STI 3A
Python Programming
Python Project

and a dash of Bash

Work in progress. Do not distribute outside of INSA CVL.[*]

Vincent Hugot — `vincent.hugot@insa-cvl.fr` — CRI 01

October 16, 2024

About this document:

This is your main work document for the course. It contains lecture notes (somewhat of a misnomer, as there will be only two lectures, properly speaking), exercises for lab classes and autonomous work (time slots are reserved specifically for that), and exercises for homework and exams. I shall probably cobble up the exam, at least partially, from leftovers, as there are more exercises here than we can handle in the alloted class time.

Dead tree version:

This document is quite large, and undergoing constant modification, so it will not be systematically printed and distributed to students.

Language:

This document is under construction and, currently, written in an unholy mixture of English and French. Why? There is a general impetus, which I take to heart, to move some courses to English, both to better prepare our own students for work in an international setting and to make it easier to host international students in the future. In that spirit, new course material should of course be written in English. However, some of the material here was

[*] Exceptions: IUT de Belfort-Montbéliard, UFR ST de l'Université de Franche-Comté (Pierre-Cyrille Héam), DIU EIL Orléans.

already written in French before I began working on these lecture notes this summer, and translation is hardly my priority at this time.

The spoken language in class will remain French, for now.

Typographical, orthographic, & grammatical errors:

The paint being very fresh still, this document is undoubtedly riddled with typos and sundry other infelicities. Do not hesitate to point them out to me. I may even consider offering a negligible bounty in terms of points, marks, and credits to any student pointing out a *significant* number of errors — preferably in one go.

If you wish to contribute a paragraph or two, let me know.

Trigger warnings:

Contains long digressions, sweeping personal opinions, attempts at humour, and traces of nuttiness.

General organisation

1 Types of classes

1.1 Lab classes

TD = TP = “travaux dirigés” = “travaux pratiques” = “lab class”. There are 11.

You work on various exercises on a machine. I hover behind your back, look at your code, and offer biting commentary when you write nonsense. Sometimes I debug your code, but only if you have been nice.

I *may* ask you to hand out your answers to some exercise at some point, which I then may or may not mark. When that happens you will be informed that the work may be marked at the beginning of the class. In that case, the work can be done in groups of 2. No more, no less. 3 is right out.

I *may* also give you a small surprise exam at any time; probably in the form of a few multiple choice questions. Those are always individual. Again, they may or may not be marked.

Attendance is mandatory.

1.2 Autonomous lab classes

There should be three of them *in toto*, all before the second and last lecture.

No teacher is present[b], but the lab room is reserved so you can finish up your work, read, test, and discuss the lecture notes, or anything else relevant to the course.

They are usually placed right after a TD or two.

Attendance is not *mandatory*, but strongly recommended.

1.3 Lectures / question sessions

CM =“cours magistral” = lecture, in an amphitheatre. There are 2.

I won’t actually give lectures in the classical sense, explaining Python in a linear way. I wrote the lecture notes to avoid having to do that, as neither the students nor I were overly fond of that system.

Instead, the “lectures”, of which there are only two, are there to give everyone a chance to ask questions on the material. This can mean re-explaining a concept that was unclear to you in the lecture notes, or offering and discussing a solution to an exercise. Note that I

[b] In theory. I may or may not linger in practice.

shall not distribute proposed solutions to TD exercises in written form; only explain or show them when asked during lectures.

Whenever you have questions, take note of them; whichever ones remain unanswered by the time the next lecture comes, ask them then. This is an experimental way of organising the class. The aim was to have more time with the machines, and less sitting on benches, listening to me speak. Hopefully, the questions-based system means that, whenever I do speak, at least one person in the amphitheatre (presumably the one who asked the question) is interested in what I am saying.

In the event that there are no questions, mayhap I shall cut the lecture short. . . Or, more likely, I shall avail myself of the remaining time to give you a "surprise" exam.

Exceptionally, if I see some specific type of mistake too often during lab classes, I may spontaneously elect to berate you about it during a lecture, even in the absence of related questions.

Attendance is mandatory.

1.4 Project presentation & project lab classes

You can read this year's Python project in Part VI[p365].

We shall begin in an amphitheatre, in which I shall answer general questions you may have about the project.

When that is done, the remainder of the project time (not counting personal homework, of course, and you *will* need some of that to get to the end of the project) will take place in labs. The lab rooms are reserved on those occasions, in contiguous slots of *at least* 2× 1h20.

I shall usually be present, going from group to group to get an idea of the general state of progress of each, and to answer questions. I leave when the questions run dry, or time runs out, whichever comes first.

Attendance is *morally* mandatory, but neither really controlled nor enforced, as some students are prompt to notice every year. Thus, if you want to stay in bed while your classmates work on the project, I suppose that is, *de facto*, your prerogative, insofar as I shall not fight to prevent it. However, it may interest you to read Section 71[p372], concerning the project's final report and individual marking scheme, to see why that might prove to be a rather poor tactical choice on your part.

2 Final examination

The final is an on-table, on-paper exam, lasting 1h20.

The only document allowed is the “memento”, a two-sided A4 sheet of paper which the whole class is tasked with writing during the semester, and which I must validate before the exam.

You won’t need to print your own; I’ll print it as part of the exam papers.

Of course, no computer, smartphone, smartwatch, R2D2 unit, etc, is allowed.

The paragraphs that follow were written with a coding exam in mind, of the same style as the lab class exercises. I have since moved away from that style, and towards Multiple-Choice Questions. Beyond direct questions of the form “Is this type mutable?”, I ask you to show that you understand how Python works, by asking you the result of the execution of various snippets of code, some trivial, some not.

This favours people who actively worked with the lecture notes and practised Python regularly, intelligently confronting bugs and figuring out their nature.

2.1 On the virtues of coding on paper

The “on-paper” aspect of the examination is regularly contested by students. My usual answers follow:

I want to enforce the habit of *thinking* about your code before executing it. Actually, you should think about your code before *writing it*.

On paper, you don’t have the luxury of writing nonsense code, running it, changing something semi-randomly in the hope that it works, and iterating until it does — or appears to.

On paper, you need to actually *understand what you are doing*, and do it right the first time — or at least the second time: you can use a draft.

“*But... when I program, I always have a computer; so what’s the point?*”, says the disgruntled student.

The point is that if you *need* the computer as a crutch for relatively simple things, such as the kind of things I ask in an exam, you are wasting time, you don’t actually know what you’re doing, and you *will* fail entirely on more complex problems, which involve juggling several subproblems.

Code written by test-mutation-iteration also tends to be far too long and complex – for lack of a global vision of what is actually going on – and thus much more time-consuming to write and fertile ground for bugs, even after the endless fiddling.

Write simple code fast, short, and right the first time, and you have more energy left over for the complex parts that really demand your attention.

To train, pretend you are on paper, even if you are not. Think, then write the code. When you’re *quite sure* it is correct, and only then, run it. Every time you were sure it was correct, and it turns out to be wrong, take it personally, find out what you *thought* wrong, and get better.

2.2 Degree of lenience with respect to syntax

Coding on paper makes it easier to commit syntax errors, and you don't have access to `help(..)` to check builtin functions' syntax, argument order, etc. . .

How lenient will I be when marking you exam?

Clearly, I shall be more lenient than the Python interpreter, but not by much. By the time you get to the exam, you should be practiced enough in Python to have a good grasp of its syntax.

The rule is: *whenever your syntax errors introduce an ambiguity, and I can interpret your code in several ways, I shall mark the code according to the* ***worst*** *way in which it can be interpreted*. Any other tactic would give weak students some degree of incentive to be as vague as possible.

Mistakes which I will not penalize are, for instance, writing

```
for e,k in enumerate(l)
```

in a context where it is clear that you mean `k` to be the index and `e` the element. (It should be `k,e`) Having well-chosen variable names helps identifying what you mean. Unless you do strange things with `e` and `k` later, this kind of mistake is not a problem for me.

Likewise, writing `for k in mydict.keys` instead of `for k in mydict.keys()` is unlikely to bother me, because it does not *usually* matter for the logic of your code whether the keys are an automatically updated attribute or returned by a function, and what you are doing is clear.

Contrariwise, if you write

```
return l.sort()
```

in a context where I expect you to return a sorted list, I will not *assume* that you meant to use `sorted(..)`, because understanding the difference between a procedure and a function, a side-effect and a return value, an in-place modification and a fresh value, and the role of `None`, is something that you are tested upon, along with everything else.

Perhaps you understand all this, and your use of `l.sort()` in that context was pure absent-mindedness. Or you *don't* understand any of this, and you confuse `l.sort()` and `sorted(..)` precisely *because* you never really understood how they differ or why that should matter. In the presence of this ambiguity, I shall assume the worst.

Very rarely, I may use knowledge from how you answered other questions to influence my interpretation. For instance, if other questions make it very clear that `l.sort()`, in your head, has `sorted(..)`'s behaviour, then I may apply a moderate, flat penalty instead of marking every question using it as wrong. I shall not use any knowledge external to the exam: nothing that you did in class, or talked with me about, will influence the mark; only what is written on the paper. For all practical purposes, I don't even pay any mind to the names on the paper.

If you *do* understand the difference between `l.sort()` and `sorted(..)`, but are not sure which is which, or even have completely forgotten their names, no problem. Just write me a short note along the lines:

```
I call "sort(l)" the builtin that returns a fresh sorted
version of l (not in-place!) -- I forgot its real name!
```

So long as you are clearly referencing a builtin that actually exists and is licit in the context of the question, you will not be penalised for it. All I need is sufficient evidence that you understand what you are doing. (What you are doing still needs to be correct, of course. . .)

However, whatever you do still needs to be in Python! If you write using C syntax (or that of any other language), attempt to "declare" variables, etc, you will be harshly penalised even if the intent is clear. This is a *Python* exam; if you blatantly advertise to me the fact that you have hardly ever touched Python, then *of course* I shall take a dim view of it, regardless of how well you solve the questions in another language.

Those are just a few examples out of thousands of possibilities — but they should sufficiently illustrate the general boundaries of what I consider admissible.

2.3 Memorising methods

The exam will not require you to know or use methods (in the Object sense) that are not discussed in this document, or explained in the exam. For instance, you are expected to know about `list.sort`, `sorted`, `sum`, `any`, `all`, `str`, `dict.values`, `dict.keys()` etcetera, because they are discussed in the document at some length. At least know they exist and what they do, even if you forget the exact name. (And if you do forget the small ones, like `sum`, `any`, and `all`, you can reimplement them in one to a few lines anyway.)

You are not expected to know about `list.insert`, `list.index` (although we use this once, for permutations), `list.count`, `dictionary.fromkeys`, or others like that, because they are not fundamental and are not discussed in the document. If a somewhat obscure method is useful for a question, I'll either give it in the exam, or reimplementing it from scratch will be the object of a question.

Students sometimes go out of their way to use obscure methods which they remember or understand only partially; this rarely plays out in their favour.

2.4 When in doubt, ask!

Unless I am indisposed or otherwise unavailable, I am always present during my examinations. Therefore, if anything about the exam seems ambiguous to you, or you are unsure whether something you have in mind is permitted, do not hesitate to ask.

The worst that can happen is that I might refuse to answer. To avoid that, formulate your query in a way that is as independent from the exam's question as possible.

For instance, if the question asks to return a sorted version of some list, you may ask "Is it licit to use `list.sort()` in this question?", and I shall probably answer "yes" or "no" (for instance if I require you to implement your own sorting function). If I answer "yes", that does not necessarily mean that it is the right tool for the job. If your query betrays a misunderstanding of *the question*, I shall take the opportunity to clarify, either privately or as an announcement for everyone.

Part I

Lecture notes: Linux shells & Bash

Though this course is ostensibly about Python, we shall spend one session on an introduction to Linux Shells, and on Bash in particular.

"What's the connection between Bash and Python?", you may well ask. Don't hurt your brain trying to find one, there is none – none worth drawing, at least.

Some might say that they are both scripting languages, which may well be true, but only if you take so loose a definition of *scripting* as to render the word nearly useless. Some might say that they are both interpreted, as opposed to compiled, but – even politely disregarding that this is a property of implementations and not of languages – that's only sort-of-true, as Python's default implementation actually compiles to byte-code, and then interprets *that*. Even were it absolutely true, it would *still* be a terrible reason to lump two languages together.

The real reason there is some Linux and Bash in this course is that several other courses depend on students having at least minimal competence on a Linux machine, but there is not enough relevant material to justify making Bash a course onto itself. Thus, you may consider this Bash class as an independent mini-course, and a prerequisite for most future lab work in other courses.

Note that only the bare minimum is covered here; you will have to experiment, ask questions, and find other sources to go further.

> **(0)** ✍ During the first lab class, look for boxes similar to this one: they contain the instructions for the class. The aim is to get familiar with shell and the Linux machines.
>
> When you have done everything – or if everything is trivial to you because you have experience in shell already – move on to the part on Python and start reading the lecture notes or jump straight to the Python exercises.
>
> The shell lab class is not marked.

3 What is a shell?

A *shell* is an otherwise perfectly ordinary interactive command-line program whose purpose is to serve as an interface to the machine's other programs. Thus it receives and interprets the user's commands, usually directly typed into the console, and executes them.

Oftentimes, this boils down to calling another program with the provided arguments, and letting it take over. For instance, `nano text.txt` just calls the external program `nano` – a barebones text editor found on most machines – with argument **`test`**`.txt`.

Shells can also receive commands from files, and support programming structures such as variables, arithmetic and Boolean operators, functions, tests and loops, et cetera, though of course the features available and their syntax vary from shell to shell.

Thus, one typically uses them to write some small scripts serving as “glue” between others, more sophisticated programs. Shell languages are well-suited for this purpose, as calling other programs is extremely easy, and there are convenient syntactic shortcuts for nearly every situation. Once you have memorised them, at least.

4 What is a *script*?

I have used the word several times already, and will many times again; it is worth defining. The first caveat is that not everybody agrees what that means. If you search for a definition, you will likely find something along the lines of (the first result of my search engine):

> A program or sequence of instructions that is interpreted or carried out by another program rather than by the computer processor (as a compiled program is).
>
> – *TechTarget.com*

This cannot immediately be said to be *wrong*, in the sense that there is no consensus or central authority against whose definition this can be compared, but I put it to you that this definition is *useless*.

It is useless because it carries no information about the program/script itself, only about the language used to write it, and that information is “it is an interpreted[c] language”. We already have perfectly good words to express this – I just used them – so with this definition the word “script” carries no new information. Every script is an interpreted program; every interpreted program is a script; we now have two different ways of saying the same thing. Great.

The notion of “interpreted language” itself is not terribly useful, because many interpreted languages, Python among them, compile to bytecode behind the scenes. That still matches the definition above, but then. . . so do Java and C#. And what to make of languages like OCaml, which support interpretation, byte-code compilation, *and* native code compilation for multiple processor architectures?

Being “interpreted” is mostly a property of a language’s implementations, not of the language itself — only *mostly* because of course some language design choices dictate what kind of implementations make sense in practice.

Here is the definition I propose:

> **A *script* is a source code file written either directly by the end-user or with the reasonable expectation that the end-user may alter it according to his needs.**

[c] And it is a rather crummy definition of “interpreted”, as well, but let’s not get into this.

– *Me*

That is the important distinction between a *script* and other types of programs. If it's *meant* to be casually tinkered with, it's a script. If it provides some sort of interface (CLI, TUI, GUI, configuration files,. . .) within the bounds of which you are expected to express what you want, and if you can't, you're out of luck, it's a regular old program.

Other characteristics of scripts, which some sources will use as definitions, stem from this:

- ◇ Scripts are almost always executed via an interpreter because altering compiled programs is tedious. But some scripts may be compiled. For instance, in the game editors for Skyrim and Neverwinter Nights, the scripts (in domain-specific languages Papyrus and NWScript) are compiled (to specialised byte-codes), either implicitly or explicitly. They're still scripts, because the end-user of the game editors is supposed to deal with them directly.
- ◇ Scripts are often short and sweet; the longer and more complex a program becomes, the less you want to casually tinker with it, unless you like pain.
- ◇ Scripts often, but not always, deal with automating tasks, or customising a system, because that's the sweet spot where you can't just toggle an option somewhere to get what you want, but you don't need to invent something completely new, just gluing together exiting functionalities will do.

Shell languages, such as Bash, are very-well suited for system automation and customisation.

5 Syntax (in)sanity warning:

The basic syntax for most common shell operations was decided a half-century ago, or thereabouts. In those days[d] "spirits were brave, the stakes were high, men were real men, women were real women, and small furry creatures from Alpha Centauri were real small furry creatures from Alpha Centauri."

In other words, accessibility, consistency, and legibility were not popular buzzwords yet; conciseness was much more highly valued. The shell syntaxes are chock-full of one-character modifiers with non-obvious effects, and the parsing rules are not always straightforward — the clear "formal (context-free) grammar + lexer + parser" approach to language design which we shall study in the "Formal Languages Theory" course this semester was not used to design those languages. Perhaps it is because that method was not yet as widespread and well-tooled when the initial design choices were made as it has been these last few decades.

The Posix shell committee wrote a grammar attempting to formalise the Bourne shell; many rules in the grammar are context-dependant. You will understand what "context dependant"

[d] and in the immortal words of Douglas Adams

means in more detail after studying formal languages. For now, let's just say that this means that shell syntax is intrinsically "more complex", in a very important sense, than that of languages like C, Java, or Python. Even then, the grammar does not exactly capture the behaviour of parser implementations.

One of the additional difficulties is that pretty much everything is handled as strings... which quickly becomes nightmarish when you have several layers of parsing and want to be robust against whitespace and special characters.

None of this means that simple tasks are necessarily complicated in shell. But complex tasks that go against the grain can become quite tricky fairly quickly.

For these reasons, among others (such as runtime efficiency and portability), it is not *advisable* to use those languages for anything *other* than "glue" scripts, though the languages are complete enough that you technically *could*, were you masochistic enough to try. If you have non-trivial algorithms to write, it is probably best to write them in some other language and call them from shell if you must, unless there are overriding constraints at play.

Things are even more chaotic with the naming conventions of standard Unix programs that you might call from a shell or within a shell script. Those are not technically part of the shell languages but in practice you will always need to use some of them to get anything done.

Nobody ever sat down and decided upon a coherent set of basic programs and naming conventions. New programs were added higgledy-piggledy by different people as time went on, under the only constraint of never reusing an existing program name. Those that proved very useful became more or less standard... and never changed names.

For instance, if you want to display the contents of a text file, do not expect to find a command named `display_text` or anything equally descriptive. That would be unimaginative and pedestrian. The relevant commands are actually named **`cat`**, `more`, and `less`. Of those, *of course*, `less` is the *most* flexible. More than `more`. Because *of course*.

There is little logic behind those names, only a *history*. **`cat`** is actually short for `concatenate`, and quite unrelated to the allegedly cute feline freeloaders that, for unfathomable reasons, old ladies like keeping as pets.

For instance, **`cat`** `f1 f2` sends the contents of the two files, one after the other, to the standard output, which can be redirected to a new file, thus concatenating the two initial files into a third. So far, so good. Since the standard output is displayed by default, the command is also (almost accidentally) suitable for displaying text, and was used for that purpose.

When the text is too large to fit in a single screen, with **`cat`**, you only see the end of it, as everything else scrolls out of view at breakneck speed, so somebody wrote a command to stop at the first screen of text, and display `--More--` on the last line until the user chooses to read the rest of the document. That command was called `more`, after that prompt. (Okay, that *almost* makes sense).

Then somebody wrote a version of `more` with even *more* features. And they called it `less`, because they thought they were being *funny*. HA. Ha. ha. (`more` could not scroll backwards, which was the main new feature, so “backwards more = less”. Get it? Instant hilarity.)

My advice: take a deep, relaxing breath and **just memorise the commands that you need**. After a while you won’t even notice how infuriating it all is anymore.

Additional good news? There is no *universal* convention regarding the syntax of the *arguments* that each program or command takes. That would be too easy. (Though there are Posix and GNU conventions, which are fairly well respected).

When faced with a new command, especially a non-standard one, always read the manual or the help page. `thecommand --help` *usually* displays the help page. But of course nothing prevents a given program from ignoring that argument outright and doing whatever it wants.

`man thecommand` will display the manual page, if installed. Since the `man` program itself is safe, you don’t run the risk of running an unknown program by using it.

6 Choose your shell:

Here are a few among the most common shells, by family:

- **Microsoft:** cmd.exe, Windows PowerShell
- **Bourne :** sh, ash, bash, ksh, zsh
- **C :** csh, tcsh
- **Perl :** perlsh, zoidberg
- **Plan9 :** rc, es
- **Secure / Restricted:** ibsh, rssh, scponly

`sh` is the historic (Bourne) shell for Unix systems, and has been present on nearly every variant of Unix since the eighties. `bash` (Bourne-Again Shell) is a strict superset of `sh`, in extremely common use. That is what you will be using here; when you launch a console on the machines, a `bash` session is opened by default. Bash is indeed the default on more Linux distributions and on MacOS (which is a Unix).

7 Bash: builtin commands and reserved words

Recall that commands fall into either of two categories: builtin shell commands, and external programs.

For reference, the following are builtin commands for Bash 4.2:

```
:         command    eval      jobs      read        times
.         compgen    exec      kill      readarray   trap
[         complete   exit      let       readonly    type
alias     compopt    export    local     return      typeset
bg        continue   fc        logout    set         ulimit
bind      declare    fg        mapfile   shift       umask
break     dirs       getopts   pushd     shopt       unalias
builtin   disown     hash      popd      source      unset
caller    echo       help      pwd       suspend     wait
cd        enable     history   printf    test
```

Everything else is an external program.

Examples include cat, less, more, cp, mv, rm, ls, mkdir, find, grep, sed, cut, ps, chmod,. . .

The following words are reserved by Bash, and thus cannot be used as variable names, among other things:

```
!         time
[[        ]]       {       }
if        then     elif    else    fi
case      esac
select    in
while     until    for     do      done
function
```

8 The command prompt

When stating a bash session, you find yourself faced with a *command prompt* (FR: invite de commande) which, by default (it is highly personalisable), looks like this:

```
vhugot@Khepri:~/Documents/3A-Python$
```

(1) ✍ Start a computer under Linux (via dual-boot or a virtual machine), and locate a terminal or console application, so that you are ready to work with a prompt.

It can be broken down into the following parts:

`vhugot`	username of current user
`@`	separator
`Khepri`	name of the current machine
`:`	separator
`~/Documents/3A-Python`	current working directory
`$`	status indicator; # for root, $ for peasant.

~ stands for "user's home directory", in my case, `/home/vhugot`.

In the remainder of this document, I'll usually just use `$` and omit the rest of the prompt.

The appearance of the prompt is dictated by the variable `PS1` (for Prompt String 1), which can be modified. On my system, it has the value

```
$ echo $PS1
\[\e]0;\u@\h: \w\a\]${debian_chroot:+($debian_chroot)}\u@\h:\w\$
```

There is some magic going on there; let's just mention that `\u` stands for username, etc, and show the effect of a simple modification of this value:

```
vhugot@Khepri:~/Documents/3A-Python$ PS1=coucou
coucouecho $PS1
coucou
coucou
```

(2) ✍ Do the same thing on your machine. Quit and start the terminal again. What happens?

9 Profile files

Prompt modifications, along with aliases and other tweaks to the behaviour of bash, are best put in the adequate configuration files that bash runs at the beginning of each session.

Those are generally `~/.bashrc` or `~/.bash_profile`.

(3) ✍ Find those files and see what's in them. Don't modify them for now.

If possible, use an editor with syntax highlighting. Note that the symbol # comments out everything that follows; syntax highlighting is particularly useful to see at a glance which lines are commented out.

It is common for Linux distributions to provide `bashrc`s with a lot of lines commented out, which the user can simply uncomment to activate a nonstandard but useful feature.

In particular, we shall come back to **`alias`**es later on.

Note: we are focusing on the Shell, but that does not entail that you can't use anything else; in particular, you don't have to edit config files with nano or vim. You can use the file browser and whatever graphical text/code editor you have available or prefer.

Knowing how to do *everything* in the terminal is extremely convenient when connected in ssh to a remote server. When working on a local machine with a graphical environment, there are usually better tools available.

Use the right tool for the task at hand.

10 Common commands and assorted tricks

You must know all these commands.

(4) ✍ Take some time to test each of these commands and skim through their manual and help pages, even is there is no specific question next to them.

Here are a few of the most useful commands. For more information about their capabilities and usage, see the help and man pages.

`man <cmd>`:

display the **man**ual page of external program `<cmd>`, if one is installed. Most, but not all, programs come with a somewhat detailed manual page.

`cd` `<path>`

Change the current working **d**irectory. `<path>` can be an absolute or relative path. `/home/vhugot` is an absolute path. `Documents/stuff` or `./Documents/stuff` (`.` means "current directory") are relative paths, as well as `../otherstuff` (`..` means "parent directory").

If the prompt displays the current working directory, which it should, you should see it updated with each **`cd`** command.

`pwd`

Print current **w**orking **d**irectory, as an absolute path. In case you forget where you are.

`cd` without argument will put you into your home directory. **`cd`** `-` will restore your previous location.

`ls <locations>`

Li**s**t files and directories at current location, if no argument, or at provided locations.

```
ls -a <locations>
```

Same thing, but list **a**ll files, including *hidden* files (files whose name begins with `.` are hidden).

> **(5)** ✍ Move around a bit. Go to your home directory, your Desktop, and Documents folders, and display the list of files there.

Special characters: `*` is a wildcard character standing for any string (including an empty string). For instance `ls *.c` will list all C files at current location. `?` stands for a single character. Character intervals can also be defined, and various patterns can be chained: `[0-9]*?[A-Z]*[0-9a-zA-Z]` would test for names beginning with a digit, followed by anything (at least one character), with at least a capital letter in the "middle", and ending with an alphanumerical character.

```
echo <string>
```

Displays the provided string. Note that the string need not be within quotes. If it is not, wildcard will be expanded. Thus, **`echo`** can be used as a cheap `ls`. Both are useful for testing your patterns before using them with dangerous commands, such as `rm`.

Note that the terminal can display colours, thanks to special escape sequences. This applies not only to **`echo`** but to any program or script that prints to the terminal. See this page.

```
printf <string>
```

Same as echo, but more flexible and has a well-defined behaviour across all shells, which can be important for the portability of scripts. It is an external command, whereas **`echo`** is internal, and thus **`printf`** is more expensive.

```
mkdir <newdirname>
```

Ma**k**e a new, empty **dir**ectory at current location.

> **(6)** ✍ Create a nice "test" directory for the purposes of this class, somewhere convenient,like your desktop. Do so entirely with shell commands, of course. Make sure it becomes your working directory.

```
nano <textfilename>
```

Create or edit a text file. Nano is a minimalist command-line text editor. Notations like `^X` in nano mean "CTRL+X".

> **(7)** ✍ Create a text file `mytext.txt`. Quit and save. Edit it again.

Note: Linux does not rely on file extensions to the same extent that Windows does, but it's still a good idea to use them regardless.

`cp <src> <dest>`, `mv <src> <dest>`, `rm <src>`

Co**p**y of **m**o**v**e file from source to destination, or **r**e**m**ove it. As with all file-manipulation commands, be very careful not to overwrite something by mistake. I would advise to almost always use the interactive mode, to receive a prompt in case the destination file already exists. The flag for interactive mode is `-i`. Thus you would type `cp -i toto tata`. This is the kind of things you should create an alias for.

To act on all files within a directory, at arbitrary depth, for instance to copy or remove a whole directory and all its files, use the `-R`, or `-r`, or `--recursive` flag (the three forms are synonyms). Avoid typing `rm -rf /`, as that means "force removal of all files on the system". Here `-rf` is equivalent to `-r -f`, where `-f` means "force". Flags can often be combined this way, though not all programs support this.

Advice: if you're planning on using non-trivial wildcard patterns with those commands, it may be prudent to test them with `ls` or **`echo`**, first.

Aliases are pretty convenient ways of enforcing certain flags; I have the following in my configuration:

```
alias rm='rm -i'
alias mv='mv -i'
alias cp='cp -i'
```

Thus, whenever I type `cp` as a command, it is as though I typed `cp -i`. I no longer have access to the original commands, but in that case `-i` can be overridden by `-f` or `-n`. I could of course have defined the aliases under different names, but here redefining the commands is deliberate, so as to avoid accidents with them.

(8) ✍ Check whether those aliases are already defined for you. If that's not the case, add them to your `.bashrc`, unless Danger is your middle name.

Note that it is also possible those aliases are already present in the `bashrc`, but commented out. In that case, simply uncomment the relevant lines.

(9) ✍ Remove the text file you created earlier.

`rmdir <dir>`

removes an empty directory. Refuses to do anything if the directory is not empty. More or less redundant with `rm -R`, but safer.

(10) ✍ Remove your test directory, which should be empty now.

`chmod <mode> <file>`

Changes file access permissions. For instance `chmod u+x s.sh` makes the file `s.sh` executable for its owner.

`top`, or **`ps`**

Displays running processes. Most desktop environment offer a GUI equivalent as well. On KDE, for instance, it is accessible via Ctrl+ESC.

There are many options depending on what you want to see. Generally, a call to **`ps`** `aux` will show most of what you would ever need.

`kill` `<processnumber>`

Kill the selected process; the process number is provided by the previous commands.

`killall <processname>`

Kill all processes bearing `<processname>` as part of their name.

`w`

Displays who is currently logged in, for how long, and current processor load.

`last`

Shows a recent history of logged users.

`history`

Shows the last run commands.

11 More complex commands that can be very useful

It is good to know these commands exist, but you will certainly need to spend some time wrestling with their man pages before getting them to do exactly what you want.

`grep`

A powerful tool to search for patterns / regular expressions in files, of filter the output of other commands.

`rsync`

Probably the best tool to reliably transfer files between computers and create mirrors and backups while detecting redundancies and minimising network traffic.

`wget`

Download files and websites from an URL.

12 Pipes and redirections

In Unix parlance, *file* is a general term designating both what we usually think of as a file and more abstract constructs such as peripherals and data streams.

Each open file is assigned a number, to keep track of it, call a *file descriptor* (or sometimes *file handle*).

In particular, there are three files that are always open: the standard input, corresponding to the stream of keyboard inputs (stdin, 0), the standard output, corresponding to what's printed in the console (stdout, 1), and finally the standard error (stderr, 2), also printed in the console, but meant specifically to be used for error messages, as opposed to the usual output of a command.

These streams can be manipulated and redirected in various ways. For instance, the output of a command can be fed as input to another, or to a file. This is fundamental to the philosophy of Unix, which consists of having many small, specialised commands which can be glued together in myriad ways as need demands, resulting in a sophisticated specialised processing pipeline.

Here are some of the various operations that can be done:

(11) ✍ Create and edit files and folders as necessary to test each of the constructs presented in this section. (And answer any specific questions along the way).

Pay attention to what you are doing and replace obvious placeholders or examples like `filename`, `contents`, or `PythonNotes.tex` with something more pertinent.

```
ls > filename
```

The character > redirects the standard output of a command (here `ls`) to a file.

(12) ✍ Using a command seen in the previous sections, create, in a single line of shell, a new file containing the line “Hello world!”. Now create an empty file. (An alternative in that last case is `touch`).

If a file named `filename` does not exist, it will be created and filed with the relevant contents – it will be kept open until the command finishes its output. If the file already exists, its contents will be erased and replaced by the output. There will be no warning.

```
ls >> filename
```

Has the same effect as above, but if the file already exists the output is appended (added at the end) to it, instead of overwriting the file.

```
grep contents < PythonNotes.tex
```

Here, the contents of `PythonNotes.tex` – the source file for the document you are reading – is redirected by < into the standard input of the command `grep someword` – the combined effect is to display all the lines of the file that contain the word "contents".

```
grep contents < PythonNotes.tex > results.txt
```

Here, input and output redirections are combined: instead of the selected lines being diplayed on the standard output, they are written in the file `results.txt`.

```
cat PythonNotes.tex | grep contents
```

The pipe operator connects the standard output of the command on the left, here **`cat`**, which outputs the contents of the file, to the standard input of the command on the right. Thus, the above has the same effect as `grep contents < PythonNotes.tex`.

> **(13)** ✍ Run the command "`sleep 9999 &`" to put a `sleep` process in the background. (see section on background processes)
>
> With **`ps`** `aux`, you can find its process number and kill it, but it's a chore. Use grep and a pipe to find the right line easily, then kill the `sleep` process.
>
> How many lines of **`ps`** `aux` where displayed? Why?
>
> Is there another way you could have used to do that without needing the process number? (That's a rhetorical question. Of course there is. What is it? Does it have any drawback from a general standpoint?)

Of course, the selling point of the pipe operator is that you can chain not just two but arbitrarily many commands together to achieve complex results:

```
cat *.txt | sort | uniq > sortedlines.txt
```

produces a file containing all lines appearing in all text files, sorted in alphabetical order and with duplicates removed.

Each process in such chains plays the role of a **filter**. When writing programs, especially in a Unix environment, it is advisable to pay attention to what their behaviour should be wrt. stdin and stdout, so that they can be used as filters.

13 Execution modes: sequential, background; jobs

(14) ✍ Test what follows on your computer.

By default, commands are executed sequentially, in the foreground. Several commands can be executed in the same line, and in sequence, by using the `;` operator.

Commands can be grouped in two ways: via `(...)` or `{... ; }` – note the mandatory `;` in the second case.

In the first case, the commands are executed in a sub-shell, and so any variable assignment will be discarded at the end of the group's execution, and the group cannot modify the current context.

In the second case, the group will be executed in the current shell context.

```
$ { date ; ls ; } > resultat.txt
$ cat resultat.txt
vendredi 17 janvier 2014, 09:48:34 (UTC+0100)
Fichier.txt
GG.txt
resultat.txt
```

A command can be run in the background by putting `&` at the end of the line:

```
$ sleep 5 &
[1] 7691
$
```

This created a process (number 7691, job number 1) running `sleep 5`, that is to say, a process that runs for 5 seconds, doing nothing, and immediately gave us back the command prompt, whereas we otherwise would have needed to wait for `sleep` to terminate before doing anything else.

`jobs` `-l`

Lists currently running jobs, their job numbers, their statuses, and process numbers (thanks to the `-l` flag).

Suppose you just launched a long process, say, `sleep 500`, in the foreground, and want to kill it. You can do that with `Ctrl+C`.

Suppose now that, while you want the prompt back, you don't want to kill the process. You can use `Ctrl+Z` to pause it and put it in the background.

```
$ sleep 500
^Z
[1]+  Stopped                 sleep 500
```

```
$ jobs
[1]+  Stopped                 sleep 500
```

From there, you can bring it back to the foreground or run it in the background with the commands **fg** and **bg**, respectively. They expect the job number as argument, not the process number.

```
$ fg 1
sleep 500
^Z      # I put it back to sleep
[1]+  Stopped                 sleep 500
$ bg 1
[1]+ sleep 500 &
$ jobs
[1]+  Running                 sleep 500 &
```

14 Quotes

Like Python, bash supports both simple and double quotes for strings. Back quotes also exist, but serve an entirely different purpose:

Single quotes are called “strong” quotes. Every character between the two quotes lose any specificity they might have had, and they are treated as a string.

```
$ echo ’This contains "double" and ‘back‘ quotes’
’This contains "double" and ‘back‘ quotes’

$ echo "This contains ’single’ and ‘back‘ quotes"
back: command not found
This contains ’single’ and  quotes
```

Double quotes, on the other hand, are “weak”. Some characters retain their specificity, and that includes `$(..)` structures and back quotes.

```
$echo ’This contains "double" and ‘back‘ quotes’
’This contains "double" and ‘back‘ quotes’

$ echo "This contains ’single’ and ‘back‘ quotes"
back: command not found
This contains ’single’ and  quotes
```

Back quotes execute the string inside them as a command, and produce, as a string, the contents of that command’s standard output. Of course, this works better with a valid command:

```
$ echo "It is `date` and all is well."
It is samedi 17 aout 2019, 10:28:35 (UTC+0200) and all is well.
```

interlaces the written text with the output of the `date` command.

The same result is obtained with the `$(..)` structure

```
$ echo "It is $(date) and all is well."
It is samedi 17 aout 2019, 10:28:35 (UTC+0200) and all is well.
```

If fact, backquotes are considered deprecated in favour of `$(..)`. Just don't use them.

```
$ echo $(echo this $(echo is $(echo easier to nest)))
this is easier to nest
```

> **(15)** ✍ Create a few `.txt` files through whatever method you find most convenient.
>
> In one line of shell, display a sentence of the form "BEGIN <list of all text files here> END". Without manually writing the names, of course.

Note that both kinds of quotes can simply be concatenated:

```
$ echo "'"'"'
'"
```

Also note that, in some contexts, there is no need to use any quotes at all to be working with strings – unless you need to deal with special characters or something like that. By default, pretty much everything is a string. There are a few caveats, however. Consider for instance

```
echo This is  a sentence
This is a sentence
```

Can you spot the problem ? There were two spaces between "is" and "a", but only one was printed. Why ? Because the shell parsed each word as a separate argument to the command **echo**, as spaces are argument separators. Echo then printed each of its arguments, separating them with spaces, but it has no way to know how many spaces were originally used to separate them. The string needs to be quoted if whitespace is to be preserved.

