is in a certain state after or during execution of the test, as in figure 5.3. We will call it a *state verification*, or *white box test*. It offers more possibilities than behavioral verification, as sometimes checking directly the state of a program is much simpler than verifying its behavior. However, it shall be used with care because changes in the program structure may require to rewrite the state tests. This will be discussed later in section 5.5.

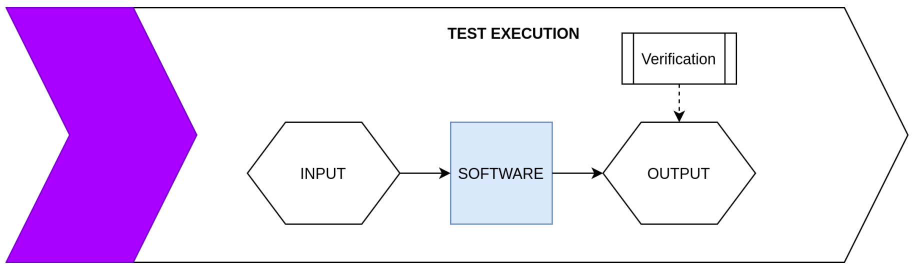


Figure 5.2: Behavioral verification

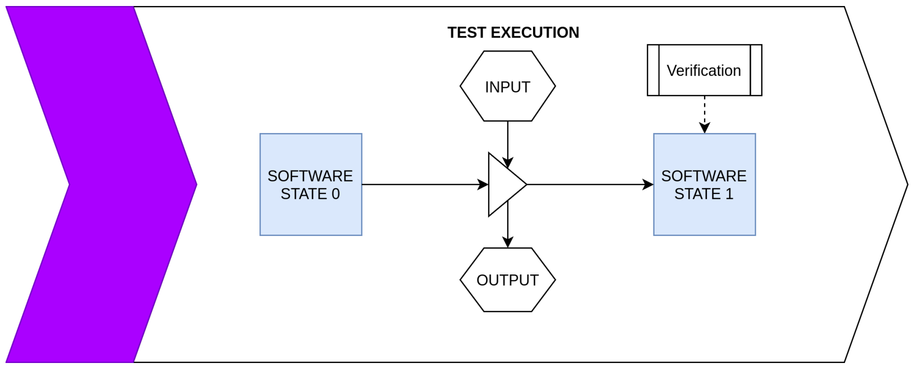


Figure 5.3: State verification

5.2.3 Test case

A test case is one way to formulate a test that aims to be really concise and precise. It focuses on a single of few inputs and performs the verification after all inputs has been sent, as show figure 5.4. It is well suitable for both behavioral and state verifications. When several test cases address the same set of requirements or are related, they are usually grouped in a *test suite*. A test case has to be as minimalistic as possible, with only required context and precondition, and fewer possible inputs. Because of this, verifications across a test suite are independent of each other, allowing testing precisely requirements and avoiding falsely marking some requirements as failed simply because test case failed. Test case usually are meant to be fully deterministic : they give fixed inputs and expects fixed outputs, excluding any randomness, in order to be fully reproducible, since they are meant to run often during development or edition of the code.

5.2.4 Test scenario

A *test scenario* is the other way to formulate a test. Instead of aiming to be concise, it will try to verify requirements in a more natural way, kind of as a user would use software. After setting the context and precondition, it sends inputs and verify output right after it, repeating the process as much as needed to verify target requirements, as it can be seen in figure 5.5. Those tests are compatible with state verifications but are much better for behavioral verifications. During

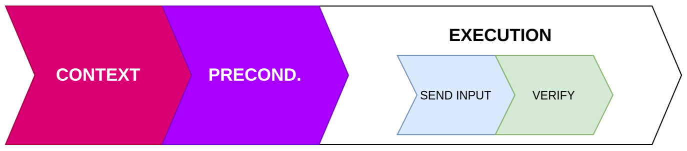


Figure 5.4: Test case

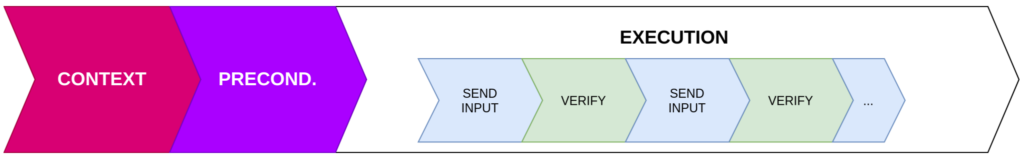


Figure 5.5: Test scenario

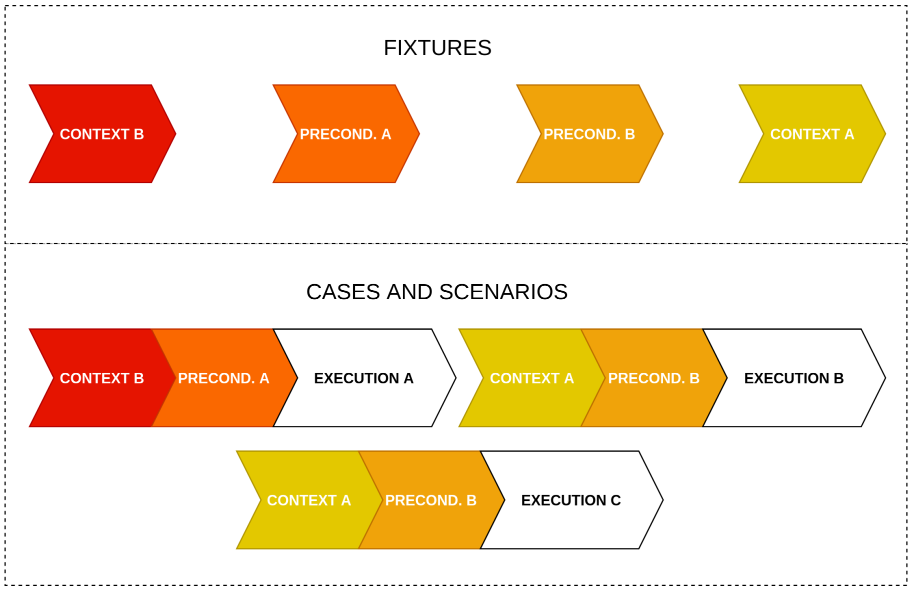


Figure 5.6: Fixtures

development or edition, they act as a guidance, informing the developer what the user will do with the program. After the development or edition, they permit validating software requirements in more realistic conditions than test cases do. Comparing to test cases, they are much easier to write when it comes to verify loosely a wide set of requirements. As they are not forcibly run often during development or edition, they can be fairly large and imprecise, as well as featuring a fair amount of random inputs. Test scenario can be split into test case and vice versa, depending on the needs. A scenario is, somehow, a collection of consecutive test cases.

5.2.5 Fixtures and Mocks

Contexts and preconditions may be fairly heavy to write or setup while being required across different test cases or scenarios. In order to keep the code clean and avoiding redundancy, we factorize contexts and preconditions of tests into *fixtures*[73][74][75], as figure 5.6 displays. Fixtures are no more than a setup for context and precondition. Fixtures help tests cases or scenario to be independent one from another by allowing to setup and tear down context and precondition for each case or scenario run.

As for a scientific experiment, we want to execute our tests in a controlled environment. We want to avoid contextual errors that would make test fail. For example, if testing a web page, we want to ensure that test will be immune to unrelated server failures. Therefore, when setting up the context, we may want to substitute actual context and input elements by artificial mimics that behaves the same way but are much simpler and thus less prone to errors. Those are called *mocks*[76][77][78][79] and are shown in figure 5.7. A mock for a large web server could for example be a tiny web server serving a static web page instead of dynamically creating it.

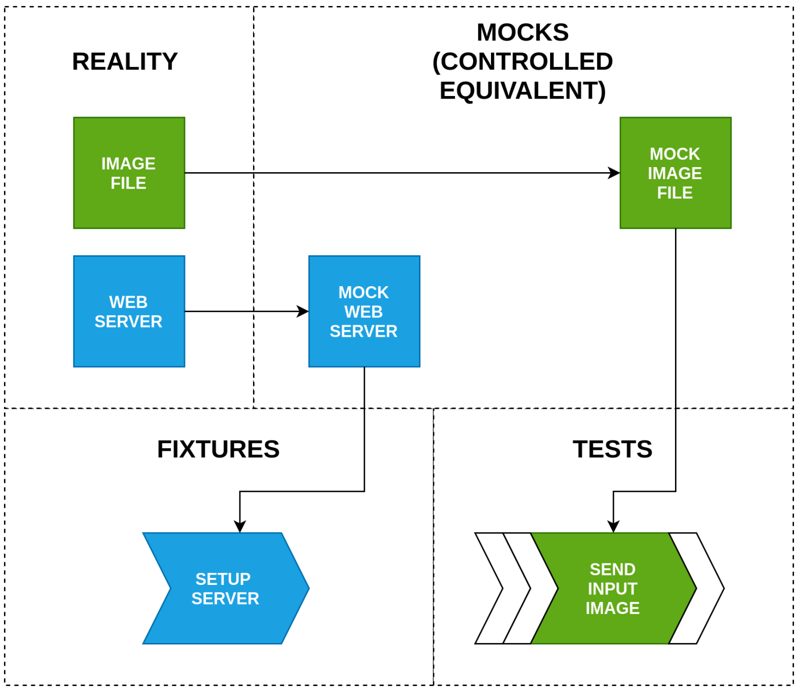


Figure 5.7: Mocks

5.2.6 Non-trivial verifications

Some requirements involving randomness or user interface, or being heavily subject to not easily controllable context elements. They may prove to be fairly complex to verify, even more to make such verifications deterministic. This section tries to address such issues.

Randomness

Randomness is the enemy of tests, and you want to avoid it as much as possible. The best way for this is to have a well-defined separation between random and deterministic code, and have as much as possible deterministic code[80].

Listing 5.1: Decoupling random code, base example

```
#this function is random
def random_based_behavior() :
    val = dice_throw() #dice_throw generates a random value
    #... here treatments based on random value
```

Listing 5.2: Decoupling random code, value as param

```
#we can make function deterministic by taking out
#dice call and instead pass it as a value
def random_based_behavior(val) :
    #... here treatments based on -eventually- random value passed as param through
        ↪ val

#calling such function would look like this :
random_based_behavior( dice_throw() ) #random
random_based_behavior( 1 ) #deterministic
```

Listing 5.3: Decoupling random code, generator abstraction

```
#we can also make it deterministic by abstracting dice as a value generator
```

```
def random_based_behavior(value_generator) :
    val = value_generator.get_value()
    #... here treatments based on val

#this way we can define either a dice or a predetermined sequence as value
    ↪ generator
class Dice :
    def get_value(self):
        return throw_dice()

class Sequence :
    def __init__(self, values):
        self.vals = values
        self.index = 0

    def get_value(self):
        val = self.vals[self.index]
        self.index += 1 #moving onto next value for last call
        #should be error code here if we reach end of sequence
        return val

#Calling such a function would look like this :
random_based_behavior( Dice() ) #random
random_based_behavior( Sequence([1, 2, 3]) ) #deterministic
```

Testing randomness is better done in a separate suite or scenario, with a lot of repetitions to allow statistical confidence in the result. Please note that most random generators, as Python random module[81], allow to provide a *seed*, making random generation deterministic. It may be useful in case decoupling is difficult.

Performances

Execution speed of a software is heavily dependent of the hardware it runs on, as well as compiler and version of libraries used. If performances are always to be evaluated by a human, this is not a problem and issuing a report with raw speed is usually fine. However, if tests are meant to *ensure* speed criterions are met, one probably wants to change from report to *fail or pass* type of result. In case of time critical applications, as robotics, where the system has to guarantee to react in a given time, taking a fixed value as reference is the right solution.

Listing 5.4: Python fixed time performance test example

```
#we will consider chrono to be a way to measure time

#main code
chrono.start()
#code to test here
chrono.end()

if chrono.time() > 1000 : #using a fixed time requirement value
    return False #test failure
else
    return True #test success
```

However, if tests are meant to measure progression of the software's performances, and that those variations shall be evaluated independently of the context, it can be better to use some trivial computing or a context-dependent value as reference.

Listing 5.5: Python relative time performance test example

```
#we will consider chrono to be a way to measure time

#main code
chrono.start()
#code to test here
chrono.end()
```

```
runtime = chrono.time()
#computing a reference time value that will scale with computer power
chrono.start()
sum = 0
for(i in range(1..100000))
    sum += i
chrono.stop()
reftime = chrono.time()

if runtime > reftime : #using a relative time requirement value
    return False #test failure
else
    return True #test success
```

Software performance can vary depending on the execution environment, as for example other processes running on the same computer. It is therefore advised to test it as if it involved randomness.

Floating-point

Floating-point numbers are subject to lack of precision, due to the fact that we represent an interval of real numbers with a limited number of values. For this reason, unless you are looking for the exact value, use intervals to compare actual value to expected one.

Listing 5.6: Python float imprecision example

```
val = 1.343
c = cos(val)
a = acos(c)

#this is unlikely to happen because of float imprecision
if a == val :
    print("equals")

#this will likely happen
epsilon = 0.00001
if val - epsilon < a and a < val + epsilon :
    print("in between")
```

UI

User interface is by far the hardest thing to test in a program, as they usually allow complex workflows and actions, and that isolating components isn't always possible and can even be unwanted. Even if both graphical and command lines interfaces testing tools will be discussed later in section 5.3.1, CLI tends to be easier to test and thus shall be preferred if software requirements allow it.

5.3 Environment

This section describes several tools that can be used in order to test a software. The aim isn't to fully explain these tools, only to make you aware of their existence and abilities. As an example, we will set up environment to test a really simple software made of the following code, split between two files.

Listing 5.7: Software sample for test environment, strop.py

```
# file strop.py
from format import concat
from format import between
from sys import argv
from sys import exit
```

```
# main code
# if not enough args, we return error and explain why
if len(argv) < 3:
    print(
        "too few arguments, invocation shall be done with: strop format str1 [...
            ↪ strn] ")
    exit(1)  # here retuning failure

# othewise, separate format from other str args
fmt = argv[1]
strs = argv[2:]

if fmt == "concat":
    print(concat(strs))
elif fmt == "between":
    print(between(strs))
else:
    print("invalid format, should be either concat or between")
    exit(1)

exit(0)
```

Listing 5.8: Software sample for test environment, format.py

```
# file format.py
def concat(strings):
    return "".join(strings)


def between(strings):
    bet = strings[0]
    others = strings[1:]
    return bet.join(others)
```

It takes two or more strings as arguments and formats them according to a selected format.

Listing 5.9: Software sample for test environment execution example

```
python strop concat ad b c d e f
adbcdef
python strop between ad b c d e f
badcaddadeadf
```

`concat` concatenates all remaining strings together. `between` takes first next string and uses it as concatenation insert for all remaining strings.

5.3.1 Automation

Testing involves performing a lot of actions and a lot of verifications. For this reason, manual testing is foolish in most cases. First, setting up and executing tests is boring and takes time. Second, when the number of tests grow, there is a high probability that you will simply not be attentive enough and miss something out verifying result, letting a bug slip away. Somehow, computers are much more reliable than humans when it comes to apply a test protocol, given the protocol can be translated as a program. And fortunately, we most of the time develop software using a computer. The goal of automation is to make testing as pleasant as possible. This way we are encouraged to write a lot of tests and run them often. In the ideal case, we only have to perform one single action to know the result of our tests.

Listing 5.10: What automated testing should look like from the outside

```
test
passing !
```

Automation may look like a bother at first, because it requires a bit of setup[82]. It can also make tests a bit costly to write, because you have to register them in automation. Let's do the math. Imagine you really struggle, it takes you three days to get your automated test environment ready. 21 hours of work. Then, when writing a test, you spend one more minute writing it in comparison to a manual test. Let's say you have 120 tests, making the total to 23 hours spent on automation. Now, consider each test you wrote to take an average of 30 seconds to execute by hand, while it takes less than 10 seconds to run them all with automation. In development phase, we can you will probably run your tests three times per hour. That's average : in the beginning of a feature, you will probably not run them, when you want the feature to match tests, you will want to run them *a lot*, like once every two minutes or so. You will not always need to run all of them, even if it would be better. We'll therefore consider you run 10 tests each time. By hand, every 20 minutes you code, you spend 5 more minutes running tests. In a day of 7 hours of work, you literally spent 1h and 30 minutes running your tests, and you only ran few of the suite. With automated testing, after 7 hours, you spent a bit more than 3 minutes running your tests, and you ran the full set each time. After roughly 15 days, any test you run manually makes you loose an incredible amount of time.

One may argue that as a scientist, you will tend to have a lot of tiny scripting projects instead of wide software development tasks, and that setting environment and tests each time isn't worth it. That would be forgetting that an estimation of three days for setting up environment is extremely pessimistic. You will quickly get used to the task. *For example, when setting up a C++ project, C++ being my main language, it rarely takes me more than 15 minutes to get a decent testing environment ready. And C++ doesn't come with a simple to set up tool set.*

General automation

General purpose automation can be done simply using a combination of scripts, as bash, and build tools, as *Make*, presented section 2.2.3. Some tests may be written in plain code, in the same language as our software is. Those are usually state verification related tests, and can be done either from scratch or with the help of a library or framework, that we will discuss in section 5.3.2. Behavioral verification oriented tests will tend to rely more on external automation tools, would it be simple scripts or interface interaction automation tools. General automation tool are good for scheduling execution of tests and then aggregating the results if needed. Here is an example using Make.

Listing 5.11: Test automation with make simple example

```
#build target to build our software
#actually not needed with python since it is fully interpreted
build :

#make automatically exits if a command fails, and a command fails if it's return
    ↪ code
#is different than zero
test : build
#first test command
        ./test_no_argument.sh
#second test command
        python -m unittest
#...
```

We can then write a simple test script using bash for testing the error message of our program if no arguments are passed.

Listing 5.12: Testing with no argument, test_no_argument.sh

```
#!/bin/bash
#file test_no_argument.sh
#we start by creating a file with expected result, this is our context setup :
printf "too few arguments, invocation shall be done with: strop format str1 [...
    ↪ strn] \n" > res.txt
```

```
#then we run the program without args and dump in a file, for a wide expected
    ↪ result,
#or with a lot of tests like that, one can use external files instead
python strop.py > exp.txt
#then run diff on both files, diff returns 0 if file are the same, another value
    ↪ otherwise.
diff exp.txt res.txt
#we store return value of the diff command, represented by variable $?
res=$?
#we finally delete res and exp, this our context cleaning
rm res.txt
rm exp.txt
#and we return result for diff
exit $res
```

Bash tests are not the best and there are many other ways to write tests, as Python for example. Some build tools[1] are specialized in building a particular language or set of languages, and can handle test automation tasks.

Command line interface automation

Doing things with bash-like scripts can become tedious when needing to test an interactive command line interface. Fortunately, there are tools to help us with this. One of them is *Expect*[83]. Expect allows you to run a program and send characters on its standard input based on what it writes on its standard output, as if someone were interacting with its CLI. Our program earlier doesn't feature such interactivity, and thus we will demonstrate it by writing a test script for the following program.

Listing 5.13: Expect script example hello world

```
user = input("who are you ?")
print("hello, ", user, " !")
```

Listing 5.14: Expect script example

```
#!/usr/bin/expect -f
#line above tells to use expect interpreter instead of bash

#running program we want to interact with
spawn python script_example.py

#now we wait for the prompt
expect "who are you ?"
#sending an input
send "expect interpreter"
#waiting for answer
expect "!"
#waiting for session to close
expect eof
```

We will see more of it in section 5.7, Example : testing the Zoo application.

Graphical interface automation

We won't do any graphical user interface testing demonstration in this course. Except when using web, or web-like technologies as *Electron*, automating GUI testing is fairly hard. Please note that some large GUI frameworks and libraries, as *Qt*[84][85], offer their own way to automate testing their graphical interfaces. In case you need to build a software with a complex graphical interface that has to be highly tested, it would be wise to use one of those. Qt and alike have ports in a lot of languages. The counterpart of Qt in Python it *pyQt*.

[1]Non-exhaustive list can be found at `https://en.wikipedia.org/wiki/List_of_build_automation_software`

5.3.2 Libraries and frameworks

Testing libraries and frameworks can greatly help writing tests that are in the same languages as your software. They provide tools to help you write tests, including fixtures and mocks, quicker and with an increased reliability. When referring to either a library or framework, we will use *framework* word for simplicity, and also because a lot of in-language testing tools are frameworks. If you choose the *unit approach* presented later in section 5.4.2, you will probably end up writing dozens, hundreds or even thousands of state verification tests, or at least behavioral tests that will act like state verification ones. In that case, not using at least a tiny test library is dangerous. Test frameworks[2] don't only provide you a nice way to write your tests, they can also ease running them, by helping you have a clear output of what is passing, what is failing, and by allowing to regroup several tests in a single program.

Python has several of those frameworks. *unittest*[86] is present in the standard library. We will therefore use it to write more test cases for our example code listings 5.7 and 5.8. Note that following example doesn't show the full potential of unittest, only a minimal example.

Listing 5.15: Python unittest example, test_format.py

```python
# file test_format.py
# let's test concat and between methods
from unittest import TestCase
from unittest import main
from format import between
from format import concat

# we define test case as a concat


class TestConcat(TestCase):
    def setUp(self):  # this method is used to setup test case
        print("setting up concat")  # no setup needed, this is only for show

    def tearDown(self):  # this method is used to teardown test case
        print("tearing down concat")

    # starting a method name by "test" tells the framework that it
    # is a test case execution
    def test_concat_single(self):
        # assertEqual allows to verify that two values are equal, and report
            ↪ otherwise
        self.assertEqual(concat(["aaa"]), "aaa")

    # the third parameter of assertion allows to display a custom message in case
        ↪ of failure
    def test_concat_multiple(self):
        self.assertEqual(concat(["aaa", "bbb", "ccc"]),
                         "aaabbbccc", "custom failure message !")


class TestBetween(TestCase):
    def setUp(self):
        print("setting up between")  # no setup needed, this is only for show

    def tearDown(self):
        print("tearing down between")

    def test_between_two(self):
        self.assertEqual(between(["a", "b"]), "b")

    def test_between_more(self):
        self.assertEqual(between(["a", "b", "c", "d"]), "bacad")
```

[2]A non-exhaustive list of such tools can be found at https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks

```
class TestBetween(TestCase):
    # test failing just for show
    def setUp(self):
        # no setup needed, this is only for show
        print("setting up between fail")

    def tearDown(self):
        print("tearing down between fail")

    def test_between_two_fail(self):
        self.assertEqual(between(["a", "b"]), "bb")

    def test_between_more_fail(self):
        self.assertEqual(between(["a", "b", "c", "d"]), "bacadd")


# if the file isn't included from another
# (eg file is run with python command directly)
# then run tests
if __name__ == '__main__':
    unittest.main()
```

Then we can simply invoke python with unittest module command in the strop root directory, `python -m unittest` and it will give a report. We can notice setup and teardown at each test run.

Listing 5.16: Python unittest example output

```
python -m unittest
setting up between fail
tearing down between fail
Fsetting up between fail
tearing down between fail
Fsetting up concat
tearing down concat
.setting up concat
tearing down concat
.
======================================================================
FAIL: test_between_more_fail (test_format.TestBetween)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/felix/courses_ressources/0423
    ↪ _software_development_methods_for_scientists/examples/testing/environment/
    ↪ test_format.py", line 57, in test_between_more_fail
    self.assertEqual(between(["a", "b", "c", "d"]), "bacadd")
AssertionError: 'bacad' != 'bacadd'
- bacad
+ bacadd
?      +


======================================================================
FAIL: test_between_two_fail (test_format.TestBetween)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/felix/courses_ressources/0423
    ↪ _software_development_methods_for_scientists/examples/testing/environment/
    ↪ test_format.py", line 54, in test_between_two_fail
    self.assertEqual(between(["a", "b"]), "bb")
AssertionError: 'b' != 'bb'
- b
+ bb
? +


----------------------------------------------------------------------
Ran 4 tests in 0.000s
```

```
FAILED (failures=2)
```

Report indicates, for each assertion, file and line where the assertion is, as well as value tested and expected result.

5.3.3 Continuous testing

Automated testing is good, but it has some limits. If a developer pushes work he forgot to run tests on, or forgets to test a merge, it can introduce bugs. Also, if your code is at rest for some time, changes in the dependencies, like libraries used, can introduce bugs as well. Fortunately, we can automate that as well. It is called *continuous testing*[87][88]. We can choose to trigger testing at two moments. First, when changes are introduced in the code base, on commits and merges. Second, at an arbitrary date or recurrent occurrence, as for example once per week, on Monday morning. Most forge software support such features a way or another. It can be set up directly in version control software as well, as for example by using *Git Hooks*[89]. Following example will be based on GitLab, since it is an open core software. To enable such feature, we need to add a *.gitlab-ci.yml* yaml file in our repository[90].

Listing 5.17: Gitlab CI file for continuous testing

```yaml
#this is a comment in yaml

#we select a docker image, to host our run
image: carterjones/manjaro

#then we create a pipeline stage
test:
  #we set it to run only when commit on main, dev and develop branches
  only:
     - main
     - dev
     - develop
  #setting name of the stage
  stage: test
  #setting setup script, here for building a C++ project
  before_script:
     #installing some libs
     - sudo pacman -Syu --noconfirm && sudo pacman --noconfirm -S gcc cmake make
         ↪ unittestpp glm
     #building
     - mkdir build
     - cd build
     - cmake ..
     - make
  #running tests
  script:
     #it only needs to show run test command, as long as the command returns 0 if
         ↪ pass and another value if fails
     - make test
```

With this, each time we push a commit on either main, dev or develop branches, tests will be executed for that commit. Recurrent occurrences of tests in GitLab can be done through the GitLab interface directly and is described in GitLab documentation[91], as well as detailed setup. Results will be displayed in a merge request's thread if the target branch is concerned by the run. Email and other alerts can be configured in case of failure of a run. Of course, similar features can be found in nearly any forge software.