Thus if something is "morally" a string, it can be helpful, to systematically quote it as such. At the very least it does not hurt, and is probably better for reasons of legibility and consistency,

15 Programming in shell / Bash

This is the part where we put our Hazmat suit on and touch very briefly on shell programming. We concentrate on features present in the basic shell, and avoid Bash-specific features, for reasons of portability.

One can program directly in the interactive shell, yes, but generally one wants to write a script in a file and execute it when needed.

Let's say that you have written `s.sh`; then you can execute it with `sh s.sh` or `bash s.sh` depending on what you want to execute it with (the smaller Bourne shell, or the larger Bash).

Scripts are usually made executable via, e.g. , `chmod u+x s.sh`, so that you can run `./s.sh` directly.

In that case, the first line should be of the form

```
#!/bin/sh
```

or, if you want Bash

```
#!/bin/bash
```

so that the shell know which program (here another shell) to use to run the script.

(16) ✍ Create and run a script that displays "Hello World". That not very original, but originality is overrated anyway.

15.1 Variables

Variables are all of type string, and do not need to be declared, just initialised. They are local to the current shell process.

`varname=value` is the syntax for initialisation.

The value of a variable can be extracted using `${varname}` or `$varname`.

```
$ x=Hello
$ echo "x='$x' y ='$y'"
x='Hello' y =''
```

Here we note that "Hello" has been treated as a string automatically, and that all uninitialised variables contain the empty string.

This is where the joy of Bash programming beings. **There must be no space around the = sign.**

```
$ x=Hello world
world: command not found
$ x = Hello
x: command not found
$ x= Hello
Hello: command not found
```

15.2 Interaction

(17) ✍ Follow the instructions and test the scripts below. Modify them a bit as inspiration strikes you.

You can prompt the user for answers by using the command **`read`** `<varname>`. Let's see it in action in our very first script:

```
#!/bin/sh
echo "My very first script. Huzzah."
echo "Current date: $(date)"
echo Dear user, what is your name?
read name
echo Please to meet you, $name.
echo In what year are you? What department?
read year dept
echo So, $name, you are in the $dept department, year $year.
echo Am I correct?
read answer
```

Note that you can read several variables at the same time. As usual, whitespace is used to separate the answers. Let's run the script.

```
./script1.sh
My very first script. Huzzah.
Current date: samedi 17 aout 2019, 11:38:45 (UTC+0200)
Dear user, what is your name?
Toto                          # my answer
Please to meet you, Toto.
In what year are you? What department?
3A    STI                     # my answer
So, Toto, you are in the STI department, year 3A.
Am I correct?
Yes                           # my answer
```

Let's play with pipes to automate the answers. Recall that everything I type goes in stdin. Let's prepare the answer ahead of time and pipe them into the script:

```
echo -e "Vincent\n3A STI" | ./script1.sh
My very first script. Huzzah.
Current date: samedi 17 aout 2019, 11:36:24 (UTC+0200)
```

```
Dear user, what is your name?
Please to meet you, Vincent.
In what year are you? What department?
So, Vincent, you are in the STI department, year 3A.
Am I correct?
```

Since we needed a line return to validate the first question, I needed to use the `-e` flag so that **echo** interprets backslash escapes.[e]

Note: the above is a fairly useful construct in lab classes. You will sometimes have to write interactive programs and test them on a given input. Often, you test the program on the same input repeatedly, until you manage to get it to work as intended. Do not waste time typing the same test input over and over again: automate it with a pipe!

15.3 Return values

(18) ✍ Test the following on your machine.

All commands and functions return a numerical value at the end of their execution.

This value is usually 0, indicating a "normal" execution. Non-zero values usually indicate an abnormal execution, but can also be used to convey some information about the execution. Each command has its own policy in that regard.

Here are two predefined commands which do noting except returning 0 or 1 (`$?` show the last return value, see next section on special variables)

```
$ true
$ echo $?
0
$ false
$ echo $?
1
```

Note that this convention is the reverse of the usual True/False ≡ 0/1!

The operator && chains commands, like `;`, but breaks the chain as soon as a command returns a nonzero value, and returns that value.

```
true        && echo Always # prints and returns 0
false       && echo Never  # does not print, returns 1
return 123  && echo Never  # does not print, returns 123
```

The operator `||` chains command, stopping at the first command that does not fail. It always returns 0.

[e] Depending on the shell, some **echo**es interpret backslashes by default. This is the case in `zsh`, which I use at home.

```
true        || echo Never   # does not print, returns 0
false       || echo Always  # prints, returns 0
```

15.4 Functions and special variables

Functions can be defined simply

```
functionname () {
  commands
}
```

or

```
function functionname {
  commands
}
```

Like scripts, they receive arguments and return a numerical value. Use the command **`return`** to return a non-zero value. Call them as any other commands, once they are defined.

All functions are variadic. It is up to them to deal with however many arguments they receive as they see fit.

A number of special variables are used to deal with return values, function arguments, etc.

`$_`	the last word / argument of the last executed command
`$?`	return result of the last executed command
`$$`	current process number
`$!`	process number of the last process put in background
`$0`	name of the current command
`$#`	in a script or function: number of received arguments
`$*`	in a script or function: list/string of arguments
`$n`	in a script or function: nth argument, $n \in 1..\$\#$

(19) ✍ Test the following on your machine.

Let us test all that in a new script `script2.sh`:

```
#!/bin/sh

f () {
  echo nb args $#
  echo current command name $0
  echo current process number $$
  echo args list $*
  echo arguments: $1, $2, $3, $4, $5.
  return 123
}

f a "b c d" e  f
echo return value $?
echo process $$: same as in function!
sleep 5 &
echo Last process in background: $!
jobs -l
```

and execute it:

```
$ ./script2.sh
nb args 4
current command name ./script2.sh
current process number 15031
args list a b c d e f
arguments: a, b c d, e, f, .
return value 123
process 15031: same as in function!
Last process in background: 15032
[1] + 15032 Running
```

15.5 Conditional branching

(20) ✍ **Shell programming aspects,** including conditional branching and loops, **are not essential**.

Continue reading and testing those constructs *if you have the time and inclination to do so*. At least skim through the last few sections, so you know that the constructs exist and can come back to them when you need to brush up on the syntax.

You can also continue familiarising yourself with the more basic aspects of the shell for the remainder of the class, or move on to Python right away.

Boolean tests in shell are done through the **test** command. For instance, and without forgetting that, in shell, the usual True/False ≡ 0/1 convention is annoyingly reversed:

```
$ test toto = Toto; echo $?
1 # false !
$ test toto = toto; echo $?
0 # true !
$ test toto != toto; echo $?
1 # false !
```

The = and != operators are string comparison, and case sensitive as the example shows.

There are other operators, given by flags (when in doubt read the manual page for **test**.) For instance, `-f` tests whether a regular (non-directory) file exists, and `-ge` stands for "greater or equal", and serves for numerical comparisons:

```
$ test -f script1.sh ; echo $?
0 # true
$ test -f noscript.sh ; echo $?
1 # false

$ test 21 -ge 20 ; echo $?
0
$ test 21 -ge 22 ; echo $?
1
$ test 21 -ge tata ; echo $?
test: integer expression expected: tata
2
```

Tests can be combined with Boolean operators `-o` OR and `-a` AND. Imbrication can be specified by parentheses, but be careful to escape the parentheses and to separate them from the rest with spaces. Likewise for the operands of =; each operand or operator is simply an argument of the **`test`** command, and must stand as its own string argument.

```
$ test \( \(  0 = 0 -a  0 = 1 \) -o 0 = 0  \); echo $?
0
```

This is rather prodigiously ugly. Fortunately there are other syntaxes for tests, which are *slightly* less execrably repugnant. Nobody with even a shred of sanity recommends using `-o` and `-a` anymore.

The syntax for conditional branching is:

```
if <test>
then
  <commands>
elif <test>
then
  <commands>
else
  <commands>
fi
```

and there is a special syntax so that you can replace

```
test <your test>
```

by

```
 [ <your test> ]
```

Note that there again, `[` and `]` must be separated from anything else by spaces.

Each "test in brackets" is a command, exactly as though written with the original syntax. This enables us to use the "boolean operators for commands" `&& ||` to combine test commands rather than using the Boolean operators internal to the **`test`** command.

Crucially, this also enables us to use command groupings, which goes a long way to making things just a tad more civilised.

For instance:

```
# if [ 0 -ge 1 ]
# if true
# if { true && false || false ; } || true
if { [ -d . ] && [ -f nothere ] || [ 0 -ge 1 ] ; } || [ 1 -ge 0 ]
then
  echo This
else
  echo That
fi
```

And *that* is about as sexy as it gets for the standard shell (`/bin/sh`). If you're willing to use Bash (`/bin/bash`) instead, you get extra features, such as the double bracket syntax `[[...]]` for conditional expressions, which can make writing complex logic in scripts almost *tolerable*.

Note, however, that this luxury is only available on desktop systems, and not quite all of them at that. You cannot count on an embedded system having Bash; pretty much everything that runs Linux at all, or any version of Unix, really, probably has a Bourne shell, though.

So for portability reasons it's probably best to bite the bullet and deal with the barebones syntax. A fatalistic way of seeing it is: "if you need complex logic, you probably shouldn't be doing it in a shell script anyway".

Let's mention in passing that `&& ||` change precedences between the single and double brackets syntaxes. They are of equal precedence in the first, and `&&` is higher precedence in the second. Moral of that story? Don't rely on precedences; use explicit groupings, always.

15.6 Arithmetic

The **`expr`** command is basically a command-line calculator, and enables the use of arithmetic computations in shell scripts.

A small example, sufficient for our purposes:

```
$ i=$(expr $i + 10)
$ echo $i
10
```

Arithmetic and Boolean tests involving only arithmetic can have their own syntax in Bash: `$((..))` and `((..))`. Let's not get into that.

15.7 Switch case

Long sequences of **elif** are best replaced by a case structure. The case structure has the following form:

```
case <variable> in
  <pattern 1>)
    <commands>
    ;;
  <pattern 2>)
    <commands>
    ;;
  <pattern n>)
    <commands>
esac
```

It executes the first branch such that the pattern matches the value of the variable.

Patterns can be simple strings or use wild card: for instance in a daemon's code, you might find:

```
case $1 in    # $1 is the script or function's first argument
start)
  echo starting
  ;;
restart)
  echo restarting
  ;;
stop)
  echo stoping
  ;;
[a-zA-Z]*42*)
  echo secret number!
  ;;
*)
  echo bad argument
  ;;
esac
```

Any string beginning with a letter and containing 42 will trigger the "secret number" clause.

Note that the last clause functions as else because * will match everything. Any clause appearing after that would never trigger.

15.8 While loops

While loops are similar in syntax to conditional branching, and the same remarks apply to tests in this context:

```
while <test>
```

```
do
  <commands>
done
```

Example:

```
k=0
while [ $k -lt $1 ]  # lt ->  less than (strict)
do
  echo $k
  k=$(expr $k + 1)
done
```

predictably, prints 0..9 when given 9 as argument.

Until loops have the same syntax – and the semantics you would obviously expect: the body is executed repeatedly until the test becomes true. **until** C is equivalent to **while** ¬C.

```
until <test>
do
  <commands>
done
```

I have seen it written – unironically and in all seriousness – that the presence of the quasi-redundant **until** construct in the shell language was a matter of *elegance*, because sometimes statements are more legible with "until". It is certainly true that, verily, shell scripts are the place where programming elegance – nay, *grace*, dare I say! – reaches its most stratospheric levels of refinement. That last part was sarcasm..

15.9 For loops

Finally, for loops:

```
for <variable> in <list>
do
  <commands>
done
```

Here, a list simply means a string, with whitespace being the list separator:

```
adjectives="convenient ugly widespread     old"

for adjective  in $adjectives
do
  echo Shell is $adjective.
done
--------------------
Shell is convenient.
Shell is ugly.
Shell is widespread.
Shell is old.
```

Conveniently, none of the strings contained whitespace.

```
adjectives="Convenient ’but(t?) ugly’ widespread     old"
```

would get you:

```
Shell is Convenient.
Shell is ’but(t?).
Shell is ugly’.
Shell is widespread.
Shell is old.
```

To deal with this, you could use a Bash array. I’m not getting into this.

Wildcards are a frequent usecase, and provide usable lists:

```
for f in *.sh
do
  echo -n "[$f] "
done
-----------------
[pre.sh] [script1.sh] [script2.sh] [script3.sh] [x.sh]
```

Note that in that case, spaces are correctly handled:

```
$ touch "space name.sh"
$ for f in *.sh
  do
    echo -n "[$f] "
  done
-----------------------
[pre.sh] [script1.sh] [script2.sh] [script3.sh] [space name.sh] [x.sh]
```

There is a convenient command to generate numbers in a range: `seq`

```
for value in $(seq 0 2 10)
do
  echo -n "$value "
done
----
0 2 4 6 8 10
```

The middle argument is the step, and is optional.

15.10 Break and continue

The commands **`break`** and **`continue`** have their usual roles: **`break`** will exit the innermost loop outright, while continue will jump to the next iteration of the loop.

15.11 Select statement

Finally, Bash offers a convenient form of loop for the frequent user interaction where the user is required to select one of a number of options, and is prompted repeatedly until he does so.

```
select <variable> in <list>
do
  <commands>
done
```

However, this construct does not exist in the barebones shell.

16 About the exam

You will not be tested on shell or Bash during the final exam of this Python class.

However, note that if you are not at least minimally proficient in shell, you will waste time in lab classes and tests, which will indirectly and severely penalise you in C programming, system programming, and any class using a Linux environment.

Part II

Lecture notes: Python

17 What is Python, again?

Python is a programming language. Let's go through a few buzzwords to describe it briefly:

- **General-purpose:** Python is not designed with a specific, specialised purpose in mind. Whatever it is you want to do, unless it is something very niche, like kernel/driver writing or something similar, you can do it in Python, in principle. It has some serious weaknesses, like performance (which can be mitigated to a large extent, as we discuss below), and there are plenty of specific kinds of tasks for which other languages are better-suited (for instance I find that algorithms on trees and other inductive structures are *ever so much* cleaner in languages with algebraic data types and pattern-matching, such as `OCaml` and `Haskell`), but there is no "non-niche" task for which Python is ill-suited to the point of tin-foil–hat absurdity, like writing a website in assembly would be.

  Python can be and has been used for projects of all sizes, from small scripts to Instagram's entire infrastructure (serving 400 million active users per day).

  As for the ecosystem around the language, while of course not all areas enjoy the same level of support, whatever you are trying to do, there is *usually* a fairly decent library for it.

- **Popular:** Python has been around since the end of the eighties. It is studied in most places where programming is studied – most universities, most "classes préparatoires", and all French high schools offering computer science options as of the coming years.

  It is extremely popular in the fields of data science and machine learning, in particular, both in academia and industry.

  It is used intensively by many companies including Instagram (quasi exclusively Python + Django), Spotify (at about 80%), Amazon (for Big Data), SurveyMonkey (migrated from C#), Dropbox (client is 10^6 lines of Python), Facebook (all image processing back-ends), Google, YouTube, PayPal,. . .

  While evaluating the popularity of languages in a meaningful fashion is tricky, it is clear that Python has been in the top ten of most popular languages for over a decade, and is currently ranked third, behind Java and C, by the TIOBE index.

  Love it or hate it, you are probably going to run into Python code during your career. Repeatedly.

- **Multi-paradigm:** Like pretty much all of modern languages, Python combines elements from multiple programming paradigms. Python is object-oriented at its core (you will have a course on that this semester), naturally supports the usual procedural programming style, without forcing an overt object-oriented architecture, and supports several tools of functional programming as well (it is a subject of great consternation for me that your syllabus does *not* include a course on that topic. I strongly invite you to study `OCaml` or `Haskell` in your spare time), although there are a few obstacles to writing primarily in a functional style, such as the lack of tail recursion optimisation.

- **High expressive power:** The words "power" and "expressiveness", when it comes to programming languages, can have several different meanings. I take this opportunity to clarify these from a general standpoint, and say a few words about Python's position along these metrics.

  - In the theory of computation: (you can skip this if you're not curious)

    The expressive power of a language is the set of problems that you can solve by writing a program, regardless of computation time, so long as it remains finite. Not all problems can be solved by a program / an algorithm.

    Those problems which admit an algorithmic solution are called decidable. Examples include "In: a list of integers. Out: is the list sorted, yes or no?".

    The others are called undecidable. Example: "In: A program's code. Out: Are there are no infinite loops in the program? That is, will it terminate for all inputs?".

    No matter how smart and talented a programmer you are, you *cannot* write a (correct) algorithm solving an undecidable problem in finite time.

    The fundamental reason is that, though there are both infinitely many programs and infinitely many problems, there are still considerably *more* problems ($\aleph_1$) than programs ($\aleph_0$), so it is unavoidable that some of them (in fact almost all of them) cannot be paired with a program that solves them. (Keywords: "(Un)Decidable problem"; "Cantor's diagonal proof"; "Aleph numbers").

    In this sense of "power", it turns out that all general-purpose programming languages are strictly equal. (Keyword: "Turing-complete").

    The idea is that, as soon as you have conditional branching and either loops or gotos, you can cobble together a "simulation" of any other structure that you want by cleverly using what you have – even if it's horrible to write and slow to execute. There is no known or even imaginable language or mode of computation – not even quantum computers – that is fundamentally beyond the reach of what you can do with those simple tools. (Keyword: "Church-Turing thesis").

    Thus if you can solve a problem in Python, you can solve it in Java, C, Brainfuck, TEX, etcetera. Conversely, if you can't solve it in one of them, you can't solve it in any of them.

    You will study some elements of this in your fourth year (and, this semester, the course on formal languages and automata theory is closely related). For now let's just say that you can't compare general-purpose languages that way, since they are all equal in that respect, and move on from this riveting topic.

▷ Elegance, clarity, conciseness, modularity, robustness, developer speed.

I have this idea for an algorithm. How many lines of code will it take to implement it in this language, as opposed to that one? How much time will it take me to come up with these lines? Once written, how clear is it what they do, and that they do what they are supposed to do? Is the code highly modular and robust, so that I can reuse solutions to subproblems elsewhere, or is it a big interconnected mess of spaghetti code held together by duct tape, creaking and swaying in the breeze, ready to come crashing down on your head if you breathe a little too hard in its general direction?

In other words, how easy is it to write good code (for several metrics of *good*)?

Here, Python mostly shines. You can do quite a bit in a single line of Python code, compared to C or Java, and its syntax and structure are regular and polished enough that it's usually quite obvious what that "quite a bit" is, and it does not take hours to come up with the line.

Thus you can often write complex algorithms in a fairly fast, compact, direct, and readable way. You don't have to jump through too many hoops. This contributes, *ceteris paribus*, to the chances of what you have written being correct.

Beyond that, Python is a dynamic language, with very little typing discipline, and does not intrinsically *force* you into writing (somewhat) correct code the way a language like `OCaml` or `Haskell` does, so if you want robust code, you have to put in some extra work.

When learning the language you have to take some care to know what you're doing, because the interpreter will accept a lot of nonsense without protest.

Since version 3.6, this can be mitigated with the use of optional type annotations and external static type checkers, such as `mypy`. This can be invaluable in large projects.

Overall, the characteristic of Python which I have seen praised most often in reports where large companies discuss why they used or switched to Python is *speed of development*, which is basically an aggregate of all these measures.

▷ Execution speed and efficiency:

How fast will your program run? Now this is not so much a property of a language as of the implementation of its interpreter or compiler, *and* of the specific way you have written your program.

But still, *generally speaking*, task for task and *mutatis mutandis*, Python is quite a bit slower than C, and even than C# and Java.

And none of that matters in the slightest unless you are limited by very weak embedded systems, are doing some serious data crunching, writing a kernel, a driver, or otherwise doing intense systems or network programming, or other fairly niche activites.

Outside of that, raw execution speed is rarely the bottleneck in any project. This usually falls on disk access, network access, display update, user interaction, or, considering the wider context of a project, programmer time (and cost!). And when execution speed is really a problem, writing smarter algorithms is usually a better answer than changing language or updating hardware.

If you can come up with a working Python prototype in a tenth the time needed to do the same in C or Java, (and in many case, you very well might), that leaves you a whole lot of time to do some performance testing and profiling, to find out what the *real* bottlenecks are, and come up with smarter algorithms. If absolutely needed, you can always write some critical parts, and only them, in C or assembly. Python's numerical computation libraries, such as numpy, are mostly written in C or Fortran behind the scene.

Let's also note that some implementations of Python are much (about 5 times on average) faster than the default one, as we'll see in the next point.

◇ **Interpreted:** Leaving aside that, *stricto sensu*, this is a property of an implementation, not of the language itself, the answer is "sort of". Python is compiled to byte code (`.pyc`), which is then executed (CPython backend). This is similar to Java and .Net. And there are Python compilers targeting both Java (Jython) and .Net (IronPython) virtual machines. There is also a Just In Time compiler in PyPy. That is to say, the code is compiled to machine code on-the-fly. PyPy is almost always faster than the default CPython, with an average speed-up of about 4.5 times — on certain computationally intensive benchmarks, like AES (cryptography) or raytracing, the speedup is much higher, about 50 times faster.

At any rate, Python has all the benefits of an interpreted language, such as an interactive mode (absent from C, Java, and the like), and what happens behind the scene (bytecode compilation etc) is entirely transparent for the user.

It also has the main drawback of an interpreted language, in that you need an external interpreter (here at least a bytecode interpreter) to run your program, as opposed to natively running it on the machine. But then the same applies to Java and .Net.

If you ever need a single executable file, you can always ship the interpreter (which is lightweight) with the code as a resource in the executable – the same principle as self-extracting archives.

The bottom line is that the distinction between interpreted and compiled can get quite blurry, especially for languages with a rich ecosystem of backends. Modern languages are rarely purely interpreted – even Lua, a lightweight language designed to be embedded in applications to provide scripting capabilities, has a bytecode virtual machine.

◇ **Not just a scripting language, please:** Python is often called a scripting language. This is mostly due to its original purpose, 30 years ago (a scripting language for the Amoeba operating system). Of course, a lot has changed since then.

There is also an often observed oversimplification along the lines of "interpreted language = scripting language", which holds perfectly in the limited ecosystem { shell, C }, but falls apart entirely today, even leaving aside that "interpreted language" can be misleading, as discussed above.

Python happens to be *fairly good* as a scripting language, because it does not require writing lots of boilerplate to do simple tasks, and has an interpreter and an interactive mode.

Having those characteristics in no way takes away from its capacity in other contexts; they just add to what makes it a *fairly good general-purpose language*. Scripting is just one among its many uses. Obviously the existence – and clear trend towards – large scale Python projects in industry shows that it is a very realistic choice in realms far removed from the usual scope of "scripting".

There are other languages that are more clearly geared and specialised towards scripting, and that you might prefer to Python depending on your scripting needs.

For glue between your programs, most of the times a shell script is all you need.

If you want to add scripting capabilities to a game or something like that, Lua is very lightweight (about 1 Mb with all standard libraries), and is specifically designed to be embedded in other applications.

18 Version 3 versus the world

18.1 Python 3.9 is required, 3.10 preferred

We deal with **Python version 3**. A very useful feature, fstrings, was introduced in **3.6**, so that is the absolute minimal version. I wrote most of this document under that version.

3.7 introduces an important change to the behaviour of generator functions. This change is explained in this document.

There is some cool new stuff in 3.8 and 3.9 but that is relatively anecdotal but nice to have.

3.9 is therefore the minimum required by the course.

3.10 introduces the **match** statement, which is awesome, and is strongly recommended from the onset. Unfortunately,. . . it's not out yet; not until October 2021.

This does not prevent your from downloading and compiling the source code for the release candidate from `https://www.python.org/download/pre-releases/`. See how to compile in Sec. 19[p57]: "How to install the damn thing *without* being admin?".

3.10 will become mandatory as soon as it is officially released.

18.2 Python 2 is right out!

Quite importantly, there are a few key differences between Python 2 and 3, making them two similar but incompatible languages. This is often a source of bugs and confusion.

To quickly ascertain whether you are running 2 or 3, try the command `print 0`. If it works, you are using Python 2, where `print` is an instruction, in the same way that `return` is, and does not require parentheses around its argument. If it throws a `SyntaxError`, you are using Python 3, where `print` is an ordinary function.

More subtle differences include the behaviour of `/`, which always yields a `float` in Python 3, but whose return type varies with the type of its operands in Python 2.

If your program worked perfectly before, and behaves weirdly on another machine, odds are you're not using the right version of Python.

Note that on most setups the `python` command if for Python 2, and you need to use the `python3` command instead.

19 How to install the damn thing *without* being admin?

19.1 I'm cool, I've got Linux

Unless your Linux distribution provides an acceptably current version via its package management system, use a local installation to ensure you have the latest version without needing any administrative right on your machine.

To do so, download the source distribution from `https://www.python.org/`.

Ensure that you have GCC installed, as well as TK and SSL development packages: the first is necessary for TkInter GUIs, including Idle, whose source is bundled with Python, and the second for integration into PyCharm, among other things for which SSL might be useful.

On a Debian-based system

```
apt-get install tk-dev libssl-dev
```

will take care of TK and SSL modules.

Then you can go to the root of Python's sources and do, as usual:

```
./configure --prefix=$HOME/Python310 # for instance
make
make install
```

Add the resulting binaries to your PATH if you like.

19.2 I've got Windows

From `https://www.python.org/`, download the installer; run it from the files manager (it seems that downloading and running from the web browser sometimes triggers an admin rights confirmation dialogue on INSA machines.)

During the installation, there is a checkbox about something like "export environment variables" or something to that effect[f], which is **not** checked by default. You must check it, otherwise, while Python will be installed on your machine, you won't be able to simply call `python3` at the prompt, but will need to use an absolute path, along the lines of `C:\Program Files\...\python3`.

Uncheck the box "for all users" (as it requires admin access)

20 How to use the damn thing?

From the shell command prompt, `python3` launches an interactive mode. `python3 yourcode.py` executes your program and exists. If you want to run your program and open an interactive Python prompt with it, `python3 -i yourcode.py` should do the job.

Python source files can be made executable in the same way as shell scripts, by including as the first line

```
#! /usr/bin/env python3
```

and making the file executable

20.1 Python Editors

Like for most if not all programming languages, Python code is just text, with all the special keywords being ASCII. Thus, you can use any raw text editor to write in Python.

That being said, it is far more convenient to use specialised editors for this purpose.

[f] Students: If someone goes through this, please tell me the exact wording, so I can update this.

By default, you can use **Idle**, which ships with Python in the Windows distribution, I believe. On a Debian-based Linux system, `apt-get install idle3` should do the trick.

The **Spyder** editor should be easy to install and use, while being an *immense* improvement over Idle. `apt-get install spyder` should do the trick on Debian/buntu, and `pacman -S spyder` for Arch. **It is my personal recommendation** if you can install it easily and don't have a favourite editor you wish to use.

For the largest projects, I personally use **PyCharm**.
`pacman -S pycharm-community-edition` for Arch.

If nothing else is available, there is also an online editor: https://repl.it/languages/python3

Be that as it may, the best editor to start with is probably whatever you are used to and feel most comfortable with, although anything that provides an interactive mode, like Idle does, is at an advantage.

20.2 Things to do in Idle

Since Idle is the default choice, a few tips:

Idle opens to an interactive Python prompt by default – unless you opened a file with it – which is fun and useful if you want to play with a calculator, but not great to do any real programming.

Use the *File* menu to create or open a file – `.py` extension please – in which to write your code. Put the window with the file on the left, and the window with the prompt on the right (or whatever floats *your* boat).

Then, you can simply press F5 from the left to clear the session on the right, and execute all your code again. Note that you can configure F5 to save the code without prompting you — Options/Settings/General/Editor Preferences.

Note that there is a slight difference between the interactive mode and the normal, coding mode: values like `42` will be printed automatically from the prompt, but not when executing the code in the left window. If you want to see the value on the right, do `print(42)` on the left.

20.3 Things *not* to do in Idle

(Also applies to any other editor with a dual prompt / file view)

Do not write complex code in the prompt! It's just for *one-time*, throwaway testing, not directly related to the function you're writing — for instance, testing how a standard function works on edge cases, before writing code using it, getting `help()` etc.

I have lost count of the number of times I have seen students, having defined

```
long_ass_function_name(with, many, parameters)
```

testing it (well *at least* they're testing it) by writing a truckload of tests with different input values in the prompt:

```
>>> long_ass_function_name(0, 0, 0)
>>> long_ass_function_name(1, 0, 0)
>>> long_ass_function_name(1, 1, 0)
...
```

Leaving aside the incoming carpal tunnel syndrome (sure you can repeat previous commands to some extent, but it's still cumbersome), every single test written that way becomes void the instant the function is modified, and you need to retype everything.

Furthermore, the execution context of your prompt may not be synchronised with your code, which can fudge up your tests. I am often called for "mysterious" bugs that only exist because the code Kevin is *actually* testing is a half-hour older than what he *thinks* he's testing. Or because a definition has been shadowed.

The decision process on whether to test your function on the left (code window) or the right (pocket calculator) is simple: either you care to keep the tests for posterity, or you don't.

If you *don't* care to keep the tests: write the test on the left, press F5 to test, change the values, press F5, etc. Then remove or comment test.

If you *do* care to keep the tests: write them on left, copy-and-paste to your heart's content until you have all the cases you need. Press F5. Profit. Then either comment them so you can easily repeat them later, or better yet, man up and turn them into `assert`s. (More on that later.) Or unit tests, later on.

Either way: **write them on the left.** The end.

20.4 A few unkind words on vim

I see a lot of students using vim, which I find mysterious, as most of them don't use any of the special features that might justify it. Perhaps it's like "Creative people use Mac; peons use PC." but with text editors? Perhaps just *using* vim makes you popular with girls at parties? I really wouldn't know.

Be that as it may, *I* don't use vim, and if you call me on your machine to debug your code, expect some annoyance if I can't CTRL+C,V and other stuff. If you are not using anything vim-specific, please, pretty please, use something else, will you. Not Emacs[g], though.

[g] Sure, Emacs is a great operating system, but what it lacks is a good text editor.
— *Everyone who does not use Emacs. Everyone who uses Emacs.* Everyone.

This to a large extent is a matter of taste, I'll admit. Much worse than that, however, is that a lot of vim users I have seen among our students just have a console with vim, displaying the code, and... that's all.

When they want to test their code – and crucially, when they call me to debug their code – they save, exit vim, run `python3 theirscript.py`, read the errors, and then run vim again, which hides everything else.

That way, they never have to see the code and the error messages at the same time. Maybe it's some psychological defence mechanism to preserve the illusion that their code is flawless. Maybe. But it drives me ab.so.lutely bananas. If you are debugging something, it is *really* helpful to have both the code and the error messages (or the output, generally speaking) available to your eyes at the same time.

You do what you want on your own time, but if you call *me* on *your* machine to sort *your* mess out, **make it so that code and output sit comfortably in their respective windows, preferably side-by-side**. And make sure the error messages I see actually pertain to the code next to it.

I have also seen people do the same "hide the output" setup with *nano*, of all things, instead of vim. I... presently lack the fortitude to comment further on the matter. Just don't.

21 A few basic points of syntax

21.1 Keywords, `help()`

Python offers an interactive help mode capable of listing and providing help for all keywords in the language:

```
>>> help()
...

help> keywords

Here is a list of the Python keywords.  Enter any keyword to get more help.

False               def                 if                  raise
None                del                 import              return
True                elif                in                  try
and                 else                is                  while
as                  except              lambda              with
assert              finally             nonlocal            yield
break               for                 not
class               from                or
continue            global              pass
```

```
async               await               # added in Python 3.7
```

As usual, any keyword is reserved by the language and cannot be used as a variable name.

The help function can be called on any function or type, and will display its documentation string:

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

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

The / at the end means the end of positional parameters of the function. This is not very important in most cases; pretend it is not there.

In the case of types, or “classes”, help displays the documentation of the class and that of all its *methods*.

```
>>> help(int)
Help on class int in module builtins:

class int(object)
 |  int(x=0) -> integer
 |  int(x, base=10) -> integer
 ...
 |  __add__(self, value, /)
 |      Return self+value.
 ...
 |  conjugate(...)
 |      Returns self, the complex conjugate of any int.
 ...
```

21.2 A few words about Object Oriented Programming (OOP)

Here we need to say a few words about object-oriented programming (OOP), since Python is object-oriented at its core. Of course you will study OOP properly, in its own course, this year, and I shall come back to this later in this document; the aim in this section is merely to put down some terminology and a basic intuition of what’s going on.

The idea is that an object, in the abstract, has a certain set of properties, and can perform a certain set of actions. For instance, a car has a weight and an age, and can start; though how it starts may well depend on, say, its age — if it’s over 30 years, it does not start.

In OOP, one would define a class `Car` — the abstract set of all possible cars — stating that a car has *attributes* `weight` and `age`, and a *method* `start()`. An *instance* of the class `Car`, that is, a specific car, yours for instance, is an *object* of type `Car`. Let’s call your car `c`.

Then your car has its own weight and age, distinct from those of any other car, which you can access as `c.weight` and `c.age`. The method `start()` is just a function that implicitly belongs to a given car; you can call it as `c.start()`.

This method call can be thought of as a function call `start(c)`; however, many other classes may define a `start()` method, such as an engine or an MP3 player, with quite different behaviours.

So if you wanted to define a function capable of doing `start(c)`, while also allowing the other starting behaviours of other types — e.g. `start(my_engine)` — you would need to test the type of the argument in the body of the function `start`, and handle all possible behaviours accordingly: if it's a car do this, if it's an engine do that, et cetera.

At the end of the day, this would result in the code pertinent to a given type being fragmented in every function that deals with it. Instead, with classes, most if not all of the code that deals with a given type is neatly grouped, and the type dispatch in shared methods is implicit. This is a form of *polymorphism*.

Consider the built-in function `len`, for instance. There are plenty of types for which it makes sense to compute the length of instances, which `len` does. That includes strings, lists, tuples, but also unordered collections such as sets and dictionaries, which contain a clear number of elements. In the latter case, `len` is understood to compute the cardinality.

If you define a new type and want `len` to work on it, you do not need to find the definition of `len` and extend it. Actually, all `len(x)` does is call a special method `x.__len__()`. If the object `x` does not implement this method, `len` raises an error accordingly. So all you need to do to support `len` with your new data type — if it makes sense to do so — is to define `__len__` for your class.

The same principle applies for many other things, such as operator overloading.

We shall continue discussion OOP in Sec. 27[p183]: "Object Oriented Programming in Python".

21.3 Whitespace

Contrary to most languages, Python uses whitespace as significant syntactic element. Returns mark the end of a statement, or "logical line", as would a semicolon "`;`" in Algol-style languages such as C, and the levels of indentation – that is to say, how far you are from the left margin – serve to delimit blocks, as do curly braces "`{ ... }`" in C. A physical line can contain several statements, if separated by a semicolon `;`.

Of course, there are exceptions, as line returns can be escaped through the use of a backslash '\' to enable a logical line to stretch over several physical lines. There are also contexts, such as within a list, where line breaks are automatically escaped without a need for an explicit backslash.

The following are all logical lines in Python:

```
n = 1 + 2

l = [1, 2, 3, 4]

n = 1 + \
    2

n = (1 +
     2)

l = [1, 2,
      3, 4]
```

Indentation at the *beginning* of a logical line determines how deeply nested it is.

```
if C:     # outer block
  if D:   # inner block
    pass  # inner-inner block
  else:   # inner block
    pass  # inner-inner block
else:     # outer block
  pass    # inner black
```

New blocks, and thus deeper levels of indentation, are always introduced by a colon `:`, as in `if <test> :`, `else:`, `def ... :`, `try... :`, et cetera.

When the block contains only one line, the line return can be omitted:

```
>>> if True: print("it works")

it works
```

Be very careful not to mix tabulation and regular spaces in your program. Depending on how your text editor represents tabulation, some things may *look* perfectly aligned which actually are not.

This can make it very difficult to control visually that the code is correctly nested, and is a common source of maddeningly annoying bugs.

It is recommended to set your text editor to translate tabs into a fixed number of regular spaces, generally between 2 and 4, the latter being more-or-less standard.

Internally, Python does not convert tabulations into spaces for the purpose of evaluating indentation, but keeps count separately. Thus the rule it follows is: "two lines are at the same level if they are indented by the same number of spaces, and the same number of tabulations".

21.4 Importing modules

Sometimes you have to import stuff from other modules to get work done. For instance, I'd like a square root function. There is one in the standard library's `math` module, named, appropriately enough, `sqrt`.

I cannot use it directly:

```
>>> sqrt(10)
NameError: name 'sqrt' is not defined
```

But what if I import the math module?

```
>>> import math
>>> sqrt(10)
NameError: name 'sqrt' is not defined
```

Still does not work, because I just imported an object call `math`, which contains a number of functions, including `sqrt`. Importing a module executes its code, so as to process all the definition, and binds the module object to its name. It does not automatically pour all of its contents into the current namespace. What I can do is get some help on the module

```
>>> help(math)
```

and access the functions in its namespace, for instance:

```
>>> math.sqrt(10)
3.1622776601683795
>>> math.ceil(math.sqrt(10))
4
```

We say that `math` is *imported* and *bound* locally.