5.4 System and unit testing

When it comes to choose what to test in a software, there is roughly two schools. The first one wants test only the behavior of the software as a whole. It is often referred to as *functional*

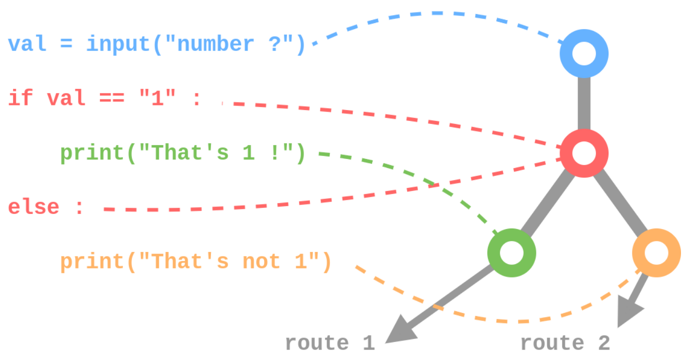


Figure 5.8: Illustration of execution route

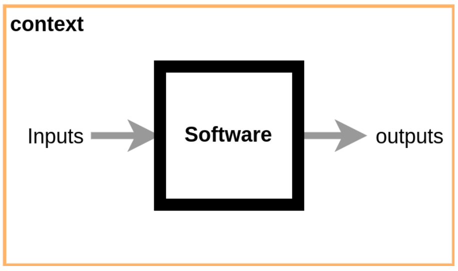


Figure 5.9: A software seen through system testing

correctness testing or as *System testing*. It's opposite school, called *code correctness testing* or *unit testing*, sees a software as an assembly of components. The main goal of both these approaches is to ensure that the software is functional. Doing so involves covering all executions routes[3] that the software can take, or, said another way, testing the behavior of the software for each possible input. See figure 5.8 for an illustration. Of course, proving a software is fully functional is nearly impossible except in few really simplistic cases, and we can only try to cover as much as executions routes as possible. The number of routes in a program can be measured as for example with *cyclomatic complexity*[92].

5.4.1 System testing

System testing considers the program as an opaque box, shown figure 5.9 receiving inputs and producing outputs according to inputs and context. It can be seen somehow as an experimental proof : to show that the program works, show that it doesn't fail in as many cases as possible. The amount of work of this approach scales with the complexity of its interface and inner structure. The more complex they are, the more tests we will need to write and the more complicated the tests will become. System testing can be extremely efficient for :

- A software that has a really simple structure and a limited amount of functionalities. As it will have only a few pairs of inputs and outputs, covering them all isn't really hard. An example can be a script to extract color profile of an image.
- A software that is mostly composed of trivial to write code. Since such code will rarely have

[3]Execution path would be a better term in most cases, but execution path usually refers to path created by control structures, and may exclude other elements. For example, attempting to divide by zero a floating point number will not result in an error but in producing a Not A Number value, and thus will be considered as being one *path*. In the case of testing, we may want to consider it as two possible paths, *division by zero* and *OK division*, division by zero often being an error. For this reason, we will use the name *route*. Keep in mind that in literature, you probably want to look for *path* and not *route*, as the latter isn't a correct term.

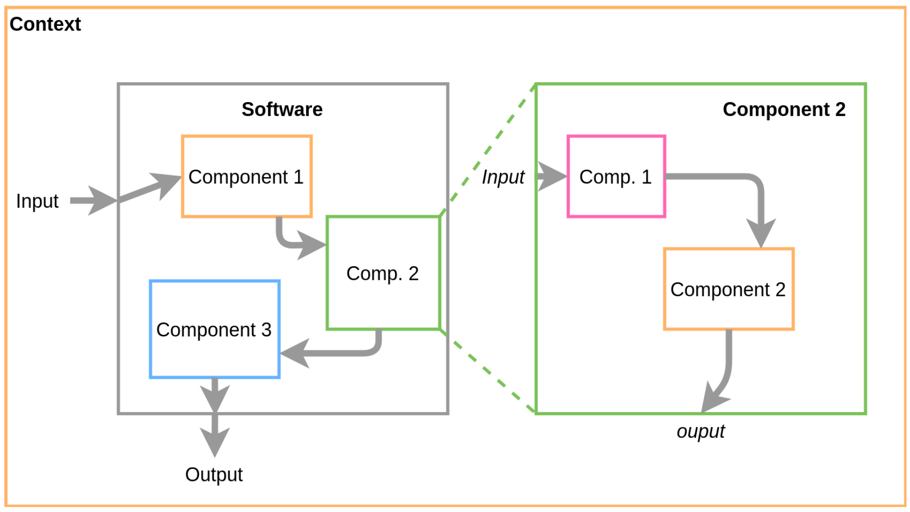


Figure 5.10: A software seen through unit testing

a lot of errors, not covered routes will probably not fail anyway. It can for example be a static web page featuring some trivial scripting.

- A software that has fairly low reliability and stability requirements. Since some errors are allowed, not removing them all is fine. A good example for this can be physics of a video game : it doesn't need more accuracy than the minimum a human, the player, can perceive.

If none of those criterions are met, then unit testing, discussed below, is probably a better choice. System testing is made of nearly only behavioral tests, thus a lot of interface tests, so covering all cases can become costly really quickly. System testing has a double-edged property : it does not take in account internal errors if they don't affect behavior of the software. This means that we may save time by not fixing non-critical errors, but it also means that we may not see growth of technical debt we talked about in section 4.3.3. Another downside is that when tests fails, we don't get much information about where the error comes from. Of course, using a debugger will ease the search process, but it still takes some effort.

5.4.2 Unit testing

Unit testing considers the program as an assembly of components, shown figure 5.10, and aims to check that all components are functioning properly. This vision is recursive : a component is made of components as well. It can be seen somehow as a mathematical proof in which to show that the program works, we show that all components of the program work. Components, referred to as units, will usually be functions, methods and classes. Unit testing requires to isolate each component and to test it with mocks. Testing components separately allows us to drastically reduce the number of execution routes to check in order to cover every possible routes, and thus drastically eases writing test cases.

Listing 5.18: Python sample for route simplification

```python
def guard(number):
	if number == 0 :
		return 1
	return number

def foo(divisor):
	return 10/divisor

def transform(input):
        #returns a float value according to some rules
        #and the "input" string

#main code
in = input("enter a string")
tr = transform(input)
gd = guard(tr)

print(foo(gd))
```

In the previous listing, the critical part is the `guard` function. This function ensures that `foo` will never receive a divisor equal to 0, which would result in an error. When testing this program, we will want to test that `guard` returns 1 when receiving zero and the original value otherwise.

In a system testing approach, we would know that one or our two cases aren't covered by using coverage tools introduced later in section 5.5.4. Such tools would tell us that `return 1` statement in `guard` has never been executed by any of the tests. However, it does not tell us *how* to cover this case. In order to write a relevant test, we have to look into `transform` code to know which input will produce the right value.

With unit approach, everything is much easier, as we directly write test cases to verify *guard* requirements, without caring about other functions. Additionally, if at some point we change the code of `guard` and break it, we will be informed directly that this specific component broke. The cost of writing unit tests scales with the number of components in the software, as well as their need for mocking. For this reason, unit testing can consume a lot of time, and therefore we want to avoid testing everything all the time. The need for unit testing increases, for each component, with :

- Complexity of the output. The harder it is to for a human to validate a pair input-output for a component, the harder it will be to live debug it with a debugger. For example, physics collision detection code is hard to debug.
- Use across the program. The more a component is used, the more testing it becomes worth it, since it will avoid later errors and reduce risk of technical debt. A core component is a good example of widely used component. Obviously, all components of a library are good candidate for unit testing.
- Relations with complexly outputting components. A component depending on or using hard to verify components will be difficult to discriminate against error. As an example, a code processing the result of collision detection and performing physics response may be trivial, however if it fails it is hard to know if collision data were valid or not.
- Ease of testing. If a component requires only few mocking and test cases, why not simply test it ?

We can rarely fully prove that a component if working. Because of this, sole unit testing is most of the time insufficient. To compensate this, we do integration testing as well.

5.4.3 Integration testing

Unit testing tests isolated components. Integration testing tests isolated groups of components together. This difference is pictured in figure 5.10. This definition is recursive, thus system testing can somehow be considered as top level integration testing, and unit testing as bottom level integration testing.

Listing 5.19: Python integration testing

```
#assuming we have these classes
class A :
        #some code here
```

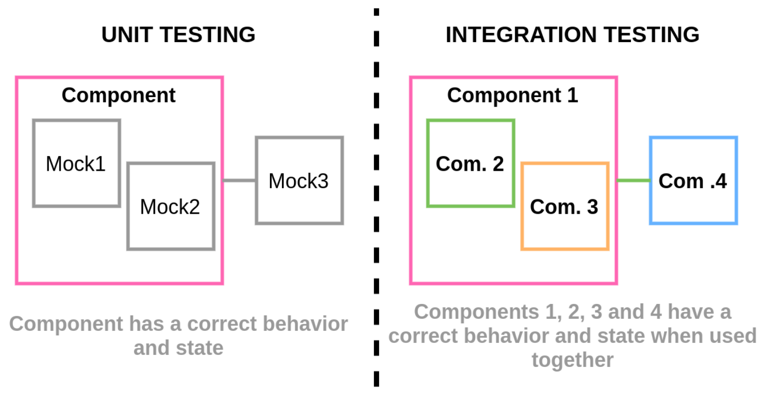

UNIT TESTING
Component
Mock1
Mock2
Mock3
Component has a correct behavior and state
INTEGRATION TESTING
Component 1
Com. 2
Com. 3
Com .4
Components 1, 2, 3 and 4 have a correct behavior and state when used together

Figure 5.11: Difference between unit and integration testing.

```
class B(A):
      #some code here

class C :
      def __init__(self, A):
      #some more code here

#an integration test for would look like that with unittest:

class TestCWithB(TestCase) :

      def test_construction(self):
            sample = C(B())
            #verify stuff here
```

The line between a unit test and an integration test isn't always clear. Is testing a component composed of a fixed set of components considered as unit testing or integrations testing ? On one side, the component is composed of others, so it looks like integration testing. On the other side, its components are fixed and thus cannot be mocked, making it a unit.

Listing 5.20: Python. Is this unit or integration testing?

```
#a component
class Point :
      #some code here

#another component, higher level
class Rectangle :
      def __init__(self, x, y, h, l):
            #point here is fixed, Rectangle cannot be tested without the actual
                ↪ component Point.
            self.origin = Point(x, y)
            self.height = h
            self.length = l
```

Integration testing can also be seen as a form of unit testing if all components but one are heavily tested. In that case, tested components act like mocks to test the remaining component.

Listing 5.21: Python. Integration testing as unit testing

```
class A :
      #some code here

#assuming that B is already heavily tested
class B(A):
      #some code here
```

system	unit and int.	conclusion
pass	pass	Everything seems right.
pass	fail	Error does not seem to affect functional correctness, fixing isn't a priority. Beware of technological debt.
fail	pass	Error seems to affect functional correctness, fixing it is high priority. It has not been detected by unit and integration testing, they are probably insufficient.
fail	fail	Error seems to affect functional correctness, fixing it is high priority. It has been detected by unit and integration testing, thus some component is wrong. Fix should be easy.

Table 5.2: Information in regard of system testing status

integration	unit	conclusion
pass	pass	Everything seems right.
pass	fail	A component has an error, fix should be easy.
fail	pass	Components may have a wrong behavior tested or are insufficiently tested. Improving tests is required.
fail	fail	A component has an error, fix should be easy.

Table 5.3: Information in regard of inner testing levels status

```
class C :
      def __init__(self, A):
      #some more code here

#a unit test using B as a mock for C
class TestC(TestCase) :

      def test_construction(self):
            #B is used as a mock to test C
            sample = C(B())
            #verify stuff here
```

Doing such construction can drastically speed up tests writing. However, if we introduce untested behavior in components acting as mocks later, there is a risk to create some undetected error on unit-like tested component.

5.4.4 Synergy

All test approaches, often called *testing levels* are important to cover. Used wisely in cohabitation, they can drastically improve the overall quality of your software. They also can help you take decisions, as fixing a bug or not. Tables 5.2 and 5.3 show a summary of conclusion we can draw regarding which combination of test levels are passing.

5.5 Reliable testing

Tests are like a protection against bugs. However, tests are imperfect and bugs can still exist, even in a thoroughly tested program. The worst thing that can happen with any safety device is to overestimate its protective power and expose oneself to danger even greater that if no safety device where here. Therefore, we need methods to build tests as reliable as possible, as well as methods to evaluate how reliable our tests are.

A set of test can face two types of problems. First one is a lack of coverage, meaning a functionality or unit isn't tested. Second one is a flaw in one of tests, leading to accepting a result with insufficient verifications.

5.5.1 Low editing

Error probability in any code increases with its size, complexity, and number of editions. It includes tests. The first thing we want with tests is reliability. Tests shall be short. Express directly what you want to verify, fixed values are usually bad in software code but perfectly fine in tests.

Listing 5.22: Python test with and without fixed values

```
class Affine :
    def __self__(self, a, b):
        self.a = a
        self.b = b

    def root(self):
        return -b/a

#this test is so-so, because we rewrite formula, it can cause errors
class TestAffine(TestCase) :
    def test_root(self):
        aff = Affine(2 , 4)
        a = aff.a
        b = aff.b
        self.assertEqual(-b/a, aff.root())

#this test is good, uses fixed value obtained by external means
class TestAffine(TestCase) :
    def test_root(self):
        aff = Affine(2 , 6)
        self.assertEqual(-2, aff.root())
```

Avoid editing tests. Sometimes, it can be better to test later than to have to rewrite tests. This will be discussed in section 5.6. Since testing user interfaces is generally hard, it is advised to decouple interfaces and inner logic. This will be demonstrated in section 5.7.

5.5.2 Sensitivity over specificity

An empty failing test is better than no test at all. If you know something has to be tested, or can't decide yet, write and empty failing test case for it. This way you will know something is left to test upon running tests. Some testing frameworks have a way to express that clearly, and thus reporting the test as not implemented instead of failing. Otherwise, it can easily be simulated.

Listing 5.23: Python not implemented test

```
#assuming we have these classes
class A :
    #some code here

class B(A):
    #some code here

class C :
    def __init__(self, A):
    #some more code here

#an integration test for would look like that with unittest:

class TestCWithB(TestCase) :

    def test_construction(self):
        self.assert(False, "Test not implemented")
```

Such construction allows for experimental and fast prototyping. If unsure whether a solution is fitting the problem or not, why not simply trying it ? In that case, developer wants to write code as few as possible to have an answer quickly, but also to have a prototype easy to convert into a definitive solution. An empty test allows the developer to send a message into the future, stating that when writing some code he skipped testing. When accepting the solution, he then simply needs to implement tests.

5.5.3 External tester

When implementing your own tests, you can be biased. Therefore, it is wise to have an external tester implementing additional tests for what you develop. Please note that tests can be included in a code review process, as described section 4.3.3.

5.5.4 Coverage

Code coverage measures how many execution routes are covered by tests. Route analysis is most of the time done by line or by branches. A branch somehow refers to machine code notions that are out of the scope of this course, but it can be summarized as a conditional execution route[4].

Listing 5.24: Python branches

```
# this conditional expression has four branches
if a and b :
    #if body
#next statement

#they can be seen like that
#1 if a is false, skip to next statement
#2 if a is true, check b
#3 if b is true, execute body
# if be is false, skip to next statement
```

Coverage gives limited information on the quality of the tests. It only points non-tested code that is *actually written*. If for example you forgot to handle some errors, coverage will not tell you anything. If such error handling added, without adding more tests, coverage will *decrease*. It can be quite surprising if you are not aware of this. As an example, the following function computes average of an array, without checking if it is empty or not.

Listing 5.25: Python coverage of non written code

```
def average(array):
    sum = 0
    for (elem in array) :
        sum += array
    return sum / len(array)
```

Let's assume we only wrote one test for it.

Listing 5.26: Python coverage of non written code test

```
class TestCWithB(TestCase) :
    def test_construction(self):
        self.assertEqual(0, average([0, 1, 2, -1, -2]))
```

The line coverage for the above function will be 100%, as all lines are executed. Coverage will not mirror the fact that we simply forgot to check if array is not empty. We see that, and add the check, without writing a test for it.

Listing 5.27: Python coverage of non written code, fixed

```
def average(array):
    if (len(array) == 0):
        return 0
    sum = 0
    for (elem in array) :
```

[4]Or *path*, as mentioned in a note in a previous section of this chapter.

```
        sum += array
    return sum / len(array)
```

The line coverage now decreased, because we added the check and thus increasing line count, without testing it. Coverage can vary in a quite unexpected manner when editing code base, and give the wrong software reliability feeling. Therefore, it is important to remember : it is only showing which written lines are not run during tests, no less, no more. Still, coverage tools are rather simple to set up and therefore having them at hand is always beneficial, as demonstrated later in section 5.7. Low code coverage is a warning that your code may not be sufficiently tested. High code coverage is not a guarantee that your code is sufficiently tested.

5.5.5 Mutation testing

To get an idea of the quality of tests, we can test them using mutants. A mutant is a version of our software where the code has been altered, as for example swapping a `+` for a `-`. A solid test base shall reject the vast majority of mutants of a program. If a mutant passes, then it means some components aren't fully tested[93]. Such practice is called *mutation testing*[94]. Mutants are clearly impossible for humans to generate in sufficient quantity to be useful, but there are tools that do that fairly well for nearly all languages. They can be much trickier to set up than coverage tools, as they are less used. They are a bit overkill for most projects, as mutation testing tends to be extremely slow. This course does not demonstrate such setup, but keep in mind the existence of mutation testing in case you need extremely high reliability on tricky to test code one day.

5.6 Test Driven Development

We have seen in this chapter that tests can act as a support for specification, describing what the software does. We know how important tests are, and how it can be difficult to resist the temptation to skip them and move onto another feature. If we want to avoid untested code, how about writing tests *before* we code ?

5.6.1 The method

Test Driven Development, or *TDD*, is a widely used[5] development method and relies extensively on unit testing. Even if it is strict in its roots, it can have a lot of flavors depending on other methods it is associated with. The main idea is to write code by tiny steps, also called *iterations*, achieving each time a tiny goal set by a short test. Following is an example were we will be writing a mean function for an array by using TDD method.

Let's specify our overall goal. We want to write a function `mean` that takes an array as parameters and returns its mean value. If the input array is empty, function shall return 0.

As we need to take tiny steps, we will address those points separately, starting by the exceptional case. Starting by exceptional case isn't mandatory, but right now it is simpler as this case is trivial. We first write a test to express it.

Listing 5.28: Python TDD writing mean function, step 1 test

```
class TestMean(TestCase) :
    def test_empty(self):
        self.assertEqual(0, mean([]))
```

When we run the test, it shall not pass at first. A test failing, failing to compile or be interpreted is considered as not passing. This is the case of our test, since we did not even write `mean` function. Our test isn't passing, so we can start writing a minimal amount of code to get our test pass.

[5]Remember, widely used isn't a proof that it is good !

Listing 5.29: Python TDD writing mean function, step 1 code

```python
def mean(array):
    if len(array) == 0:
        return 0
```

Now we run our test, it passes. We then run all our test base, it passes as well, that's to be expected since we have only one test. We review our code, it is high quality. A step has been completed, time to move on. Average of not empty array is not that trivial, there are a lot of possibilities to cover. We'll therefore try to have few tests that are representative of the mass, and do one step per test. Why not start by an array with only one element ? First, add a test.

Listing 5.30: Python TDD writing mean function, step 2 test

```python
class TestMean(TestCase) :
    #already existing test
    def test_empty(self):
        self.assertEqual(0, mean([]))

    #new test
    def test_single_element(self):
        self.assertEqual(33, mean([33]))
        self.assertEqual(49, mean([49]))
```

We wrote a new test with two new assertions. TDD purists would probably want a single assertion by step. As long as tests are concise and precise, we can consider that multiple assertions are fine, given they validate the same requirement. Test isn't passing, we can move onto writing our code to pass the test. Minimalistic as always.

Listing 5.31: Python TDD writing mean function, step 2 code

```python
def mean(array):
    if len(array) == 0:
        return 0
    if len(array) == 1 :
        return array[0]
```

Now we run our test, it passes. Then we run all our test base, it passes as well. We review our code, it is high quality. We completed a step. Let's move onto the last one. We finally want to check we can handle more than one element. Time for test writing.

Listing 5.32: Python TDD writing mean function, step 3 test

```python
class TestMean(TestCase) :
    #already existing tests
    def test_empty(self):
        self.assertEqual(0, mean([]))
    def test_single_element(self):
        self.assertEqual(33, mean([33]))
        self.assertEqual(49, mean([49]))

    #new test
    def test_multiple_elements(self):
        self.assertEqual(20, mean([10, 20, 30]))
        self.assertEqual(100, mean([100, 0, 200, 300, -100]))
        self.assertEqual(-3, mean([-3, -3, -3, -3]))
```

Test is not passing. We can implement, always with minimal effort.

Listing 5.33: Python TDD writing mean function, step 3 code

```python
def mean(array):
    if len(array) == 0:
        return 0
    if len(array) == 1 :
        return array[0]
    sum = 0
```

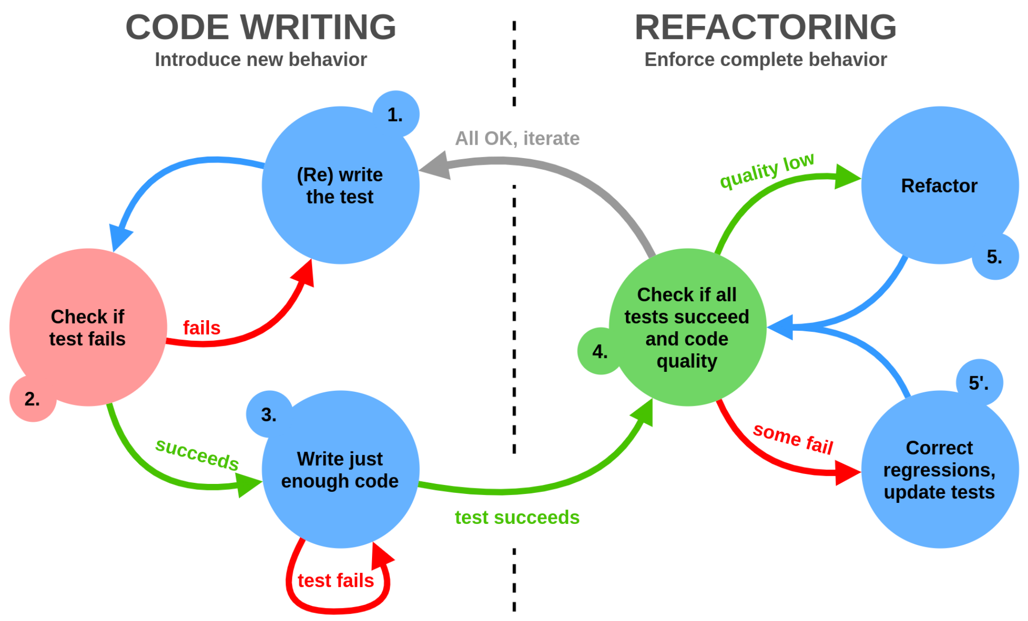


Figure 5.12: Overview of TDD development cycle

```
    for(elem in array) :
        sum += array
    return sum / len(array)
```

Now we run our test, it passes. Then we run all our test base, it passes as well. We review our code, and we find it could be better quality. The for loop handles the 1 element case as well, so we can correct that with a refactoring.

Listing 5.34: Python TDD writing mean function, step 3 refactoring

```
def mean(array):
    if len(array) == 0:
        return 0
    for(elem in array) :
        sum += array
    return sum / len(array)
```

We run our test, it passes. We run all our test base, it passes as well. After a review our code, we conclude it is high quality. One more step completed. Figure[6] 5.12 shows an overview of the TDD development cycle.

5.6.2 Rationale and concerns

Test driven development is, test wise, an extremely reliable development method. By writing tests first, you avoid the risk of forgetting to test what needs to be tested. By unit testing extensively, you remove the subjective decision to choose what to test or not based on your feelings.

Unit testing everything encourages you to decouple components of your software. It also emphases design and refactoring phases we have seen in section 4.3.3. Writing a test before coding is well fit to serve as a support for a design phase, where you consider *what* you want to achieve and not *how*, and review and refactoring are already at the end of a TDD iteration. Small iterations and refactoring often also help prevent technical debt.

[6]Inspired from Xavier Pigeon, `https://en.wikipedia.org/wiki/Test-driven_development#/media/File:TDD_Global_Lifecycle.png`

If you are not extremely confident in your rigor to test your software, and if you plan to use your software in a way where you need reliability, then TDD is likely to be a good choice.

TDD shows its limits when code structure is likely to change a lot, as for example in a prototyping context. A fast changing code structure may lead to intensive test editing, it increases the risk of introducing mistakes in them. In the case of fast prototyping, only low amount of testing is wanted, and TDD therefore isn't fit. Once prototype is validated, actual solution can start back from zero using TDD.

5.7 Example : testing the Zoo application

As a larger example, we will set up a test suite for the zoo program we wrote in chapter 3, section 3.5, Architecture : animal creation.

5.7.1 Preparing for the tests : decoupling

In order to ease testing, we want to decouple our code more than it was before, especially, we want to decouple user interface from inner logic. We will create a class responsible for all inputs and outputs. Let's call it `UserInterface`. Its implementation is trivial.

Listing 5.35: Python Zoo UserInterface

```
class UserInterface :
    def __init__(self):
        pass

    def input(self, message) :
        return input(message)

    def print(self, message):
        print(message)
```

It simply forwards calls to `input` and `print`, and will act as an abstraction allowing us to mock user interface. All functions and methods requiring an interaction with user will then receive a `UserInterface` instance. With these changes, an animal part will look as follows.

Listing 5.36: Python animal part using UserInterface

```
class PawsLicking:
    def __init__(self):
        pass

    def name(self):
        return "paws_licking"

    #ui refers to user interface class
    def be_cute(self, ui):
        ui.print("*licks paws*")
```

Please note that this is not an absolute solution, and not the best solution in a lot of case. It would have been at least as nice to have only a few classes handling interfaces, and have other methods instead returning and gathering values, display being done at a single place at once.

Listing 5.37: Python animals returning and gathering values

```
#please method names are now much less relevant with those changes
class PawsLicking:
    #...
    def be_cute(self):
        return "*licks paws*"

class Animal :
    #...
    def display(self):
        str = ""
        str += self.cuteness.be_cute()
        return str

#somewhere else in code, actual ui code
for animal in animals :
    print(animal.display())
```

The first one is more fit in our case since it will have us perform much less transformation to existing code. You can find full transformed zoo code and all test files in appendix C.