If the module name is too long, or conflicts with my local definitions, I can bind it under a nicer name:

```
>>> import math as m
>>> m.sqrt(10)
3.1622776601683795
```

If all I am interested in is `sqrt`, I can just import that, and only that, directly bound under that name:

```
>>> from math import sqrt
>>> math.sqrt(10)
NameError: name 'math' is not defined
>>> sqrt(10)
3.1622776601683795
```

The whole module is still *executed* during the import, even if you only want and *bind* part of it. This matters if modules have side effects, like **print**s, or unit tests, which shouldn't be the case in production.

You can also import just a function and rebind it:

```
>>> from math import sqrt as s
>>> s(10)
3.1622776601683795
```

Finally, you can import everything into the current namespace:

```
>>> from math import *
>>> ceil(6.7)
7
```

There can be modules inside modules, like foo.bar.baz; this does not present any specific difficulty.

21.5 Defining variables and functions

21.5.1 Variable assignment

Variables in Python do not need to be declared. They do not have any default value, not even None.

```
>>> IdontExist
NameError: name 'IdontExist' is not defined
>>> IdontExist = "And now I do!"
>>> IdontExist
'And now I do!'
```

Note: assignment uses =; comparison uses ==.

Python is case sensitive. X and x are not the same variable.

Identifiers (i.e. variable names) can contain alphanumeric characters[(h)] (though that cannot begin with a number) and _ (underscore AltGr+8 on an AZERTY keyboard). No whitespace, no other special symbols. No reusing language keywords. This is all pretty standard.

Avoid redefining identifiers like len, or help.

Variables can be removed using the **del** statement:

```
>>> x= 2
>>> x
2
>>> del x
>>> x
```

[(h)] Including Unicode alphanumerics, since version 3.0.

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

This does not immediately remove the structure the variable `x` points to from memory; this merely removes the binding, the pointer, from `x` to that structure. Python's garbage collector will free the memory at some point, assuming there is no other pointer to that same structure.

You will almost certainly *never* need to use **`del`** in that fashion. The keyword finds its real use in removing elements from lists and dictionaries.

21.5.2 Parallel variable assignment

Assignment can be parallel. Actually this is a special case of some slight pattern-matching capabilities on tuples, which we shall see later:

```
>>> x=1; y=2
>>> x,y = y,x
>>> print(x,y)
2 1
```

21.5.3 Incrementation and similar operators

There is no `++` or `--` operator in Python, but you can write

```
i += 1
```

as a shortcut for

```
i = i + 1
```

and variants `%= /= //= -= += *= **=` exist for many other operators. Those shortcuts are always defined on types for which the corresponding operator is defined. In the presence of mutable structures, the equivalence between `x = x+y` and `x += y` may no longer hold, as we shall see when we discuss lists.

21.5.4 Defining functions; predicates and procedures

Functions are defined with the following syntax (there is more to it; once we have seen tuples and dictionaries we shall see in more detail, in Sec. 25.1[p171]: "Variadic function definition", how to define variadic functions and keyword arguments)

```
def <functionname> (<arg1>, ..., <argN>):
  """optional doc string for help(<functionname>)"""
  # you do your stuff here
  return <something nice>     # optional; no return => return None
```

Note that the colon `:` opens a block, which is the function body: you must indent, and keep the indentation consistent.

Procedures are functions that do stuff but return nothing (or, in Python, `None`).

Predicates are functions that return Booleans (`True` or `False`).

```
def f(a,b,c):
    """ My function f
    does some cool stuff, dude. """
    print("Call f:", a,b,c)
    return a + b - c

print( f(1,2,3) )
----------------------------
Call f: 1 2 3
0
>>> help(f)
Help on function f in module __main__:

f(a, b, c)
    My function f
    does some cool stuff, dude.
```

21.5.5 Functions are first-class citizens

When we say that functions are first-class citizens, we mean that they should not be thought of as fundamentally different from any other type of objects in the language. Integers, Booleans, lists, etc are all types of objects with their own properties, that can be passed as arguments to functions, and returned from a function.

In Python – and any language supporting some degree of functional programming, which is to say nearly all modern languages – functions are no different from anything else, and can be passed as arguments and returned, thus forming what is called “higher-order functions”.

For instance, let us pass functions as arguments:

```
def f(x): print("F",x)
def g(x): print("G",x)

def do(x,f):
    f(x)

do(0,f)
do(1,g)
----------------------
F 0
G 1
```

No special syntax is needed. A function `f` is an ordinary object, just one you can “call” with `f(..)` syntax. A “callable”, in Python terminology. If needed, you can test whether an object `o` is callable by writing `callable(o)`.

Let us try returning a function, then:

```
def hFactory(letter):
    def my_h(x):
        print(letter,x)
    return my_h

do(2, hFactory('H'))
-----------------------
H 2
```

Note that the version of `my_h` that is returned still has access to `letter`. More generally, a returned function still has access to all local variables. This is called a *lexical closure*.

Note: Python is still a dynamic language, though, so the resulting behaviour can in some case differ from that of statically typed functional languages that make heavy use of lexical closures, such as OCaml and Haskell. Sec. 64.3[p355]: "Foncteurs et décorateurs de mémoïsation" provides an (advanced) exercise that illustrates that.

21.5.6 Anonymous functions: **lambda**

When using higher-order functions, it is often convenient to be able to define functions on-the-fly, as an expression, without having to use a **def** statement and finding a name for it. That is the use-case for anonymous functions, using the **lambda** keyword.

As an aside, that keyword comes from λ-calculus, a formal, universal model of computation, and the direct inspiration of functional programming languages (LISP, Scheme, Standard ML, OCaml, Haskell,. . .).

The syntax is the following:

```
lambda <arg1>, ..., <argN> : <returned expression>
```

For instance, continuing the previous examples:

```
do(3, lambda x: print("L",x))
-----------------------------
L 3
```

There is no meaningful semantic difference between **lambda**-defined functions and **def**-defined ones, apart from one *intrinsically* having a name, and the other not:

```
def add(x,y):
    return x + y
```

and

```
add2 = lambda x,y : x+y
```

are equivalent, for all intents and purposes. Note that the **return** keyword is implicit in the second form. They are of the same function type:

```
>>> add
<function add at 0x7f873ffaff28>
>>> add2
<function <lambda> at 0x7f873ffbf048>
```

There is actually a *syntactic* restriction to **lambda**-expressions. Since they are *expressions*, as opposed to *statements* – see Sec. 22.5[p98]: "Nihilism: NoneType: expression versus statement" – they cannot contain any instruction, such as **if..elif**. However, they can use the ternary operator .. **if** .. **else** .. and comprehension expressions, so this is not a very stringent restriction. (It just serves as a hint that Python is not *really* a functional programming language, even though you can do some stuff in that style.)

lambda-expressions are best use for very short, very simple throwaway functions anyway. For anything more meaty, use **def**, and name them, even if the name will not actually be used again.

lambda-expressions can be chained:

```
>>> f = lambda x: lambda y: lambda z: f"{x}{y}{z}"
>>> f(1)(2)(3)
'123'
>>> ( (f(1))(2) )(3)
'123'
```

f is a function that takes an argument x, and returns another function. What that function is and does depends on the value of x; let us call it f_x. This function f_x itself takes an argument y, and returns yet another function f_{xy}, which takes an argument z, and returns a value depending on all three values x,y,z.

This is an instance of a general technique to reduce all n-ary functions to unary functions, called *currying*. It is prevalent in functional languages of the ML family (OCaml, Standard ML, Haskell,. . .), where all functions are implicitly curried, which is key to a number of interesting techniques. In Python, however, chaining **lambda**s like that should generally be avoided; we have unfortunately neither the syntactic sugar nor the tooling necessary to make such constructs worthwhile.

21.5.7 Optional arguments

Again, we will come back to that in more detail later, but it is worth mentioning optional arguments right now, as some common functions, such as **print**, have them.

A more general pattern for defining functions is this:

```
def <functionname> (<arg1>, ..., <argN>,
                   <optarg1> = <defval1>,..., <optargM> = <defvalM>):
```

Mandatory arguments come first, followed by any number of optional arguments, to which a default value is passed.

Thereafter, whenever the function is called with some optional argument being passed a value, the provided value is used by default.

In a call, the arguments can be provided positionally (first, second, etc) or by keyword, in which case they can be passed in any order.

```
def g(a,b=2,c=3):
    print("Call g:", a,b,c)
    return a + b - c
-----------------------------
>>> g(0)
Call g: 0 2 3
-1
>>> g(0,20)
Call g: 0 20 3
17
>>> g(0,20,30)
Call g: 0 20 30
-10
>>> g(0,c=30)
Call g: 0 2 30
-28
>>> g(0,c=30,b=20)
Call g: 0 20 30
-10
>>> g(c=30,a=100)
Call g: 100 2 30
72
```

In all cases, positional arguments must come first, and keyword arguments follow:

```
>>> g(a=1,2,3)
SyntaxError: positional argument follows keyword argument
```

21.5.8 Order of evaluation of arguments and expressions

In Python, arguments are evaluated in the order in which they are passed, left-to-right:

```
def x(n): print("arg",n)
print(x(1), x(2), x(3))
------------------------
arg 1
arg 2
arg 3
None None None # see the section on None
```

This is the case for all kinds of expressions, not only functions; the only real exception is assignment: `z,a = x,y`, where `x,y` is evaluated before `z,a`, of course.

This behaviour is part of of Python's specification. Nevertheless, it is somewhat inadvisable to write code that relies on this.

21.5.9 Scopes: local, `global`, and `nonlocal`

The *scope* of a variable is the context in which such a variable is visible. In Python, variables are local unless otherwise specified, which means they exist only in the scope of the function in which they are defined. The scope outside of any function is called the global scope.

In practice, you will only very rarely need to worry about scope. Simply remember that the variables inside a function are local to it, and cease to exist after the function has returned.

Using global variables can be handy in a few circumstances, but it is generally a very bad idea and strongly discouraged in any language. Likewise, `nonlocal` can be useful, but the situations in which that is the case are quite complex (decorators with own state,. . .). Furthermore, scoping rules in Python have some rather prodigiously ugly edge cases.

Thus, **you can skip this section until you need it**, especially given that, in order to make the most out of this section, you need to understand some data structures (lists, dictionaries, strings) and adjacent notions (mutability,. . .).

21.5.9.1 The `global` scope

Let us take an example to experiment with different scenarios, to see how Python handles global and local scopes.

```
a,b,c,d = 10, 20, 30, 40
l = [1,2]
print(globals())

def fun(a):
    a = 12
    b = 21
    global c
    c = 31
    global e
    e = 51
    ee = 61
    l.append(3)
    print(a,b,c,d,e,ee,l)
    print(locals())

fun(11)
print(a,b,c,d,e,l)
print(globals())
print(ee)
```

```
------------------------

{'__name__': '__main__',..., 'a': 10, 'b': 20, 'c': 30, 'd': 40, 'l': [1, 2]}
12 21 31 40 51 61 [1, 2, 3]
{'a': 12, 'b': 21, 'ee': 61}
10 20 31 40 51 [1, 2, 3]
{'__name__': '__main__',..., 'a': 10, 'b': 20, 'c': 31, 'd': 40,
    'l': [1, 2, 3], 'fun': <function fun at 0x7fa8def98dc0>, 'e': 51}
NameError: name 'ee' is not defined. Did you mean: 'e'?
```

There is a lot to unpack, here. First, we declare and initialise a,b,c,d,l in the global scope. For the next line, we take a peek at Python's global scope thanks to the built-in **globals** function:

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

globals()
    Return the dictionary containing the current scope's global variables.

    NOTE: Updates to this dictionary *will* affect name lookups in the current
    global scope and vice-versa.
```

It is not necessary right now, but you can look up what a dictionary is in Sec. 24.4[p144]: "Dictionaries: class dict" — Python uses this data structure extensively behind the scenes. We see in that global scope a lot of technical information that we don't really need to worry about, along with our variables and their values.

Now, inside our fun function, we manipulate a, which is an argument, and therefore a local variable: anything we do to it in the function is forgotten afterwards. The global a is completely shadowed within fun, we never have access to it.

With b=21, we create a new local variable b, and affect a value to it. This new definition is local, and shadows the global b: it is a different variable. Nothing we do to our local b has any effect on the global one.

With **global** c, we declare c as global, meaning that the c in the function is the same variable as the global c. Any modification done to it in fun is done as well in the global scope.

With **global** e, we *create* a new global variable e, which, unlike c, did not previously exist.

With ee=61, we create a new local variable, that does not shadow any existing global variable.

With the local **print**, we observe that the global variable d is read, though we have neither declared it as global nor shadowed it locally. Our local scope can *read* variables from the global scope just fine, it just cannot *rebind* that variable in the global scope, unless the variable is declared as **global**.

Note that not being able to rebind a global variable does not mean that a function cannot modify the *contents* of global scope variables. Take the list `l` (cf. Sec. 24.2[p130]: “Lists: class `list`”). With `l.append(3)`, the value of `l` is read. That value, however, is fundamentally a pointer towards an object, and that object can modify its own state, if asked to do so.

Thus, we can modify the contents of the list. The variable `l` itself is unchanged: it still contains the same pointer towards the same zone in memory, but the contents of that zone have changed during the execution of `fun`. Thus, at the higher level where we do not think in terms of pointers, the value of `l` has changed, by in-place modification.

As you can guess, `locals()` is the local equivalent of `globals()`, and tells you the contents of the local scope. We observe that global variables do not appear in the local scope at all.

Another (advanced) difference is that `locals()` is read-only, whereas you *can* alter the global namespace by acting directly on the dictionary returned by `globals()`. It’s probably not something you *should* do, but there *may* possibly be valid use cases for programmatically defining new variables.

This can’t be done with `locals()` in Python3[i], because local variables are not internally represented as a dictionary, but as an array computed at compile-time, for performance reasons. The dictionary returned by `locals()` is merely a *view* of that static array. Technically, you *can* modify `locals()`, but the changes won’t be passed on to the array structure, so it is utterly useless to do so.

Again, this (the read-write / read-only behaviour of `globals()` / `locals()`) is a rather advanced point. I don’t expect you to manipulate those structures very often, if at all. Certainly, no exercise or exam question will require that kind of fine knowledge. The lesson here is that this sort of introspective structure (that is to say, structure dealing with the internals of the language itself) can help you better understand how scopes work and debug your program (if you don’t have a nice IDE to visualise that kind of things), but please don’t start using them willy-nilly in your programs.

21.5.9.2 *The ugly: the scoping heuristic*

Let us execute that unassuming little snippet of code:

```
a = "Am I global or local?"
def f():
    print(a)
    # a = 2

f()

---------------

Am I global or local?
```

[i] it could be done in Python 2 under some circumstances

So far, so good. What could *possibly* go wrong with code that is so terribly simple? Now uncomment the `a=2` which, morally, should just define a local variable `a`, shadowing the global `a` in the remainder of `f`, changing nothing to the output. You get

```
UnboundLocalError: local variable 'a' referenced before assignment
```

What is going on, here? Python does not have a notion of “`a` is global at the beginning of the scope of `f`, and is shadowed by a local declaration from <*line number*> onward”.

To Python, a given variable in a given scope must be local or not, period. It makes that determination based on a simple heuristic: in that scope, is the variable ever bound? A variable can be bound by an assignment, simple or in-place or multiple or nested, a **`for`** loop, an **`import`**, an **`as`** following **`with`** or **`except`**. . .

Without `a=2`, `a` is never bound locally, so Python considers it to be global. With `a=2`, `a` is now bound in the local scope, at *some* point, therefore it is *local*, and since there is no `a` in the local scope at the time `print(a)` is executed, you get an error.

21.5.9.3 *The* `nonlocal` *keyword*

The **`global`** keyword should be used extremely rarely, in most circumstances. The **`nonlocal`** keyword is rarer still, as it only occurs in nested functions. Whereas **`global`** always gives access to the **`global`** scope, no matter where it appears, **`nonlocal`** gives a nested function access to the local scopes of the functions within which it is nested.

Let us take an example:

```
def f ():
    a,b,c,d = "abcd"
    print(locals())
    def g():
        print(a)
        nonlocal b
        b = b + "g"
        d = "D"
        def h():
            nonlocal b
            b = b + "h"
            nonlocal c
            c = c + "h"
            nonlocal d
            print(d)
        h()

    g()
    print(locals())

f()
```

```
---------------------------------------------------

{'a': 'a', 'd': 'd', 'b': 'b', 'c': 'c'}
a
D
{'a': 'a', 'd': 'd', 'b': 'bgh', 'c': 'ch',
    'g': <function f.<locals>.g at 0x7f5502a54ee0>}
```

The variable a is not global; nor is is bound locally, so it is not local; nor is it declared as **nonlocal**. However, g does have read access to it, same as for global variables.

Unlike that, however, what happens internally is a bit confusing, with locals() containing copies of nonlocal variables based on the same kind of heuristic discussed in the previous section. Therefore, the contents of locals() may depend on lines that have not been executed yet. You can verify this by modifying the beginning of g:

```
    def g():
        print(locals())
        # print(a) # comment and uncomment
        ....
```

a appears in locals() iff print(a) is uncommented. Thus locals() is actually a poor tool to understand nested scoping. Here you feel that nested scoping was added to Python long after the initial design phase; it is best not to think too much about how it is handled internally.

nonlocal b gives g access to the b of its youngest ancestor, here h; it is rebound. Note that I avoided += to exclude any suspicion that what happens is an in-place modification — strings do not allow in-place modification anyway, so such suspicion would be misplaced anyway.

d = "D" creates a g-local variable that shadows the f-local d. This will be useful to determine which one is visible from the scope of h.

In h, a new **nonlocal** b gives us access to the original b again, which we know because of the eventual 'bgh' output, but we can wonder whether we access b directly from the scope of f, or from the scope of g, where it happens to also point to the scope of f.

nonlocal c, keeping in mind that c is defined in f but not at all in g, enables us to test whether we can access the scope of f directly: and indeed we can, as attested by the 'ch' output.

print(d) is the final test of nested scoping: will we get access to the d in the scope of f or g? The **nonlocal** d statement changes nothing to that question — we do not modify d anyway. The D output shows that we get access to g's scope.

This is logical: g's version of d shadows that of f: we get access to the scope of the closest ancestor that binds the variable we are interested in.

22 Basic data types

To program is to manipulate data. Python provides various data types. We present here the most fundamental of those.

The type of an object – that is to say, the class of which it is an instance – can be obtained thanks to the `type` function:

```
>>> type(42)
<class 'int'>
>>> type(42.)
<class 'float'>
>>> type(True)
<class 'bool'>
>>> type("Python")
<class 'str'>
>>> type([1,2])
<class 'list'>
>>> type( (1,2,3) )
<class 'tuple'>
>>> type(None)
<class 'NoneType'>
>>> type(print("Hello")) # see section on NoneType
Hello
<class 'NoneType'>
```

You shouldn't do this very often, if at all, but types *can*, technically, be tested like this

```
>>> x=5
>>> type(x) == int    # avoid that
True
>>> type(x) is int    # is exactly of that class
True
>>> isinstance(x,int) # is that or subclass of that
True
```

Python subscribes to some extend to a "duck typing" philosophy. "If it walk like a duck and quacks like a duck, it's probably a duck; treat it as such". Thus rather than testing the type of something explicitly, you should just try to make it behave the way you actually want (quacking, for instance), and see if that works.

The `try .. except` construction can come in handy for this. See Sec. 23.5[p117]: "`try .. except`".

Even better, rely on polymorphism, which, as we have discussed above, is precisely a neat way to perform type dispatch.

22.1 Integers: int

22.1.1 Integer literals

Integers behave pretty much as you would expect, and can be entered as usual in bases 10, 16, and 8 using the standard notations:

```
>>> 42          # decimal
42
>>> 0xDEADBEEF  # hexadecimal
3735928559
>>> 0o18        # octal -- here the digit '8' is out of place...
SyntaxError: invalid syntax
>>> 0o15
13
```

22.1.2 Operators on integers

Integers support the following standard operators, listed by classes of increasing precedence:

+	-	addition, subtraction
*	/	multiplication, floating-point division
//	%	integral division (quotient) and modulo (remainder)
-		negation
**		exponentiation

There are also bitwise operators on integers, whose precedences are all lower, and comparison operators – see the section on Booleans – with the lowest precedence.

An important note regarding the two division operators:

/ is the floating point division. It will always return a floating point value instead of an integer, even if the result happens to be exact. Contrariwise, // returns an integer if both its operands are integers.

```
>>> 18 / 6
3.0
>>> 18 / 7
2.5714285714285716
>>> 18 // 7
2
>>> 18. // 7
2.0
>>> 18 % 7
4
>>> 18. % 7
4.0
```

`//` and `%` provide the standard quotient and remainder of euclidean division, regardless of whether the output type is `int` or `float`.

That is to say, for any $a, b \in \mathbb{Z}$, we have the identity

```
a == (a // b) * b + (a % b)
```

22.1.3 Integer arithmetic is exact

In most languages, the range of integers which can be manipulated is restricted by the fixed amount of memory allocated to it. Operations going beyond that result in an integer overflow, whereby the value exceeding the maximum wraps back to the minimum.

In Python, the native `int` type is a more sophisticated variable-size data structure, capable of dynamically allocating however much memory is needed to represent the integer values being calculated.

```
>>> 2**10000
19950631168807583848837421626835850838234968318861924454
85200894985294388302219466319199616840361945978993311294...
...68269497871413160632106863915116817743047925965709376
```

Note that this does **not** apply to floating point values.

22.2 Floating-point numbers: `float`

22.2.1 Writing floating point numbers

Floating point numbers behave as usual and can be entered with the usual `e` "exponent" notation; arithmetic operators other than `/` – which always produces `float`s – will produce a floating point number if any of their operands is floating point:

```
>>> 3.141592
3.141592
>>> 42**69  # integer
1009720183288035557387579086321483322 6
  8961863698723269942503985703768774 33
  686009543845316266007917815719968899072
>>> 42.**69 # float; note the .
1.0097201832880356e+112
>>> 42.**-69
9.903733891340244e-113
>>> 1e14
100000000000000.0
>>> float('inf') # infinity
inf
>>> float('inf') - float('inf') # Not a Number, undefined
nan
>>> float('inf') > 10**9999 # infinity is bigger than any number
True
```

22.2.2 Floating point computation is inexact

Under the hood, those are the same floating point numbers as in C or any other programming language: they follow the IEEE 754 norm, double precision (64 bits).

This means, that, unlike integers, the same caveats apply as in any other languages.

You should *almost never* test the equality two floating point numbers directly, for there might be a loss of precision due to rounding:

Consider:

```
>>> 1/3 * 6
2.0
```

So far, so good. But let's unpack that into six additions, now:

```
>>> 1/3 + 1/3 + 1/3 + 1/3 + 1/3 + 1/3
1.9999999999999998
```

Another example:

```
>>> 1.1 + 2.2 - 3.3 == 0
False
>>> 1.1 + 2.2 - 3.3
4.440892098500626e-16
>>> 1e19 + 1000 == 1e19
True
```

Instead of using == you should test whether the distance between them is smaller than some small value that still accounts for the possibility of error: $|x - y| < \varepsilon$. I often take `e-13`, but that really depends on what what kind of computation you are doing, and it is *very* difficult to properly evaluate what the margin of error should be, as that requires detailed knowledge of the IEEE 754 representation and quite a bit of maths. Some computations (e.g. chaotic systems) are immensely sensitive to errors, as even a small initial error will "snowball" out into utter nonsense.

The very smallest representable value by **`float`** is

```
>>> import sys
>>> sys.float_info.min  # smallest normalised
2.2250738585072014e-308

>>> from math import ulp
>>> ulp(0.0)            # smallest denormalised
5e-324
```

Note that the error in even the simple addition `1.1+2.2` ends up a whooping 292 *orders of magnitude* greater than `sys.float_info.min`.

It is *sometimes* acceptable to test equality directly. For instance if a value tends towards zero, it will get there eventually:

```
>>> x = 1000
>>> i = 0
>>> while x != 0:
...     x /= 2
...     i+=1

>>> print(i)
1085 # steps to get to zero
```

Note how backwards this is compared to mathematics: in $\mathbb{R}$, this loop is infinite: there is no n such that $\frac{1000}{2^n} = 0$. But with `float`, eventually you will get a value smaller that the smallest representable value, and it will approximate to 0:

```
>>> 5e-324/2
0.0
```

So, in Python as in every language, **beware** of floating point numbers. If a problem does not *require* their use, do not use them. If it *does*, think carefully of all that could go wrong.

Concrete example: do not *ever* consider representing money, that is, an exact quantity of euros and cents, using floating point numbers. Suppose you have 1€; you spend 42 cents. You have. . . not quite 58 cents:

```
>>> 1 - .42
0.5800000000000001

>>> 1 - .42 == .58
False
```

Floating-point numbers are better suited for things like temperature, pressure, age, and other types of measurements, which are intrinsically inexact anyway. Even then it is easy to produce nonsense if your computations are not continuous.

Fun fact: real numbers are so complicated that, for almost all of them, it is impossible to write a program of any finite length displaying all their digits. (To say nothing of trying to fit them in 64 measly bits). (Keyword: “computable number”).

Real numbers are convenient in maths, but in computer science, they are to be *feared*. Integers are all right, though; especially in Python, where we do not risk overflow.

Python provides the `decimal` and `fractions.Fraction` types for exact, but slower and less convenient computation, if you need that (e.g. currency,. . .).

22.3 Complex numbers: class `complex`

The addition of a j to a numerical literal produces an imaginary number, which can then be added or otherwise manipulated along with other numerical types:

```
>>> 27.1j
27.1j
>>> type(27.1j)
<class 'complex'>
>>> 2 + 3.14j
(2+3.14j)
>>> type (2+3.14j)
<class 'complex'>
```

Note that j is just another notation for $\sqrt{-1}$; in France we tend to use i for that.

Operators involving different numerical will tend to produce results of the "widest" type; numerical types, from narrowest to widest, are **`int`**, **`float`**, and **`complex`**.

We shall probably not use complex numbers in this class, but it is good to know that they exist and are very straight-forwardly supported by Python.

22.4 Strings: class `str`

22.4.1 Writing string literals

There isn't really a "character" type in Python. There are just strings of length 1. Thus the string type is both primitive and a sequence type: a string is a sequence of strings (each of length 1). Roughly, a sequence type is a type that supports element indexes – starting from 0, as usual in programming – and has a length.

Strings in Python support Unicode, and can be written either with simple or double quotes, as in shell. Unlike in shell, there is absolutely no difference in semantics between the two ways; the only thing that changes is that you need not escape single quotes in a double-quoted string and vice versa.

```
>>> "This is a 'single quote' in a double"
"This is a 'single quote' in a double"
>>> 'This is a "double quote" in a single one'
'This is a "double quote" in a single one'
```

There is another type of string syntax: triple quoted! This can use either triple double quotes or triple single quotes. Either way, in those you need not escape either single or double quotes, nor do you need to escape carriage returns (`\n`):

```
>>> """This is a triple quoted string,
where I can write carriage returns
without problem, and use 'single'
and "double" quotes as well."""
```

```
'This is a triple quoted string,\nwhere I can write carriage
  returns\nwithout problem, and use \'single\'\nand "double"
  quotes as well.'
```

The triple-quoted syntax is often use in “doctrings” – strings that decorate functions and can be used by several tools, including Python’s own `help` function, for purposes of documentation. They naturally tend to span several lines.

Like in shell, string literals can be concatenated arbitrarily, and this is even more flexible as they can be separated by whitespace:

```
>>> "This i" 's a string' """ in multiple bits"""
'This is a string in multiple bits'
```

Sometimes, you want a long string to be broken into several lines of Python, independently of any newline characters that it may contain. As in other contexts, you can do that with backslashes:

```
>>> "This is a \
long string."
'This is a long string.'
```

The drawback here is that it messes up the indentation, visually at least, since you cannot put whitespace at the beginning of the second line without making it part of the string:

```
>>> "This is a \
     long string."
'This is a      long string.'
```

In that case, you can take advantage of the literal concatenation feature, combined with the extension of logical lines within expressions:

```
>>> ("This is a "
     "long string.")
'This is a long string.'
```

Or, you can escape the returns:

```
>>> "This is a "\
    "long string."
'This is a long string.'
```

22.4.2 Raw, format, and binary string literals

Prefixing a string literal, regardless of quotation style, with `r` (or `R`) makes it a **raw string**. In raw strings, all escape sequences, like `\n`, are ignored:

```
>>> x = 21*2
>>> print("a\nb{x}c")
a
b{x}c
>>> print(r"a\nb{x}c")
a\nb{x}c
```

This is quite useful if you want to manipulate strings containing a lot of backslashes, like LaTeX code.

Prefixing with `f` (or `F`) makes it a **formatted string**, of fstring for short, which makes it possible to include the value of expressions directly in the string, in braces `{}`:

```
>>> print(f"a\nb{x}c")
a
b42c
>>> f"Hello, x={x}, x**2={x*x}"
'Hello, x=42, x**2=1764'
```

There is a lot more to say about fstrings, and we'll come back to them in Sec. 22.4.10.4[p94]: "The good stuff: formatted string literals".

Raw and formatted strings can be combined through an `rf` (or `fr`, or `Rf`, or. . .) prefix, in which case all escape codes are ignored except for braces:

```
>>> print(rf"a\nb{x}c")
a\nb42c
```

Prefixing with `b` (or `B`) makes it a **binary string**, of type `bytes` rather than `str`:

```
>>> print(b"a\nb{x}c")
b'a\nb{x}c'

>>> print(b"a\nb{x}c".hex())
610a627b787d63
```

Binary strings are a sequence of bytes, interpreted as ASCII characters for visualisation purposes, and are used for binary manipulation. They can also be made raw via an `rb` prefix:

```
>>> print(rb"a\nb{x}c")
b'a\\nb{x}c'
```

We shall not do any binary manipulation in this course, so you won't have to worry about those.

You will also find an `u` prefix, standing for "Unicode" in old Python 2 code, and it is supported but redundant in Python 3, as all strings are Unicode.

22.4.3 User interaction: the `input` procedure

You may at some point want to read a string typed by the user during an interaction with your program: the command for this is **`input`**.

```
>>> x=input("? ")
? Hello ! # Here I type 'Hello !'
>>> x
'Hello !'
```

If you want something other than a string, you need to perform a conversion. Generally speaking, the name of a class, such as **`int`**, **`float`**, or **`bool`**, acts as a *constructor*, a function that builds an element of that type, given some arguments. In particular, they are used for purposes of conversion.

```
>>> n=int(input("Number please ? "))
Number please ? 42
>>> n + 8
50
```

For some reason, a lot of students seem to *love* the **`input`** command. When I ask for, say, a *function* converting Celsius into Fahrenheit, many will write a *procedure* prompting the user for a value, and display the result of the conversion. Which is a completely different thing. (More on that in the section on `NoneType`). And this is despite my going out of my way to *not* mention **`input`**'s existence too early in my lectures, precisely to avoid this – it still manages to creep up.

Let's be clear: when I speak of the *input* of a function, I am not referring to this command, but to the function's parameters or arguments. If I don't ask you explicitly to prompt the user, don't use this.

22.4.4 The `in`/∈ operator

The **`in`** operator, which corresponds roughly to the mathematical ∈ in container types, is extended in strings: not only can it test whether a character is present in a string, it can test whether a (contiguous) substring can be extracted from the string:

```
>>> "b" in "abcd"
True
>>> "bc" in "abcd"
True
>>> "ac" in "abcd"
False
```

Note that since the empty string is a substring of any string, we have

```
>>> "" in "abcd"
True
```

```
>>> "" in ""
True
```

Which is a bit of a trap in some circumstances: for instance, testing whether the user inputs “yes” at a prompt, one can easily write

```
input() in "yY"
```

meaning “lowercase and uppercase Y mean yes”. But actually, this returns `True` if the user enters nothing, which is not the intended behaviour. Instead, you need to write something like

```
input() in list("yY")
```

22.4.5 Length and indexing in sequential types

The length of a string — or of any type for which the notion of length makes sense and has been implemented, such as lists, tuples, sets (where it means “cardinality”) etc – can be obtained thanks to the `len` function:

```
>>> len("Python")
6
```

Strings are indexed starting from 0, as usual, and with the usual notation:

```
>>> "Python"[0]
'P'
>>> "Python"[1]
'y'
```

22.4.6 Slicing and dicing, concatenation, repetition

Slightly less usual is the extension of this notation to “slices”, allowing to extract substrings:

The slice `s[start:end]` means the substring of `s` beginning at position `start` and ending *just before* position `end`. If `start` is omitted it is assumed to be 0, and if `end` is omitted, it is assumed to be the length of the string: `len(s)`. Thus `s[:]` is always the same string as `s`.

```
>>> "Python"[1:4]
'yth'
>>> "Python"[:]
'Python'
>>> "Python"[:42] # an overly large end index is truncated
'Python'
```

The customary zero-based indexing makes reasoning from the beginning of the string easy, but it takes some light arithmetic to reason backwards, from the end of the string. For those cases, Pythons allow the use of negative numbers in index accesses or slices. Thus each element is indexed by two different numbers, one positive, one negative.

However, the last element is of negative index -1, and not -0 — because distinguishing $+0$ and -0 would be somewhat tricky. To remember that, consider that, under the usual positive index, the last element of a string `s` is at position `len(s) - 1`, the second-to-last at `len(s) - 2`, etc. Therefore, this convention simply spares us the hassle of writing `len(s)` all the time.

P	**y**	**t**	**h**	**o**	**n**
0	1	2	3	4	5
-6	-5	-4	-3	-2	-1

```
>>> "Python"[-1]
'n'
>>> "Python"[-4]
't'
>>> "Python"[1:-4]
'y'
>>> "Python"[:-4]
'Py'
```

The slice syntax admits a third, optional argument `step` — bringing the full syntax to `s[start:end:step]` — that dictates the increment between the selected indices:

```
>>> "Python"[0:6:2]
'Pto'
>>> "Python"[::2]
'Pto'
>>> "Python"[::3]
'Ph'
>>> "Python"[1::2]
'yhn'
```

The `step` can be negative; in that case `end` is best understood as the index which, when met, immediately ends the construction of the slice. It is still *exclusive* in its semantics.

```
>>> "Python"[5::-2]
'nhy'
>>> "Python"[5:1:-2]
'nh' # excludes 'y' at index 1

>>> "Python"[4::-2]
'otP'
>>> "Python"[4:0:-2]
'ot' # note that end is not implicitly zero!

>>> "Python"[4:6:-2]
''
```

It should also be noted that, in slices, any `start` or `end` argument with out-of-bounds indexes is silently ignored and the start or end of the string is used instead, exactly as though the argument had been omitted.

This can result in surprising behaviours if your `step` is greater than 1, as this is not *quite* the same as saying that "out-of-bounds indexes are ignored".

```
>>> "Python"[4::-2]
'otP'
>>> "Python"[5::-2]
'nhy'
>>> "Python"[6::-2]
'nhy'  # you could legitimately expect 'otP'...
>>> "Python_"[6::-2]
'_otP' # ... just ignoring out-of-bounds index 6

>>> "Python"[7::-2]
'nhy'
>>> "Python"[-6::2]
'Pto'
>>> "Python"[-7::2]
'Pto'
```

Slices are actually objects in Python, which are generally constructed implicitly from the slice syntax, but can be named and manipulated explicitly. The constructor for that type is `slice(start, stop, step)`:

```
>>> myslice = slice(3, 10, 2)
>>> myslice
slice(3, 10, 2)  # does not do much except store those values

>>> l = list(range(15)) # list of numbers 0 <= .. < 15
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]

>>> l[myslice] # but can be passed as index!
[3, 5, 7, 9]
>>> l[3:10:2]
[3, 5, 7, 9]
```

A theorem of some interest is that, for any string `s` and any integer `n`, even negative or out of bounds, it holds that `s[:n] + s[n:] == s`.

This is useful when "modifying" strings, for instance... which you cannot do, strictly speaking. Unlike in most languages you may be used to, strings in Python are immutable. That is to say, once a string is created, you can never modify it in-place. What you do instead is create a new version of the string, incorporating the modification you want. The slice notation makes this process relatively painless. There are advantages to strings being immutable, as we shall see later when dealing with sets and dictionaries.

As an aside, in purely functional programming languages such as Haskell, *everything* is immutable. While that may seem strange at first, this has great benefits for program proof or rewriting, as you never have to worry about any value being sneakily modified as a side effect of some other piece of code.

Back to Python strings, you can concatenate them with `+`, which is not unusual, but you can also multiply them with integers, concatenating them with themselves repeatedly:

```
>>> "Python" + " rules!"
'Python rules!'
>>> "Python" * 6
'PythonPythonPythonPythonPythonPython'
```

Thus, to “replace” a character in a string, here capitalising the ’t’ in “Python” for no reason whatsoever, we can write:

```
>>> s="Python"
>>> s[:2] + 'T' + s[3:]
'PyThon'
```

Slices also have an optional third argument indicating the increment by which to select characters; it is of course 1 by default.