5.7.2 Unit testing animal parts

Animal parts are really simple to test, therefore we will start with them. Create a file `test_parts.py` in `animals/` directory. As part name is trivial to test, we will only demonstrate with the behavior the part carries on. For `cuteness` parts, it is the method `be_cute`. As this method requires an `UserInterface` object, we will mock one by relying on Python ducktyping.

Listing 5.38: Python UserInterface mock for be_cute test

```
class MockUI:
# Mock class that will register output
    def __init__(self):
        self.out = []

    def print(self, message):
        self.out.append(message)
```

This mock simply stores all messages sent to it through its `print` method.
All test will need it, and so we will use the mock as a fixture, by creating a new test class.

Listing 5.39: Python UserInterface as a fixture

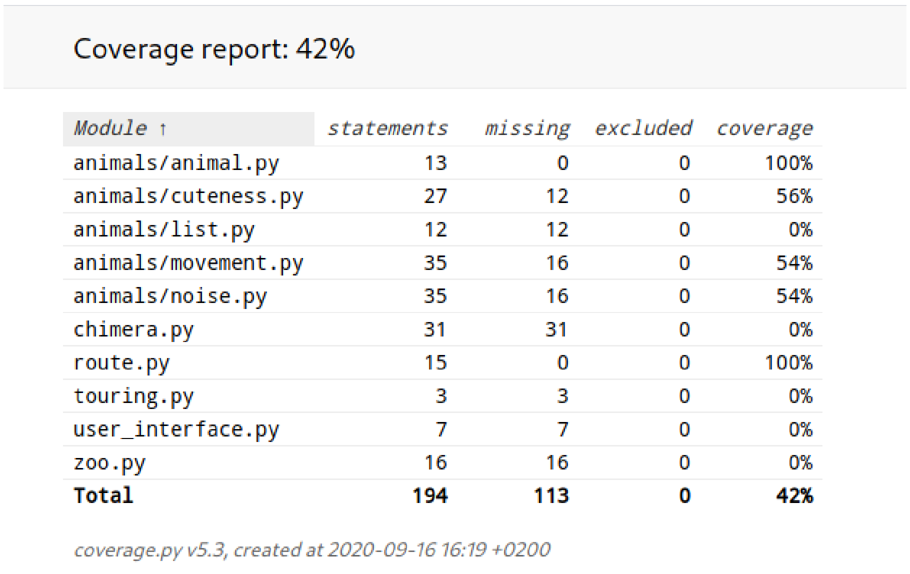
```
class TestPart(TestCase):
    # base class for all part tests that will setup and teardown UI

    def __init__(self, *args, **kwargs):
        #this strange * ** syntax is to simply adapt to any call made on the
            ↪ constructor
        #and transfer arguments to TestCase constructor
        super().__init__(*args, **kwargs)
        self.ui = None #we reserve a slot for ui mock

    def setUp(self):
        self.ui = MockUI() #we create a new UI at setup

    def tearDown(self):
        del self.ui #we delete ui at teardown
```

We will inherit from this class to build our tests.

Listing 5.40: Python testing a part

```
class TestCutenessBeCute(TestPart):
    def test_paws_licking(self):
        pl = PawsLicking()
        pl.be_cute(self.ui) #ui inherited from TestPart
        self.assertListEqual(["*licks paws*"], self.ui.out)
    #more tests here
```

After writing all our tests, we can run them using `python -m unittest` command. Please note we have to use `from animals.cuteness import *` and not `from cuteness import *`. It is mandatory to allow tests to be run from root directory of the project, otherwise we would get a `ModuleNotFoundError`.


Coverage report: 42%

Module ↑	statements	missing	excluded	coverage
animals/animal.py	13	0	0	100%
animals/cuteness.py	27	12	0	56%
animals/list.py	12	12	0	0%
animals/movement.py	35	16	0	54%
animals/noise.py	35	16	0	54%
chimera.py	31	31	0	0%
route.py	15	0	0	100%
touring.py	3	3	0	0%
user_interface.py	7	7	0	0%
zoo.py	16	16	0	0%
Total	**194**	**113**	**0**	**42%**

coverage.py v5.3, created at 2020-09-16 16:19 +0200

Figure 5.13: Example of a html coverage report, front page

5.7.3 Coverage and other setups

Since we have some tests, it will be simpler for us to set coverage tool up, `Coverage.py`[95]. We can install it really quickly with `pip install coverage`. To invoke coverage, we run `coverage run -m` ↪ `unittest`. It should have created a `.coverage` file. This file contains data on the coverage. It can be used to produce a quick report, using `coverage report` command.

Listing 5.41: Python zoo coverage report

```
coverage report
Name                        Stmts   Miss  Cover
-----------------------------------------------
animals/__init__.py             0      0   100%
animals/cuteness.py            27     12    56%
animals/test_parts.py          23      1    96%
-----------------------------------------------
TOTAL                          50     13    74%
```

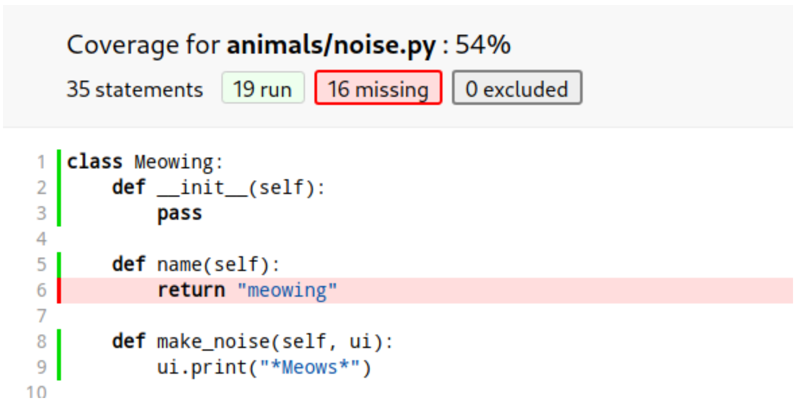
We can also produce a more detailed HTML report with `coverage html` command. It will create a `htmlcov` directory containing all report files. Report can be read using any navigator : `chromium` ↪ `htmlcov/index.html`. Result is seen on figures 5.13 and 5.14. To erase `.coverage` files, we can call `coverage erase` command. Please note it does not remove HTML report files.

We can also notice that coverage actually counts tests as a source file, while non run files are considered as not in the software. To select files to include or exclude, we can write a configuration file for coverage, `.coveragerc` at the root of the project.

Listing 5.42: Zoo .coveragerc

```
[run]
omit =
     */__init__.py
     */test_*

source = .
```

`source = .` tells coverage tool to include any files in the directory, recursively. `omit = ...` specifies to exclude all files matching specified patterns.


Coverage for **animals/noise.py** : 54%

35 statements | 19 run | 16 missing | 0 excluded

```
class Meowing:
    def __init__(self):
        pass

    def name(self):
        return "meowing"

    def make_noise(self, ui):
        ui.print("*Meows*")

```

Figure 5.14: Example of a html coverage report, file details

Listing 5.43: Python zoo coverage report with all files

```
Name                      Stmts   Miss  Cover
---------------------------------------------
animals/animal.py            13     13     0%
animals/cuteness.py          27     12    56%
animals/list.py              12     12     0%
animals/movement.py          35     35     0%
animals/noise.py             35     35     0%
chimera.py                   31     31     0%
route.py                     15     15     0%
touring.py                    3      3     0%
user_interface.py             7      7     0%
zoo.py                       14     14     0%
---------------------------------------------
TOTAL                       192    177     8%
```

To simplify the process of testing and coverage, we can use make tool.

Listing 5.44: Zoo testing makefile

```
default :
#nothing to do actually

#running tests
test :
        python -m unittest

#coverage
cov :
        coverage run -m unittest

#generating coverage report
covr : cov
        coverage report

#generating coverage html report
covhtml : cov
        coverage html

#removing extra files
#|| true allows to bypass error if html cov isn't here, while still informing
clean :
        coverage erase
        rm -r htmlcov || true
```

This way, we can access all we need with simple commands, as `make test`.

5.7.4 Unit and integration testing Animal

Unit testing `Animal` class isn't much harder than testing a part, it only requires a bit more mocking. We have to write a mock for each part type. But before that, let's create a `test_animal.py` file in `animals` package.

Listing 5.45: Python animal unit testing mocks

```python
class MockCuteness:
    def be_cute(self, ui):
        ui.print("cuteness")


class MockMovement:
    def move(self, ui):
        ui.print("movement")


class MockNoise:
    def make_noise(self, ui):
        ui.print("noise")
```

We will also use the same `MockUI` class we used for unit testing parts. We can then write two simple tests.

Listing 5.46: Python unit testing Animal

```python
class TestAnimal(TestCase):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.ui = None
        self.animal = None

    def setUp(self):
        self.ui = MockUI()
        self.animal = Animal("anim", MockNoise(),
                             MockMovement(), MockCuteness())

    def tearDown(self):
        del self.ui
        del self.animal

    def test_introduce(self):
        self.animal.introduce(self.ui)
        self.assertListEqual(["This animal is a anim"], self.ui.out)

    def test_display(self):
        self.animal.display(self.ui)
        self.assertListEqual(
            ["This animal is a anim", "noise", "movement", "cuteness"], self.ui.out
              ↪ )
```

As we use the same `animal` for both tests, it can be considered a fixture, manipulated through `setUp` and `tearDown`. To further check, why don't we implement a simple integration test ? It isn't much different from unit one, simply does not use mocks but parts instead.

Listing 5.47: Python integration testing Animal

```python
class TestAnimalIntegrationAsCat(TestCase):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.ui = None
        self.animal = None

    def setUp(self):
        self.ui = MockUI()
        self.animal = Animal("Cat", Meowing(), AgileRun(), PawsLicking())
    def tearDown(self):
        del self.ui
        del self.animal

    def test_introduce(self):
        self.animal.introduce(self.ui)
        self.assertListEqual(["This animal is a Cat"], self.ui.out)

    def test_display(self):
        self.animal.display(self.ui)
        self.assertListEqual(
            ["This animal is a Cat", "*Meows*", "*Runs and jumps with agility*", "*
                ↪ licks paws*"], self.ui.out)
```

Even if in this example we have put both integration and unit test in the same file, it would probably be better to separate them in a real development context.

5.7.5 Unit testing route planning

Unit testing route planning, e.g. function `ask_route` in lstinlineroute.py file is much harder to test. It is in these cases that a good decoupling and abstraction of the user interface is primordial. We need to use a much more complicated mock interface than what we did before to simulate sequential inputs from the user. But first, we want to create a `test_route.py` file alongside `route.py` file.

Listing 5.48: Python zoo route mock ui

```
class MockUI:
    # Mock class that will register output and provide input sequence
    def __init__(self):
        self.out = []
        self.inputs = []
        self.currentInput = 0

    def input(self, message):
        # each call, we advance in the sequence and return res
        self.out.append(message)
        res = self.inputs[self.currentInput]
        self.currentInput += 1
        return res

    def print(self, message):
        self.out.append(message)

    def set_inputs(self, inputs):
        self.inputs = inputs
```

After that, we will set up a fixed list of animals for all our tests, animals in our case being only string representing their names.

Listing 5.49: Python zoo route test setup

```
class TestAskRoute(TestCase):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.ui = None
        self.animals = None

    def setUp(self):
        self.ui = MockUI()
        self.animals = ["animal1", "animal2", "animal3"]

    def tearDown(self):
        del self.ui
        del self.animals
```

When creating a test, we can then specify several inputs that will be sent one after another in each `ui.input` call.

Listing 5.50: Python zoo route test example

```
    def test_prompt_repeat_on_failure(self):
        self.ui.set_inputs(["animal, animal2", "animal1, animal2"])
        route = ask_route(self.animals, self.ui)
        expected = [
            "please list animals you want to see in order, separated by a comma"]
        expected.append("animals are : animal1, animal2, animal3")
        expected.append("your answer : ")
        expected.append(
            "please list animals you want to see in order, separated by a comma")
        expected.append("animals are : animal1, animal2, animal3")
        expected.append("your answer : ")
        self.assertListEqual(expected, self.ui.out)

    def test_result_on_failure(self):
        self.ui.set_inputs(["animal2, animal", "animal3, animal2"])
        route = ask_route(self.animals, self.ui)
        self.assertListEqual(["animal3", "animal2"], route)
```

In these tests, we send two string consecutively, the first one isn't correctly formatted while the second is.

5.7.6 System testing zoo

System testing zoo will not be made using Python, even if it would be possible, as the main idea is to show how to test any command line interface. It can be done using a bit of `Bash` and `TCL/Expect` ↪ [83] scripting. We will use Bash to run the overall structure of our tests, setup and teardown, while Expect will allow us to send interactive inputs to our zoo. Let's place these tests into a `tests` directory at the root of our project. We will first create a `compare_output.sh` script.

Listing 5.51: Bash script for system testing zoo

```
#!/usr/bin/bash
#$1 is name of script/expected couple
src="./${1}.exp" #name of the script (software call)
res="${1}.res" #name of the result
exp="${1}.expr" #name of expected

#creating commands
run_cmd="${src} > ${res}" #dump script into result
check_cmd="diff ${res} ${exp}" #run diff on result vs expected
clean_cmd="rm ${res}" #remove res

#actually calling commands
eval $run_cmd
eval $check_cmd
r=$? #storing result for later, remember, 0 if diff found no difference
eval $clean_cmd
exit $r #returning result
```

This shall be called with `compare_output.sh testname`, where testname refers to the name of a couple of a `testname.exp` Expect script and a `testname.expr` expected result file. Let's start by Expect script. We need to install Expect. On Manjaro, it can be done using `pacman -S expect` command. Then, we can write an Expect script in a `create_chimera.exp` file.

Listing 5.52: Expect script for system testing zoo

```
#!/usr/bin/expect -f
#line above tells to use expect shell interpreter

#spawns our program to interact with it
```

```
spawn python ../zoo.py

#wait for program to write a ?
expect "?"
#send a y then enter press
send "y\r"
#and so on
expect "your choice :"
send "crawing\r"
expect "your choice :"
send "agile_flight\r"
expect "your choice :"
send "paws_licking\r"
expect ":"
send "tigercraw\r"
expect "(y,n) ?"
send "n\r"
expect "your answer :"
send "tigercraw, cat\r"
#wait for our program to return
expect eof
```

We can now run the Expect script and check it is producing the right output manually.

Listing 5.53: Expect script for system testing zoo output

```
spawn python ../zoo.py
animals : cat, tiger, whale, alpinechough, fishingcat
Do you want to create a chimera (y/n) ?y
Available noises : crawing, ultrasonic_noises, roaring, meowing
your choice : crawing
Available movements : agile_flight, powerful_swimming, powerful_run, agile_run
your choice : agile_flight
Available cutenesses : paws_licking, geysering, feather_shaking
your choice : paws_licking
please name you animal : tigercraw
Do you want to create another chimera (y,n) ?n
please list animals you want to see in order, separated by a comma
animals are : cat, tiger, whale, alpinechough, fishingcat, tigercraw
your answer : tigercraw, cat
This animal is a tigercraw
*Craws*
*Flies, wisely using air currents*
*licks paws*
This animal is a cat
*Meows*
*Runs and jumps with agility*
*licks paws*
```

We will then create a text file called `create_chimera.expr` by dumping Expect script result in a file `./create_chimera.exp > create_chimera.expr` Finally, we can update our Makefile by adding a `systest` target.

Listing 5.54: Makefile update for zoo system testing

```
test : utest systest

#was test target previously
utest :
    python -m unittest

systest :
    #calling compare_output.sh on relevant pairs
    cd tests && ./compare_output.sh create_chimera
```

We manually checked the result of our software and then used it as a test. This method, has to be used wisely as it is prone to create erroneous tests. However, if user with care, it can drastically increase writing speed of certain tests, as is sometime much simpler to verify a result rather than anticipating it, especially when dealing with user interfaces.

Changelog

Version 1.0

Initial version :

- Introduction and acknowledgments
- Added *About this course* chapter
- Added *Development environment* chapter
- Added *Programming* chapter
- Added *Code quality* chapter
- Added *Testing* chapter

Contributors

Contributors are not shown in a particular order.

Edward Andó, edward.ando@3sr-grenoble.fr
v1.0 : Helped with early thinking on needs and plan.

Marc Sousbie, marc.sousbie@protonmail.com
v1.0 : *Programming* chapter review.

Nicolas Pamart, nicolaspmt@gmail.com
v1.0 : *Code quality* chapter review.

Remy Kilindjian, rem.kilindj@gmail.com
v1.0 : *Programming* chapter beginner review.

François Dupont, francoisdupont05@gmail.com
v1.0 : *Development environment* chapter review, especially Python flaws.

Loïc Wisniewski, lpw.wisniewski@gmail.com
v1.0 : *Testing* chapter review.

Funders

Funders are not shown in a particular order. They greatly helped existence of this course by ordering paid services related to it from author, as for example course supervision or student support.

3SR Laboratory, Grenoble
Supervised course, 4 sessions of 4 hours

Free software used

List is non-exhaustive and not in a particular order.

LaTeX, `https://www.latex-project.org/`
Typesetting support for the whole document and PDF rendering.

---

latexmk, `https://www.ctan.org/pkg/latexmk/`
Easier and automated LaTeX building

Inkscape, `https://inkscape.org/`
Used for diagrams.

Draw.io desktop, `https://github.com/jgraph/drawio-desktop`
Used for diagrams.

Draw.io desktop, `https://rsync.samba.org/`
In build stage, used to gather examples with python cache exclusion.

Bibliography

Wikipedia

[1] *Wikipedia page of software library.* URL: https://en.wikipedia.org/wiki/Library_(computing). (accessed: 04.10.2020).

[2] *Wikipedia page of software framework.* URL: https://en.wikipedia.org/wiki/Software_framework. (accessed: 04.10.2020).

[3] *Wikipedia page of software programming language.* URL: https://en.wikipedia.org/wiki/Programming_language. (accessed: 04.10.2020).

[4] *Wikipedia page of software programming paradigm.* URL: https://en.wikipedia.org/wiki/Programming_paradigm. (accessed: 04.10.2020).

[13] *Wikipedia page of Make.* URL: https://en.wikipedia.org/wiki/Make_(software). (accessed: 04.10.2020).

[51] *Wikipedia page of the switch statement.* URL: https://en.wikipedia.org/wiki/Switch_statement. (accessed: 05.10.2020).

[54] *Wikipedia page about side effect.* URL: https://en.wikipedia.org/wiki/Side_effect_(computer_science). (accessed: 05.10.2020).

[68] *Wikipedia page about elegance.* URL: https://en.wikipedia.org/wiki/Elegance. (accessed: 06.10.2020).

[87] *Wikipedia page about continuous testing.* URL: https://en.wikipedia.org/wiki/Continuous_testing. (accessed: 11.10.2020).

[92] *Wikipedia page about cyclomatic complexity.* URL: https://en.wikipedia.org/wiki/Cyclomatic_complexity. (accessed: 05.10.2020).

[94] *Wikipedia page about mutation testing.* URL: https://en.wikipedia.org/wiki/Mutation_testing. (accessed: 11.10.2020).

Online resources

[5] *Medium article on compilation and interpretation.* URL: https://medium.com/basecs/a-deeper-inspection-into-compilation-and-interpretation-d98952ebc842. (accessed: 04.10.2020).

[7] *Stack Overflow thread about speeding up python.* URL: https://stackoverflow.com/questions/138521/is-it-feasible-to-compile-python-to-machine-code. (accessed: 04.10.2020).

[17] *The register article about of slavery related terms in Linux kernel.* URL: https://www.theregister.com/2020/07/13/linux_adopts_inclusive_language/. (accessed: 04.10.2020).

[18] *Git commit introducing recommendation related to inclusive terms in Linux kernel.* URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=49decddd39e5f6132ccd7d9fdc3d7c470b0061bb. (accessed: 04.10.2020).

[19] *ZDNet article on GitHub master term changes.* URL: https://www.zdnet.com/article/github-to-replace-master-with-alternative-term-to-avoid-slavery-references/. (accessed: 04.10.2020).

[20] *Stack Exchange discussion about Git master term.* URL: https://english.stackexchange.com/questions/474419/does-the-term-master-in-git-the-vc-system-refer-to-slavery. (accessed: 04.10.2020).

[24] *VSCode GitHub repository.* URL: https://www.jetbrains.com/community/education/#students. (accessed: 04.10.2020).

[28] *GitLab difference between CE and EE explained.* URL: https://about.gitlab.com/install/ce-or-ee/. (accessed: 05.10.2020).

[38] *Text of the Apache License 2.0.* URL: http://www.apache.org/licenses/LICENSE-2.0. (accessed: 05.10.2020).

[39] *Summary of the GNU GPL license family.* URL: https://www.gnu.org/licenses/licenses.html. (accessed: 05.10.2020).

[40] *GNU GPLv3 license explained.* URL: https://www.gnu.org/licenses/quick-guide-gplv3.html. (accessed: 05.10.2020).

[41] *GNU GPLv3 license text.* URL: https://www.gnu.org/licenses/gpl-3.0.txt. (accessed: 05.10.2020).

[42] *GNU AGPL license explained.* URL: https://www.gnu.org/licenses/why-affero-gpl.html. (accessed: 05.10.2020).

[43] *GNU AGPL license text.* URL: https://www.gnu.org/licenses/lgpl-3.0.txt. (accessed: 05.10.2020).

[44] *GNU LGPL licence usage concerns.* URL: https://www.gnu.org/licenses/why-not-lgpl.html. (accessed: 05.10.2020).

[45] *Python community wiki about Python 3 and Python 2.* URL: https://wiki.python.org/moin/Python2orPython3. (accessed: 05.10.2020).

[47] *Wikibooks showing extremely simple scripts in Python.* URL: https://en.wikibooks.org/wiki/Python_Programming/Creating_Python_Programs. (accessed: 05.10.2020).

[48] *Stack Overflow discussion about Python main method.* URL: https://stackoverflow.com/questions/22492162/understanding-the-main-method-of-python. (accessed: 05.10.2020).

[60] *jitterphysics GitHub wiki page about sweep and prune algorithm.* URL: https://github.com/mattleibow/jitterphysics/wiki/Sweep-and-Prune. (accessed: 05.10.2020).

[61] *Code Climate documentation article about their way to measure code cognitive complexity.* URL: https://docs.codeclimate.com/docs/cognitive-complexity. (accessed: 05.10.2020).

[71] *CodeClimate's quality pricing page, free for open source projects.* URL: https://codeclimate.com/quality/pricing/. (accessed: 05.10.2020).

[73] *Stack Exchange thread discussing what a fixture is.* URL: https://softwareengineering.stackexchange.com/questions/211959/what-are-the-different-meanings-of-fixture. (accessed: 11.10.2020).

[74] *Stack Overflow thread discussing what a fixture is.* URL: https://stackoverflow.com/questions/12071344/what-are-fixtures-in-programming. (accessed: 11.10.2020).

[75] *Stack Exchange thread discussing what a fixture is.* URL: https://sqa.stackexchange.com/questions/13039/what-is-a-fixture. (accessed: 11.10.2020).

[76] *Stack Overflow thread about what mocking is.* URL: https://stackoverflow.com/questions/2665812/what-is-mocking?. (accessed: 11.10.2020).

[77] *Stack Overflow thread on clarifying mocking.* URL: https://stackoverflow.com/questions/214092/what-is-a-mock-and-when-should-you-use-it. (accessed: 11.10.2020).

[78] *Medium article discussing use of mocks.* URL: https://medium.com/javascript-scene/mocking-is-a-code-smell-944a70c90a6a. (accessed: 11.10.2020).

[79] *Mocks Aren't Stubs, clarification of what are mocks and stubs.* URL: https://martinfowler.com/articles/mocksArentStubs.html. (accessed: 11.10.2020).

[80] *Stack Exchange thread discussing how to test randomness.* URL: https://softwareengineering.stackexchange.com/questions/147134/how-should-i-test-randomness. (accessed: 11.10.2020).

[82] *Quora thread about use of automation testing.* URL: https://www.quora.com/When-should-you-use-automation-testing. (accessed: 11.10.2020).

[88] *Continuous integration, article presenting it.* URL: https://martinfowler.com/articles/continuousIntegration.html. (accessed: 11.10.2020).

[93] *Article demonstrating mutation testing in JavaScript.* URL: https://opensource.com/article/19/9/mutation-testing-example-definition. (accessed: 11.10.2020).

Free and Open Source Software tools

[8] *Vim website.* URL: https://www.vim.org/. (accessed: 04.10.2020).

[9] *Emacs website.* URL: https://www.gnu.org/software/emacs/. (accessed: 04.10.2020).

[10] *VSCode website.* URL: https://code.visualstudio.com/. (accessed: 04.10.2020).

[11] *VSCode GitHub repository.* URL: https://github.com/microsoft/vscode. (accessed: 04.10.2020).

[12] *VSCodium, VSCode without telemetry, GitHub repository.* URL: https://github.com/VSCodium/vscodium. (accessed: 04.10.2020).

[14] *SVN website.* URL: https://subversion.apache.org/. (accessed: 04.10.2020).

[15] *Git website.* URL: https://git-scm.com/. (accessed: 04.10.2020).

[16] *Mercurial website.* URL: https://www.mercurial-scm.org/. (accessed: 04.10.2020).

[21] *CodeBlocks website.* URL: http://www.codeblocks.org/. (accessed: 04.10.2020).

[23] *Intellij Idea community edition GitHub repository.* URL: https://github.com/JetBrains/intellij-community. (accessed: 04.10.2020).

[25] *GitLab website.* URL: https://about.gitlab.com/. (accessed: 05.10.2020).

[27] *GitLab's full open source GitLab repository.* URL: https://gitlab.com/gitlab-org/gitlab-foss/. (accessed: 05.10.2020).

[29] *Gitea website.* URL: https://gitea.io/en-us/. (accessed: 05.10.2020).

[30] *Gitea GitHub repository.* URL: https://github.com/go-gitea/gitea. (accessed: 05.10.2020).

[65] *Doxygen website.* URL: https://www.doxygen.nl/index.html. (accessed: 05.10.2020).

[69] *pylint pypi project page.* URL: https://pypi.org/project/pylint/. (accessed: 05.10.2020).

[72] *Clang Format main documentation page.* URL: https://clang.llvm.org/docs/ClangFormat.html. (accessed: 05.10.2020).

[83] *Expect website.* URL: https://core.tcl-lang.org/expect/home. (accessed: 11.10.2020).

[84] *Qt website. Qt has an open source and a proprietary version.* URL: https://www.qt.io/. (accessed: 11.10.2020).