```
>>> "Python"[::2]
'Pto'
>>> "Python"[::-1]
'nohtyP'
>>> "Python"[::-2]
'nhy'
```

The behaviour of slices mirrors that of the `range` function. See the section on `for .. range`.

Note that `len`, slices, `+`, `*`, and more, work not only with strings, but with any “sequence” type in Python, such as lists and tuples. Actually, `len` works on any “collection” type, that is, anything containing a number of other things; that includes strings, lists, tuples, but also sets and dictionaries.

22.4.7 Python strings use Unicode

Python strings are coded in Unicode (utf8), and as such can contain all kinds of special characters, while being compatible with the ASCII encoding (see table) and the larger ISO-8859-1/Latin1 encodings. You can write the Unicode symbols directly, by whatever means are available (honestly I often just copy and paste from a web page or something), or you can enter the Unicode code point (or ordinal) for the symbol you want, if you know it, after the \u escape:

```
>>> 'Σ is the sum symbol'
'Σ is the sum symbol'
```

```
>>> "\u2211 is the sum symbol"
'Σ is the sum symbol'
```

The functions **chr** and **ord** effect the conversion from Unicode code point to character (“string of length 1”) and vice versa:

```
>>> chr(65)
'A'
>>> ord('A')
65
>>> ord('a')
97
>>> ord('Python')
TypeError: ord() expected a character, but string of length 6 found
```

Python also supports Unicode in identifiers, with some restrictions. You can only use symbols that are classified as letters (and numbers, of course). Thus Σ (U+03A3), the greek letter, will be acceptable, but ∑ (U+2211), the mathematical operator, will not.

```
>>> Σ = sum
>>> Σ(n for n in range(10))
45

>>> ∑
SyntaxError: invalid character '∑' (U+2211)
>>> ∫ = 2
SyntaxError: invalid character '∫' (U+222B)
```

Having Greek letters is great for implementing maths, although it’s a shame mathematical operators are not allowed.

Lest you think that this restriction was a necessary sacrifice for the sake of clarity, note that, even with letters only, Unicode can be a source of great joy:

```
>>> ord("A")
65
>>> ord("Α")
913
>>> "A" == "Α"
False
```

“What on Earth is going on here?”, you ask, bewildered... well, you see, one A is the bog-standard ordinary latin letter A, and the other Α is the capital Greek α. They just *happen* to be visually identical in most fonts, but they are entirely different symbols from different alphabets. Make sure to put Unicode capital αs and βs (Β) and so on everywhere in your code for endless laughs.

BITS			b5	0	1	0	1	0	1	0	1
				CONTROL		SYMBOLS NUMBERS		UPPER CASE		LOWER CASE	
b4	b3	b2	b1								
0	0	0	0	0 NUL 0x0	16 DLE 0x10	32 SP 0x20	48 0 0x30	64 @ 0x40	80 P 0x50	96 ‘ 0x60	112 p 0x70
0	0	0	1	1 SOH 0x1	17 DC1 0x11	33 ! 0x21	49 1 0x31	65 A 0x41	81 Q 0x51	97 a 0x61	113 q 0x71
0	0	1	0	2 STX 0x2	18 DC2 0x12	34 ” 0x22	50 2 0x32	66 B 0x42	82 R 0x52	98 b 0x62	114 r 0x72
0	0	1	1	3 ETX 0x3	19 DC3 0x13	35 # 0x23	51 3 0x33	67 C 0x43	83 S 0x53	99 c 0x63	115 s 0x73
0	1	0	0	4 EOT 0x4	20 DC4 0x14	36 $ 0x24	52 4 0x34	68 D 0x44	84 T 0x54	100 d 0x64	116 t 0x74
0	1	0	1	5 ENQ 0x5	21 NAK 0x15	37 % 0x25	53 5 0x35	69 E 0x45	85 U 0x55	101 e 0x65	117 u 0x75
0	1	1	0	6 ACK 0x6	22 SYN 0x16	38 & 0x26	54 6 0x36	70 F 0x46	86 V 0x56	102 f 0x66	118 v 0x76
0	1	1	1	7 BEL 0x7	23 ETB 0x17	39 ’ 0x27	55 7 0x37	71 G 0x47	87 W 0x57	103 g 0x67	119 w 0x77
1	0	0	0	8 BS 0x8	24 CAN 0x18	40 (0x28	56 8 0x38	72 H 0x48	88 X 0x58	104 h 0x68	120 x 0x78
1	0	0	1	9 HT 0x9	25 EM 0x19	41) 0x29	57 9 0x39	73 I 0x49	89 Y 0x59	105 i 0x69	121 y 0x79
1	0	1	0	10 LF 0xA	26 SUB 0x1A	42 * 0x2A	58 : 0x3A	74 J 0x4A	90 Z 0x5A	106 j 0x6A	122 z 0x7A
1	0	1	1	11 VT 0xB	27 ESC 0x1B	43 + 0x2B	59 ; 0x3B	75 K 0x4B	91 [0x5B	107 k 0x6B	123 { 0x7B
1	1	0	0	12 FF 0xC	28 FS 0x1C	44 , 0x2C	60 < 0x3C	76 L 0x4C	92 \ 0x5C	108 l 0x6C	124 | 0x7C
1	1	0	1	13 CR 0xD	29 GS 0x1D	45 – 0x2D	61 = 0x3D	77 M 0x4D	93] 0x5D	109 m 0x6D	125 } 0x7D
1	1	1	0	14 SO 0xE	30 RS 0x1E	46 . 0x2E	62 > 0x3E	78 N 0x4E	94 ˆ 0x5E	110 n 0x6E	126 ˜ 0x7E
1	1	1	1	15 SI 0xF	31 US 0x1F	47 / 0x2F	63 ? 0x3F	79 O 0x4F	95 _ 0x5F	111 o 0x6F	127 DEL 0x7F

22.4.8 Unicode defines the order on characters

Characters are ordered according to their code points. That is to say, given two characters c and d, we have

$$\mathtt{c} < \mathtt{d} \iff \mathtt{ord(c)} < \mathtt{ord(d)}\ .$$

22.4.9 The order on characters defines the lexicographical order on strings

This total order on characters is extended to a total order on strings, by deriving the lexicographical order: let $u, v \in \Sigma^*$ be strings[j], then

$$u < v \iff u \text{ is prefix of } v \ \lor\ \exists x, y, z \in \Sigma^*,\ a, b \in \Sigma : \begin{cases} u = xay \\ v = xbz \\ a < b. \end{cases}$$

The following are all `True`:

```
>>> 'aa' < 'aaa'
>>> 'aaa' < 'ab'
>>> 'Etudiant' < 'Prof'
>>> 'Prof' < 'etudiant'    # case matters!
```

As we shall see later on, other sequential collection types follow the same method to lift an order on their elements into an order on homogeneous collections.

22.4.10 Formatting strings

A common task with strings is to format data. There are several ways of going about that in Python, depending on the complexity of the task.

Only **print** and the basics of fstrings will be tested in the exam.

22.4.10.1 *The **print** procedure*

At its most simple, and if you just want to display the result, you can simply use the variadic **print** procedure.

```
>>> x = 1 ; y = None
>>> print("And then", "there was", x, ",and then there were", y,".")
And then there was 1 ,and then there were None .
```

print adds spaces between its arguments, and a carriage return at the end. This is controlled by optional arguments `sep` and `end`, respectively:

[j] Here I am using notations that we shall see this semester in formal languages theory. It should still be pretty intuitive: Σ is the set of characters, and Σ^* the set of strings. xay means the concatenation of x, a, and y.

```
>>> print(x,y,y,x,sep='..',end='!!')
1..None..None..1!!
```

What if you want non only to display this, but to get a new string? In that case, again for basic tasks, you can concatenate what you need, but note that you must convert non-string elements manually:

```
>>> "And then " + " there was " + x + ", and then there were " + y + "."
TypeError: must be str, not int
>>> "And then" + " there was " + str(x) + ", and then there were" + str(y) + "."
'And then  there was 1, and then there were None.'
```

Is there something more convenient and powerful, along the lines of `printf` and `sprintf`? Yes, there are ways, three of them to be specific, thought none are called `printf`.

22.4.10.2 *The % operator*

In chronological order of introduction to Python, they are the `%` operator, the `format` string method, and fstrings, which share syntax with the `format` method and can be see as an improved way to access it. Not that it matters much, but fstrings are also the most efficient of the bunch.

We are going to touch briefly on each of those, with more emphasis on fstrings, which are the recommended way to proceed.

`%` is a binary operator taking a string with special formatting syntax inside, very similar to `printf` syntax, and a tuple of values, and returning the corresponding formatted string:

```
>>> "Like %.2f a %d printf %s." % (3.14159265359, 42, "dream")
'Like 3.14 a 42 printf dream.'
```

Since it is a perfectly normal operator producing a perfectly normal string, nothing prevents you from passing all that as argument to, say, `print`, essentially making `print` into a `printf` equivalent:

```
>>> print("Like %.2f a %d printf %s." % (3.14159265359, 42, "dream"))
Like 3.14 a 42 printf dream.
```

22.4.10.3 *The `format` method*

Back in the good-ol'-days of Python 2, that was the way things were done. Then `format` was introduced in Python 2.6. It uses a more streamlined method call, extensible on new types via the `__format__` method, and has a new, original specification syntax, based on curly braces, that breaks with some of the conventions of `printf`, but retains some of them:

```
>>> "Like {:.2f} a {:d} printf {:s}.".format(3.14159265359, 42, "dream")
'Like 3.14 a 42 printf dream.'
```

For now, we don't see any improvement over printf that would justify the change of syntax. But what if you want to repeat a given argument? Sure, you could duplicate it in the call to format, but there is more elegant. Before the semicolon :, you can actually put the index of the argument you are referring to. Absent this indication, they will be taken in order; but if you wish, nothing prevents you from reusing an argument, perhaps with different formatting:

```
>>> "Like {0:.2f} a {1:d} printf {2:s}.".format(3.14159265359, 42, "dream")
'Like 3.14 a 42 printf dream.'
>>> "Like {0:.2f} a {1:d} printf {2:s} {0:.3f}.".format(
      3.14159265359, 42, "dream")
'Like 3.14 a 42 printf dream 3.142.'
```

Digression: you can actually do something like that with printf in C, with a $-based syntax

```
printf("%1$s%1$s\n", "hello");
```

but that is a Posix extension, that works only on Unix systems, and is not included in the C99 standard. You can also used named arguments instead of positional indices (and for course the formatting part, such as :.2f, is optional), which can result in much more readable patterns:

```
>>> "{x:d}{y}{y}{x}".format(x=0,y=1)
'0110'
```

This is about as good as it gets for format, and while it is a vast improvement over printf style, there is still a level of redundancy and clunkiness.

22.4.10.4 *The good stuff: formatted string literals*

Then, with Python 3.6, formatted string literals, or *fstrings*, entered the fray. An fstring is special string literal of the form f"...", containing formatting instructions in a similar syntax to format, which is evaluated at runtime and converted into a normal string. It can reference any variable in its current scope.

```
>>> x = 'Python'
>>> v = 3.6
>>> f"Introduced in {x} version {v}."
'Introduced in Python version 3.6.'
```

It would be difficult to imagine something simpler and cleaner than that. But that is not the end of it: Since fstrings are evaluated at runtime, you can put (almost) any valid python expression inside them:

```
>>> f"Before {v} was {v-0.1}, and after was {v+0.1}."
'Before 3.6 was 3.5, and after was 3.7.'
```

fstrings concatenate with other string literals, including other fstrings, without problems,

```
>>> f"fstring {v} " "regular string"
'fstring 3.6 regular string'
```

Thus you can use the same techniques as usual to span an fstring over multiple logical lines. Also note that all the different syntaxes for string literals apply as well for fstrings; single quotes:

```
>>> f'Introduced in {x} version {v}.'
'Introduced in Python version 3.6.'
```

and triple quotes (whether single or double):

```
>>> f"""Introduced in {x}
version {v}."""
'Introduced in Python\nversion 3.6.'
```

This is even more useful in fstrings that in regular strings – where it's just a nice but inessential convenience – since you may want to execute code involving string literals inside fstrings, and you cannot escape special symbols there, as \ cannot appear inside the formatting curly braces. By using a style of quote for fstrings and another one for the string literals inside it, things go smoothly:

```
>>> d = { 'key' : 42 }
  # this is a dictionary; a key:value association
  # see the relevant section
>>> f"Value for 'key' is {d['key']}."
"Value for 'key' is 42."
>>> f'Value for \'key\' is {d['key']}.' # string eds at ...{d['
SyntaxError: invalid syntax
>>> f'Value for \'key\' is {d[\'key\']}.'
SyntaxError: f-string expression part cannot include a backslash
```

Note that you can use backslashes just as usual in the string part of the fstring, just not in the formatting/expression parts between braces. If that restriction becomes cumbersome, do not hesitate to evaluate the offending expression beforehand, put its value in a variable, and reference *that* in the fstring.

Do not hesitate to use triple quotes even if you don't need line returns, because then you need not worry about escaping any type of quote:

```
>>> f"""Value for 'key' or "key" is {d['key']} or {d["key"]}."""
'Value for \'key\' or "key" is 42 or 42.'
```

Given the special nature of braces in fstrings, they must be escaped if you want these characters to appear in the string, but you can't do that with a backslash: you need to use double braces `{{..}}` [k]:

```
>>> f"{3.1415:.2f}"     # a formated expression
'3.14'
>>> f"{{3.1415:.2f}}"   # now it's just a literal string
'{3.1415:.2f}'
>>> f"{{{3.1415:.2f}}}" # combine the two.
'{3.14}'
```

Finally, note that, as of the current version of Python, fstrings have much higher performance than other ways of formatting strings – not that this is usually a bottleneck, but when it's *both* the most elegant and the fastest way to do this, why use anything else?

It is highly recommended to get used to them. For a complete reference on their syntax, see the official documentation:

https://docs.python.org/3/reference/lexical_analysis.html#f-strings

https://docs.python.org/3/library/string.html#formatspec

As you can see, there is quite a lot going on in the formatting syntax. I'll just insist on a very common thing to do with string formatting: aligning things in monospace fonts. The relevant formating instructions are:

<	left-align text in the field
^	center text in the field
>	right-align text in the field

In those examples `x` still contains `'Python'`:

```
>>> f"[{x:<20}]"
'[Python              ]'
>>> f"[{x:>20}]"
'[              Python]'
>>> f"[{x:^20}]"
'[       Python       ]'
>>> f"[{x:*^20}]"   # fill with any character, here *
'[*******Python*******]'
>>> f"[{x:#^20}]"
'[#######Python#######]'
>>> f"[{x:\^20}]"   # it looks bigger but that's because \ is escaped
                    # in the string literal below
'[\\\\\\\\\\\\\\Python\\\\\\\\\\\\\\]'
>>> print(f"[{x:\^20}]") # when printed it's the right size
[\\\\\\\Python\\\\\\\]
```

[k] Speaking from personal experience, this makes using fstrings to generate LaTeX code, which has an extremely high concentration of braces, a small nightmare.

Of course for numerical values the usual specifiers for the decimal type suffice to produce quite legible tables:

```
>>> for i in range(0,10+1,2):
      print(f"{i:6d} {i**2:6d} {i**3:6d} {i**4:6d}")

     0      0      0      0
     2      4      8     16
     4     16     64    256
     6     36    216   1296
     8     64    512   4096
    10    100   1000  10000
```

For debugging purposes, it is often useful to print the values of variables; to easily keep track of which value corresponds to which variable, you can use the “=” formatter:

```
>>> x,y,z = "abc"
>>> f"{z=} {y=} {x=}"
"z=’c’ y=’b’ x=’a’"
```

Note that the formatting instructions after the : can use the value of variables. For this, it suffices to surround the variable name with braces:

```
lines= [ "a", "bbbbbbb", "ccc"]
maxl = max(len(l) for l in lines)
for l in lines:
    print(f"|{l:^{maxl}}|")
---------------------------------
|   a   |
|bbbbbbb|
|  ccc  |
```

Furthermore, fstrings can be nested; the usual rules regarding quote escapes apply:

```
>>> db = dict()
>>> y = 2021
>>> db[f"{y}_budget"] = 42e5

>>> print(f"{y}_budget : {db[f’{y}_budget’]}")
2021_budget : 4200000.0
```

Python 3.12 makes nesting more flexible[1].

Finally, let us illustrate what I meant when I said that that types could implement the __format__ method to roll their own formatting specifications:

```
>>> import datetime
>>> today = datetime.datetime.today()
```

[1] https://docs.python.org/3.12/whatsnew/3.12.html#pep-701-syntactic-formalization-of-f-strings

```
>>> today
datetime.datetime(2019, 8, 20, 11, 53, 33, 122991)
>>> print(f"{today:%B %d, %Y}")
August 20, 2019
```

Here, `%B %d, %Y` is a special formatting specification that works only on objects the `datetime` class, where its meaning is defined. It works because at runtime, it is known that `today` is of this type.

22.5 Nihilism: `NoneType`: expression versus statement

It is quite important to distinguish expressions and instructions/statements[m]. Year after year I belabour this point, and year after year students forget all about it by the time the exam comes. Insert sad emoji here.

An *expression*, like `1+2`, has a data type (here, integer), and a **value**.

Expressions can be nested: e.g. `3*(1+2)`. Fundamentally, an expression is either a base value (`1`, `2`,. . .) or a combination of several sub-expressions under an operator or a function).

Arithmetical and logical expressions are two kinds that you are familiar with, but there are endlessly many variants.

A *statement* (FR: instruction), on the other hand, represents an **order** given to the machine:

```
x = 42      # create a memory space encoding 42, and bind it to x!
print(42)   # display that on the standard output!
pass        # do nothing!
```

A statement does not, morally, have any value attached to it, in the way that `2+2` does. There is a fundamental difference of concept between the *value* 42, and the execution of the order "take a bucket of paint and write this value, 42, on the wall!". That 42 be written on a wall or on the standard output is immaterial to this distinction.

Because statements do not carry an intrinsic value, there is generally no point in allowing them to be combined or nested. The only thing you can do with them is put one after the other.

In Python, some statements, like `return 42` cannot be combined or nested in any way, under pain of `SyntaxError`. Others, like `print(42)`, can; sort of.

```
>>> print(print(42), print(69))
42
69
None None
```

[m] Technically, in Python an expression is also a particular form of statement; I am more interested in the semantics, here.

What happens here? **`print`** is a function. Functions typically *return* something, some value, but this is not the case here; **`print`** *does* something. In that case we prefer the word *procedure* to *function*.

It is important to distinguish the cases where a function returns something from those when it does not, so as to catch errors. The way Python does that – and many other languages as well – is to have a "placeholder" type for "this has no type", "absence of data". A value to return when you want to say "I returned nothing". Thus procedures are naturally handled in exactly the same way as every other functions, no special cases needed.

In Python, this special value is called `None`, and it is the only value belonging to the type `NoneType`. Any function that does not return anything implicitly ends with `return None`.

And now we can understand what happens in `print(print(42), print(69))`: the outermost **`print`** begins by evaluating its arguments, in order. `print(42)` evaluates 42, displays it, and returns `None`, which becomes the first argument value of the outermost **`print`**. Then `print(69)` evaluates 69, displays it, and returns `None`. Finally, `print(None,None)` executes, displaying the string representation of both `None`s, which happens to be `"None"`.

22.6 Booleans: `bool`

Booleans are a very simple data type, containing only two values: `True`, and `False`.

22.6.1 Comparison operators

A function that returns a Boolean is called a *predicate*.

You usually get Booleans as a result of a comparison operator. In Python, they are the following:

Mathematics	Python
$=$	`==`
$\neq$	`!=`
$<$	`<`
$>$	`>`
$\leqslant$	`<=`
$\geqslant$	`>=`

All comparison operators have the same precedence, unlike in C, and that precedence is lower than that of any arithmetic operation (as in pretty much all any language). They compare the *values* of their operands, not whether they are the same memory object — the operator for that is **`is`**.

An interesting and very unusual thing about the Python incarnation of the comparison operators is that they can be chained as in maths: for instance, I can write

```
0 < x <= y < 1
```

to mean

```
0 < x and x <= y and y < 1
```

The semantics is only exactly the same in the absence of side effects, as we shall see in Sec. 22.6.5[p104]: "The semantics of **and** and **or**, & implicit Boolean conversion".

22.6.2 in and is

in and **is**, as well as their negations **not in** and **is not**, can be considered comparison operators and have the same precedence as them.

in (and its negation **not in**), corresponding to ∈ in maths, tests whether an element appears in a collection – so long as the collection type implements the relevant methods:

```
>>> 2 in {1,3,4}
False
```

Recall that we saw that, in the case of strings, it also tests for substrings:

```
>>> "bc" in "abc"
True
```

That does not apply to other collection types, such as tuples, lists, etc, since, unlike strings, they can be nested:

```
>>> (2,3) in (1,2,3)
False
>>> (2,3) in (1,(2,3))
True
```

Then there is also **is** (and **is not**), testing if two objects point to the same location in memory – much more rarely used. Examples of common uses: `x is None`, `type(x) is int` (though `isinstance(x,int)` is often to be preferred).

The operands of comparison operators need not be of the same type. The rule is that, when comparing two objects of different types:

- If they are both a numerical type, they are converted to a common type and compared:

  ```
  >>> 42 == 42.
  True
  ```

  It should be noted that, technically, **bool** is a numerical type – in fact it is a subclass of **int**. Thus we have:

```
>>> 0 == False and 1 == True
True
>>> 2 == True or 2 == False
False
>>> 2 * True + True
3
```

This can result in some strangeness in the case of collections that rely on hashes of the values of its items, such as sets and dictionaries:

```
>>> {True, 1, 0, False}
{0, True}
```

- Otherwise, they are automatically unequal.

```
>>> "42" == 42
False
```

- Unless the appropriate method has been defined in one of the operands, they cannot be ordered:

```
>>> "42" < 42
TypeError: '<' not supported between instances of 'str' and 'int'
```

Note that since they are comparison operators, they can be chained as well, and with any other comparison operators.

Whereas in maths, almost all instances of comparison chaining are ordered, e.g.

$$e_1 < e_2 \leqslant e3 = x \leqslant \infty \,,$$

and when you see such expressions, it's quite intuitively obvious what they mean.

Python has no restrictions on which operators can be chained. Consider the expression `None==None is None`. What does it mean? If you forget that `is` can be chained, nothing makes sense:

```
>>> None==None is None
True
>>> (None==None) is None
False
>>> None==(None is None)
False
```

Both == and `is` are, nominally, binary operators, but neither of the two possible interpretations of that expression match what you obtain.

What it really means is:

```
>>> None==None and None is None
True
```

The same goes for `in`:

```
>>> None==None is None in {None}
True
```

22.6.3 Boolean operators

Boolean expressions involve Boolean operators in the same way that arithmetical expressions involve arithmetical operators. The Boolean operators are as follows:

Maths / Logic	Python
$\vee$	`or`
$\wedge$	`and`
$\neg$	`not`
$\iff$	`==`

The last one is actually the standard comparison operator, not really a Boolean operator, that just happens to work because logical equivalence is simply equality of Boolean values.

The truth tables of these operators are as usual.

All three true Boolean operators have lower precedence than any comparison operator, and are here listed in order of increasing precedence. The only thing in Python with lower precedence than `or` is `lambda`.

Precedence warning: if you find yourself writing

```
>>> reply='y'
>>> reply == 'y' or 'yes'
True
```

you have just introduced a large bug in your program: the test will always pass:

```
>>> reply='n'
>>> reply == 'y' or 'yes'
'yes' # non-empty, therefore True
```

The precedence rules evaluate that as `(reply == 'y') or 'yes'`, which is `True`.

Either write

```
reply == 'y' or reply == 'yes'
```

or – less classical but more convenient – use the **in** operator and a collection, preferably a tuple:

```
reply in ('y', 'yes')
```

22.6.4 A fantastic fear of Booleans

As a point of syntax or style, there is a anti-pattern I see often – quasi systematically, in fact, in some form or other – and dislike considerably: writing predicates with unnecessary **if** statements and/or ==True tests. Here is an example of what I mean:

```
def p(a,b):
    test =    a == 1 and b == 2
      # or any computation that yields True or False
    if test == True:
        return True
    else:
        return False
```

There are two things wrong with this picture; let's start with the test. It is a Boolean expression. Its value is *already* True or False. For any Boolean b, it holds that b is true if and only of b==True is true. It does not get extra truer with extra tests ==True. If you go down that road, why stop there? Why not go full retard, as they say, and write

```
>>> (b == True) == True
>>> ((b == True) == True) == True
>>> # you get the idea
```

just for those tasty incremental bits of truthiness? Just don't, please. There is *never* a good reason to write **if** test == True instead of **if** test. The same goes for **return** test==True versus **return** test.

Now on to the second wrong thing:

```
if test :
    return True
else:
    return False
```

What are we doing, here? If test is True we return True. If test is False we return False. test being a Boolean, we have covered all the values it can ever take. In other words, we return the value of test, no matter what. Let's do that: **return** test.

All in all, the predicate above should be written:

```
def p(a,b):
    return a == 1 and b == 2
```

In exams, I have a policy of taking away points whenever I see that.

22.6.5 The semantics of `and` and `or`, & implicit Boolean conversion

Morally, Boolean operators act on two Booleans, and yield a Boolean. `False or True` is a well-defined Boolean expression, of value `True`. Something like `False or 42` should raise an error along the lines of “hey, I expected a `bool`, and I got an `int`. What gives?”. Right? Let’s ask Python:

```
>>> False or 42
42
>>> True and 42
42
>>> 42 or True
42
>>> True or 42
True
>>> not 42
False
>>> not not 42
True
>>> False or 0
0
>>> [] and 42
[]
```

Oh. Okay. “*C’est pas faux.*” Let’s stay calm and figure out what’s going on.

In Boolean contexts, that is to say, in a `if` of `while` test, or as operand to a Boolean operator, everything that is not already a Boolean is forcibly converted to one, along the following rules:

Everything becomes `True` unless it is on the following list, which intuitively captures the “empty” object of each type:

- `False` itself, of course.
- `None`, because you can’t get emptier than that.
- Numerical values equal to zero: `0`, and `0.0`
- the empty string `''`, the empty list `[]`, the empty tuple `()`, the empty set `set()`, the empty dictionary `{}`, and any empty container type, generally.
- anything that says so in its `__bool__` method. If you write your own type, give some thought to what conversion makes sense.

That actually makes a lot of sense: implicitly, testing an object becomes testing non emptiness.

But wait, that explains `True or 42` being true, because that is equivalent to `True or True`, and that explains `not 42` being false, because that boils down to `not True`, but why does `False or 42` yield `42` instead of `True`?

First note that this is consistent, as in a Boolean context, 42 will be converted to `True` anyway.

The reason for that lies in the precise behaviour of **`or`** and **`and`**, which actually differs in subtle but important ways from that of their mathematical counterparts.

In mathematics, both $\wedge$ and $\vee$ are commutative: the statements

$$a \neq 0 \quad \wedge \quad \frac{1}{a} > 10$$

and

$$\frac{1}{a} > 10 \quad \wedge \quad a \neq 0$$

are both well-defined and strictly equivalent for all values of $a \in \mathbb{R}$. Let us test that in Python for $a = 0$:

```
>>> a=0
>>> a != 0 and 1/a > 10
False
>>> 1/a > 10 and a != 0
ZeroDivisionError: division by zero
```

Why this behaviour? In mathematics, all operands are "processed" simultaneously. In Python – as in most languages – the operands are processed in order, left to right.

Furthermore, not all operands need be evaluated: In mathematics, given the expression $\top \vee x$, whatever the Boolean x, the result is $\top$. Likewise, given $\bot \wedge x$, the result is $\bot$.

Python follows that convention, and never evaluates the second argument if the first suffices. Thus, in the first version, `1/a > 10` is never evaluated, and can thus yield no error. In the second version, it is evaluated first, and the second operand cannot "protect" it.

Finally, whenever Python examines the value that concludes the evaluation of a binary Boolean operator, it returns that value itself, and not its Boolean conversion. Thus, for instance, `False or 42` first looks at `False`, which is not enough to conclude in an **`or`**; then it looks at `42`, and returns that. In fact, if it gets to the second operand, it does not even *try* to convert it to **`bool`**, it just returns it as is. There is no need, as $\bot \vee x$ and $\top \wedge x$ are both equivalent to x.

In conclusion, Python's binary Boolean operators are only commutative in the absence of exceptions (and side effects).

Another way of expressing the behaviour of those operators is as follows, with $\approx$ denoting implicit Boolean conversion:

$$x \;\mathbf{or}\; y \quad = \quad \begin{cases} x & \text{if } x \approx \texttt{True} \\ y & \text{otherwise} \end{cases}$$

and

$$x \;\texttt{and}\; y \;=\; \begin{cases} x & \text{if } x \approx \texttt{False} \\ y & \text{otherwise} \end{cases}$$

Again, neither operator returns the Boolean conversion of their operands, but instead the operands themselves; in both cases, y is evaluated only in the case where it must be returned.

A pattern that you may sometimes see that makes use of this behaviour looks like this:

```
default = 42
user = input('Enter a number, or press Enter for the default: ')
nb  = user or default
print('The number is', nb, user)
```

It has the behaviour you would expect. Note that it works if the user enters `0`, because `input` yields a string, and `'0'` is non-empty. It is not necessarily recommended to write in this way.

Let us come back to chained comparison operators. I said earlier that they were translated into an `and` statement, but with a slight difference in semantics in the presence of side effects. Let us clarify and demonstrate that fact.

Let us define a test function with no purpose except having a side effect — printing the value it returns:

```
def x(x): print(x,end=" "); return x
```

First, let us see a similarity:

```
>>> x(10) <= x(2) <= x(3)
10 2 False

>>> x(10) <= x(2) and x(2) <= x(3)
10 2 False
```

Line `and`, the chain comparison stops as soon as it is broken; we see this because `x(3)` is never executed.

And now, the difference:

```
>>> x(1) <= x(2) <= x(3)
1 2 3 True

>>> x(1) <= x(2) and x(2) <= x(3)
1 2 2 3 True
```

Each operand is only evaluated once in a chain comparison.

22.6.6 Assertions: cheap unit testing and preconditions enforcing

Boolean expressions serve of course as tests in the usual control flow structures, but another cool thing you can do with them is **assert**ions.

An assertion is a statement about the state of your program that must hold if the state is correct. It can be used as a mechanism for defensive programming – ensuring that the precondition of a function are met, e.g. this input representing an hour is between 0 and 24, it does not make sense otherwise – or for testing – my function must return *this* on *that input*, otherwise it is incorrect.

The syntaxes for the **assert** instruction are the following:

```
assert <condition>
assert <condition> , <optional error data or message>
```

Its effect is to test the condition, and do nothing if it is `True`. Indeed, the condition states the normal, expected behaviour of the program. If the condition is false, on the other hand, something is very wrong with the state of the program, and an exception is raised, interrupting the program.

It is very good practice to use **assert**s in your code whenever convenient. Like unit tests, they can catch bugs at their source and prevent regressions.

Of course, the code within the assertions may have a negative impact on performance, in which case they should be deactivated in a production environment. Assertions are only executed in debug mode, which is on by default, and can be deactivated by passing the `-O` (capital O, for Optimisation) flag to Python.

The behaviour of **assert** is thus equivalent to

```
if __debug__:  # you can't assign to this; use -O
    if not condition: raise AssertionError #(optional_message)
```

Note that it is a bad idea to try to catch an assertion error, and have the behaviour of the program depend on that. Assertions only trigger if the logic of the program is violated; they are fundamentally *outside* of the program's logic. They are a safeguard against incoherent behaviour, not a control flow mechanism. That is why they can be deactivated or removed at will.

For instance, you should probably not use an assertion to test the sanity of a user's **input**; dealing intelligently with your users' inability to formulate correct inputs is very much in your program's bailiwick; "user not very smart" is not an error condition, it's a Tuesday.

You *should* use an assertion to ascertain that programmers (including you) use your function correctly. Dealing with *programmers'* inability to read and understand a function's documentation is *not* your responsibility. If your function only makes sense when `n` is positive, and you have documented that fact, then by all means throw in an `assert n>=0` at the beginning of it. If the function is ever called with n negative, it's a programmer error, everything should stop right now so the programmer notices and fixes the error.

The optional message is not all that useful, as the traceback displayed when an exception is raised reproduces the assertion's line:

```
>>> n = 52
>>> assert n == 42
Traceback (most recent call last):
  File "<pyshell#14>", line 1, in <module>
    assert n == 42
AssertionError
```

I often see error messages that paraphrase the condition:

```
>>> assert n == 42, "n should be 42"
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    assert n == 42, "n should be 42"
```

As you can see it is completely redundant; I can read the condition already. It might serve some use in that capacity if I *caught* the **AssertionError**, and did not have the traceback, but as discussed above that goes against the grain of what an assertion *means*.

A better use for it that I can see is to provide the value that failed to meet the condition, if applicable:

```
>>> assert n == 42, n
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
    assert n == 42, n
AssertionError: 52
```

Here I can see that the actual value of n that triggered the assertion was 52, which is non-redundant and actually helpful for debugging purposes.

Let's take a silly example to illustrate the kind of use of **assert** I would like to see in your code. Suppose I want to implement a function that computes the square of a number (very difficult, that. . .).

I propose the following implementation:

```
def square(n): return n**2
```

To illustrate the use of assertions to test preconditions, I'll add the requirement that my function must only be called with nonnegative values. This condition is entirely artificial here, but again, it's just to showcase an assertion. I would then write:

```
def square(n):
    assert n >= 0
    return n**2
```

Once the function is written, I want to make sure it is correct. I know how to compute a square *independently of my implementation*, so I do that for a few values – it's good to test edge cases, so for integers the first few and a large one generally suffice – and write

```
assert square(0)   ==    0
assert square(1)   ==    1
assert square(2)   ==    4
assert square(10)  ==  100
```

Nothing explodes when I run this code, so I am that much more confident that my function is correct.

Do not remove or comment out your assertions! Again, they can be deactivated globally for performance reasons if need be, and they protect against code regressions. Very often I see students removing the assertions after having (allegedly) tested their function. I cannot fathom the reasoning under that act — nor can they account for it, when I ask. And of course, *of course!*, two times out of three, when the assertions are put back in, it turns out that the code was incorrect.

An even better way to proceed would be Test-Driven Development (TDD). The idea is to write the tests *first*, when you have decided what your function *should* do on paper, and only then to implement the function, until all the tests pass.

An interesting case is when you have two implementations of the same function. It's very common to have an obvious, clearly correct but inefficient way to implement something, which you later replace by a more finicky optimised version, perhaps operating on completely different principles. Rather than getting rid of the old, correct but slow implementation, you can use it in tests to check that the new implementation is equivalent to the old – and prevent regressions while you continue to optimise the new version.

With two independent implementations, you can use a loop to test equivalence on a large number of values: Here I would write:

```
for n in range(10): assert square(n) == n * n
```

This passes as well, so the **-based and *-based implementations seem equivalent.

There is a drawback to the form above: if debug mode is deactivated, this does not disappear completely, but becomes

```
for n in range(10): pass
```

Which is still a loop for nothing. Thus the following form should be preferred:

```
assert all( square(n) == n*n  for n in range(10) )
```

We shall study that syntax later on, in Sec. 24.5.1[p154]: "Comprehensions for every type; first contact with generators". Specifically, see Sec. 24.5.3.4[p159]: "Reductions", especially the part about any and all.

In this syntax, it is harder to see for which value the assertion fails (if fail it does). We can use the := operator, which assigns a value to a variable in the course of evaluating an expression, to store the last values computed before the assertion failed, and return them with the AssertionError:

```
def square(n): return 8 if n==3 else n*n

assert all( (es:=square(en:=n)) == (ee:=n*n)
            for n in range(10) ), (en, es, ee)

----------------------------------------------

AssertionError: (3, 8, 9)
```

Here en, es, and ee store, respectively, the value of n that caused the failure, the value that was computed by square, and the value that was expected.

22.6.7 Beware: a trap in assert's syntax

assert is an instruction, like return, not a function, like print, and thus it requires no parentheses. Of course, you can always use parentheses around any expression without changing its meaning, and so assert False and assert(False) are strictly equivalent. Likewise, the following, with an error message, seems perfectly innocuous:

```
assert (False, "It's a trap!")
```

Yet, you receive a warning (in some contexts; Idle seems to eat them, but I see them when using Python from the terminal in 3.10.8) when running the code:

```
Warning (from warnings module):
SyntaxWarning: assertion is always true, perhaps remove parentheses?
```

What is happening here? (e1,e2,...,en) is the Python syntax for tuples; ordered immutable lists of elements. They can also be written simply e1,e2,...,en in some contexts. assert may potentially take two comma-separated arguments, but the comma is part of its syntax, there is no tuple involved. Putting parentheses around both arguments turns

```
(False, "It's a trap!")
```

into a single tuple – here, a couple, so non-empty, and therefore equivalent to True. Hence the warning.

`=`	$(\circ\cdot(\bullet\cdot\bullet))$
`+=, -=, *=, /=, //=, %=, &=,^= |=, <<=, >>=`	none
`:=`	none
`lambda`	$(\circ\cdot(\bullet\cdot\bullet))$
`x if C else y`	$(\circ\cdot(\bullet\cdot\bullet))$
`or`	$((\bullet\cdot\bullet)\cdot\circ)$
`and`	$((\bullet\cdot\bullet)\cdot\circ)$
`not`	$(\circ\cdot(\bullet\cdot\bullet))$
`in, not in, is, is not, <, <=, ==, !=, >=, >`	$((\bullet\cdot\bullet)\cdot\circ)$
`|`	$((\bullet\cdot\bullet)\cdot\circ)$
`^`	$((\bullet\cdot\bullet)\cdot\circ)$
`&`	$((\bullet\cdot\bullet)\cdot\circ)$
`<<, >>`	$((\bullet\cdot\bullet)\cdot\circ)$
`+, -`	$((\bullet\cdot\bullet)\cdot\circ)$
`*, @, /, //, %`	$((\bullet\cdot\bullet)\cdot\circ)$
`+x, -x, ~x`	$(\circ\cdot(\bullet\cdot\bullet))$
`**`	$(\circ\cdot(\bullet\cdot\bullet))$
`await x`	
`x[i], x[i:i′], x(a,...), x.a`	
`(x...), (x,...), [x,...], {x,...}, {k:v,...}`	

Figure 1: Precedence and associativity

22.6.8 Synthetic table of operator precedence and associativity

The table in Figure 1[p111] presents all precedence classes for Python constructs — including some not yet introduced in this document — in order of increasing priority. Where applicable, the associativity is indicated. When in doubt, refer to it.

A few notes are required in order to clarify some of the entries. Strictly speaking, the concept of associativity is only classically defined for binary operators. We extend the definition to the following intuition:

The associativity *of a syntactic construct is whichever side (left or right) the parentheses accumulate on, if made explicit.*

By that loose definition, a unary operator, like $-$, is right associative, because $--1 = -(-(1))$, and the parentheses accumulate on the right. The same idea applies to **`lambda`** and `.. if .. else ...`

23 Basic control flow

Here we recall the basic control flow structures. They mostly behave as expected and have few game-changing Pythonic specificities that I am aware of, so there is not much to say beyond giving the syntax.

The exception to that is the **for** loop. There is much more to it than meets the eye at first glance, and we shall come back to it later.

23.1 Conditional branching instruction: if

This is your usual **if**/**elif**/**else** statement, with the usual semantics.