[85] *Qt GitHub mirror. Qt has an open source and a proprietary version.* URL: https://github.com/qt/qt5. (accessed: 11.10.2020).

Proprietary tools

[22] *PyCharm website.* URL: https://www.jetbrains.com/pycharm/. (accessed: 04.10.2020).

[31] *GitHub website.* URL: https://github.com/. (accessed: 05.10.2020).

[70] *CodeClimate's quality home page.* URL: https://codeclimate.com/login/github/join. (accessed: 05.10.2020).

Documentation

[32] *GitLab permissions documentation page.* URL: https://docs.gitlab.com/ce/user/permissions.html. (accessed: 05.10.2020).

[33] *GitLab merge request documentation page.* URL: https://docs.gitlab.com/ce/user/project/merge_requests/. (accessed: 05.10.2020).

[34] *GitLab issues documentation page.* URL: https://docs.gitlab.com/ce/user/project/issues/. (accessed: 05.10.2020).

[35] *GitLab issue boards documentation page.* URL: https://docs.gitlab.com/ce/user/project/issue_board.html. (accessed: 05.10.2020).

[36] *GitLab wiki documentation page.* URL: https://docs.gitlab.com/ce/user/project/wiki/. (accessed: 05.10.2020).

[37] *GitLab web IDE documentation page.* URL: https://docs.gitlab.com/ce/user/project/web_ide/. (accessed: 05.10.2020).

[46] *w3schools page on Python comments.* URL: https://www.w3schools.com/python/python_comments.asp. (accessed: 05.10.2020).

[49] *Official Python documentation page about data structures.* URL: https://docs.python.org/3/tutorial/datastructures.htm. (accessed: 05.10.2020).

[50] *w3schools page on Python variable scopes.* URL: https://www.w3schools.com/python/python_scope.asp. (accessed: 05.10.2020).

[52] *Golang switch explanation.* URL: https://tour.golang.org/flowcontrol/9. (accessed: 05.10.2020).

[53] *C++ switch documentation.* URL: https://en.cppreference.com/w/cpp/language/switch. (accessed: 05.10.2020).

[55] *Python 3.5 modules official documentation.* URL: https://docs.python.org/3.5/tutorial/modules.html. (accessed: 05.10.2020).

[56] *Python namespace packages official documentation.* URL: https://docs.python.org/3.5/tutorial/modules.html. (accessed: 05.10.2020).

[57] *C++ inheritance documentation.* URL: http://www.cplusplus.com/doc/tutorial/inheritance/. (accessed: 05.10.2020).

[58] *w3school page on java abstractions.* URL: https://www.w3schools.com/java/java_abstract.asp. (accessed: 05.10.2020).

[59] *w3school page on java interfaces.* URL: https://www.w3schools.com/java/java_interface.asp. (accessed: 05.10.2020).

[62] *Python development guidelines.* URL: https://www.python.org/dev/peps/pep-0008/#a-foolish-consistency-is-the-hobgoblin-of-little-minds. (accessed: 06.10.2020).

[63] *C++ Core Guidelines.* URL: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuideliness. (accessed: 06.10.2020).

[64] *Python docstrings conventions.* URL: https://www.python.org/dev/peps/pep-0257/. (accessed: 06.10.2020).

[66] *Doxygen documentation on comment formatting.* URL: `https://www.doxygen.nl/manual/docblocks.html`. (accessed: 06.10.2020).

[67] *UML standard specifications reference page.* URL: `https://www.omg.org/spec/UML`. (accessed: 06.10.2020).

[81] *Python random module documentation.* URL: `https://docs.python.org/3/library/random.html`. (accessed: 11.10.2020).

[86] *Python unittest documentation main page.* URL: `https://docs.python.org/3.8/library/unittest.html`. (accessed: 11.10.2020).

[89] *Git Hooks documentation.* URL: `https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks`. (accessed: 11.10.2020).

[90] *GitLab CI documentation.* URL: `https://docs.gitlab.com/ce/ci/`. (accessed: 11.10.2020).

[91] *GitLab CI pipeline schedule documentation.* URL: `https://docs.gitlab.com/ce/ci/pipelines/schedules.html`. (accessed: 11.10.2020).

[95] *Coverage.py documentation.* URL: `https://coverage.readthedocs.io/en/coverage-5.3/`. (accessed: 11.10.2020).

Code repositories

[6] *GitHub of Cling, a C++ interpreter.* URL: `https://github.com/root-project/cling`. (accessed: 04.10.2020).

[11] *VSCode GitHub repository.* URL: `https://github.com/microsoft/vscode`. (accessed: 04.10.2020).

[12] *VSCodium, VSCode without telemetry, GitHub repository.* URL: `https://github.com/VSCodium/vscodium`. (accessed: 04.10.2020).

[23] *Intellij Idea community edition GitHub repository.* URL: `https://github.com/JetBrains/intellij-community`. (accessed: 04.10.2020).

[26] *GitLab's GitLab repository.* URL: `https://gitlab.com/gitlab-org/gitlab`. (accessed: 05.10.2020).

[27] *GitLab's full open source GitLab repository.* URL: `https://gitlab.com/gitlab-org/gitlab-foss/`. (accessed: 05.10.2020).

[30] *Gitea GitHub repository.* URL: `https://github.com/go-gitea/gitea`. (accessed: 05.10.2020).

[85] *Qt GitHub mirror. Qt has an open source and a proprietary version.* URL: `https://github.com/qt/qt5`. (accessed: 11.10.2020).

Appendix A

Chimera zoo Python code

A.1 Basic programming full example code

Listing A.1: Basic programming zoo.py

```python
# defining animals
def display_cat():
    print("This animal is a cat :")
    print("*meows*")
    print("*runs and jumps with agility*")
    print("*licks paws*")


def display_whale():
    print("This animal is a whale :")
    print("*inaudible ultrasonic noises*")
    print("*swims with ample and powerful movements*")
    print("*shoots water in the air*")


def display_tiger():
    print("This animal is a tiger :")
    print("*roars*")
    print("*walks with grace and confidence*")
    print("*licks paws*")


def display_alpine_chough():
    print("This animal is an alpine chough :")
    print("*craws loudly*")
    print("*shows his mastery of air flow along the cliff*")
    print("*shakes feathers*")


# listing animals
def list_animals():
    # simply listing their names
    animals = ["cat", "whale", "tiger", "alpinechough"]
    return animals


# route stuff
def ask_route(animals):  # note that now we pass the animal list as a parameter

    valid = False  # set to false to execute while loop once
    route = []

    while not valid:
        valid = True
        print("please list animals you want to see in order, separated by a comma")
        answer = input("your answer : ")
        answer = answer.replace(" ", "")
        route = answer.split(",")
        for animal in route:
            valid = valid and (animal in animals)
    # at this point, we know route is valid
    return route  # return route


# touring stuff
def take_tour(route):
    for animal in route:
        if(animal == "cat"):
            display_cat()
        elif(animal == "whale"):
            display_whale()
        elif(animal == "tiger"):
            display_tiger()
        elif(animal == "alpinechough"):
            display_alpine_chough()
        else:
            print("animal ", animal, " not found")


# main code
animals = list_animals()
route = ask_route(animals)
take_tour(route)
```

A.2 Basic programming full example code using dictionnary

Listing A.2: Basic programming zoo.py using dictionnary

```
# in this version, animals are represented using a dictionnary where
# actions they perform are their keys

# animal stuff
def create_animal(animals, name, noise, movement, cute_action):
    # adding a value to the animals dict and returning it
    dict = animals
    animal = {"name": name, "noise": noise,
              "move": movement, "cute": cute_action}
    dict[name] = animal
    return dict


def list_animal_names(animals):
    list = []
    for key in animals:
        list.append(key)
    return list


def display_animal(animal):
    print("This animal is a", animal["name"])
    print("*", animal["noise"], "*")
    print("*", animal["move"], "*")
    print("*", animal["cute"], "*")


# route stuff
def ask_route(animaldict):  # note that now we pass the animal list as a parameter
```

```python
    animals = list_animal_names(animaldict)
    sep = ", "  # for displaying list of animals
    animal_string = sep.join(animals)

    valid = False  # set to false to execute while loop once
    route = []

    while not valid:
        valid = True
        print("please list animals you want to see in order, separated by a comma")
        print("animals are : ", animal_string)
        answer = input("your answer : ")
        answer = answer.replace(" ", "")
        route = answer.split(",")
        for animal in route:
            valid = valid and (animal in animals)

    # at this point, we know route is valid
    return route  # return route

# touring stuff


def take_tour(route, animals):
    for animal_name in route:
        display_animal(animals[animal_name])


# main code
animals = {}
# creating animals, changing descriptions to keep lines short
animals = create_animal(animals, "cat", "meows", "agile run", "licks paws")
animals = create_animal(animals, "whale", "whale noise",
                        "powerful swimming", "water geyser")
animals = create_animal(animals, "tiger", "roar", "powerful run", "licks paws")
animals = create_animal(animals, "alpinechough", "craws",
                        "agile fly", "shakes feathers")
route = ask_route(animals)
take_tour(route, animals)
```

A.3 Object programming full example code

Listing A.3: Object programming zoo.py

```python
# animals
class Animal():
    def __init__(self):
        pass  # nothing to do

    def display(self):
        self.introduce()
        self.make_noise()
        self.move()
        self.be_cute()

    def introduce(self):
        print("This animal is a ", self.name())

    def name(self):
        return "unknown specie"

    def be_cute(self):
        print("*does nothing*")
    def move(self):
        print("*does nothing*")

    def make_noise(self):
        print("*stays silent*")

# feline class : feline licks their paws and that's cute


class Feline(Animal):
    def __init__(self):
        pass  # nothing to

    def be_cute(self):
        print("*licks paws*")

# specific feline


class Cat(Feline):
    def __init__(self):
        pass  # nothing to do

    def name(self):  # redefining name method to change name of the class
        return "cat"

    def make_noise(self):
        print("*meows*")

    def move(self):
        print("*runs and jumps with agility*")


class Whale(Animal):
    def __init__(self):
        pass  # nothing to do

    def name(self):  # redefining name method to change name of the class
        return "whale"

    def make_noise(self):
        print("*ultrasonic noises*")

    def move(self):
        print("*swimms with powerful noises*")

    def be_cute(self):
        print("*expels water from its back*")


class Tiger(Feline):
    def __init__(self):
        pass  # nothing to do

    def name(self):
        return "tiger"

    def make_noise(self):
        print("*roars*")

    def move(self):
        print("*runs with powerful jumps*")


class AlpineChough(Animal):
    def __init__(self):
        pass  # nothing to do
    def name(self):
        return "alpinechough"

    def make_noise(self):
        print("*craws*")

    def move(self):
        print("*flies with agility following airflow*")

    def be_cute(self):
        print("*shakes feathers*")


def list_animals():
    animals = {}  # dictionnary
    # simply creating the class corresponding to the animal we want
    animals["cat"] = Cat()
    animals["whale"] = Whale()
    animals["tiger"] = Tiger()
    animals["alpinechough"] = AlpineChough()
    return animals

# extracting animals names


def list_animals_names(animals):
    names = []
    for key in animals:  # iteration over a dictionnary is done using keys
        names.append(key)  # simply listing keys, easy !
    return names

# route stuff


def ask_route(animaldict):  # note that now we pass the animal list as a parameter

    sep = ", "  # for displaying list of animals
    animal_string = sep.join(animals)

    valid = False  # set to false to execute while loop once
    route = []

    while not valid:
        valid = True
        print("please list animals you want to see in order, separated by a comma")
        print("animals are : ", animal_string)
        answer = input("your answer : ")
        answer = answer.replace(" ", "")
        route = answer.split(",")
        for animal in route:
            valid = valid and (animal in animals)

    # at this point, we know route is valid
    return route  # return route


# touring
def take_tour(route, animals):
    for animal in route:
        # accessing the animal by its name and displaying it
        animals[animal].display()


# main code
animals = list_animals()
animals_names = list_animals_names(animals)
```

```
route = ask_route(animals_names)
take_tour(route, animals)
```

A.4 Architecture full example code

Add stuff here !

Appendix B

Chimera zoo architecture full C++ code

please fill

Appendix C

Testing the zoo Python code

C.1 Root of the project

Listing C.1: .coveragerc

```
[run]
omit =
     */__init__.py
     */test_*

source = .
```

Listing C.2: chimera.py

```python
from animals.cuteness import list_cutenesses
from animals.movement import list_movements
from animals.noise import list_noises
from animals.animal import Animal


def list_keys_str(dict):
    list = []
    for key in dict:
        list.append(key)
    glu = ", "
    return glu.join(list)


def create_chimeras(animals, ui):
    noises = list_noises()
    movements = list_movements()
    cutenesses = list_cutenesses()

    noise_query = "Available noises : " + list_keys_str(noises)
    cuteness_query = "Available cutenesses : " + list_keys_str(cutenesses)
    movement_query = "Available movements : " + list_keys_str(movements)

    ui.print("animals : " + list_keys_str(animals))
    answer = ui.input("Do you want to create a chimera (y/n) ?")
    cont = (answer == "y")
    while cont:
        ui.print(noise_query)
        noise = ui.input("your choice : ")
        ui.print(movement_query)
        movement = ui.input("your choice : ")
        ui.print(cuteness_query)
        cuteness = ui.input("your choice : ")
        name = ui.input("please name you animal : ")
        animals[name] = Animal(name, noises[noise],
                               movements[movement], cutenesses[cuteness])
        answer = ui.input("Do you want to create another chimera (y,n) ?")
        cont = (answer == "y")
```

Listing C.3: Makefile

```
default :
	#nothing to do actually

#running tests
test : utest systest

utest :
	python -m unittest

systest :
	cd tests && ./compare_output.sh create_chimera

#coverage
cov :
	coverage run -m unittest

#generating coverage report
covr : cov
	coverage report

#generating coverage html report
covhtml : cov
	coverage html

#removing extra files
#|| true allows to bypass error if html cov isn't here, while still informing
clean :
	coverage erase
	rm -r htmlcov || true
```

Listing C.4: route.py

```
def ask_route(animals, ui):  # note that now we pass the animal list as a parameter

    sep = ", "  # for displaying list of animals
    animal_string = sep.join(animals)

    valid = False  # set to false to execute while loop once
    route = []

    while not valid:
        valid = True
        ui.print("please list animals you want to see in order, separated by a
            ↪ comma")
        ui.print("animals are : " + animal_string)
        answer = ui.input("your answer : ")
        answer = answer.replace(" ", "")
        route = answer.split(",")
        for animal in route:
            valid = valid and (animal in animals)

    # at this point, we know route is valid
    return route  # return route
```

Listing C.5: test_route.py

```
from unittest import TestCase
from unittest import main
from route import ask_route


class MockUI:
    # Mock class that will register output and provide input sequence
    def __init__(self):
        self.out = []
        self.inputs = []
        self.currentInput = 0

    def input(self, message):
        # each call, we advance in the sequence and return res
        self.out.append(message)
        res = self.inputs[self.currentInput]
        self.currentInput += 1
        return res

    def print(self, message):
        self.out.append(message)

    def set_inputs(self, inputs):
        self.inputs = inputs


class TestAskRoute(TestCase):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.ui = None
        self.animals = None

    def setUp(self):
        self.ui = MockUI()
        self.animals = ["animal1", "animal2", "animal3"]

    def tearDown(self):
        del self.ui
        del self.animals

    def test_prompt(self):
        self.ui.set_inputs(["animal1, animal2"])
        route = ask_route(self.animals, self.ui)
        self.assertListEqual(
            ["please list animals you want to see in order, separated by a comma",
            ↪ "animals are : animal1, animal2, animal3", "your answer : "],
            ↪ self.ui.out)

    def test_result_with_space(self):
        self.ui.set_inputs(["animal1, animal2"])
        route = ask_route(self.animals, self.ui)
        self.assertListEqual(["animal1", "animal2"], route)

    def test_result_no_space(self):
        self.ui.set_inputs(["animal3,animal1"])
        route = ask_route(self.animals, self.ui)
        self.assertListEqual(["animal3", "animal1"], route)

    def test_prompt_repeat_on_failure(self):
        self.ui.set_inputs(["animal, animal2", "animal1, animal2"])
        route = ask_route(self.animals, self.ui)
        expected = [
            "please list animals you want to see in order, separated by a comma"]
        expected.append("animals are : animal1, animal2, animal3")
        expected.append("your answer : ")
        expected.append(
            "please list animals you want to see in order, separated by a comma")
        expected.append("animals are : animal1, animal2, animal3")
        expected.append("your answer : ")
        self.assertListEqual(expected, self.ui.out)

    def test_result_on_failure(self):
        self.ui.set_inputs(["animal2, animal", "animal3, animal2"])
        route = ask_route(self.animals, self.ui)
        self.assertListEqual(["animal3", "animal2"], route)


class TestRouteCorrection(TestCase):
    pass


if __name__ == '__main__':
    unittest.main()
```

Listing C.6: touring.py

```
# touring
def take_tour(route, animals, ui):
    for animal in route:
        # accessing the animal by its name and displaying it
        animals[animal].display(ui)
```

Listing C.7: user_interface.py

```
class UserInterface:
    def __init__(self):
        pass

    def input(self, message):
        return input(message)

    def print(self, message):
        print(message)
```

Listing C.8: zoo.py

```
from route import ask_route
from touring import take_tour
from chimera import create_chimeras
from animals.list import list_animals
from user_interface import UserInterface


def list_animals_names(animals):
    names = []
    for key in animals:  # iteration over a dictionnary is done using keys
        names.append(key)  # simply listing keys, easy !
    return names


ui = UserInterface()
animals = list_animals()
create_chimeras(animals, ui)
animals_names = list_animals_names(animals)
route = ask_route(animals, ui)
take_tour(route, animals, ui)
```

C.2 *animals* directory

Listing C.9: animal.py

```python
class Animal:
    def __init__(self, name, noise, movement, cuteness):
        self.name = name
        self.cuteness = cuteness
        self.noise = noise
        self.movement = movement

    def introduce(self, ui):
        ui.print("This animal is a " + self.name)

    def display(self, ui):
        self.introduce(ui)
        self.noise.make_noise(ui)
        self.movement.move(ui)
        self.cuteness.be_cute(ui)
```

Listing C.10: cuteness.py

```python
class PawsLicking:
    def __init__(self):
        pass

    def name(self):
        return "paws_licking"

    def be_cute(self, ui):
        ui.print("*licks paws*")


class Geysering:
    def __init__(self):
        pass

    def name(self):
        return "geysering"

    def be_cute(self, ui):
        ui.print("*shoots water in the air")


class FeatherShaking:
    def __init__(self):
        pass

    def name(self):
        return "feather_shaking"

    def be_cute(self, ui):
        ui.print("*shakes feathers*")


def list_cutenesses():
    ctns = {}
    ctns["paws_licking"] = PawsLicking()
    ctns["geysering"] = Geysering()
    ctns["feather_shaking"] = FeatherShaking()
    return ctns
```

Listing C.11: list.py

```python
from .noise import (Crawing, Roaring, Roaring, Meowing, UltrasonicNoises)
from .movement import (AgileFlight, PowerfulRun, PowerfulSwimming, AgileRun)
from .cuteness import (PawsLicking, Geysering, FeatherShaking)
from .animal import Animal
```

```python
def list_animals():
    animals = {}
    animals["cat"] = Animal("cat", Meowing(), AgileRun(), PawsLicking())
    animals["tiger"] = Animal("tiger", Roaring(), PowerfulRun(), PawsLicking())
    animals["whale"] = Animal(
        "whale", UltrasonicNoises(), PowerfulSwimming(), Geysering())
    animals["alpinechough"] = Animal(
        "alpinechough", Crawing(), AgileFlight(), FeatherShaking())
    animals["fishingcat"] = Animal("fishingcat", Meowing(),
                                   PowerfulSwimming(), PawsLicking())
    return animals
```

Listing C.12: movement.py

```python
class AgileRun:
    def __init__(self):
        pass

    def name(self):
        return "agile_run"

    def move(self, ui):
        ui.print("*Runs and jumps with agility*")


class PowerfulRun:
    def __init__(self):
        pass

    def name(self):
        return "powerful_run"

    def move(self, ui):
        ui.print("*Runs with powerful movements*")


class PowerfulSwimming:
    def __init__(self):
        pass

    def name(self):
        return "powerful_swimming"

    def move(self, ui):
        ui.print("*Swims with powerful movements*")


class AgileFlight:
    def __init__(self):
        pass

    def name(self):
        return "agile_flight"

    def move(self, ui):
        ui.print("*Flies, wisely using air currents*")


def list_movements():
    mvt = {}
    mvt["agile_flight"] = AgileFlight()
    mvt["powerful_swimming"] = PowerfulSwimming()
    mvt["powerful_run"] = PowerfulRun()
    mvt["agile_run"] = AgileFlight()
    return mvt
```

Listing C.13: noise.py

```python
class Meowing:
    def __init__(self):
        pass

    def name(self):
        return "meowing"

    def make_noise(self, ui):
        ui.print("*Meows*")


class Roaring:
    def __init__(self):
        pass

    def name(self):
        return "roaring"

    def make_noise(self, ui):
        ui.print("*Roars*")


class UltrasonicNoises:
    def __init__(self):
        pass

    def name(self):
        return "ultrasonic_noises"

    def make_noise(self, ui):
        ui.print("*ultrasonic noises*")


class Crawing:
    def __init__(self):
        pass

    def name(self):
        return "crawing"

    def make_noise(self, ui):
        ui.print("*Craws*")


def list_noises():
    noi = {}
    noi["crawing"] = Crawing()
    noi["ultrasonic_noises"] = UltrasonicNoises()
    noi["roaring"] = Roaring()
    noi["meowing"] = Meowing()
    return noi
```

Listing C.14: test_animal.py

```python
from unittest import TestCase
from unittest import main
from animals.animal import Animal
from animals.cuteness import PawsLicking
from animals.movement import AgileRun
from animals.noise import Meowing


class MockUI:
    # Mock class that will register output
    def __init__(self):
        self.out = []
    def print(self, message):
        self.out.append(message)


class MockCuteness:
    def be_cute(self, ui):
        ui.print("cuteness")


class MockMovement:
    def move(self, ui):
        ui.print("movement")


class MockNoise:
    def make_noise(self, ui):
        ui.print("noise")

# base class for all part tests that will setup and teardown UI


class TestAnimal(TestCase):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.ui = None
        self.animal = None

    def setUp(self):
        self.ui = MockUI()
        self.animal = Animal("anim", MockNoise(),
                             MockMovement(), MockCuteness())

    def tearDown(self):
        del self.ui
        del self.animal

    def test_introduce(self):
        self.animal.introduce(self.ui)
        self.assertListEqual(["This animal is a anim"], self.ui.out)

    def test_display(self):
        self.animal.display(self.ui)
        self.assertListEqual(
            ["This animal is a anim", "noise", "movement", "cuteness"], self.ui.out
                ↪ )


class TestAnimalIntegrationAsCat(TestCase):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.ui = None
        self.animal = None

    def setUp(self):
        self.ui = MockUI()
        self.animal = Animal("Cat", Meowing(), AgileRun(), PawsLicking())

    def tearDown(self):
        del self.ui
        del self.animal

    def test_introduce(self):
        self.animal.introduce(self.ui)
        self.assertListEqual(["This animal is a Cat"], self.ui.out)

    def test_display(self):
        self.animal.display(self.ui)
        self.assertListEqual(
            ["This animal is a Cat", "*Meows*", "*Runs and jumps with agility*", "*
                ↪ licks paws*"], self.ui.out)


if __name__ == '__main__':
    unittest.main()
```

Listing C.15: test_parts.py

```
from unittest import TestCase
from unittest import main
from animals.cuteness import *


class MockUI:
    # Mock class that will register output
    def __init__(self):
        self.out = []

    def print(self, message):
        self.out.append(message)


class TestPart(TestCase):
    # base class for all part tests that will setup and teardown UI
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.ui = None

    def setUp(self):
        self.ui = MockUI()

    def tearDown(self):
        self.ui = None


class TestCutenessBeCute(TestPart):
    # starting a method name by "test" tells the framework that it
    # is a test case execution
    def test_paws_licking(self):
        pl = PawsLicking()
        pl.be_cute(self.ui)
        self.assertListEqual(["*licks paws*"], self.ui.out)


if __name__ == '__main__':
    unittest.main()
```

C.3 *tests* directory

Listing C.16: compare_output.sh

```
#!/usr/bin/bash
#$1 is name of script/expected couple
src="./${1}.exp" #name of the script (software call)
res="${1}.res" #name of the result
exp="${1}.expr" #name of expected

#creating commands
run_cmd="${src} > ${res}" #dump script into result
check_cmd="diff ${res} ${exp}" #run diff on result vs expected
clean_cmd="rm ${res}" #remove res
```

```
#actually calling commands
eval $run_cmd
eval $check_cmd
r=$? #storing result for later, remember, 0 if diff found no difference
eval $clean_cmd
exit $r #returning result
```

Listing C.17: create_chimera.exp

```
#!/usr/bin/expect -f
#line above tells to use expect shell interpreter

#spawns our program to interact with it
spawn python ../zoo.py

#wait for program to write a ?
expect "?"
#send a y then enter press
send "y\r"
#and so on
expect "your choice :"
send "crawing\r"
expect "your choice :"
send "agile_flight\r"
expect "your choice :"
send "paws_licking\r"
expect ":"
send "tigercraw\r"
expect "(y,n) ?"
send "n\r"
expect "your answer :"
send "tigercraw, cat\r"
#wait for our program to return
expect eof
```

Listing C.18: create_chimera.expr

```
spawn python ../zoo.py
animals : cat, tiger, whale, alpinechough, fishingcat
Do you want to create a chimera (y/n) ?y
Available noises : crawing, ultrasonic_noises, roaring, meowing
your choice : crawing
Available movements : agile_flight, powerful_swimming, powerful_run, agile_run
your choice : agile_flight
Available cutenesses : paws_licking, geysering, feather_shaking
your choice : paws_licking
please name you animal : tigercraw
Do you want to create another chimera (y,n) ?n
please list animals you want to see in order, separated by a comma
animals are : cat, tiger, whale, alpinechough, fishingcat, tigercraw
your answer : tigercraw, cat
This animal is a tigercraw
*Craws*
*Flies, wisely using air currents*
*licks paws*
This animal is a cat
*Meows*
*Runs and jumps with agility*
*licks paws*
```