```
if <$c_1$>:
    <execute if $c_1$>
elif <$c_2$>:
    <execute if $\neg c_1 \wedge c_2$>
...
elif <$c_n$>:
    <execute if $\bigwedge_{k=1}^{n-1} \neg c_k \wedge c_n$>
else:
    <execute if $\bigwedge_{k=1}^{n} \neg c_k$>
```

Just recall that the conditions are converted to Booleans if they are not already, with the consequences discussed in the section on that data type.

As in any language (apart from, say, pure functional languages) be mindful of side effects:

The following two blocks of code are only equivalent under a certain assumption. What is it?

```
if test():
    instr1()
else:
    instr2()
```

```
if test():
    instr1()
elif not(test()):
    instr2()
```

23.2 Conditional expression: .. if .. else .. ternary operator

Sometimes, you have a simple test that is best written in one line, typically when you are trying to return a conditional value or assign it to a variable, or perhaps use it in the middle of a computation.

This can be done with the following syntax, where B is a Boolean and v_{true} and v_{false} are expressions:

$< v_{\mathtt{true}} >$ **if** $< B >$ **else** $< v_{\mathtt{false}} >$

which stands for the *expression*, or the *value*:

$$\begin{cases} v_{\text{true}} & \text{if B holds} \\ v_{\text{false}} & \text{otherwise} \end{cases}$$

This corresponds to what is often called “the” ternary operator `<cond> ? t : f` in C and derived languages. The use of the definite article in this terminology is a bit imprecise: it is an operator with three operands, hence ternary, but not the only possible one.

Note that since this is an expression, and not an instruction, it can be used in the middle of a computation

```
>>> (1 if False else 2) **2
4
```

and you you cannot use return (or any instruction) within it: write

```
return 1 if True else 2
```

or

```
if True:
  return 1
else:
  return 2
```

but never

```
return 1 if True else return 2
```

This construct can be chained, of course, though doing so is not recommended for reasons of legibility. There are two ways to interpret the expression `1 if C1 else 2 if C2 else 3`, depending on the associativity of this – slightly weird – operator. Let us test that in a systematic way:

```
def default(x,y): return  1 if x else  2  if y else 3
def left   (x,y): return (1 if x else  2) if y else 3
def right  (x,y): return  1 if x else (2  if y else 3)

for x in (0,1):
    for y in (0,1):
        print(x,y,"  ", default(x,y), left(x,y), right(x,y))

------------------------------------------------------------
```

```
0 0    3 3 3
0 1    2 2 2
1 0    1 3 1
1 1    1 1 1
```

Thus we see that the ternary operator is associative to the right which, after a minute of reflection, appears as the most natural option.

Functions being first-class objects, this construct is perfectly capable of switching between functions:

```
>>> (str.lower if True  else str.upper)("Abacus")
'abacus'

>>> (str.lower if False else str.upper)("Abacus")
'ABACUS'
```

This is to be used with parsimony, if at all.

The ternary operator has the lowest precedence of all Python operators.

23.3 While loop

The `while` loop is as usual:

```
while <condition>:
    <intructions block>
```

`break` and `continue` statements can be used in a for loop, with the usual semantics.

There is one Pythonic surprise, though: `while` may be paired with an `else` clause, executed at the natural end of the loop — that is to say, after the condition turns to `False` — but not in the event of a `break`. This also works with `for` loops, and an example is provided in the corresponding section.

This syntactic construct is rarely used, and the choice of `else` for the keyword is widely acknowledged as injudicious.

You may mentally substitute `nobreak` for `else` when trying to wrap your head around that concept.

If you use that construct at all, I'd recommended commenting the `break` and `else` lines to make the logic clear.

23.4 `for .. in .. range` loop

The syntax of the `for` loop that most closely resembles the classical "`i++`" approach is the `range` construction:

```
for k in range(0,10,2):
    print(k, end=' ')
-----------------------
0 2 4 6 8
```

As you can guess, the last argument, the increment, is optional and defaults to 1. Following the same convention as for slices, the starting point is inclusive, and the end point is exclusive.

```
for k in range(10,0,-2):
    print(k, end=' ')
------------------------
10 8 6 4 2
```

There is also a syntax with just one argument: `range(n)` is interpreted as `range(0,n)`. This often appears when generating the indices of a string or list:

```
s = "Python"
for k in range(len(s)):
    print(s[k], end=' ')
------------------------
P y t h o n
```

This syntax appears a bit restrictive, and it is not immediately clear whether the **range** syntax is actually an intrinsic part of the **for** loop's own syntax or something else.

As it happens, **range** is not tied to **for**'s syntax:

```
>>> range(10)
range(0, 10)
>>> type(range(7))
<class 'range'>
```

but it's not clear at this point what it is and what you can do with it. For now let us just say that the **range** type is an iterable and indexable/subscriptable sequence type — you can use **for** loops, `r[i]` indexed access and `r[i:j:step]` slice notation on them.

We shall come back to this in greater detail when speaking of *generators* later on, but know that **for** in Python is really a "for each", that iterates over every item in a collection type that supports the operation – an *iterable* type. Ranges are just one such type, but strings, lists, tuples, sets, etc, are as well.

```
for <var> in <collection>
    <block in which var takes the value...
    ... of an element of the collection. >
```

So you can just write

```
for c in "Python":
    print(c, end=' ')
---------------------
P y t h o n
```

Not only is this clearer and less error-prone, since there is no point in reasoning on indices if your logic does not depend on them, but this is actually slightly more efficient in general, as you don't have to generate an extra **range** object.

It is important to note that the iteration variable exists outside the scope of the **for** loop, and holds the last value it took during the loop. For instance, after iterating on "Python" as above, we have:

```
>>> c
'n'
```

This does *not* apply to **for** loops appearing in comprehension expressions.

The iteration variable can actually be replaced by a structure of variable names, in which case the same kind of pattern matching as in standard assignments on nested structures (cf. Sec. 24.1[p126]: "Tuples: class tuple") is performed. For instance, here we iterate over lists of couples:

```
>>> l = [ (n, n**2) for n in range(5) ]

>>> l
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16)]

>>> for x,y in l:
      print(f"{x} -> {y}   ", end='')

0 -> 0   1 -> 1   2 -> 4   3 -> 9   4 -> 16
```

This is often used with, for instance, the enumerate construct, which provides automatic indexing of an iterable:

```
>>> list(enumerate("Python"))
[(0, 'P'), (1, 'y'), (2, 't'), (3, 'h'), (4, 'o'), (5, 'n')]

>>> for k,c in enumerate("Python"):
      print(f"{k}:{c}  ",end='')

0:P  1:y  2:t  3:h  4:o  5:n
```

Also useful is the zip function, to consume several iterables at the same time:

```
>>> for x,y,z in zip([1,2,3], "abcd", "XYZ"):
...     print(x,y,z)

1 a X
```

```
2 b Y
3 c Z
```

Note that it stops when the shortest iterable is exhausted.

Also occasionally useful is the `reversed` function, which reverses an (ordered) iterable:

```
>>> for x in reversed(range(3)):
...     print(x)

2
1
0
```

Again, we shall examine the underlying notions in more detail in the sections on iterables, generators, etc.

break and **continue** statements can be used in a for loop, with the usual semantics.

Like **while**, **for** loops may also have an **else** clause in Python, executed at the end of the loop but not in the event of a **break**.

This example, lifted from Python's own documentation, – with some additional comments – illustrates the use of this construct:

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(n, 'equals', x, '*', n//x)
            break # found a factor
    else:# if no break
         # loop fell through without finding a factor
        print(n, 'is a prime number')
-----------------------------------------------------
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

23.5 try .. except

You may already be familiar with exceptions if you have used Java or any other somewhat modern, mid-to-high level language. Whenever something goes wrong with your program, rather than crashing outright, it "raises" and "exception".

If the exception is not caught / handled at some point by code that invoked the faulty sub-program, the execution is interrupted. The exception itself is an object that can carry some information about the type and parameters of the failure.

try is a flow-control structure dealing with exceptions. Its syntax admits many variants, as follows:

```
try:
  <code that may fail>

except <ExceptionName1> :
  <what to do if this exception is raised>

except <ExceptionName2> as <var>: # as is optional,
  <what to do in that case>       # exception is bound as <var>

except (<Ex3, Ex4, ...):          # catch any of those
  <what to do in that case>

except :                          # catches all other exceptions
  <what to do in that case>

else:                             # optional
  <executes if the try block does not raise any exception>

finally: # optional
  <always executes after, regardless of exceptions and breaks>
```

As you can see, it is somewhat similar to a switch/case structure (which did not exist in Python before version 3.10), but specialised for exceptions.

Exception names — actually classes — are among **Exception** (of which all others are subclasses), **AssertionError**, **NameError**, **TypeError**, **IndexError**, **KeyError**, **ValueError**, **OverflowError**, **ZeroDivisionError**, etc.

A classical example, seen whenever user input is involved:

```
n = None
while n is None:
  try:
      n = int(input("Enter a number: "))
  except ValueError:
      print("Invalid number. Retry.")
```

You can define your own exception types by subclassing **Exception**. This not essential, and will be clearer after reading the section on objects.

If you want a catch-all that can manipulate the raised exception, unlike **except:**, you can use, for instance:

```
except Exception as e: print(repr(e))
```

23.6 Pattern matching: match..case

For the longest time, Python had no equivalent to any kind of switch/case construct. Alternatives included chaining if..elif constructs and using dictionaries, none of which was very satisfactory. Then, for version 3.10, the developers suddenly woke up and added the match..case statement. This turned out to be a very significant addition.

Not only does match cover everything that is expected of a switch construct, but it goes deeper, and enables structural pattern matching, a powerful tool more often seen in statically typed functional languages such as OCaml or Haskell. match is not *quite* as convenient as the equivalent in those languages, but we are 90% of the way there, and I for one welcome our new match-ing overlords.

In this section, we shall take a very superficial view of what it can do. We shall come back to it later, after Sec. 24.6[p164]: "Packing and unpacking" and Sec. 27[p183]: "Object Oriented Programming in Python", and explore the power of structural pattern matching a bit more in depth. That will be the object of Sec. 28[p202]: "Advanced structural pattern matching".

23.6.1 Syntax overview

For now, the syntax:

```
match expr:
    case pattern1: <execute if expr matches pattern1>
    case pattern2: <execute if expr matches pattern2 but not 1>
    ...
    case patternN: <execute if expr matches patternN but not 1..N-1>
```

Patterns are a new class of syntactic constructs, which can be thought of as a generalisation of assignment targets. In the statement $x = 2$, x is the assignment target, and now x is bound to the value 2.

More complex assignment targets can be found in Sec. 24.6[p164]: "Packing and unpacking": for instance [a,b] = [1,2] breaks down [1,2], and binds a to 1 and b to 2. This is structural pattern matching: finding, if possible, an assignment of the variables in the left-hand side that matches the structure of the right-hand side. Patterns in the match statement generalise that.

Intuitively, the match statements attempts to make pattern1 = expr happen, then pattern2 = expr if the first didn't work, and so on. If a pattern works, it bind variables (if the patterns contains variables), and executes the code corresponding to the *case*.

Syntax note: Unlike all other keywords, such as def, if, etc, match and case keywords are so-called *soft* keywords, which is to say you can still use them as variable or argument names.

```
>>> def = 7
SyntaxError: invalid syntax
>>> if = 7
SyntaxError: invalid syntax
>>> match = 1
>>> match
1
```

This does not prevent them from being recognised as keywords when they are used in the right grammatical context, which is to say that of a statement, rather than an expression.

23.6.2 The different types of patterns, by example

Let us take an example that covers the different types of patterns, if only superficially. We shall go through it pretty much line by line.

```
    match x:
        case 0:                         return "Ze Zero or Neo"
        case 1 | -1:                    return "Neo or Negative Neo"
        case int():                     return "integer != 0"
        case "INSA":                    return "lotta homework"
        case str() as s:                return f"a string '{s}'"
        case "a", 1, 3.0:               return "a very specific sequence"
        case [x, y, z]:                 return f"3 element sequence {x}-{y}-{z}"
        case [1|2 as x, 3|4 as y] as l: return f"or/as {x} {y} {l}"
        case x, [*l], y, z:             return f"4 elem seq, 2nd is seq {l}"
        case x, *rest:                  return f"at least 1 element {x}:{rest}"
        case {1:v, 2:V, 3:x} if x==v+V: return f"dict 123"
        case {2:v, **r}:                return f"dict 2 -> {v}; {r}"
        case {3:8}:                     return "38"
        case _:                         return "who knows?"

for x in [-1, 0, 1, 2,
          "INSA", "Meh",
          (1,2,3), ["a", 1, 3.0], ("a", 1, 3.0), {"a", 1, 3.0},
          [1, 3], [1, 4], [3, 1],
          (1,), [1,2], [1,2,3], [1,2,3,4], [1, 2, [3, 4], 5],
          [1, [2, 3], 4, 5], [1, (), 4, 5],
          {1, 2, 3}, {1:"a", 2:"b", 3:"c"},
          {1:4, 2:3, 3:7}, {1:4, 2:3, 3:8}, {1:4, 3:8},
          ]:
    print(f"{repr(x):>30} -> {match(x)}")

------------------------------------------------------------------------------

                            -1 -> Neo or Negative Neo
                             0 -> Ze Zero or Neo
                             1 -> Neo or Negative Neo
                             2 -> integer != 0
                        'INSA' -> lotta homework
```

```
                   'Meh' -> a string 'Meh'
               (1, 2, 3) -> 3 element sequence 1-2-3
           ['a', 1, 3.0] -> a very specific sequence
           ('a', 1, 3.0) -> a very specific sequence
           {1, 3.0, 'a'} -> who knows?
                  [1, 3] -> or/as 1 3 [1, 3]
                  [1, 4] -> or/as 1 4 [1, 4]
                  [3, 1] -> at least 1 element 3:[1]
                    (1,) -> at least 1 element 1:[]
                  [1, 2] -> at least 1 element 1:[2]
               [1, 2, 3] -> 3 element sequence 1-2-3
            [1, 2, 3, 4] -> at least 1 element 1:[2, 3, 4]
       [1, 2, [3, 4], 5] -> at least 1 element 1:[2, [3, 4], 5]
       [1, [2, 3], 4, 5] -> 4 elem seq, 2nd is seq [2, 3]
           [1, (), 4, 5] -> 4 elem seq, 2nd is seq []
               {1, 2, 3} -> who knows?
{1: 'a', 2: 'b', 3: 'c'} -> dict 2 -> b; {1: 'a', 3: 'c'}
      {1: 4, 2: 3, 3: 7} -> dict 123
      {1: 4, 2: 3, 3: 8} -> dict 2 -> 3; {1: 4, 3: 8}
            {1: 4, 3: 8} -> 38
```

It should not be too surprising to see

```
0 -> Ze Zero or Neo
```

This is the standard behaviour of a `switch/case` statement, as seen in other languages: the case is triggered if the expression is equal to the **literal pattern** `0`.

Next, we see an **OR pattern** `1 | -1` matching either of two constants.

```
-1 -> Neo or Negative Neo
 1 -> Neo or Negative Neo
```

More generally, you can use `p | q` to create a pattern that matches if either `p` or `q` matches — tested in order, which can matter if the patterns bind variables. Pretty straightforward so far.

```
2 -> integer != 0
```

Now, 2 is not covered by our previous patterns, and it somehow matches `int()`. Note that replacing `int()` by `int` would fail with error message

```
    case int:                     return "integer != 0"
         ^^^
SyntaxError: name capture 'int' makes remaining patterns unreachable
```

This is because `int` is an ordinary variable name, like `x` or `y`, that just happens to be bound to the class for integers by default. As a pattern, a variable name simply matches everything, and is bound to the matched object.

This is called a **capture pattern**. Writing `case int:` in our `match x:` would therefore be tantamount to writing `int = x`, with the result of binding the variable `int` to the value of `x`.

The reason Python protests is that, since a capture pattern always matches, every `case:` below that line becomes useless: they can never be tested. Recall from the syntax definition that match executes the *first* `case` *that matches*, and only the first.

So, back to what is actually written, what does `int()` mean? This is a **class pattern**. It matches if the expression is an instance of the class `int`, as tested with `isinstance()`.

Note that this means that anything in a sub-class of `int` matches as well. Recall that `bool` is a subclass of `int`; we have:

```
>>> isinstance(True, int)
True
>>> match True:
...     case int(): print("yes")

yes
```

Class patterns go far beyond just testing the type, though; between the parentheses, one can test the values of attributes as well. We shall come back to that in Sec. 28[p202]: "Advanced structural pattern matching". Meanwhile, just remember that you need a `()` suffix to test a type in a pattern, and leave it at that.

```
                    'INSA' -> lotta homework
                     'Meh' -> a string 'Meh'
```

The first corresponds to a literal pattern again, and the second to a class pattern, with a twist: `str() as s`. `str()` is obviously the class pattern, and `as` is a keyword that can be used within a pattern to bind all or part of the matched expression to a variable name.

In that case, we match any string and bind it to `s`. This is not the most useful use of `as`, though, as we could have reused `x`. In the case where the matched expression has no convenient name already available, e.g. in a `match [1,2,3]:` binding the matched value that way can be quite useful.

We shall soon see cases where `as` binds only *parts* of the matched value, rather than all, where the keyword is much more obviously useful.

The next cases are where things start getting a bit more complex. *You might want to pause here and come back after reading Sec. 24[p126]: "Container data types" if you're not entirely clear on what lists, tuples, sets, and dictionaries are, and how they are written.*

Let us focus on a subset of the rules:

```
        case "a", 1, 3.0:   return "a very specific sequence"
        case [x, y, z]:     return f"3 element sequence {x}-{y}-{z}"
        case _:             return "who knows?"
```

```
--------------------------------------------------------------------
                    (1, 2, 3) -> 3 element sequence 1-2-3
                ['a', 1, 3.0] -> a very specific sequence
                ('a', 1, 3.0) -> a very specific sequence
                {1, 3.0, 'a'} -> who knows?
```

`"a", 1, 3.0` is a **sequence pattern**, and one that happens to only contain constants. It is very important to note that it could equally have been written `("a", 1, 3.0)` or `["a", 1, 3.0]`, with no difference in semantics whatsoever! That means there is no direct type distinction between `list` and `tuple` in patterns. Any ordered sequential type will match.

This is a bit shocking, but not too much when you consider that this is coherent with how unpacking works: `(a,b) = [1,2]` has the same effect as `[a,b] = [1,2]`, for instance.

Also note that a `set` will not match, even with the right values. In fact, there is no pattern support for sets at all:

```
>>> match 1:
...     case {"a", 1, 3.0}: pass
...
SyntaxError: invalid syntax
```

`case [x, y, z]` also presents a sequence pattern, this time with variables. It works exactly as expected from an unpacking, binding its variables to the components of any three-element sequence. There again, it could have been written `(x, y, z)` or simply `x, y, z` with no change in meaning.

What of `{1, 3.0, 'a'}`? Being a set, it does not match any of the two sequence patterns, but it matches `_`, which is the **wildcard**, or **catch-all pattern**. As its name indicates, it matches everything.

It does not bind, though, which is a bit of a subtlety, as `_` is a valid variable name in Python, and thus this could as well be a capture pattern, matching anything but also binding to `_`. `case _` will often be the last line of your match statement. If you need to bind, `case x: <do something with x>` can serve equally well.

Let us now focus on this:

```
        case [x, y, z]:                 return f"3 element sequence {x}-{y}-{z}"
        case [1|2 as x, 3|4 as y] as l: return f"or/as {x} {y} {l}"
        case x, [*l], y, z:             return f"4 elem seq, 2nd is seq {l}"
        case x, *rest:                  return f"at least 1 element {x}:{rest}"
-----------------------------------------------------------------------------
                          [1, 3] -> or/as 1 3 [1, 3]
                          [1, 4] -> or/as 1 4 [1, 4]
                          [3, 1] -> at least 1 element 3:[1]
                            (1,) -> at least 1 element 1:[]
                          [1, 2] -> at least 1 element 1:[2]
                       [1, 2, 3] -> 3 element sequence 1-2-3
```

```
             [1, 2, 3, 4] -> at least 1 element 1:[2, 3, 4]
         [1, 2, [3, 4], 5] -> at least 1 element 1:[2, [3, 4], 5]
         [1, [2, 3], 4, 5] -> 4 elem seq, 2nd is seq [2, 3]
```

Here we have sequence patterns, with a few twists. `[1|2 as x, 3|4 as y] as l` demonstrates `as`-bindings of both parts of the matches value, in `x` and `y`, and the entire value itself, in `l`.

`x, *rest` and `x, [*l], y, z` both demonstrate the use of packing (Sec. 24.6[p164]: "Packing and unpacking"), and the latter shows deep exploration of the structure: the second element must be a sequence type.

Sequence patterns are even more powerful than in unpacking contexts, though, because each element can be any pattern, as already demonstrated by `[1|2 , 3|4]`. For instance, `[2,"a"]` would not match the pattern `[str(), _]`, because the first element is not of type `str`.

An important limitation of sequence patterns is that at most one starred name may appear in them. Otherwise, the matching would be ambiguous. For instance, the pattern `[*a, *b]` may match `[1,2]` as `[], [1,2]`, or `[1], [2]`, or `[1,2], []`. Likewise, `[*a, "x", *b]` is ambiguous, as there may be several instance of `"x"` in the list. The "one starred name" limitation avoids all such problems.

If you want to do that kind of stuff — for instance, splitting a list along certain keywords — you should use manual programming with index searches, or regular expressions, or even a full-fledged parser generator. That kind of tasks has a way of getting complex quite quickly, and `match`/`case` is not meant to handle them on its own.

Finally, we deal with dictionaries, or, more generally, mappings:

```
    case {1:v, 2:V, 3:x} if x==v+V: return f"dict 123"
    case {2:v, **r}:                return f"dict 2 -> {v}; {r}"
    case {3:8}:                     return "38"
    case _:                         return "who knows?"
----------------------------------------------------------------
                {1, 2, 3} -> who knows?
   {1: 'a', 2: 'b', 3: 'c'} -> dict 2 -> b; {1: 'a', 3: 'c'}
      {1: 4, 2: 3, 3: 7} -> dict 123
      {1: 4, 2: 3, 3: 8} -> dict 2 -> 3; {1: 4, 3: 8}
            {1: 4, 3: 8} -> 38
```

The set `{1, 2, 3}` is not matched by anything — except the wildcard, of course — because a set is not a mapping. `{2:v, **r}` is an interesting **mapping pattern**.

It matches any mapping (in particular, `dict` and its derivatives) that contain *at least* the key `2`, whose corresponding value is bound to `v`. The remainder of the matched mapping. is bound to `r`, following the `**` syntax for dictionary unpacking. `**` can only appear at the end of a mapping pattern.

However, note that `{2:v}` alone would also match the same values! It just wouldn't bind the remainder of the dictionary. This is what happens with `case {3:8}`: it matches `{1: 4, 3: 8}`.

I really do not like this syntax — or rather its semantics. It is misleading. It looks like `case {3:8}` is a constant — not a literal, not an atomic value, but a constant nonetheless. You would expect the *pattern* `{3:8}` to match the *value* `{3:8}`, and nothing else, but that's not how it works. Instead it matches any *extension* of the pattern.

At least it is not inconsistent with assignment semantics, because no unpacking exists for dictionaries:

```
>>> {a:b} = {1:2}
SyntaxError: cannot assign to dict literal here. Maybe you meant '==' instead of
'='?
```

Still, I would rather have seen `case {3:8,...}` or a mandatory `case {3:8, **r}` rather than this. It is what it it; keep that in mind if you match dictionaries.

That said, you can do some pretty neat things in practice, so long as you are clearheaded about what the syntax means: let us extract the name and first phone number of a student, from a record containing other, irrelevant information, which we ignore:

```
>>> match {"name":"Toto", "phones":[123,911], "sex":"safe"}:
...     case {"name":n, "phones":[p,*r]}: print(n,p)
...
Toto 123
```

23.6.3 A simple application: handling a command line

Let us use `match` to handle a basic command line interface:

```
def cmdmatch(c):
    ops = {"cp":"copy", "mv":"move"}
    match c.split():
        case ["cp"|"mv" as c, *options, source, target]:
            for o in options:
                match o:
                    case "-v": print("I'm verbose")
                    case "-i": print("I'm interactive")
                    case _   : raise ValueError(o)
            print(f"I {ops[c]} {source} to {target}")
        case ["cp"|"mv" as c, *r]:
            raise TypeError(f"{ops[c]} needs at least 2 arguments")

--------------------------------------------------------------------

>>> cmdmatch("cp -i -v toto tata")
I'm interactive
```

```
I'm verbose
I copy toto to tata

>>> cmdmatch("mv toto tata")
I move toto to tata

>>> cmdmatch("mv -x toto tata")
ValueError: -x

>>> cmdmatch("mv toto")
TypeError: move needs at least 2 arguments
```

As you can see, `match` is a rather natural and straightforward tool for handling that type of problem.

We shall see more advanced applications of structural pattern matching in Sec. 28[p202]: "Advanced structural pattern `match`ing".

24 Container data types

So far, we have seen basic, *atomic* types – with the weird exception of the string type, which is both atomic and a sequence, depending on length. Now we focus on *composite* data types, which specifically serve as containers for groups of elements of (other?) types.

24.1 Tuples: class `tuple`

Tuples in Python work pretty much in the same way as they do in mathematics, and share the same syntax:

```
>>> t = (1, "toto", 3.14) # note the heterogeneous types
>>> t
(1, 'toto', 3.14)
>>> type(t)
<class 'tuple'>
```

In some contexts, the surrounding parentheses are optional:

```
>>> t = 1, "toto", 3.14
>>> t
(1, 'toto', 3.14)
```

Elements are grouped in a specific, sequential order. Therefore they are indexable (subscriptable) and slice-able, following the same syntax as seen for strings.

```
>>> t[1]
'toto'
```

```
>>> t[:2]
(1, 'toto')
>>> t[:-1]
(1, 'toto')
>>> t[:0]
()    # empty tuple
>>> t[:1]
(1,)  # singleton tuple
```

Note the strange syntax for singleton tuples; this is necessary, because (1) is just the expression 1 in parentheses, and should be equivalent to it – you should always be able to put an expression in parentheses without changing its meaning. Having a weird syntax in the specific case of singleton tuples is an acceptable compromise to avoid confusion between tuple parentheses and expression parentheses.

Tuples can be nested:

```
>>> t = (1, (2,3) , 4)
>>> t[1][0]
2
```

They can be concatenated and multiplied, like strings, again because those operations make sense on any sequential container type:

```
>>> (1,2) + (3,4)
(1, 2, 3, 4)
>>> (1,2) * 5
(1, 2, 1, 2, 1, 2, 1, 2, 1, 2)
```

Like strings, and unlike, say, lists, they are immutable:

```
>>> t[0]=8
TypeError: 'tuple' object does not support item assignment
```

If you want to modify a tuple, just construct a new tuple from the old one, following the same recipe as seen previously for strings:

```
>>> (8,) + t[1:]
(8, (2, 3), 4)
```

There is a degree of pattern-matching in some contexts, which enables you to perform assignments on nested structures: for instance

```
>>> (a, (b,c), d) = (1, (2, 3), 4)
>>> print (a, b, c, d)
1 2 3 4

>>> (a, *m, b) = (1, 2, 3, 4)
>>> a, b, m
(1, 4, [2, 3])
```

See Sec. 24.6[p164]: “Packing and unpacking” for more details on the role of the `*` in the `(a, *m, b)`, and note that the result would be the same with `[a, *m, b]` or `a, *m, b` on the left-hand side.

As mentioned for basic types, the name of a class acts as a constructor for it, and can therefore be used for purposes of conversion from another to this one. For instance:

```
>>> tuple("Python")
('P', 'y', 't', 'h', 'o', 'n')
```

Of course, that only works if the conversion makes enough sense that somebody thought of implementing it:

```
>>> tuple(1)
TypeError: 'int' object is not iterable
```

We shall see what “iterable” means in more detail soon, but intuitively it means being a container whose elements can be enumerated in some arbitrary order, one after the other, to build a tuple. An object that cannot do that cannot be converted into a tuple.

Tuples themselves are iterables, of course, and so they support for each loops:

```
for x in (1,2,3):
  print(x,end='')
---------------------
123
```

Like pretty much all containers, they support the `in` and `not in` operators to test whether an element is contained within them. Or rather, whether they contain an element of equal value (as opposed to equal memory location).

```
>>> 2 in (1,3,4)
False
>>> 3 in (1,3,4)
True
```

Of course, they have a length:

```
>>> len( (1,2,3) )
3
>>> len(1,2,3) # don't forget the parentheses!
TypeError: len() takes exactly one argument (3 given)
```

Tuples can be compared of course, with the semantics that two tuples are equal if and only if they are of equal length and elements of equal index are equal. More clearly:

$$(a_1, \dots, a_n) \quad = \quad (b_1, \dots, b_m) \quad \iff \quad n = m \;\;\wedge\;\; a_k = b_k \;\forall k$$

The same applies to all sequential containers.

```
>>> t = (1,2,3)
>>> t == (1,2,3)
True
>>> (1, 2, 3) == (2, 1, 3)
False
>>> (1,2) == (1,2,3)
False
```

Again, be careful with parentheses:

```
>>> 1,2 == 1,2,3
(1, False, 2, 3)
```

Note that two tuples of equal values do not necessarily occupy the same memory space, even when written as literal values:

```
>>> (1,2,3) is (1,2,3)
False
```

Contrast to integers and string literals

```
>>> 3 is 3
True
>>> "abc" is "abc"
True
```

An optimised execution might detect that and optimise memory. As of version `3.6.3`, even `python3 -O` does not do so. Since tuples are immutable, I cannot imagine any point in ever using **`is`** for them anyway.

What about inequalities? As seen with strings, and as applicable to any sequential, indexable (subscriptable) type, the total order on elements is lifted into the lexicographical order on containers. Let $u, v \in \Sigma^*$ be words representing the containers[n], then

$$u < v \quad \Longleftrightarrow \quad u \text{ is prefix of } v \quad \vee \quad \exists x, y, z \in \Sigma^*,\ a, b \in \Sigma : \begin{cases} u = xay \\ v = xbz \\ a < b. \end{cases}$$

```
>>> (1,2) < (1,2,3)
True
>>> (2,2) < (1,2,3)
False
```

[n] Here I am using notations that we shall see this semester in formal languages theory. It should still be pretty intuitive: Σ is the set of characters, and Σ^* the set of strings. xay means the concatenation of x, a, and y.

Note that you should only attempt to compare *homogeneous* tuples, as comparison is undefined between different types. In the following example, at some point, Python attempts to compute `2 < "a str"`, and it does not go well:

```
>>> (2,3) < ("a str",1,2,3)
TypeError: '<' not supported between instances of 'int' and 'str'
```

As always with a dynamic language, such errors may not appear immediately, as the type is only checked at runtime: no error is raised in the following code

```
>>> (2,3) < (1,2,3,"a str")
False
```

because, since $2 > 1$, the comparison is stopped immediately, and `2 < "a str"` is never actually run.

24.2 Lists: class `list`

Lists function almost exactly the same as tuples, with a bracket-based syntax:

```
>>> l = [1, "toto", 3.14]
>>> type(l)
<class 'list'>
```

The only meaningful difference is that **lists are mutable**, whereas tuples are not. That is to say, they can be modified in-place:

```
>>> l[1] = True
>>> l
[1, True, 3.14]
```

The `del` keyword can be used to remove elements from lists, in indexed notation:

```
>>> l = [0, 1, 2, 3, 4]
>>> del l[2]
>>> l
[0, 1, 3, 4]
```

Be careful when deleting several elements in succession, as indexes change after the first deletion:

```
>>> l = [0, 1, 2, 3, 4]
>>> del l[1], l[3]
>>> l
[0, 2, 3]
```

We can also assign to entire slices, replacing the entire sublist:

```
>>> l = [1, 2, 3]
>>> l[1:] = list("abc")  # or simply l[1:] = "abc", any iterable will do
>>> l
[1, ’a’, ’b’, ’c’]
```

Note that the right-hand–side of the assignment need not be a list, specifically; any iterable type will do:

```
>>> l[1:3] = (1,2)
>>> l
[1, 1, 2, ’c’]
>>> l[1:3] = range(5)
>>> l
[1, 0, 1, 2, 3, 4, ’c’]
>>> l[1:3] = 5
TypeError: can only assign an iterable
```

Note that in the edge cases where the slice is of length one or zero, the behaviour remains that of subsequence replacement:

```
>>> l = list(range(5))
>>> l
[0, 1, 2, 3, 4]
>>> l[1:1] = list("abc")
>>> l
[0, ’a’, ’b’, ’c’, 1, 2, 3, 4]
>>> l[1:2] = list("xyz")
>>> l
[0, ’x’, ’y’, ’z’, ’b’, ’c’, 1, 2, 3, 4]
```

Compare the last line to

```
>>> l[1] = list("xyz")
>>> l
[0, [’x’, ’y’, ’z’], ’y’, ’z’, ’b’, ’c’, 1, 2, 3, 4]
```

Replacing an element by a list nests the list, modifying the outer list in-place. Replacing a slice of length one by a list concatenates the list before the element to the replacing list, and that to the list after the element. Replacing a length of length 0 inserts the new list in place.

24.2.1 Lists versus tuples

I have read in several sources that tuples are best used for small heterogeneous data structures or records like

```
>>> student = ("Julius", "Caesar", 55)
```

Components are accessed through indexing:

```
>>> student[2]
55
```

Lists, on the other hand, should be used for homogeneous collections of unbounded length: lists of **int**, or lists of strings, etc.

I never found any concrete justification for that received wisdom, and, as I can figure out, is (almost) completely baseless, and hold the real question to be that of mutability.

Obviously any structure of unbounded length should *preferably* be homogeneous – otherwise iterating on them requires type dispatch. That is true of tuples as well as of lists. Of course, if it is of unbounded length, you are *probably* in a context where you want to add elements, which implies mutability, which implies using lists.

I find it a dangerous idea, under *most* circumstances, to use indexing to access elements of a record – unless it is of *very* small size, follows a *very* obvious order, and will *never* need extension.

Indeed, if you ever want to add data to your records, or reorder the fields? A lot of code will need refactoring. You are much better off using a dictionary (class `dict`, see the relevant section), a singleton (a class, singleton or not, see the section on objects), or a named tuple[(o)]. Thus, generally, neither lists not tuples should be used for that purpose. Exception include packing and unpacking (see relevant section) function arguments and multiple return values; in that case tuples should be preferred.

Immutability being the only concrete difference between the two types, the real criterion when deciding what type you should use for your "lists" is: "do you need mutability?" – i.e. do you need to add or modify elements. If not, go with `tuple`s, as they are a bit more efficient, can be used in sets and dictionaries, as we shall see shortly, and by using them you *know* that your code does not contain hard-to-debug side effects.

If you *do* need mutability for something specific, by all means go with lists. Usually, you don't. If you think you don't, and it turns out there is a case where you end up needing it, convert your tuple into a list, and you are set; since lists support all operations tuples do, the part of your code that uses tuples should not see the difference.

To expend on the question of efficiency, tuples are generally more memory-efficient and faster than lists in almost all respects – so long as you don't need to modify or append to them, of course.

What Python calls lists are actually dynamic arrays of pointers (even when the elements are of basic types): they overallocate memory exponentially to allow for fast appending, and are reallocated in a larger memory space when that excess capacity becomes insufficient. A tuple is allocated once, with exactly the right amount of memory.

[(o)] https://docs.python.org/3/library/collections.html#collections.namedtuple

```
>>> from sys import getsizeof
>>> help(getsizeof)

getsizeof(...)
    getsizeof(object, default) -> int
    Return the size of object in bytes.

>>> t = tuple(range(10))
>>> l = list(range(10))
>>> getsizeof(t)
128
>>> getsizeof(l)
200
```

As a parenthesis, "list" is really a misnomer for Python `list`s. In programming, "list" *usually* (systematically in functional languages) refers to *linked lists*: cells containing a pointer to the next cell, which have very different algorithmic properties from dynamic arrays. There are also structures explicitly called arrays in Python, which are mainly useful for interfacing with C code, and for intensive numerical computations (e.g. in NumPy).

24.2.2 Pointers and memory

To be clear, in Python, **every variable is a pointer**, except for basic types: `int`, `float`, `str`, `bool`. They are implicitly dereferenced when used.

Note about terminology: In the Python community, people often use the word *reference* instead of pointer.

24.2.2.1 *The danger of multiple pointers*

When you write `t = (1,2,3)`, you create a variable `t`, which points to a location in memory containing the structure. In the case of tuples, since the memory is never modified, there is hardly any reason to give it much thought. When it comes to lists, this becomes crucial.

Consider the following code:

```
>>> l = list(range(5))
>>> l
[0, 1, 2, 3, 4]

>>> m = l
>>> m
[0, 1, 2, 3, 4]

>>> m[1] = "Hello"

>>> m
[0, 'Hello', 2, 3, 4]
```

```
>>> l

[0, 'Hello', 2, 3, 4]

>>> m is l  # test whether they point to the same memory address
True
```

The list `l` was modified, though no line of code explicitly did so. But since `m` and `l` are pointers, `m = l` just means "copy the pointer `l` into a new pointer `m`". This does not create a deep copy of the target list; you just end up with two different pointers, pointing to the same memory location. They become two names for the same list.

24.2.2.2 *Case study: nested lists/ matrices*

Occurrences of shared pointers can become harder to see in nested lists:

```
>>> l = list(range(5))
>>> ll = [l,l]
>>> ll
[[0, 1, 2, 3, 4], [0, 1, 2, 3, 4]]
>>> ll[0][2] = "X"
>>> ll
[[0, 1, 'X', 3, 4], [0, 1, 'X', 3, 4]]
```

This is a source of danger in many common situations. Say that you want to initialise a $n \times n$ matrix with 0: there is a seemingly very elegant way to do so: `[0] * n` yields a list with n zeros; thus repeating `[ [0] * n ]` n times yields the desired matrix

```
n = 5
>>> M = [ [0] * n ] * n
>>> mprint(M) # a matrix printing function I defined.
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 0]
```

So far so good. However, when you actually alter the matrix, it does not behave in the desired way:

```
>>> M[2][1] = 1
>>> mprint(M)
[0, 1, 0, 0, 0]
[0, 1, 0, 0, 0]
[0, 1, 0, 0, 0]
[0, 1, 0, 0, 0]
[0, 1, 0, 0, 0]
```

Thinking in terms of pointers, what happens is not surprising: `[0] * n` is evaluated, and is handled as a pointer; let us denote it by `l`. The list `[ [0] * n ]` is therefore equivalent to `[l]`, and `M` is `[l,l,l,l,l]`.

In order to properly initialise a matrix, you need to evaluate the expression "`[0]*n`" n times, to create n different lists in memory. This can be done in a loop — or most elegantly using list comprehensions, which we shall see in more detail in Sec. 24.5[p153]: "Comprehension expressions".

```
>>> M = [ [0] * n  for _ in range(n) ]
>>> M[2][1] = 1
>>> mprint(M)
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 0]
[0, 1, 0, 0, 0]
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 0]
```

Additional problems of a similar nature appear when trying to copy matrices: we discuss this in Sec. 24.2.3[p137]: "Shallow copies and deep copies".

24.2.2.3 *In-place assignment on mutable structures*

Another thing that should be noted is the behaviour of +=:

```
>>> l = [1,2] ; m = ['a', 'b'] ; oldl = l
>>> l += m
>>> l
[1, 2, 'a', 'b']
>>> oldl
[1, 2, 'a', 'b']
```

As you can see, `l` is appended to in-place. Thus `l += m` is not actually equivalent to `l = l + m`:

```
l = [1,2] ; m = ['a', 'b'] ; oldl = l

>>> l = l + m
>>> l
[1, 2, 'a', 'b']

>>> oldl
[1, 2]
```

Indeed, `l = l + m` first evaluates `l + m`, which creates a new list in memory – let us call its pointer `p` – then performs `l = p`, redefining `l` to point to the new memory location, while the original is untouched.

24.2.2.4 *Infinitely deep lists*

Finally, note that it is possible to create infinite looping structures:

```
>>> il=[1]
>>> il.append(il)

>>> il
[1, [...]]

>>> len(il)
2
```

The list `il` is infinite in the sense that, though it only has two elements, its second element is always equal to itself, which makes it infinitely deep.

If you tried to write the entire list, you would write, endlessly, something along the lines of:

```
[1, [1, [1, [1, .... ]... ]]]
```

The only reason Python does not choke when trying to display it in the interactive mode — i.e. trying to get a `repr(..)` — is because it is careful to memorise which pointers it has already encountered, and thus detects such loops, printing an ellipsis `...` instead.

Any recursive function that does not perform similar bookkeeping will loop on such lists. Here is a naïve list printer:

```
def recf(l):
    print(end='[')
    for e in l:
        if type(e) is list:
            recf(e)
        else:
            print(e,end='; ')
    print(end='] ')
------------------------------
>>> recf([1,2,3])
[1; 2; 3; ]

>>> recf([1,list('abc'),3])
[1; [a; b; c; ] 3; ]

>>> recf(il)
[1; [1; [1; [1; [1; [1; [1; [1; [1;
KeyboardInterrupt  # I interrupted the program.
```

Needless to say, although it is important to know that this is possible, it is not recommended to define infinite lists unless you have a very, *very* good reason for it.

24.2.3 Shallow copies and deep copies

What if you want to copy a list, so as to alter two versions of it independently? The simplest way to proceed is to use the type constructor `list(..)`. There is also a `.copy()` method.

Recall that the `list(..)` constructor takes any iterable, including other lists, and creates a list containing the same elements, in the order of iteration. Thus we have:

```
>>> l = list(range(5))
>>> ll = list(l)
>>> l is ll # they are indeed different objects in memory
False
>>> l[2] = 'X'
>>> l,ll
([0, 1, 'X', 3, 4], [0, 1, 2, 3, 4])
```

Now let us do the same thing with matrices:

```
>>> n = 5
>>> M = [ [0] * n  for _ in range(n) ]
>>> MM = list(M)
>>> M is MM
False
>>> M[2][1] = 1
>>> mprint(M)
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 0]
[0, 1, 0, 0, 0]
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 0]

>>> mprint(MM)
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 0]
[0, 1, 0, 0, 0]   # what?
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 0]
```

Despite `M` and `MM` being different objects in memory, we *still*, somehow, managed to alter one through the other. What's going on here?

The reality of it is... we did *not* actually modify one through the other. Strictly speaking, we actually modified neither `M` nor `MM`, as objects.

Recall that a list is actually a list of pointers to the objects it contains. A pointer is basically an integer. When `M` was copied, a new list containing the same values, the same integers, the same pointers, was created. Let us visualise this by using a command revealing the memory locations of each object:

```
>>> help(id)
```

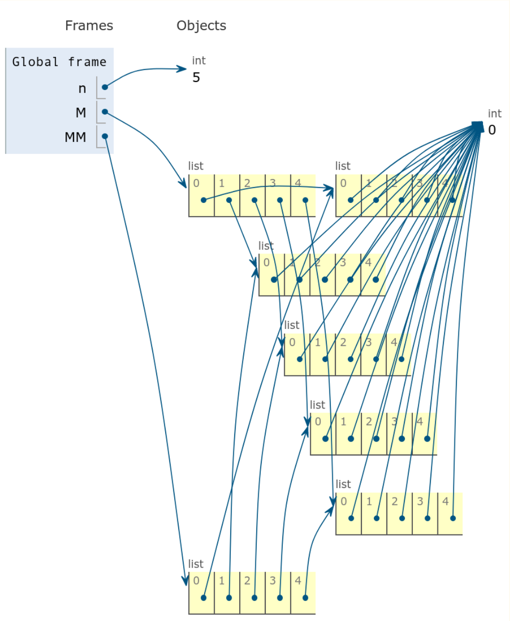


Figure 2: *In-memory representation of shallow matrix copy, using PythonTutor.*

```
id(obj, /)
    Return the identity of an object.

    This is guaranteed to be unique among simultaneously existing objects.
    (CPython uses the object's memory address.)

>>> [ id(e) for e in M ]  # list comprehension
[140180209944968, 140180127633608, 140180210018824,
  140180210019208, 140180209944520]
>>> [ id(e) for e in MM ]
[140180209944968, 140180127633608, 140180210018824,
  140180210019208, 140180209944520]
```

As expected, M and MM contain the same pointers. We represent this graphically using PythonTutor in Figure 2[p138]. Thus, M[2] and MM[2] are actually the same list.

What we did is called a *shallow copy*: we copied the list of pointers, but we did not bother to dereference the pointers to copy the objects we point to. A copying operation that recursively copies contained objects is called a *deep copy*. This operation is provided by the small, aptly names copy module:

```
>>> M = [ [0] * n  for _ in range(n) ]
>>> from copy import deepcopy
>>> MM = deepcopy(M)
>>> M[2][1] = 1
>>> mprint(M)
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 0]
[0, 1, 0, 0, 0]
```

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

>>> mprint(MM)
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 0]

>>> [ id (e) for e in M ]
[140180233630728, 140180127751240, 140180127750856,
  140180127750664, 140180127751368]
>>> [ id (e) for e in MM ]
[140180127749320, 140180165209800, 140180127751560,
  140180127751752, 140180127751944]
```

As the **id**s show, this time every single sublist has been fully duplicated in memory. At last our matrix copy behaves as expected.

24.2.4 How not to iterate on lists

Let me I share with you some terrible code I have seen in the 2018–2019 resit exam, in answer to exercise 51[p258]. (I have adapted the exercise and the code somewhat). It illustrates a very common mistake with in-place modification of iterable structures, whether in Python or in C.

The aim of the exercise is to write a function `remove_bad(l,bad)` returning a *new* list identical to `l` except in that elements equal to `bad` have been removed from it.

Here is (an adaptation of) the code the student produced:

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

def remove_bad(l,bad):
    for i in range(len(l)):
        print(i,l) # for debugging purposes
        if l[i] == bad:
            l.remove(l[i])  # removes the first occurrence, in-place

print(remove_bad(l,2))

----------------------------
0 [1, 2, 3, 2, 2, 1]
1 [1, 2, 3, 2, 2, 1]
2 [1, 3, 2, 2, 1]   # we have successfully removed the first 2
3 [1, 3, 2, 1]      # and another
4 [1, 3, 2, 1]
IndexError: list index out of range
    ... in remove_bad, if l[i] == bad:
```

There is a grave problem here — leaving aside that removing elements of l in-place does not answer the question, which demands a new list.

The algorithmic problem is that the length of l is computed *once*, at the beginning of the loop. When removing elements from the list in-place, its length diminishes with each element that is removed. At the end of the loop, if any element was removed, i takes index values that no longer exist.

The only case where this algorithm behaves correctly – again leaving aside that it does not answer the question – is when the list does not contain any element that should be removed.

The moral of the story is: be extremely careful about in-place alteration, especially in the presence of iteration.

24.2.5 Sorting

The **sorted** function takes any iterable (tuples, lists, sets, . . .), and returns a fresh sorted list of their elements:

```
>>> l = [2, 7, 4, 0, -6]
>>> sorted(l)
[-6, 0, 2, 4, 7]
>>> l
[2, 7, 4, 0, -6]
```

This is not to be confused with the sort method, which is a *procedure*, that sorts in place:

```
>>> print(l.sort())
None
>>> l
[-6, 0, 2, 4, 7]
```

The order can be reversed through use of the optional, keyword argument reverse:

```
>>> l = [2, 7, 4, 0, -6]

>>> sorted(l,reverse=True)
[7, 4, 2, 0, -6]

>>> l.sort(reverse=True)
>>> l
[7, 4, 2, 0, -6]
```

If there is a need to order a sequence according to a non-standard ordering, the optional keyword argument key can be provided. key is a function converting the elements of the sequence to be sorted to *keys*. It is according to their keys, then, that the elements are ordered. For instance, if we wish to order strings according to their length instead of the usual Unicode-based lexicographical ordering, we can do:

```
>>> l = ['School', 'Platypus', 'Sleep']

>>> sorted(l) # the usual order
['Platypus', 'School', 'Sleep']

>>> sorted(l,key=len)
['Sleep', 'School', 'Platypus']

>>> sorted(l,key=lambda s:s[2])
['Platypus', 'Sleep', 'School']
```

Here `len` is the usual function returning the length of a string, and `lambda s:s[2]` is merely an anonymous function associating to each string its third character.

Both `sort` and `sorted` implement *stable* sorting algorithms, which means that values that compare equal are kept in their previous order. This is extremely useful when sorting in multiple passes. Imagine having a list of students sorted in the usual alphabetical order. Then sort it by project group; an unstable sort could shuffle the students' names in each group.

The sorting algorithm used by Python is actually Timsort, a sophisticated blend of merge sort and insertion sort. It is *very* efficient.

24.3 Sets: class `set`

Sets in Python play the same role as sets in mathematics: they are unordered but iterable collections, without duplicates.

They support the standard set operators:

Mathematics	Python	
$=$	`==`	*set equality:* $A \subseteq B$ *and* $B \subseteq A$
$\neq$	`!=`	
$\subseteq$	`<=`	*inclusion:* $x \in A \Rightarrow x \in B$
$\supseteq$	`>=`	
$\subset$	`<`	
$\supset$	`>`	
$\cap$	`&`	
$\cup$	`|`	
$\setminus$ or $-$	`-`	*set difference:* $\{x \in A \mid x \notin B\}$
$\ominus$	`^`	*symmetric difference:* $A \cup B \setminus A \cap B$

```
>>> s = set(range(5)) ; ss = set(range(3,8))
>>> s, ss
({0, 1, 2, 3, 4}, {3, 4, 5, 6, 7})
```

```
>>> 1 in s , 7 in s
(True, False)

>>> s & ss    # intersection
{3, 4}
>>> s | ss    # union
{0, 1, 2, 3, 4, 5, 6, 7}
>>> s - ss    # difference
{0, 1, 2}
>>> s ^ ss    # symmetric difference
{0, 1, 2, 5, 6, 7}

>>> {}          # you can’t define an empty set that way
{}
>>> type({})
<class 'dict'>  # it’s actually the empty dictionary
>>> set()       # this is how you make an empty set
set()

>>> s <= ss , s >= ss
(False, False)  # inclusion is of course a *partial* order
>>> set() <= s  # empty set is smaller than everybody
True
```

They are not a sequential type, which means that there is no indexed access to elements, nor slices.

```
>>> s[2]
TypeError: ’set’ object does not support indexing
```

When iterated upon, each element is visited once, but the order of iteration is not guaranteed:

```
>>> for e in s:
  print(e,end=’’)

01234  # iteration follows the same order as display

>>> { 'Python’, 'abba’, 'ABBA’, 'A’,'a’}
{'a’, 'ABBA’, 'A’, 'Python’, 'abba’}

>>> { 'Python’, 'abba’, 'ABBA’, 'A’}
{'A’, 'ABBA’, 'Python’, 'abba’} # order has changed

>>> { 10010, 1, 86}
{1, 10010, 86} # order does not correspond to order on elements
```

For all intents and purposes, the order in which the elements of a set are displayed or iterated on should be considered random – or, more judiciously, *undefined*. It is not really random, but entirely dependent upon the internal implementation.

Duplicates are meant with respect to value, not memory location.

```
>>> t = (1,2)
>>> tt = (1,2)
>>> t is tt
False
>>> { t, tt }
{(1, 2)}
```

Beware: since Booleans are actually a subclass of integers, that actually have the same *values* as 0 and 1 – not merely as an implicit conversion to Boolean, unlike other numbers – and thus they will be confused in a set:

```
>>> { 0, False, True, 1 }
{0, True}
```

0 is added first, then `False`, but since it has the same value as 0, is is treated as a duplicate, despite being of a different (sub)type. Then `True` is added, and 1 is detected as a duplicate of it, which explains the result.

Sets are mutable, but can only contain immutable objects, as they are implemented using hash tables. In particular, that excludes sets of sets.

```
>>> { 1, 2, { 4, 5 } , 3 }
TypeError: unhashable type: 'set'
```

In the next section, we shall explore what *hashable* means.

Even if you are working exclusively with lists, a quick jaunt through sets can be extremely useful to efficiently remove duplicates from a collection:

```
>>> l = [2, 1, 2, 1, 1, 1, 3, 3, 2, 2]

>>> list(set(l))
[1, 2, 3]
```

This elegant solution, as anything involving sets, requires the elements to be hashable. A method working on *anything* would be of quadratic complexity — here using a comprehension expression, cf. Section 24.5[p153]:

```
>>> l = [[1], [2], [2], [1], [1], [1], [3], [1], [2], [1]]

>>> list(set(l))
TypeError: unhashable type: 'list'

>>> [ e for k,e in enumerate(l) if e not in l[:k] ]
[[1], [2], [3]]
```

24.3.1 Frozen sets: class frozenset

But what if you *want* sets of sets?

Let us first understand what that 'unhashable' error means.

The idea of hash tables, which Python sets use internally, rests on the computation of a "summary" (called *hash*) of the value of an object, which is then used to determine the address in memory where they are stored.

The hash is a deterministic function – although highly chaotic – so equal values imply equal hashes. The converse is not always true because some information is lost, as the hash is short. In a hash table, the hash of the value of an object determines the memory location where the object is stored.

When checking whether an object is already in the table, you just hash it, and use direct memory access to see if it is there, as opposed to, say, iterating over the container and testing equality against each value, as you would in a list. Membership testing ($\in$, **in**) is therefore very efficient for large collections, as it is amortised to $O(1)$ – constant time – with some trickery.

All of this rests on the assumption that the value of an object is constant. If your object is mutable, its value may change over time (that's what being mutable is all about), and thus its hash may change as well. But any set that would contain that object has no way of knowing whether or when a change takes place. Its hashes will not be recomputed. Thus the set ends up being inconsistent. This is why such structures are restricted to hashable (that is to say, generally, immutable) elements.

frozenset is an immutable alternative to set, though otherwise compatible with it – for instance, comparison operators ignore whether a set is frozen or not.

```
>>> S = frozenset(s) ; SS = frozenset(ss)
>>> S, SS
(frozenset({0, 1, 2, 3, 4}), frozenset({3, 4, 5, 6, 7}))
>>> s == S, ss == SS
(True, True)
```

You can now create a set of frozen sets:

```
>>> { S, SS }
{frozenset({0, 1, 2, 3, 4}), frozenset({3, 4, 5, 6, 7})}
```

24.4 Dictionaries: class dict

Dictionaries are a "key"/"value" structure, encountered in various languages under diverse names, such as associative arrays, associative memory, mappings, etc.

A dictionary should be regarded as a set of *keys*, which can take any (immutable) value, to each of which another (unique) *value* is associated. There is no restriction upon the mutability of associated values, only of keys, and this for reasons identical to the restriction on sets, for they share the same type of hashtable-based implementation — with a few specialised optimisations to accounts their different roles.[p]

Like sets, dictionaries are fundamentally unordered, though from Python 3.7 onwards their implementation is guaranteed to preserve the order in which elements are inserted[q]. Like all other collections we have see so far, they can contain elements of heterogeneous types — though this should be used parsimoniously.

Recall that, counter-intuitively, we could not use `{}` to define empty sets: this is because `{}` stands for the empty dictionary:

```
>>> {}
{}
>>> type({})
<class 'dict'>
>>> dict() == {}
True
```

Dictionaries can be initialised *in extenso* – or in *display form* – in the same way as sets et cetera, by listing their entries, separated by commas. Each entry, however, has a special, colon-separated `<key> : <value>` syntax:

```
>>> age = {’Toto’:15, ’Tata’:27, ’Mamie’:97 }
>>> age
{’Toto’: 15, ’Tata’: 27, ’Mamie’: 97}
```

Alternatively, they can be defined using the type constructor `dict()`, and a tuple of couples, or any other suitable sequence types:

```
>>> dict( [ (’Toto’, 15), (’Tata’, 27), (’Mamie’, 97 ) ] )
{’Toto’: 15, ’Tata’: 27, ’Mamie’: 97}
```

Note that only one value, at most, can be associated to any given key. If several associations are given, only the last one counts:

```
>>> {’Toto’: 15, ’Tata’: 27, ’Mamie’: 97, ’Toto’:99}
{’Toto’: 99, ’Tata’: 27, ’Mamie’: 97}
```

The key values are treated, syntactically, as indexes:

```
>>> age[’Tata’]
27
```

[p] The most frequent operation on a set is to test whether a given element belongs to it; sets are optimised for that. In the case of dictionaries, it is expected that you you are most often looking for values associated to a valid key, so they are *slightly* less efficient in the case where the key you are looking for is not there.

[q] This was already an “accidental” property of the implementation of CPython 3.6. You need to use an `OrderedDict` if you want another order or compatibility across all versions.

This makes dictionaries an indexable type, *stricto sensu*, as it implements the `__getitem__` method that underlies the `object[index]` syntax. However, they remain fundamentally unordered, like sets. You can choose keys in an ordered way if you like – though if your keys are 0..n, you are better off using a list or a tuple. At any rate, do not attempt to use slice notations on dictionaries.

```
>>> age[1:3]
TypeError: unhashable type: 'slice'
```

See Sec. 22.4.6[p86]: "Slicing and dicing, concatenation, repetition" on slices to understand why that message is what it is. Regardless, no slices on dictionaries.

In standard dictionaries, trying to access undefined keys results in an error.

```
>>> age['IDONTEXIST!']
KeyError: 'IDONTEXIST!'
```

There are, however, various ways to handle notions of "default values", as we shall see later.

Dictionaries are a mutable structure, like sets and lists. Entries can be added, altered, or removed outright, all using the index-like notation:

```
>>> age['IDONTEXIST!'] = "Well, *now*, I do!"
>>> age
{'Toto': 15, 'Tata': 27, 'Mamie': 97, 'IDONTEXIST!': 'Well, *now*, I do!'}

>>> age['IDONTEXIST!']
'Well, *now*, I do!'

>>> del age['IDONTEXIST!']
>>> age
{'Toto': 15, 'Tata': 27, 'Mamie': 97}

>>> age['IDONTEXIST!']
KeyError: 'IDONTEXIST!'

>>> del age['IDONTEXIST!']
KeyError: 'IDONTEXIST!'
```

Note that, in particular, dictionaries cannot appear as keys to other dictionaries. They can appear as associated values, though:

```
>>> { age : 21 }
TypeError: unhashable type: 'dict'
>>> { 21 : age }
{21: {'Toto': 15, 'Tata': 27, 'Mamie': 97}}
```

Like sets, dictionaries are iterable. For all iteration-related intents and purposes, they are treated as the set of their keys:

```
>>> for k in age:
      print(k,age[k])

Toto 15
Tata 27
Mamie 97
```

Consistently with this view of dictionaries as sets of keys, their length is defined as their number of defined keys, or, equivalently, or stored key/value associations:

```
>>> len(age)
3
```

Likewise, you can easily tests the presence of a key with **in**:

```
>>> 'Toto' in age
True
>>> 'toto' in age
False
```

If you want to iterate on key/value pairs directly, you can use the `items` method to get a "view" object – not an object any of the usual types – containing those pairs:

```
>>> age.items()
dict_items([('Toto', 15), ('Tata', 27), ('Mamie', 97)])

>>> for k,v in age.items():
      print(k,v)

Toto 15
Tata 27
Mamie 97
```

Likewise, if you want to iterate on the values only, there is a method for that:

```
>>> age.values()
dict_values([15, 27, 97])
```

In Python 3.9+, you can merge two dictionaries using the union (`|`) operator, as for sets. However, this operation is not commutative: if the two dictionaries define different values for the same key, only the last value is taken into account:

```
>>> {'Toto': 15, 'Tata': 99} | {'Tata': 27, 'Mamie': 97}
{'Toto': 15, 'Tata': 27, 'Mamie': 97}
```

There is also an `|=` operator, which updates a dictionary in place; the dictionary operators `|` and `|=` therefore play the same role for dictionaries as + and += do for lists. `|=` is a nicer syntax for the `dict.update` method, just as += is syntactic sugar for `list.extend`.

```
>>> d = {'Toto': 15}
>>> d |= { 'Tata': 27 }
>>> d
{'Toto': 15, 'Tata': 27}
>>> d |= [ ("Mamie", 97) ]
>>> d
{'Toto': 15, 'Tata': 27, 'Mamie': 97}
```

The last line shows that update accepts in the right-hand side any iterable convertible to a dictionary, whereas the union expect two dictionaries:

```
>>> {'Toto': 15, 'Tata': 27} | [ ("Mamie", 97) ]
TypeError: unsupported operand type(s) for |: 'dict' and 'list'
```

Prior to Python 3.9+, creating a fresh, merged dictionary required the use of dictionary unpacking, as discussed in section Sec. 24.6.2.1[p169]: “Merging two dictionaries”.

Leaving aside dictionary merging, let us move on to the set of keys (ignoring associated values). There is a method to get a set-like objects representing the set of keys, which enables you to use set operators, which are not usable directly on dictionaries (with the exception of | in Python 3.9+):

```
>>> age.keys()
dict_keys(['Toto', 'Tata', 'Mamie'])

>>> age & {'Toto', 'xx'}
TypeError: unsupported operand type(s) for &: 'dict' and 'set'

>>> age.keys() & {'Toto', 'xx'}
{'Toto'}
```

In the case where associated values are hashable, the dict_items object returned by the items method is also set-like:

```
>>> {'Toto': [] }.items() & age.items()
TypeError: unhashable type: 'list'

>>> {'Toto': 15 }.items() & age.items()
{('Toto', 15)}
```

Those three view objects, keys, items, values, are dynamic, in the sense that they are always updated along with the underlying dictionary:

```
>>> K = age.keys()
>>> age['Banana'] = 10
>>> K
dict_keys(['Toto', 'Tata', 'Mamie', 'Banana'])

>>> del age['Banana']
>>> K
dict_keys(['Toto', 'Tata', 'Mamie'])
```

However, they cannot be used to modify the underlying dictionary indirectly.

Two dictionaries are considered equal for the purpose of == if they contain identical pairs key/value. Other comparison operators are not supported:

```
>>> {'Toto': 15 } <= age
TypeError: '<=' not supported between instances of 'dict' and 'dict'
```

If you want to reason about the "inclusion" of a dictionary into another, you can use its items view:

```
>>> {'Toto': 15 }.items() <= age.items()
True
>>> {'Toto': [] }.items() <= age.items()
False  # this works, despite [] not being hashable
```

Since there are a few subtleties with those view objects, for instance `values` being unsuitable for any comparison,

```
>>> age.values() == age.values()
False # == always returns False on that view
```

I would advise, for purposes of comparison, to explicitly and systematically convert views – especially `values` and, to a lesser extent, `items` – to sets or whatever you need; that may be suboptimal in terms of execution time, but at least the semantics of the comparisons become entirely clear:

```
>>> set(age)
{'Toto', 'Mamie', 'Tata'}
>>> set(age.items())
{('Tata', 27), ('Mamie', 97), ('Toto', 15)}
>>> set(age.values())
{97, 27, 15}
>>> set({'Tata': 27 }.items()) <= set(age.items())
True
```

24.4.1 Handling default values

24.4.1.1 The `get(key, default)` method

Suppose that we want to count the number of appearances of each letter in a string. From

```
s = "AABaaBAAA"
```

we expect to obtain

```
{’A’: 5, ’B’: 2, ’a’: 2}
```

With what we have seen so far, we cannot write

```
def count(s):
    d = {}
    for c in s:
        d[c] += 1
    return d
----------------------
Traceback in d[c] += 1
  KeyError: ’A’
```

because the first time a letter is encountered, it has no corresponding associated value; we cannot increment something that is yet undefined. We have to test whether the key has been encountered previously, and initialise the field if not:

```
def count(s):
    d = {}
    for c in s:
        if c in d:
            d[c] += 1
        else:
            d[c] = 1
    return d
```

Would it not be better if we could say that, by default, we have encountered a key zero times? As it happens, we can, using the `get` method:

```
>>> age.get(’Toto’)  # this item exists
15
>>> age.get(’XXX’)   # this one does not
None # here I wrote it explicitly;
     # the interactive mode won’t print it unless requested
>>> age[’XXX’]
KeyError: ’XXX’
```

`get` is thus a more forgiving version of indexed access, returning `None` – by default – when the key is unknown. The default returned value can be changed through an optional argument:

```
>>> age.get(’Toto’,"I’m not here!")
15
>>> age.get(’XXX’,"I’m not here!")
"I’m not here!"
```

Thus our `count` function becomes:

```
def count(s):
    d = {}
    for c in s:
        d[c] = d.get(c,0) + 1
    return d
```

There is also the `setdefault` method, occasionally useful, which behaves in all ways like `get`, except that it not only returns the default value, but inserts it in the dictionary if needed.

Thus, one can write, for instance

```
d.setdefault(key, []).append(value)
```

to add to a list of values that may not exist yet.

24.4.1.2 *`defaultdict`, from `collections`*

There is a variant of the dictionary class that is created with a "default factory". This factory is a function which is called when a key is missing, and returns the value now associated with this key. Let us begin by importing the type constructor:

```
from collections import defaultdict
```

Let us create a dictionary such that all default values are zero:

```
>>> d = defaultdict(lambda: 0)
```

Note in passing that you could also create a `defaultdict` and initialise it with the values of another dictionary `D` like so:

```
>>> d = defaultdict(lambda: 0, D)
```

In both cases **`lambda`**`:0` is merely a niladic function (i.e. it has no arguments) that returns zero. This is our factory – the advantage of having a function in that role rather than simply a default value will become clear later.

```
>>> d
defaultdict(<function <lambda> at 0x7fb0a51bde18>, {})
```

The display is a bit verbose, but only the `{}` at the end matters: for now, the dictionary is empty.

```
>>> d[5]
0
>>> d
defaultdict(<function <lambda> at 0x7fb0a51bde18>, {5: 0})
>>> d['hey']
0
>>> d
defaultdict(<function <lambda> at 0x7fb0a51bde18>, {5: 0, 'hey': 0})
```

Key/default values associations are created on the fly whenever a key is looked up.

Thus our function becomes:

```
def count(s):
    d = defaultdict(lambda: 0)
    for c in s:
        d[c] += 1
    return d
```

Suppose now that instead of simply counting occurrences, we wish to list the indexes where they appear: on the previous example `s = "AABaaBAAA"` we would expect

```
  {'A': [0, 1, 6, 7, 8], 'B': [2, 5], 'a': [3, 4]})
```

Before encountering a letter, we have seen it at no index; thus our default should be the empty list. We can then append to it the relevant indexes. Our default factory is therefore the function of no argument returning a new list. We could write it `lambda:[]`, but we already have a function (actually, a callable) for that, and it is the list constructor `list`, which does return a new empty list when called with no arguments. (Actually, we could have used `int` in place of `lambda:0` as well, as it returns 0 with no argument. Most type constructors, when called with no argument, return whatever makes sense as the "zero" or "empty" element for that type.)

Thus we obtain the function:

```
def occs(s):
    d = defaultdict(list)
    for k,c in enumerate(s):
        d[c].append(k)
    return d
```

Note how essential it is in this application that a default *factory*, a function, be provided, and not merely a default *value*. Let us simulate passing a default value by defining a list and giving a factory that always returns a pointer to that same list:

```
l=[]
def occs(s):
    d = defaultdict(lambda:l)
    for k,c in enumerate(s):
        d[c].append(k)
    return d
print(occs(s))
----------------------------------------------------------------
defaultdict(<function occs.<locals>.<lambda> at 0x7f4c9fe3cea0>,
  {'A': [0, 1, 2, 3, 4, 5, 6, 7, 8],
   'B': [0, 1, 2, 3, 4, 5, 6, 7, 8],
   'a': [0, 1, 2, 3, 4, 5, 6, 7, 8]})
```

All values point to the same list `l`. Thus, when values are mutable, it is essential to be able to create a fresh value for each key. Hence the need for a “factory”.

The `defaultdict` structure enables some pretty nice tricks. See for instance a trick to represent trees, in this snippet of code.

24.4.1.3 *`Counter`, from `collections`*

For the specific use-case of counting elements, there is actually a nice variant of `dict` called `Counter`. Still with `s = "AABaaBAAA"`, we have:

```
>>> Counter(s)
Counter({'A': 5, 'B': 2, 'a': 2})
```

Counters in Python play the role of multisets, or bags, in mathematics. That is to say, sets where elements may appear multiple times. Thus they support some specific arithmetic operations:

```
>>> Counter(s)+Counter(s)
Counter({'A': 10, 'B': 4, 'a': 4})
```

As a fun example, an *anagram* of a word or sentence is another word or sentence using exactly the same letters, and the same number of each. We ignore case. Generally, in the case of sentences, we ignore whitespace and punctuation as well. Testing if two strings are anagrams is easy thanks to counters:

```
>>> def is_anagram(u,v):
      u,v = ( e.replace(" ","").upper() for e in (u,v) )
      return Counter(u) == Counter(v)

>>> u = "Counting Eh Tv"
>>> v = "Vincent Hugot"
>>> is_anagram(u,v)
True
```

The `u,v = ..` line makes use of a comprehension expression to avoid repeating the same code twice; cf. Section 24.5[p153].

24.5 Comprehension expressions

Implementing any non-trivial algorithm means performing operations on collections of elements: initialising, filtering, transforming. . .

Say that you need to build the set S of all even numbers less than 10; in mathematics, this is written using the set-builder notation – also called *set abstraction* or *set comprehension* or *set intension*:

$$S = \left\{ n \in [\![1, 9]\!] \;\middle|\; \exists k \in \mathbb{N} : n = 2k \right\}$$

So far, you know of two ways to define this set in Python. First, you can compute it yourself and write it *in extenso* – this is called a *display* expression in Python:

```
>>> S = { 0, 2, 4, 6, 8 }
```

Of course, that is only realistic for small collections, and is unworkable if it depends on another variable.

Second, you can write some code to programmatically compute it, by initialising the container and augmenting it in a loop:

```
>>> S = set()
>>> for n in range(10):
      if n%2==0:
        S.add(n)
>>> S
{0, 2, 4, 6, 8}
```

This is what you would do in most circumstances.

There is a third way, available in a few languages, Python included: *comprehension expressions*. The idea is to use pretty much the same set-builder notation as in mathematics, where useful, to achieve greater conciseness and clarity. For instance, we could write:

```
>>> { n for n in range(10) if n%2==0 }
{0, 2, 4, 6, 8}

>>> { 2*k for k in range(5) }
{0, 2, 4, 6, 8}
```

24.5.1 Comprehensions for every type; first contact with generators

Let us see how this works, and how flexible it is. The example so far was for sets, but this construction works for any collection type; at least with a few syntactic tweaks. The idea is that the delimiters determine the type of collection being created, and indeed that holds for sets, as we have seen, and lists as well:

```
>>> [n for n in range(10) if n%2==0]
[0, 2, 4, 6, 8]
```

This also holds for dictionaries, with a `key : value` syntax:

```
>>> {n : n**2 for n in range(10) if n%2==0}
{0: 0, 2: 4, 4: 16, 6: 36, 8: 64}
```

However, if you follow this pattern for tuples, you get

```
>>> (n for n in range(10) if n%2==0)
<generator object <genexpr> at 0x7fb79904d468>
```

instead. Despite not having a nice, textual representation, generators – or, more generally, iterators – are actually the central iterable structure powering every **for** loop behind the scene, and are so fundamental they get to use the basic `(..)` delimiters. To get a tuple, you have to use the tuple constructor and write:

```
>>> tuple(n for n in range(10) if n%2==0)
(0, 2, 4, 6, 8)
```

Using the type constructor in this way always works as expected.

```
>>> list(n for n in range(10) if n%2==0)
[0, 2, 4, 6, 8]

>>> set(n for n in range(10) if n%2==0)
{0, 2, 4, 6, 8}
```

What actually happens here is that this is equivalent to passing the generator as argument to the constructor:

```
>>> tuple((n for n in range(10) if n%2==0))
(0, 2, 4, 6, 8)
```

Since this is not a very nice-looking syntax, in contexts where you pass a generator expression as the single argument to a function, its parentheses can be omitted.

A small syntactic exception if for dictionaries: the `key : value` syntax is only a trick to distinguish set comprehensions from dictionary comprehensions. Using couples `(key, value)` would be ambiguous, as a set of couples does not have the same behaviour as a dictionary. What happens behind the scenes is that dictionary comprehensions are converted to `dict(..)` constructor calls on a collection of couples:

```
>>> {n:n**2 for n in range(10) if n%2==0}
{0: 0, 2: 4, 4: 16, 6: 36, 8: 64}

>>> dict(n:n**2 for n in range(10) if n%2==0)
SyntaxError: invalid syntax

>>> dict((n,n**2) for n in range(10) if n%2==0)
{0: 0, 2: 4, 4: 16, 6: 36, 8: 64}

>>> {(n,n**2) for n in range(10) if n%2==0}    # not the same thing
{(6, 36), (0, 0), (8, 64), (4, 16), (2, 4)}
```

In all cases, why does passing a generator to a constructor work? Because generators are iterable, and collection constructors just iterate over them to build the required collection, in the same way they would, say, a list, or a range.

We shall see what a generator *is*, exactly, in Sec. 29[p208]: “Iterables, iterators, and generators”.

In the meantime, let us just say that a generator is an object that produces values on demand, until they are exhausted:

```
>>> G=(n for n in range(5) if n%2==0)
>>> type(G)
<class 'generator'>
>>> next(G)
0
>>> next(G)
2
>>> next(G)
4
>>> next(G)
StopIteration # exception: the generator is exhausted
```

This is why they do not have any nice textual representation: you can't know that a generator will produce without asking them to produce a value, and doing so alters their state. Values are produced once, on demand, and are not stored in memory. Thus a generator can only be read once, and is then lost. This makes them very lightweight in memory, and enables us to have *infinite* generators as well.

24.5.2 Loop nesting in comprehensions

So far we have seen that comprehensions have two components: a generator expression, which does all the heavy lifting, and, optionally, delimiters – and, in the case of `dict`, a special `key : value` syntax – to specify which type constructor the resulting generator should be fed to.

Now, let us see what we can do with generator expressions. In

```
>>> [n // 2 for n in range(10) if n%2==0]
[0, 1, 2, 3, 4]
```

we use an expression `n // 2`, a **for** loop, which defines `n`, and a conditional expression. More complex expressions can involve additional variables and levels of nesting:

```
>>> [ (x,y) for x in 'ABCD' if x != 'D' for y in (0,1,2) ]
[('A', 0), ('A', 1), ('A', 2), ('B', 0), ('B', 1), ('B', 2),
  ('C', 0), ('C', 1), ('C', 2)]
```

Unlike in mathematics, however, you cannot put clauses in any order:

```
>>> [ (x,y) for y in (0,1,2) if x != 'D' for x in 'ABCD']
UnboundLocalError: local variable 'x' referenced before assignment
```

Is there a general rule of thumb to fully understand this syntax? Of course there is! The generator expression

```
[ (x,y) for x in 'ABCD' if x != 'D' for y in (0,1,2) ]
```

is actually “morally”[r] equivalent to a function containing the code

```
for x in 'ABCD':
    if x != 'D':
        for y in (0,1,2):
            yield (x,y)
```

We shall see the **yield** keyword in greater detail in Sec. 29.2[p211]: “**yield** and **yield from**”. Let us just say for now that it’s very much like a function’s **return**, but instead of completely exiting the function, it *pauses* it after yielding a value, so that you can later “unpause” it, causing it to resume execution to just after the **yield** keyword that paused it, until it yields another value and pauses again.

If **yield** is confusing you, think in terms of adding to a list for now:

```
L = []
for x in 'ABCD':
        if x != 'D':
            for y in (0,1,2):
                L.append( (x,y) )
print(L)
---------------------------------
[('A', 0), ('A', 1), ('A', 2), ('B', 0), ('B', 1), ('B', 2),
  ('C', 0), ('C', 1), ('C', 2)]
```

Seeing this, the general rule of thumb for comprehensions is clear: **you nest for and if clauses in exactly the same manner as though you were writing normal loops and conditionals**. The only things that change are that

(1) you write everything on one logical line, without opening a new block each time with a colon **:**

(2) the expression in the innermost **for** or **if** is put at the very beginning of the expression instead

(3) the outermost construct must be a **for**

(4) you cannot have an **else** clause in your **if**s.

Regarding the restriction on **else** clauses, do not forget that you can use the ternary operator syntax:

```
>>> [ "Even" if n%2==0 else "Odd"    for n in range(5) ]
['Even', 'Odd', 'Even', 'Odd', 'Even']
```

For legibility reasons, I would add space or indentation when using this construct to visually separate the ternary operator from the loops.

[r]. . . and also *technically*, but I have omitted a few steps to clarify the exposition here.

Of course, the logical line can be broken up following the usual rules; in te case of comprehensions, you can actually use line breaks pretty arbitrarily. For instance, the following works quite well:

```
L = [ (x,y) for x in 'ABCD'
            if x != 'D'
            for y in (0,1,2) ]
```

24.5.3 Common comprehension patterns

Let us see a few common comprehension patterns – and mistakes

24.5.3.1 Cartesian product

We have seen examples of cartesian products in the previous section already, without naming them as such.

As a reminder, the cartesian product of two sets A and B is the set

$$A \times B \;=\; \left\{ (x,y) \;\middle|\; x \in A,\; y \in B \right\},$$

and we generalise this notion to other structures, such as lists. This is written quite easily:

```
>>> [ (x,y) for x in (1,2,3) for y in (4,5,6) ]
[(1, 4), (1, 5), (1, 6), (2, 4), (2, 5), (2, 6),
  (3, 4), (3, 5), (3, 6)]
```

Note that you cannot write that using **and**:

```
>>> [ (x,y) for x in (1,2,3) and y in (4,5,6) ]
NameError: name 'y' is not defined
```

Comprehensions actually allow the use of Boolean expressions after **for .. in**, but the semantics is not what you would intuitively expect:

```
>>> [ x for x in (1,2,3) and (4,5,6) ]
[4, 5, 6]
```

To understand what is happening here, please refer to Sec. 22.6.5[p104]: “The semantics of **and** and **or**, & implicit Boolean conversion” on the semantics of **and** and **or** chains. The bottom line is, you will probably never have to use Boolean operators after **for .. in**. You may of course freely use them after **if**.

24.5.3.2 Mapping / element by element transformation

A common need is to transform a collection, applying a function to each of its elements: thus

$$e_1, \ldots, e_n$$

becomes

$$f(e_1), \ldots, f(e_n) \,.$$

In functional programming, we use the **map** higher-order function[s] for that. This exists in Python as well, but it is just as convenient to use comprehensions:

```
>>> def f(x): return f"f({x})"
>>> [ f(x) for x in "abc" ]
['f(a)', 'f(b)', 'f(c)']
```

Of course, you don't need to define a function for that, as the transformation can be performed directly in the comprehension's leftmost expression:

```
>>> [ f"g({x})" for x in "abc" ]
['g(a)', 'g(b)', 'g(c)']

>>> [ (x, x//3, x%3) for x in range(5) ]
[(0, 0, 0), (1, 0, 1), (2, 0, 2), (3, 1, 0), (4, 1, 1)]
```

24.5.3.3 Filtering

Another common need is to filter a collection C, keeping only those elements that satisfy a predicate P: C becomes – using set notation –

$$C' = \left\{ e \in C \,\middle|\, P(e) \right\}$$

In functional programming, there is a higher-order **filter** function for that, and again, it exists in Python, but is just as convenient using comprehensions, where we have for instance:

```
>>> def P(x): return 65 <= ord(x) <= 65+26
>>> [ c for c in "loUPwPerER" if P(c) ]
['U', 'P', 'P', 'E', 'R']
```

There again, defining a separate predicate is not at all necessary:

```
>>> [ c for c in "loUPwPerER" if 97 <= ord(c) <= 97+26 ]
['l', 'o', 'w', 'e', 'r']
```

24.5.3.4 Reductions

It is common to want to reduce a collection to a single value. Of course that value can be something complex, like another collection, but often it is something simple, like an integer. Either way, in functional programming, there is a `reduce` higher-order function for that –

[s] You may indirectly have heard of it in the context of Map/Reduce frameworks.

also extent in Python, though somewhat hidden. Here we focus on some simple reductions, using comprehensions.

First the sum function:

$$\mathtt{sum}(C) \quad = \quad \sum_{e \in C} e \,.$$

This very useful function has the good taste to be predefined in Python. It takes any iterable collection as argument. This means that, like the list, tuple, and other collection type constructors, it can be used quite elegantly with generator expressions. Let us compute

$$\sum_{k=0}^{10} k :$$

```
>>> sum( k for k in range(10+1) )
55
```

It would be a bad idea, although not a *wrong* one, to create a list to pass as argument to sum – or any function taking an iterable:

```
>>> sum( [ k for k in range(10+1) ] )
55
```

Not only is this less legible, but behind the scene you now need to compute all the values of k, and store them all in memory in a list, before even staring the computation of the sum.

In contrast, the first version computed the values of k one at a time, incrementing the sum along the way, in the same efficient way you presumably would if you wrote the code for $\sum_{k=0}^{10} k$ directly.

Rule of thumb: If you are writing a comprehension expression for the purpose of a reduction, always use generators rather than wrapping it into an intermediary structure.

There is no predefined prod function that performs the product:

$$\mathtt{prod}(C) \quad = \quad \prod_{e \in C} e \,.$$

That should not stop us from defining one, and computing

$$5! \quad = \quad \prod_{k=1}^{5} k \quad = \quad 1 \times 2 \times \cdots \times 5 \quad = \quad 120 :$$

```
def prod(C):
    r = 1 # neutral element for *; just like 0 for +
    for e in C: r *= e
    return r
```

```
>>> prod(range(1, 5+1))
120

>>> prod( 1/k for k in range(1,5+1) )
0.008333333333333333

>>> 1/120
0.008333333333333333
```

Two interesting reductions are the builtin `min` and `max` functions, which, interestingly, can be used either as variadic functions, returning a minimal/maximal element out of their arguments, or as a reduction on a single iterable:

```
>>> min(1)
TypeError: ’int’ object is not iterable
>>> min(3,1,2)
1
>>> min([3,1,2])
1
```

Like `sorted(..)` and `list.sort()`, they take an additional `key` argument to define the order according to which the notion of minimal/maximal element should be defined:

```
>>> l = [’Platypus’, ’School’, ’Sleep’] # in the usual order

>>> min(l), max(l)
(’Platypus’, ’Sleep’)

>>> sorted(l,key=lambda s:s[2])
[’Platypus’, ’Sleep’, ’School’]

>>> max(l,key=lambda s:s[2])
’School’
```

Let us use this to find, for instance, the index of the minimal element of a list:

```
>>> l = [75, 76, 99, 74, 11, 98, 85, 7, 5, 87]

>>> min(l)
5
>>> l.index(5)
8
>>> min(range(len(l)), key=lambda i:l[i])
8
```

The most frequent element in a list:

```
>>> l = [’A’, ’C’, ’B’, ’B’, ’C’, ’C’, ’C’, ’C’, ’C’, ’A’]
>>> max(set(l), key=l.count)
'C' # l would work instead of set(l), but less efficient
```

Be mindful of the fact that not all objects are *totally*, or *linearly*, ordered. Thus there is not always *a* smallest, or greatest, element. Sets for instance, are only partially ordered with respect to ⊆. min and max return *a* minimal/maximal element; in fact the first one they encounter. In the case of partial orders, the actual order in which elements are encountered matters, and reordering the arguments or the collection can change the result:

```
>>> min({0},{0,1})
{0}

>>> min({0,2},{0,1})
{0, 2}

>>> min({0,1},{0,2})
{0, 1}
```

Another common reduction is the concatenation of multiple strings, provided by the str.join method:

```
>>> "".join( chr(k) for k in range(65,65+26) )
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

For Booleans, there are the two functions all and any, acting on iterable collections of (implicit or explicit) Booleans:

$$\mathtt{all}(C) \;=\; \bigwedge_{b\in C} b \;=\; \forall b \in C,\, b\,.$$

and

$$\mathtt{any}(C) \;=\; \bigvee_{b\in C} b \;=\; \exists b \in C : b\,.$$

```
>>> all([])
True  # neutral element for *and*
>>> all([True, False, True, True])
False
>>> all([True, True, True])
True

>>> any([])
False # neutral element for *or*
>>> any([True, False, True, True])
True
>>> any([False, False])
False
```

Of course, they can be used with any types of values, with implicit Boolean conversion:

```
>>> any([ (), () ])
False
>>> any([ (), (1,2) ])
True
```

We have seen a typical use in the section on assertions:

```
assert all( square(n) == n*n  for n in range(10) )
```

This compactly translates the statement:

$$\forall n \in [\![0, 9]\!],\ \mathtt{square}(n) = n^2$$

24.5.3.5 *Index manipulation, permutations*

You can use comprehension expressions to act on indexes as well as elements: for instance, the inversion

```
>>> l = list(range(5))

>>> [ l[-i-1] for i in range(5) ]
[4, 3, 2, 1, 0]
```

Of course inversion in particular can be done more elegantly with the slice `l[::-1]`.

Let us try more general permutations. The application of the permutation σ on a list $l = [e_1, \dots, e_n]$ is the list

$$l\sigma^{-1} = \left[e_{\sigma^{-1}(1)}, \dots, e_{\sigma^{-1}(n)}\right]$$

Let

$$\sigma = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 1 & 2 & 0 & 4 & 3 \end{pmatrix}$$

be a permutation – in Cauchy's notation – on $[\![0, 4]\!]$. That is to say, we send the element of index 0 onto index 1, the element of index 1 onto index 2, the element of index 2 onto index 0, and exchange the last two.

```
l = list("ABCDE")

def σ⁻¹(i): return (1,2,0,4,3).index(i) # get the index where i appears
# I use σ⁻¹ for clarity. In actual code, use a valid identifier

lσ⁻¹ = [ l[σ⁻¹(i)] for i in range(5) ]

print( lσ⁻¹ )
```

```
----------------------------------------------------------------

['C', 'A', 'B', 'E', 'D']
```

24.5.3.6 *Flattening a sequence of sequences*

Say that you have a sequence of sequence, and want to flatten all those sequences into one, concatenating them all. You can do so easily with a comprehension:

```
>>> l = [ (1,2,3), (4,5), (6,7,8) ]

>>> [ e for sl in l for e in sl ] # sl = sublist
[1, 2, 3, 4, 5, 6, 7, 8]
```

24.5.3.7 *Element repetition / stutter*

Sometimes, you want to "stutter" a sequence, repeating an element a number of times. It is not immediately trivial how to do so:

```
>>> [ (c,c,c) for c in list("ABCD") ]
[('A', 'A', 'A'), ('B', 'B', 'B'), ('C', 'C', 'C'), ('D', 'D', 'D')]
```

does not give the expected result by itself. You still need to flatten that. There is, to my knowledge, no magical syntactic trick to automatically flatten the tuple:

```
>>> [ c,c,c for c in list("ABCD") ]
SyntaxError: invalid syntax

>>> [ *(c,c,c) for c in list("ABCD") ]  # unpacking attempt
SyntaxError: iterable unpacking cannot be used in comprehension
```

However, repetition is actually very easy to write, with the right idea:

```
>>> [ c for c in list("ABCD") for _ in range(3) ]
['A', 'A', 'A', 'B', 'B', 'B', 'C', 'C', 'C', 'D', 'D', 'D']
```

In this code, _ is just the name one traditionally uses for a variable whose name does not actually matter, since one does not actually ever use its value. It could as well have been called `i`.

24.6 Packing and unpacking

24.6.1 Starred expressions: sequence types

At the most basic level, *packing* is the act of grouping several elements in a sequence type, like so

```
>>> t = ("Hello", 24, True)
```

and *unpacking* is to “explode” the package back into its individual components, through pattern matching:

```
>>> s,i,b = t      # or, equivalently, (s,i,b) = t
>>> print(s,i,b)
Hello 24 True
```

The pattern-matching works on any sequence type – concretely, that means lists and tuples – but not on unordered types, as it would be ambiguous:

```
>>> [s,i,b] = t
>>> s,i,b
(’Hello’, 24, True)

>>> {s,i,b} = t
SyntaxError: can’t assign to literal
```

The * operator enables the programmer to selectively pack or unpack elements in some contexts:

```
>>> r = range(7)
>>> first, second, *middle, end = r

>>> first
0
>>> second
1
>>> middle
[2, 3, 4, 5]
>>> end
6

>>> [type(e) for e in (first,second,middle,end)]
[<class ’int’>, <class ’int’>, <class ’list’>, <class ’int’>]
```

Here, we have used * to pack the middle elements in a list while unpacking others. As you can see, this is quite a convenient way of isolating the first and last few elements from the rest. The alternative would be to write multiple slices

```
>>> r = range(7)

>>> first   = r[0]
>>> second  = r[1]
>>> middle  = r[2:-1]
>>> end     = r[-1]

>>> (first, second, middle, end)
```

```
(0, 1, range(2, 6), 6)

>>> (first, second, list(middle), end)
(0, 1, [2, 3, 4, 5], 6)
```

Let us note here that *-packing provides us with a list, regardless of the initial type of the container being unpacked:

```
>>> s = set(range(5))

>>> a, *rest = s

>>> a, rest
(0, [1, 2, 3, 4])
```

In contrast, slicing preserves the type of the underlying indexable. Note as well that *-packing works on all iterables, whereas slices require indexable types. In the code above, a is only first in the arbitrary order of iteration – we merely isolated an arbitrary element from the rest, similar to a call to the `pop()` method, but without altering the original set. We could not have used slices to achieve the same effect, as sets are not indexable:

```
>>> a = s[0] ; rest = s[1:]
TypeError: 'set' object does not support indexing
```

Only a single starred expression can appear on the left-hand side. The reason for this restriction should be pretty clear: how the two starred variables should share the elements would be difficult to define.

```
>>> *a,*b = r
SyntaxError: two starred expressions in assignment
```

We have covered the uses of * in left-hand sides of assignments. Let us now see where we may find them in expressions.

In a comma-separated element enumeration context – this is to say, when enumerating the elements of a list, tuple, or set, or when writing arguments to a function call – a * can be used to unpack a collection directly within the enumeration:

```
>>> r = range(5)

>>> ['A', *r, 'Z']
['A', 0, 1, 2, 3, 4, 'Z']

>>> {'A', *r, 'Z'}
{0, 1, 'A', 2, 3, 4, 'Z'}

>>> print('A', *r, 'Z', sep=',')
A,0,1,2,3,4,Z
```

For the last line, recall that print is a variadic function, printing the separator sep between each two of its successive arguments. Its output shows that each element of r was passed as a separate argument to the function.

Contrary to the restrictions of left-hand side packing, in the context of expression unpacking there is no restriction to how many starred expressions may appear:

```
>>> print(*{3.15, 6.7, 90.}, *r, ’A’, *r, ’Z’, sep=’ , ’)
90.0 , 3.15 , 6.7 , 0 , 1 , 2 , 3 , 4 , A , 0 , 1 , 2 , 3 , 4 , Z
```

The only restriction is of course that any function call must in the end be made with the right number of arguments. **print** being variadic, we have not run into this:

```
>>> def f(a,b,c): print(a,b,c)

>>> f(*[1,2], 3)
1 2 3

>>> f(*[1,2,3,4])
TypeError: f() takes 3 positional arguments but 4 were given
```

24.6.2 Doubly-starred expressions: dictionaries

What about dictionaries?

```
>>> age = {’Toto’:15, ’Tata’:27, ’Mamie’:97 }

>>> print(*age)
Toto Tata Mamie
```

As you can see, single star syntax unpacks the list of the dictionary’s keys – a behaviour entirely consistent with the view that, for iteration purposes, a dictionary is assimilated to the set of its keys.

However, this usual single-star syntax does not work well in the context of dictionary item enumeration:

```
>>> { ’Jojo’:15 , *age }
SyntaxError: invalid syntax
```

This is to be expected; in this context, both keys and values are needed. For this, there is a special double-star unpacking syntax that does work as hoped:

```
>>> { ’Jojo’:15 , **age }
{’Jojo’: 15, ’Mamie’: 97, ’Tata’: 27, ’Toto’: 15}
```

However, it works only in that context, and does not generate, say, key/value couples usable in classical contexts:

```
>>> [*age]
['Toto', 'Tata', 'Mamie']

>>> [**age]
SyntaxError: invalid syntax
```

Nor can it be used in left-hand sides:

```
>>> {'Mamie': 97, **d } = age
SyntaxError: can't assign to literal
```

This stands in contrast to single-star packing:

```
>>> [ a, *b ] = age
>>> b
['Tata', 'Mamie']
```

Working only in the context of dictionary item enumeration does not make double-star syntax nearly so niche a tool as you would expect, though, as dictionaries are present in unexpected places. . .

Consider the following code, where we define two functions with the same named arguments, which, incidentally, correspond to the keys of our toy dictionary:

```
def f(Toto, Tata, Mamie):
  print(Toto, Tata, Mamie)

def g(Mamie, Toto, Tata):
  print(Toto, Tata, Mamie)
```

Recall how we used the single-star unpacking syntax in function calls to pass positional arguments. We can actually do the same thing with double-star unpacking syntax to pass keyword arguments, this time:

```
>>> f(**age)
15 27 97
>>> g(**age)
15 27 97
```

What we have done here is equivalent to having written

```
>>> f(Toto=age['Toto'], Tata=age['Tata'], Mamie=age['Mamie'])
15 27 97
```

Internally, functions actually use dictionaries to handle the values of their keyword arguments, and we make use of that fact.

Be careful not to confuse single- and double-starred unpacking: as f accepts both positional and keyword arguments, it is also possible to unpack the set of keys in a function call, with very different results:

```
>>> f(*age)
Toto Tata Mamie
>>> g(*age)
Tata Mamie Toto
```

Note that, in f(**age) and g(**age), the order of arguments does not matter — which is fortunate, since dictionaries are unordered structures anyway. Recall that order does not matter either when arguments are passed by keyword — which is one of the selling points of calling functions in such a way, as it avoids some silly programming mistakes.

The f(argname=argvalue, ...) syntax could in principle have been defined as f('argname' : argval instead, and though the latter is not actually valid python syntax, it is still "morally" equivalent to what takes place.

Some standard dictionary methods make use of that, such as the update method, which updates a dictionary in-place according to the contents of another one, passed as a argument:

```
>>> age.update( { 'a' : 1, 'b': 2 } )
>>> age
{'Toto': 15, 'Tata': 27, 'Mamie': 97, 'a': 1, 'b': 2}
```

Thanks to the implementation of update, the above could equally well be written:

```
>>> age.update(a=1,b=2)
>>> age
{'Mamie': 97, 'Tata': 27, 'Toto': 15, 'a': 1, 'b': 2}
```

We shall see how such a function might be defined in more detail in the Sec. 25.1[p171]: "Variadic function definition".

24.6.2.1 *Merging two dictionaries*

If you want to (non-recursively) merge two dictionaries into a new one without altering the original, you can **(1)** in Python 3.9+, simply use the union (|) operator, as for sets **(2)** prior to 3.9, to obtain the same behaviour, unpack both of them into a fresh one:

```
>>> age = {'Toto':15, 'Tata':27, 'Mamie':97 }
>>> d = { 'AA' * i : 10*i for i in range(1,3+1) }
>>> d
{'AA': 10, 'AAAA': 20, 'AAAAAA': 30}

>>> { **age, **d }
{'Toto': 15, 'Tata': 27, 'Mamie': 97, 'AA': 10, 'AAAA': 20, 'AAAAAA': 30}
```

Note that, should the two dictionaries share keys, associations in `d` will overwrite those in `age`:

```
>>> { **{0:1}, **{0:2} }
{0: 2}
```

In some sources, I have seen this pattern recommended instead:

```
>>> dict(**age, **d)  # or dict(age, **d)
{'Toto': 15, 'Tata': 27, 'Mamie': 97, 'AA': 10, 'AAAA': 20, 'AAAAAA': 30}
```

Yes, it also works for that purpose, but only to some extent. But why does it work at all? Because, like `update`, `dict()` treats its keyword arguments as dictionary associations, for the purpose of building a new dictionary:

```
>>> dict(a = 1, b = 2)
{'a': 1, 'b': 2}
```

This makes manually defining dictionaries with string keys a little more elegant. But beware: the keyword argument syntax is not as flexible as the `k:v` dictionary item syntax:

```
>>> dict(a = 1, b = 2)
{'a': 1, 'b': 2}

>>> dict(1=a, 2=a)
SyntaxError: keyword can t be an expression

>>> dict(**{1 : 'a' , 2 : 'b'})
TypeError: keyword arguments must be strings

>>> dict(**{'a':0},**{'a':1})
TypeError: type object got multiple values for keyword argument 'a'
```

Thus this method of merging will fail on dictionaries whose keys are not strings, or who have keys in common.

The `{**d1, **d2}` syntax is very clear and very efficient. Use it. The only real alternative is to copy the first dictionary, then update that with the second; this is both much less elegant, and almost twice as slow in tests. Comprehension flattening patterns or `itertools.chain` methods are even slower.

25 Advanced function definitions

25.1 Variadic function definition

This follows Sec. 21.5.4[p67]: "Defining functions; predicates and procedures" and Sec. 21.5.7[p70]: "Optional arguments".

Variadic functions such as print are pretty neat, but so far we have not seen how we can define our own. Let us come back to function definitions, in full generality; we shall need dictionaries and packing/unpacking.

Recall the last pattern we gave for function definitions, including optional parameters:

```
def <functionname> (<arg1>, ..., <argN>,
                    <optarg1> = <defval1>,... <optargM> = <defvalM>):
```

So far, all of our arguments, whether mandatory or optional, could be passed either as keyword or positionally. We shall see that, in all generality, arguments can also be exclusively positional or exclusively keyword.

The more general definition pattern is either of two patterns below, where I name the parameters according to the following conventions:

`pk`	positional and keyword parameters without default value
`pkd`	positional and keyword parameters with default value (optional parameters)
`p`	exclusively positional parameters — traditionally written `args`
`k`	exclusively keyword parameters — traditionally written `kwargs`

```
def <fun> (
           <pk1>, ..., <pkN>,
           <pkd1> = <def1>,..., <pkdM> = <defM>,
           *<p>,
           **<k>
          ):
```

or, perhaps more rarely,

```
def <fun> (
           <pk1>, ..., <pkN>,
           *<p>,
           <pkd1> = <def1>,..., <pkdM> = <defM>,
           **<k>
          ):
```

In both cases, all variables are assigned a value during a function call, according to the following discipline:

- the optional parameters <pkd_> have their default values, if not overridden later.
- the first `N` positional arguments are affected, in order, to positional parameters <pk_>.
- excess positional arguments are all affected, as a tuple, to <p>. If there is no such excess, <p> is the empty tuple. This is the nub of variadic functions, such as **`print`**.

  <p> absorbs this excess at the point where it appears in the definition. This means that in the first form, excess positional arguments first override the default values of the <pkd_>, whereas in the second form, the defaults can only be overridden by keyword arguments.
- then come keyword arguments, which can specify the value of missing positional arguments <pk_> or override the default value of optional arguments <pkd_>.
- any excess keyword arguments, that is to say arguments neither <pk_> nor <pkd_>, are then absorbed, as a dictionary, by <k>, which is the empty dictionary by default.

Let us illustrate all this with examples: let us define a function according to the first form:

```
def f( pk1, pk2,
       pkd1='d1', pkd2='d2',
       *p, **k
     ):
    print("f ",pk1,pk2,pkd1,pkd2,p,k)
```

We have:

```
>>> f(1,2)          # ‘‘normal’’ positional call
f  1 2 d1 d2 () {}

>>> f(pk2=2,pk1=1) # keyword call
f  1 2 d1 d2 () {}

>>> f(1,pk2=2)      # partial positional, partial keyword
f  1 2 d1 d2 () {}

>>> f(1,pkd2=99,pk2=2) # overriding default via keyword
f  1 2 d1 99 () {}

>>> f(1,pkd2=99,pk2=2,a=78) # excess keyword is absorbed
f  1 2 d1 99 () {'a': 78}

>>> f(1,2,3)        # positional; override some defaults
f  1 2 3 d2 () {}

>>> f(1,2,3,4,5,6) # override all defaults; excess positionals absorbed
f  1 2 3 4 (5, 6) {}

>>> f(1,2,3,4,5,6,a=90,b=55) # same; excess keywords absorbed
f  1 2 3 4 (5, 6) {'a': 90, 'b': 55}
```

And now with the second form:

```
def g( pk1, pk2, *p,
       pkd1='d1', pkd2='d2', **k
     ):
    print("g ",pk1,pk2,p,pkd1,pkd2,k)
```

We have:

```
>>> g(1,2)
g  1 2 () d1 d2 {}

>>> g(pk2=2,pk1=1)
g  1 2 () d1 d2 {}

>>> g(1,pk2=2)
g  1 2 () d1 d2 {}

>>> g(1,pkd2=99,pk2=2)
g  1 2 () d1 99 {}

>>> g(1,pkd2=99,pk2=2,a=78)
g  1 2 () d1 99 {'a': 78}

>>> g(1,2,3)    # excess positional absorbed: defaults untouched
g  1 2 (3,) d1 d2 {}

>>> g(1,2,3,4,5,6) # defaults cannot be overridden positionally
g  1 2 (3, 4, 5, 6) d1 d2 {}

>>> g(1,2,3,4,5,6,pkd2=777) # but can be overridden by keyword
g  1 2 (3, 4, 5, 6) d1 777 {}

>>> g(1,2,3,4,5,6,a=90,b=55)
g  1 2 (3, 4, 5, 6) d1 d2 {'a': 90, 'b': 55}
```

Let us now illustrate that by writing our own toy versions of **print** and dict(..):

```
def myprint(*args):
    for a in args:
        print(a,"",end="")
    print() # newline
```

We have:

```
>>> myprint()

>>> myprint(1)
1
>>> myprint(1,2)
1 2
```

```
>>> myprint(*range(5))
0 1 2 3 4
```

Now for `dict(..)`, we have:

```
def mydict(**kwargs):
    return kwargs
-----------------------------------------------------
>>> d,D = dict(a=1,b=2), dict(c=3,d=4)

>>> mydict(**d,**D)
{'a': 1, 'b': 2, 'c': 3, 'd': 4}

>>> mydict(**d,**D,another='stuff')
{'a': 1, 'b': 2, 'c': 3, 'd': 4, 'another': 'stuff'}
```

With this, we have covered function definition in all generality, except for two advanced features: decorators, and annotations.

25.2 Function decorators

Function decorators are a means of altering how a function, method, or class works, without modifying its definition — or even needing access to it. Examples include automatically logging function calls and debug information, measuring time elapsed — cf. Sec. 31.4[p221]: "Examples and performance tests" — automatic memoisation — cf. question (233)[p355] and functools.cache, — defining read-only attributes, getters and setters[t], and more.

Fundamentally, a decorator is nothing more than a higher order function, taking a function as input and yielding another function as output. Python supports a special "`@`" syntax making this process more elegant, but it is not required.

25.2.1 A plain decorator

For a first contact, we shall not use any special syntax at all. Let us make a decorator `announce_call` that modifies the behaviour of a function by printing "Hello" and "Goodbye" before and after it is executed. While this is a bit silly in and of itself, you can substitute those printing operations for any kind of pre- and post-processing you want; think of this as a template for more useful decorators.

The general idea of decoration is to "wrap" the function inside another function, called a *wrapper*. The wrapper is in charge of calling the original function, to which it passes along any argument it receives. Before and after that, it does whatever it needs to do; here, printing. The decorator then returns the wrapper, which in effect becomes the decorated function.

```
def announce_call(f):
    def wrap(*a,**k):
```

[t] See `https://docs.python.org/3/library/functions.html#property`.

```
        print("Hello!")
        res = f(*a,**k)
        print("Goodbye!")
        return res
    return wrap
```

To use it, without any pretty syntax, we can define our function as usual, and after that rebind it to the modified version of itself.

```
def e(x): return  f"e({x})"
e = announce_call(e)

---------------------------

>>> e
<function announce_call.<locals>.wrap at 0x7fa6758c7be0>

>>> e(1)
Hello!
Goodbye!
'e(1)'
```

It works as expected, of course, but is a tad cumbersome: the name of the altered function is repeated three times. Instead, you can and should use the special @-syntax for decorators: simply write `@<your decorator>` on the line preceding the definition of your function.

```
@announce_call
def f(x): return  f"f({x})"

@announce_call
def g(x): return f"g({x})"

print([f(x) + g(x) + g(x) for x in range(2)])

---------------------------------------------

Hello!
Goodbye!
Hello!
Goodbye!
Hello!
Goodbye!
Hello!
Goodbye!
Hello!
Goodbye!
Hello!
Goodbye!
['f(0)g(0)g(0)', 'f(1)g(1)g(1)']
```

25.2.2 Decorators with their own states

Now let us write a more sophisticated decorator `count_calls`, such that functions not only say hello and goodbye, but also say their name, display their arguments, as well as their return values on goodbye, and count the total number of times they have been called; each function must have its own independent counter.

Getting the name of a function is easy: a function is an object with a `__name__` attribute that contains just this information.

The counter is more delicate. This can be done in an OOP style, defining a callable class with a counter attribute, but this is much simpler and cleaner in a more functional style, using a lexical closure.[u]

The idea is that the decorator is called once, and once only, for each decorated function — at definition time — while the wrapper is called once for each call of the decorated function. With good reason, since the wrapper *is* the decorated function.

Thus, our counter shall be a variable initialised by the decorator, and used, **`nonlocal`**ly, by the wrapper. Each wrapper shall therefore have its own independent counter.

```
def count_calls(f):
    i = 0
    def w(*a,**k):
        nonlocal i
        i += 1
        call = f"{f.__name__}[{i}]({','.join(map(repr,a+tuple(k.items())))})"
        print(f"@< {call}...")
        res = f(*a,**k)
        print(f"   {call} = {repr(res)} >@")
        return res
    return w

@count_calls
def f(x): return f"f({x})"
@count_calls
def g(x): return f"g({x})"

print([f(x) + g(x) + g(x) for x in range(2)])

----------------------------------------------------

@< f[1](0)...
   f[1](0) = 'f(0)' >@
@< g[1](0)...
   g[1](0) = 'g(0)' >@
@< g[2](0)...
   g[2](0) = 'g(0)' >@
@< f[2](1)...
```

[u] Using on-the-fly function attributes is also a possibility: `f.my_counter = ....`.

```
   f[2](1) = 'f(1)' >@
@< g[3](1)...
   g[3](1) = 'g(1)' >@
@< g[4](1)...
   g[4](1) = 'g(1)' >@
['f(0)g(0)g(0)', 'f(1)g(1)g(1)']
```

This is a nice little debugging decorator; let us apply it to a function with a more complex behaviour: the Fibonacci sequence:

```
@count_calls
def fib(n):
    return n if n <= 1 else fib(n-1) + fib(n-2)

print(fib(3))

-----------------------------------------------

@< fib[1](3)...
@< fib[2](2)...
@< fib[3](1)...
   fib[3](1) = 1 >@
@< fib[4](0)...
   fib[4](0) = 0 >@
   fib[2](2) = 1 >@
@< fib[5](1)...
   fib[5](1) = 1 >@
   fib[1](3) = 2 >@
2
```

25.2.3 Decorating wrappers to preserve function metadata

The use of decorators has an undesirable side effect, in that decorated functions forget their original identity: recall that they *are* their wrapper. Let us consider a well-documented little function, that happens to be decorated:

```
@count_calls
def helpful():
    "I'm well-documented!"
    return "Happy!"

---------------------------------------------------

>>> helpful
<function count_calls.<locals>.w at 0x7fddbda6fa30>

>>> helpful.__name__
'w'
```

```
>>> help(helpful)
Help on function w in module __main__:

w(*a, **k)
```

The nice documentation has been lost, along with the name of the function. Bad decorators! What is the solution to bad decorators, you ask? Why, *more* decorators, of course!

Let us import the `wraps` decorator from the `functools` module:

```
from functools import wraps
```

Now we can decorate our wrapper functions to overwrite their metadata — name, documentation — by that of the function they decorate, which is passed as argument to `wraps`; let us use that in `count_calls`:

```
def count_calls(f):
    i = 0
    @wraps(f)
    def w(*a,**k):
        nonlocal i
        ...
```

This time all helpful information is preserved:

```
>>> helpful
<function helpful at 0x7fe59e3bba30>
>>> helpful.__name__
'helpful'
>>> help(helpful)
Help on function helpful in module __main__:

helpful()
    I'm well-documented!
```

As a bonus, `wraps` adds a `__wrapped__` attribute to the wrapper it decorates, ponting to the *original*, undecorated function. This can be useful; maybe we do not want to count *some* calls in the case of `count_calls`; maybe we want to bypass the cache in the case of `memoise`, etcetera.

Let us demonstrate:

```
>>> helpful()
@< helpful[1]()...
   helpful[1]() = 'Happy!' >@
'Happy!'

>>> helpful.__wrapped__
<function helpful at 0x7f3aad8a7d90>
```

```
>>> helpful.__wrapped__()
'Happy!'

>>> helpful()
@< helpful[2]()...
   helpful[2]() = 'Happy!' >@
'Happy!'
```

Decorators may be chained; this presents no difficulty whatsoever:

```
@announce_call
@count_calls
def helpful():
    ...

--------------------------

>>> helpful()
Hello!
@< helpful[1]()...
   helpful[1]() = 'Happy!' >@
Goodbye!
'Happy!'
```

25.2.4 A fun decorator: trace

Let us come back to the Fibonacci function. Here is a cool decorator to print its call tree — or indeed that of any function:

```
def trace(f):
    lvl = 0
    @wraps(f)
    def w(*a,**k):
        nonlocal lvl
        tree = "|  " * lvl + "+"
        print(f"{tree} {f.__name__}({', '.join(map(repr,a))})")
        lvl += 1
        res = f(*a,**k)
        lvl -= 1
        return res
    return w

@trace
def fib(n):
    return n if n <= 1 else fib(n-1) + fib(n-2)

print(fib(5))

----------------------------------------------------------------
```

```
+ fib(5)
|  + fib(4)
|  |  + fib(3)
|  |  |  + fib(2)
|  |  |  |  + fib(1)
|  |  |  |  + fib(0)
|  |  |  + fib(1)
|  |  + fib(2)
|  |  |  + fib(1)
|  |  |  + fib(0)
|  + fib(3)
|  |  + fib(2)
|  |  |  + fib(1)
|  |  |  + fib(0)
|  |  + fib(1)
5
```

Do you understand why and how it works?

Note: I do not have the necessary LaTeX font to display that here, but the tree can be made much prettier by using Unicode ordinal 9474 in place of | and ordinals 9500 and 9472 in place of +.

25.2.5 Parametric decorators

What if you need to have a decorator that takes parameters? For instance, let us write a decorator `who_says`, behaving much like `announce_call`, except that the messages it prints can be customised by a parameter:

```
@who_says("Simon")
def f(x): return  f"f({x})"

print(f(0))

----------------------------

Simon says Hello!
Simon says Goodbye!
f(0)
```

To implement that, we are going to nest functions three levels deep . . . but don't panic, it is actually all fairly straightforward when you think about it.

In order for things to work properly, the expression `who_says("Simon")` must be a decorator. Therefore, `who_says` must be a function that takes an argument and returns a decorator. The decorator itself is done as usual, by declaring and returning a wrapper function.

Thus we have a pattern:

```
def deco_with_params(params):
    def deco(f):
        @wraps(f)
        def wrap(*a,**k):
            ... params ... f(*a,**k)...
        return wrap
    return deco
```

Let us apply it to our problem:

```
def who_says(who):
    def deco(f):
        @wraps(f)
        def wrap(*a,**k):
            print(who, "says Hello!")
            res = f(*a,**k)
            print(who, "says Goodbye!")
            return res
        return wrap
    return deco
```

We get the expected behaviour.

25.2.6 Class decorators

Decorators can be used on classes, and can be — callable — classes. This is outside the scope of this document.

26 Reading and writing files

For now I shall just give you the bare minimum you will need in the exercises.

Files are opened with the **open** function, which is quite rich in functionality. Let us say that `mytext.txt` is an existing text file. It contains the text:

```
This␣is
a
text␣file.
```

Let us open it; by default, we are in *read-only* mode:

```
>>> f = open("mytext.txt")

>>> f
<_io.TextIOWrapper name='mytext.txt' mode='r' encoding='UTF-8'>
```

Once the file is open, it acts as an iterator on the text's lines, which is both convenient and very efficient, as we do not load all the file in memory at once, but just a single line of it. By default, line returns are normalised to `'\n'` regardless of the file's own CR/LF style.

```
>>> [l for l in f]
['This is\n', 'a\n', 'text file.\n']
```

Note that doing so moves the imaginary reading head in the file, and thereafter the stream is exhausted. If you want to read the file again, you need to position the stream back at the beginning, which can be done with `seek(0)`:

```
>>> [l for l in f]
[]

>>> next(f)
StopIteration

>>> f.seek(0)
0 # I'm now at position zero. Again.

>>> [l for l in f]
['This is\n', 'a\n', 'text file.\n']
```

The `str.split` method is very convenient, in conjunction with line iteration, to extract information from text files:

```
>>> [l.split() for l in f]
[['This', 'is'], ['a'], ['text', 'file.']]
```

Alternatively, you *can* read all the text at once:

```
>>> f.read()
'This is\na\ntext file.\n'
>>> f.read()
'' # stream is exhausted
```

Note that both methods rely on the same stream:

```
>>> f = open("mytext.txt")
>>> [l for l in f]
['This is\n', 'a\n', 'text file.\n']
>>> f.read()
''

>>> f.seek(0)

>>> f.read()
'This is\na\ntext file.\n'
>>> [l for l in f]
[]
```

When you are done, close the file.

```
>>> f.close()
```

Writing is much the same; you must specify write-mode by passing `"w"` to **`open`**. The file need not exist yet. In write mode, you have, unsurprisingly, access to the `write` method:

```
>>> g = open("blah.txt","w")

>>> g.write('This is\na\ntext file.\n')
21 # number of characters written
   # always equal to the length of the string.

>>> g.close()
```

26.1 The `with .. as` statement, for files

The recommended way to handle files is through the use of context managers and the **`with .. as`** statement.

Without going into detail or generalities, the idea is to minimise the work that needs doing to ensure your files are properly closed and you don't leak file descriptors.

`with` introduces a code block, and you can open as many files as you want at once:

```
with open(ROfile) as f, open(RWfile,"w") as g:
  <block with f and g opened>
```

Within this block, you can do whatever you want with `f` and `g`. Any variable defined with this block will remain accessible outside of it.

The point of this construction is that, no matter what happens, whether you exit the block normally or exceptions are raised, the files are properly closed. Thus you don't need to close them manually or worry about them much.

This is a good pattern. Use it.

27 Object Oriented Programming in Python

This continues Sec. 21.2[p62]: "A few words about Object Oriented Programming (OOP)".

You will have a course on OOP (in Java) during the first semester. While the general concepts are the same, the details can change significantly from one language to the next.

27.1 Empty class, dynamic attributes

Let us define our own class, representing a person. For now, let us start by defining an empty class we can play around with:

```
class Person:
    pass
```

Now that we have defined a class, we can *instantiate* it into *objects*:

```
>>> p = Person()
>>> p
<__main__.Person object at 0x7f82091eab70>
>>> type(p)
<class '__main__.Person'>
```

__main__ is of course the name for the current, main module. The class we just defined belongs to it.

Now that we have an object, we can play around with it. For now, it has no attributes, but in Python, you can *sometimes* define those on the fly:

```
>>> p.a
AttributeError: 'Person' object has no attribute 'a'

>>> p.a = 21  # on-the-fly new attribute. It works here,
              # but not on all objects.
>>> p.a
21

>>> del p.a

>>> p.a
AttributeError: 'Person' object has no attribute 'a'
```

27.2 Why we have constructors

Let us add some attributes, with default values, to our class:

```
class Person:
    name = ''
    age = 0
```

Playing around with instances, we see that each object has those default values, and its own namespace once defined:

```
>>> p, q = Person(), Person()
>>> p.name
''
```

```
>>> p.name = "Toto"
>>> p.age = 25

>>> f"{p.name} {p.age} {q.name} {q.age}"
'Toto 25  0'
```

However, consider this:

```
class Person:
    name = ''
    age = 0
    hobbies = ['Knitting', 'Reading']
```

All instances actually share the same list, which is very limiting:

```
>>> p, q = Person(), Person()
>>> p.hobbies
['Knitting', 'Reading']
>>> del p.hobbies[0]
>>> p.hobbies
['Reading']
>>> q.hobbies
['Reading']
```

We have encountered that kind of problem before, for instance for default values of the `defaultdict` type. The solution is to call a function to make new instances of the needed objects. This is the notion of *constructor* which we have already discussed. A constructor is a function or method that initialises an object, that instantiates a class. In Python, each time you use a constructor, like `list(...)`, the method named `__init__` plays that role.[v]

Note that this "double-underscore" naming style is that used, by convention, by Python to denote special methods. We already mentioned `__len__`, for instance, which is the special method actually handling `len(..)`'s length or cardinality computations. There are many more special methods, some of which we shall discuss in good time, and others we shall list in Sec. 27.10[p197]: "Special, magic, dunder methods".

27.3 Static attributes, and the `type` metaclass

The attributes we have defined so far are actually *static* attributes. That is to say, they are attached to the class, more so than to instances of it.

You can and should preferably access them directly from the class:

```
>>> Person.age
0
```

[v] There is another, `__new__`, that I shall not discuss, and which can be ignored in 99% of use cases.

Actually, what this means is that a class is itself an object. But if that is so, then what class is `class` itself an instance of?

```
>>> type(Person)
<class 'type'>
```

So `type` is really also a class — a metaclass, of which all other classes are instances. But what is `type` an instance of? Is that an object as well?

```
>>> type(type)
<class 'type'>
```

Type is a perfectly ordinary instance of... itself. The `type` function is therefore a constructor, which can be used to define new types, new classes, on the fly, since they are perfectly ordinary objects:

```
>>> help(type)
...
|  type(name, bases, dict) -> a new type
...

>>> C = type('C', (object,), dict(a=1))
>>> C
<class '__main__.C'>

>>> type(C)
<class 'type'>

>>> C.a
1
```

Here we see that, internally, the attributes – and methods – associated with an object are actually handled by a dictionary, like function calls.

`type` is also an instance of object. All objects are derived from `object` by inheritance. And `object` is a `type`. It's objects all the way down indeed...

```
>>> isinstance(type,object)
True
>>> type(object)
<class 'type'>
```

The idea of inheritance, which we shall see later, is that some classes are specialised instances of others, adding and altering behaviour, and inheriting the rest. For instance `frozenset` shares a lot of behaviour with `set`, with a few twists of its own.

27.4 Constructors: beware of mutable structures

Let us add a constructor to our class. It is its job to initialise attributes, given a number of pertinent arguments. Its first argument, as for all non-static methods, is generally called `self`[w]and is a pointer to the object being defined itself. Within the definition of a method, you *must* refer to any local attribute as `self.something`.

```
class Person:
    def __init__(self, name='', age=0, hobbies=[]):
        self.name = name
        self.age = age
        self.hobbies = hobbies

p = Person("Toto", 25, ['Knitting', 'Reading'])

q = Person("Tata", 97, ['Snoozing'])
------------------------------------------------
>>> p.hobbies
['Knitting', 'Reading']
>>> q.hobbies
['Snoozing']
```

This time, we have a much more convenient way of defining new objects, and it seems like we have different lists at last. We must, since we provide those new lists to the constructor. But what happens if we *don't*, and use the default value?

```
>>> p, q = Person(), Person()
>>> p.hobbies.append(0)
>>> p.hobbies
[0]
>>> q.hobbies
[0] # oops, still the same list.
>>> r = Person()
>>> r.hobbies
[0] # everyone shares the same
```

What happens here is that the default values were evaluated once, when the line

```
def __init__(self, name='', age=0, hobbies=[]):
```

was run, yielding a pointer which will then be shared among all objects using the default value. We need the new list to be generated at each call of `__init__`. A solution would be to write `self.hobbies = hobbies.copy()` or `self.hobbies = list(hobbies)`, so that every list of hobbies passed to the constructor is copied. This may or may not be what you want. If you want the lists passed to the constructor not to be copied, but want a fresh empty list by default, write something like this:

[w] though I am very lazy and often write it `s` for short, which you might notice in my code; use `self` in production code; do as I say, not as I do.

```
class Person:
    def __init__(self, name='', age=0, hobbies=None):
        self.name = name
        self.age = age
        self.hobbies = list() if hobbies is None else hobbies

p = Person("Toto", 25, ['Knitting', 'Reading'])
q = Person("Tata", 97, ['Snoozing'])
r, s = Person(), Person()
---------------------------------------------------------------
>>> r.hobbies.append(0)
>>> r.hobbies
[0]
>>> s.hobbies
[]
```

27.5 matching attributes

Recall that in Sec. 23.6[p119]: "Pattern matching: **match..case**", we used class patterns such as **int**() and str(), and said we can go farther and match attributes as well?

Let us do that:

```
p = Person("Toto", 25, ['Knitting', 'Reading'])

match p:
    case Person(age=25, hobbies=[x,y]): print(p.name, y)
---------------------------------------------------------
Toto Reading
```

It follows about the same conventions as for mappings: so long as at least the attributes you are looking for are present, and each one matches its assigned pattern, it matches.

Note, crucially, that although Person(age=25, hobbies=[x,y]) *looks* like a constructor invocation, it is not. In this example it happens that the attributes and the constructor arguments are the same, which is good practice where possible, but that need not be the case. The name argument is missing anyway.

Like mapping patterns, class patterns syntactically look like they define an object, but in fact match only part of it.

A *class pattern* C(a=1,b=2) has no guarantee to match an *object* C(a=1,b=2):

```
class C:
    def __init__(s,a,b):
        s.x = a+b

myobj = C(a=1, b=2)
```

```
match myobj:
    case C(a=1, b=2): print("yes")
    case C(x=3):      print("Beware!")
----------------------------------
Beware!
```

Also note that attributes are matched *by name*, exclusively. A pattern `C(x,y)` would be rejected with

```
TypeError: C() accepts 0 positional sub-patterns (2 given)
```

despite `C` having a two-arguments constructor, which is expected as constructors have nothing to do with matching, as belaboured above, and so would `C(x)`, with

```
TypeError: C() accepts 0 positional sub-patterns (1 given)
```

The problem is that attributes are not intrinsically ordered, so "bind `x` to the first attribute" makes no sense, absent a notion of "first".

We shall see in Sec. 28[p202]: "Advanced structural pattern **matching**" that we can use data classes to enforce an order between attributes, and match accordingly.

27.6 Instance methods and static methods

A *method* – or *instance method* – is a function associated to an object, and potentially acting on it, altering it.

Methods are not fundamentally different from attributes — when it comes down to it, they are simply attributes that happen to be callable.

Let us define the `getold(years)` method for instances of `Person`. It increments their age by `years`, and causes them to state their new age.

Like `__init__`, instance methods always take `self` as their first argument. The rule is that, if `c` is an instance of class `C`, then a call `c.f(x)` is equivalent to `C.f(c,x)`.

When presenting the code, I shall not repeat previously defined methods, so as to save space, and just write `...` to indicate some code has been elided.

```
class Person:
    ... # I don't repeat __init__

    def getold(self,years):
        self.age += years
        print(f"{self.name} says: I am now {self.age} years old.")
```

Let us test this:

```
>>> p = Person("Toto", 25, ['Knitting', 'Reading'])
>>> p.age
25
>>> p.getold(1)
Toto says: I am now 26 years old.
>>> p.age
26
```

The second form works as well:

```
>>> Person.getold(p, 3)
Toto says: I am now 29 years old.

>>> p.age
29
```

What if you want your class to contain a method that does not rely on a specific instance? For instance, let us say that we want to compute the average age of many persons? Then, while this operation is clearly associated with the type `Person`, is is just as clearly not tied to a specific person, to a specific *instance*; especially if you want a specific behaviour for the empty population.

In that case, you define a *static method*. Such a method does not have privileged access to any of the instances of the class, only to its static namespace. Therefore, it should not take `self` as its first argument but whatever you want, as would a normal function. In the end, it is merely a function in the namespace of the class.

Let us define our variadic static method `avgage` so that it returns the average age of its arguments, or `None` if called with no arguments.

```
class Person:
    ...
    def avgage(*ps):
        return sum( p.age for p in ps ) / len(ps) if ps else None
-----------------------------------------------------------------
p = Person("Toto", 25, ['Knitting', 'Reading'])
q = Person("Tata", 97, ['Snoozing'])
r, s = Person(), Person()
```

Let us test this:

```
>>> Person.avgage()   # None
>>> Person.avgage(p)
25.0
>>> Person.avgage(p,q)
61.0
>>> Person.avgage(p,q,r,s)
30.5
```

But wait, what happens if we call it from an instance of the class?

```
>>> p.avgage()
25.0
>>> p.avgage(q)
61.0
```

So far, there is nothing distinguishing our “static” method from any other, beyond the fact that we didn’t call the first argument `self` — but this is a convention, not a syntactical requirement. So when the method is called from an instance, its first argument, as always, is that instance. The above is therefore equivalent to

```
>>> Person.avgage(p)
25.0
>>> Person.avgage(p,q)
61.0
```

And maybe that is the behaviour you want; but if not, if you want the method to behave the same regardless of whether it is invoked from the class or an instance, then you should use the **`@staticmethod`** decorator, like so:

```
class Person:
    ...
    @staticmethod
    def avgage(*ps):
        return sum( p.age for p in ps ) / len(ps) if ps else None
```

Then the instance is no longer passed as argument:

```
>>> p.avgage()
>>> p.avgage(q)
97.0
```

A related decorator is **`@classmethod`**, which passes the class of the instance object as first argument, instead of the instance — this allows calling other class or static methods, and is useful in the case of inheritance.

27.7 String representations `str` and `repr`

So far, our objects have been somewhat cryptic to look at in the interactive mode:

```
>>> p
<__main__.Person object at 0x7f370d00dac8>
>>> q
<__main__.Person object at 0x7f3710d95b70>
```

Sure, it tells us the main information: type and memory address, allowing us to see that those are two different objects, but it would be nice to have palatable string representations of our persons. So far, conversion to string yields the same display, wrapped in a string:

```
>>> str(p)
'<__main__.Person object at 0x7f370d00dac8>'
```

There are actually special methods which we can override to change this: `__repr__` and `__str__`.

The first one is called to get a representation of the object for Python — the canonical string representation which, if at all possible, should be valid Python code to build a copy of the object. It is called by the builtin `repr` function.

The second one is called by `str` to convert an object in a "pretty", human friendly, representation. Lacking a specific implementation, it defaults to using the `repr`.

To fully understand the difference between the two, consider this:

```
>>> str('Hello')
'Hello'

>>> repr('Hello')
"'Hello'"

>>> print(str('Hello'), "and",repr('Hello'))
Hello and 'Hello'

>>> repr(repr('Hello'))
'"\'Hello\'"'
```

The `repr` version contains the Python quotes and escapes necessary to encode a `str` object, and would be used by a programmer trying to generate Python code in a string, say, for later execution by `timeit`. The `str` version contains what you actually want to see when you decide to print a string. `print(str(x))` is equivalent to `print(x)`.

Let us define a canonical string representation for `Person`:

```
class Person:
    ...
    def __repr__(self):
        return f"Person({repr(self.name)}, {self.age}, {self.hobbies})"
```

Let us test this:

```
>>> p
Person('Toto', 25, ['Knitting', 'Reading'])
>>> repr(p)
"Person('Toto', 25, ['Knitting', 'Reading'])"
>>> str(p)
"Person('Toto', 25, ['Knitting', 'Reading'])"
>>> p.getold(10)
Toto says: I am now 35 years old.
```

```
>>> p
Person(’Toto’, 35, [’Knitting’, ’Reading’])
```

But, before we defined this, why did we have a `repr` at all? Why do we have a `str`, when we have not defined it? How does it know that it should default to `repr`? Because no class stands on its own; all inherit from `object`, and it defines a number of standard behaviours that we can override if we so choose.

27.8 Inheritance

Inheritance is the process through which new classes inherit the behaviour of other classes. They can then choose to redefine and override some of them, and add their own new behaviours. The syntax to inherit from a class is as follows:

```
class <DerivedClass> (<ParentClass>):
```

Implicitly, all classes inherit, at least, from `object`, so

```
class Person:
```

was strictly equivalent to

```
class Person(object):
```

Let use derive a class from `Person`:

```
class Teacher(Person):

    def lecture(self):
        print(f’{self.name} lectures: "You are doing it wrong!"’)
```

Let us test that and see that a teacher can indeed behave in all respects as a person, but, they can also lecture people.

```
>>> VH = Teacher("V. Hugot", 33, [])

>>> VH
Person(’V. Hugot’, 33, [])

>>> VH.getold(1)
V. Hugot says: I am now 34 years old.

>>> VH.lecture()
V. Hugot lectures: "You are doing it wrong!"

>>> p.lecture()
AttributeError: ’Person’ object has no attribute ’lecture’
```

We have

```
>>> isinstance(VH, Teacher)
True
>>> isinstance(VH, Person)
True # it is a subclass
>>> isinstance(VH, object)
True # of that too; always
>>> isinstance(VH, list)
False # just checking isinstance can return False ;-)
```

This is all well and good; however, we would like to override some of the inherited behaviours to make them more specific to the case of teachers. For one thing, the string representation still states `Person`; let us update it to say `Teacher`.

Note: *this is a somewhat artificial example to illustrate inheritance, and in this specific case, you could just bypass that and get the current class name with* `obj.__class__.__name__`*. But let us pretend that does not exist.*

```
class Teacher(Person):
    ..
    def __repr__(self):
        return f"Teacher({repr(self.name)}, {self.age}, {self.hobbies})"
```

Now it behaves as it should:

```
>>> VH
Teacher('V. Hugot', 33, [])
```

Since only the first word changes, could we not have used the previous representation, and simply changed the first word? Yes we could have, for we can call methods from parent classes:

```
    def __repr__(self):
        return Person.__repr__(self).replace("Person","Teacher",1)
```

has the same effect as before. Instead of calling `Person` explicitly, and needing to pass `self`, would it not be great if we could, for an instant, treat `self` as an instance of the parent class only? We can, thanks to `super()`, which returns a delegate object to the parent class, with full access to all applicable attributes. The code becomes:

```
    def __repr__(self):
        return super().__repr__().replace("Person","Teacher",1)
```

with, again, the same results. Super is preferred for single inheritance — for instance, should you decide to change the base class, you can do so transparently, without having to alter all the methods that used a call to a parent, provided that the new parent class is compatible with the calls that are made. It is also a little bit more readable.

In the case of multiple inheritance, it is a powerful tool, but requires careful design and consideration to work correctly. In Sec. 27.9[p195]: "Multiple inheritance", we shall assume that we are using versions of `__repr__` that do not use `super()`.

Let us create another derived class: what about students? Well, they can't lecture people, but they can goof around, acquiring new hobbies in the process:

```
class Student(Person):

    def goof(self,hobby):
        self.hobbies.append(hobby)
        print(f'{self.name} goofs around with {hobby.lower()}')

    def __repr__(self):
        return super().__repr__().replace("Person","Student",1)
```

Let us see Bob discover the joys of knitting:

```
>>> Bob = Student("Bob", 19, [])

>>> Bob.goof('Knitting')
Bob goofs around with knitting

>>> Bob
Student('Bob', 19, ['Knitting'])
```

Let us not forget that everyone is still a `Person`, and that `Persons`, `Teachers`, and `Students` can be mixed in any context that merely needs a `Person`:

```
>>> all ( isinstance(o,Person) for o in (p,q,VH,Bob,PVH) )
True

>>> Person.avgage(p,q,VH,Bob,PVH)
41.4
```

27.9 Multiple inheritance

Advanced notion: *skip this section unless you have a clear and present need to use the technique — in which case you will need a lot more documentation.*

Python supports *multiple inheritance*, which means that a class can inherit from several — though inheriting just from one is much more common. The syntax is as follows:

```
class <DerivedClass> (<ParentClass1>,...,<ParentClassN>):
```

Multiple inheritance has a reputation for difficulty and potential danger, to the point that Java does not support it at all, as opposed to, say, C++. Whether that reputation is fully deserved, whether the danger is intrinsic to multiple inheritance as a concept or whether it is the result of poor implementations and uses of it, remains a matter of some debate. Let us just see briefly how it works in Python.

Consider the case of professors; clearly, they are a subset of teachers, but, as researchers, they are eternal students. Thus, they can behave as both – where that does not conflict.

```
class Prof(Teacher,Student):
    pass
----------------------------
>>> PVH = Prof("V. Hugot", 33, [])

>>> PVH.goof('Video Games')
V. Hugot goofs around with video games

>>> PVH.lecture()
V. Hugot lectures: "You are doing it wrong!"
```

So far so good. But what about methods shared by the two parent classes, but implemented differently? In that case `Teacher` and `Student` have conflicting definitions of `__repr__`. Which one prevails?[x]

```
>>> PVH
Teacher('V. Hugot', 33, [])
```

It turns out that `Teacher` prevails. Why? Because Python's method to resolve this conflict – known as the *diamond problem* – is to use left-to-right priority in the order of inheritance. `Teacher` comes first in the definition `Prof(Teacher,Student)`, so that is what is used.

The details are a bit more complicated. Python builds a *method resolution order* (MRO), which is a linear order among classes derived from the inheritance graph according to algorithms we shall not get into. When a method is called, Python invokes it from the first class, with respect to the MRO, that implements it.

You can consult the MRO of any class, as it is stored as an attribute of the class object:

```
>>> Prof.__mro__
( <class '__main__.Prof'>,
  <class '__main__.Teacher'>, <class '__main__.Student'>,
  <class '__main__.Person'>,
  <class 'object'>)
```

In other words, when `Prof` calls a method, Python first looks for an implementation in `Prof` itself; if `Prof` does not implement the method, it looks into the parents, in the order `Teacher`, then `Student`, then it looks into the common ancestor `Person`, and as a last resort, into `object`.

[x] Here we are assuming there is no use of `super()`.

27.10 Special, magic, dunder methods

Special, magic, or dunder – for double underscore – methods are methods with a special role in Python. By convention,they are named according to the template `__methodname__`.

What is magical or special about them? Nothing intrinsic; they are ordinary methods, without any special syntax or mechanics. What makes them special is that important parts of the Python language *calls* them implicitly and automatically, in certain circumstances.

So far we have seen `__len__`, which is called by the `len` function; `__repr__`, which is called by the `repr` function, and thus any time a value is displayed in the Python interactive mode; `__str__`, which is called any time you print something; `__init__`, which is called any time a new object is created. We shall see `__next__` and `__iter__`, which are central to iterable structures in Sec. 29[p208]: "Iterables, iterators, and generators", and there are many more.

Operators such as + are actually syntactic sugar for method calls. If you want to define a type that supports those operators — "operator overloading", — all you have to do is implement the corresponding methods.

Binary operators:

`+`	`__add__(self,other)`
`-`	`__sub__(self,other)`
`*`	`__mul__(self,other)`
`//`	`__floordiv__(self,other)`
`/`	`__truediv__(self,other)`
`%`	`__mod__(self,other)`
`**`	`__pow__(self,other[,modulo])`
`<<`	`__lshift__(self,other)`
`>>`	`__rshift__(self,other)`
`&`	`__and__(self,other)`
`^`	`__xor__(self,other)`
`|`	`__or__(self,other)`

Binary comparison operators:

<	`__lt__(self,other)`
<=	`__le__(self,other)`
==	`__eq__(self,other)`
=!	`__ne__(self,other)`
>=	`__ge__(self,other)`
>	`__gt__(self,other)`

Note that every binary operator has a normal version and a "reversed", or "reflected" version. That means there are also methods `__radd__`, `__rsub__`, `__rmul__`, `__rdiv__`, `__rlt__`, `__req__`, etc.

The difference between, say, `__add__` and `__radd__` is this: suppose I want to roll my own wrapper for **int** — for some reason — with support for addition. I get the class

```
class INT:
    def __init__(self, a): self.a = a
    def __add__(self, other):
        print("__add__")
        return self.a + other
```

When an object of my class is the left operand of +, everything works as expected. But not when my object is on the right:

```
>>> INT(1) + 2
__add__
3

>>> 1 + INT(2)
TypeError: unsupported operand type(s) for +: 'int' and 'INT'
```

Why is that? Well, recall the rule seen in Sec. 27.6[p189]: "Instance methods and static methods":

> The rule is that, if c is an instance of class C, then a call c.f(x) is equivalent to C.f(c,x).

That means that `INT(1) + 2` is translated into `INT.__add__(INT(1), 2)` — without recomputing `INT(1)`, of course. Then, how is `1 + INT(2)` translated? Into `int.__add__(1,INT(2))`. The problem is **int** is not programmed to deal with operands of type `INT`; it does not know that class exists!

Must we modify the builtin class **int** to get support for our homemade `INT`? Fortunately no. It suffices to add an `__radd__` method to our `INT` class:

```
class INT:
    ...
    def __radd__(self, other):
        print("__radd__")
        return other + self.a
```

and things work as expected:

```
>>> 1 + INT(2)
__radd__
3
```

Why and how does that work? When **int** fails to handle the other operand, of type INT, it obligingly returns a very special value NotImplemented. Think of it like a little brother for None: like None, it is the only one of its type:

```
>>> type(NotImplemented)
<class 'NotImplementedType'>
```

What's special about that value is that the Python interpreter will actively be on the lookout for it as the return value when executing magic methods for binary operators, such as __add__. If it does not detect it, the buck stops there. If it does detect it, it tries to reverse the operands and invoke __radd__ from the other type. If that method is not implemented, or is but ends up returning NotImplemented for those particular arguements, only then does it give up and raise a TypeError.

If that is not clear, play with the following code:

```
class A:
    def __add__(s,o):
        print("Aadd")
        if isinstance(o,int): return "intOK"
        return NotImplemented

class B:
    # pass
    def __radd__(s,o):
        print("Bradd")
        if isinstance(o,A): return "yes"
```

Try executing A()+1 and A()+B() with and without the **return** NotImplemented line and the __radd__ method in B, until you understand what is going on.

That means that any *proper* implementation of a binary operator, that wishes to let other classes fully support it with their own operators, should be of the form:

```
if isinstance(other, sometype):
    return # something
elif isinstance(other, someOtherType):
    return something_else
...
return NotImplemented
```

Without the last line, the method would return None, which is as valid a return value as any other, and the interpreter would stop there.

Incrementation / in-place assignment operators:

+=	__iadd__(self,other)
-=	__isub__(self,other)
*=	__imul__(self,other)
/=	__idiv__(self,other)
//=	__ifloordiv__(self,other)
%=	__imod__(self,other)
**=	__ipow__(self,other[,modulo])
<<=	__ilshift__(self,other)
>>=	__irshift__(self,other)
&=	__iand__(self,other)
^=	__ixor__(self,other)
|=	__ior__(self,other)

Not that there are NotImplemented shenanigans at work here again: when executing a += b, Python first tries __iadd__ — there is some expectations that it implements an in-place version of __add__, cf. Sec. 24.2.2.3[p135]: "In-place assignment on mutable structures" — and if that is not implemented or returns NotImplemented, it tries a = a + b, which invokes __add__. That too can fail, in which case it tries __radd__, as seen before.

Consider the following code for an illustration:

```
class A:
    def __add__(s,o):
        print("A.add")
        if isinstance(o,int): return "Aa"
        return NotImplemented
    def __iadd__(s,o):
        print("A.iadd")
        return NotImplemented

class B:
    def __radd__(s,o):
        print("B.radd")
        if isinstance(o,A): return "yes"
----------------------------------------
>>> a = A()
>>> a += B()
A.iadd
A.add
B.radd
>>> a
'yes'
```

Unary operators, conversions, etc:

`-`	`__neg__(self)`
`+`	`__pos__(self)`
`abs()`	`__abs__(self)`
`~`	`__invert__(self)`
`complex()`	`__complex__(self)`
`int()`	`__int__(self)`
`float()`	`__float__(self)`
`oct()`	`__oct__(self)`
`hex()`	`__hex__(self)`
`repr()`	`__repr__(self)`
`str()`	`__str__(self)`
`bool()`	`__bool__(self)`
`hash()`	`__hash__(self)`
`len()`	`__len__(self)`
`reversed()`	`__reversed__(self)`

Other special operators

`X[index]`	`__getitem__(self, index)`
`X[index] = v`	`__setitem__(self, index, v)`
`x in X`	`__contains__(self, other)`
`f(x)`	`__call__(self, *args, **kwargs)`

Note that your classes can support some operations even if they are not explicitly defined. For instance, if `__contains__` is not implemented, `x in X` will default to iterating on the structure `X` via `__iter__` in search of `x`. If `__iter__` is not implemented either, it will try to iterate on `X` anyway by testing indexes `0, 1, 2,...` via `__getitem__`. Only if that is not implemented either does it give up.

We shall see more of those specific iteration problems in Sec. 29[p208]: "Iterables, iterators, and generators". Just remember, with this example and that of `__add__` and `__radd__` seen at the beginning of this section, that there is plenty of implicit behaviour involved in translating what you write into magic method calls.

27.11 Against paradigmatic integralism

The following advice may not be universally accepted, to put it mildly: Do not force yourself to put *everything* into objects. While cars and such offer nice examples where the object abstraction works fairly well, and there are plenty of cases where methods should belong to specific types, things break down quickly in more complex cases.

For instance, when several types of objects interact, it often becomes difficult to decide *which* of the objects should own the actions that are jointly performed; that is, who owns the method, and who is merely a parameter of it. For instance, should the operation "concatenating a list of strings" belong to the type of lists or that of strings? What about operations involving three or four different types?

Use classes where it is clear that the abstraction work, and stick with standard procedural code for everything else. The ratio of the two will vary widely depending on the specific problem.

If all you need are namespaces, use modules.

28 Advanced structural pattern **`match`**ing

Now that we have seen more of OOP, let us come back to pattern matching. Be sure to have read Sec. 23.6[p119]: "Pattern matching: **`match..case`**" and Sec. 27.5[p188]: "**`match`**ing attributes" before this.

As a guiding example, we shall implement the transformation of formulæ of propositional logic into Negation Normal Form (NNF). You should have seen this already in your Logic course.

If propositional logic does not excite you overmuch in and of itself, understand that, at the end of the day, what we are doing is manipulating inductive structures by applying precise structural rules. A considerable number of very interesting things depend on the very same techniques.

Any arithmetic expression, such as $2 \times (x - 1) + 3$, is an inductive structure; any computation done on it is done by breaking down its structure and applying rules to rewrite it or compute its value.

Any program code is also an inductive structure — refer to the Languages Theory course, especially the section on grammars — and program compilation proceeds inductively, by breaking down the code's structure. You will see that next semester during the Compilation course.

Therefore, the techniques used there are not at all tied to propositional logic, but are extremely general and important for a vast class of problems. And the **`match`** keyword provides an extremely good way of tackling such problems. In Sec. 45[p274]: "A smidgen of Computer Algebra", you will implement a rudimentary Computer Algebra System capable of computing derivative functions symbolically.

With this out of the way, recall that formulæ of propositional logic (PL) are those built with operators $\wedge, \vee, \neg, \implies, \iff, \dots$, the first three being the only indispensable ones, since $\equiv$

and $\iff$ can be refined in terms of those three[y].

In other words, PL formulæ φ are generated by the grammar

$$\varphi \;::=\; x \;\mid\; \varphi \wedge \varphi \;\mid\; \varphi \vee \varphi \;\mid\; \neg\varphi \;\mid\; \varphi \Rightarrow \varphi \;\mid\; \cdots\,.$$

For instance, let us consider the formula

$$\varphi \;=\; \neg\big(x \Rightarrow (y \vee z)\big)\,.$$

A formula is in NNF when it only uses $\wedge, \vee, \neg$ and all negations appear on atoms (here, variables). So φ above is not in NNF, since it is the negation of an implication, which is not an atom — and not even allowed as an operator.

To put a formula in NNF without changing its truth table, it suffices to rewrite it according to the following rules, until nothing is left to be rewritten:

$$\begin{aligned} x \Rightarrow y &\quad\to\quad \neg x \vee y \\ \neg(x \vee y) &\quad\to\quad \neg x \wedge \neg y \\ \neg(x \wedge y) &\quad\to\quad \neg x \vee \neg y \\ \neg\neg x &\quad\to\quad x \end{aligned}$$

The first is the definition of $\Rightarrow$, which we get rid of, the next two are the De Morgan's laws, and the last is the elimination of the double negation. Notice that negation is systematically pushed inwards or removed.

Let us implement this transformation. The first question is: "how do I represent my formulæ?".

For the purpose of pattern matching, we could take variable names to be strings, define constants `AND`, `OR`, `NOT`, and represent a formula as a tuple

```
(<operator>, <left operand>, <right operand>)
```

so that φ becomes

```
(NOT, (IMPLIES, "x", (OR, "y", "z")))
```

That would work quite nicely, using only sequence patterns, but we can do better by using data classes and their patterns. It would be nice if `Or` was an object, and we could write `Or("y","z")`, nest it in bigger formulæ and enjoy the same kind of pattern matching as for tuples above. Let's make it so!

[y] Alternatively, all operators can be defined in terms of a single, less intuitive operator: either NOR or NAND. But this is just a bit of trivia, outside of the scope of the current discussion.

The @dataclass decorator applies to a barebones class definition, just a class name and a list of (typed[z]) attributes, and automatically generates the magic methods implementing the obvious constructor and repr. By obvious, I mean that an attribute x becomes a constructor argument x, and the constructor does self.x = x.

Furthermore, @dataclasse specifies the order of the attributes for the purpose of class patterns as the order in which the attributes were written in the code.

Let us partake in all this syntactic sugar for the benefit of our formulæ:

```
from dataclasses import dataclass

@dataclass
class BinOp:
    l: object
    r: object

class And       (BinOp):
    symb = "&"
class Or        (BinOp):
    symb = "|"
class Implies   (BinOp):
    symb = "->"

@dataclass
class Not:
    f: object

f = Not(Implies("x", Or("y", "z")))

------------------------------------

>>> f
Not(f=Implies(l='x', r=Or(l='y', r='z')))
```

A binary operator is just something with two attributes l and r — and in practice the names won't matter much, since dataclasses have ordered attributes, and we shall match according to order and not name. And, Or, and Implies, as well as equivalence and whatever else you want, are particular binary operators, to which I add a symb attribute that will be useful later on, to prettify the string representation of the formula.

Not is the only unary operator possible, so we don't bother abstracting it with a class UnaryOp, that would be overkill.

With this, using the constructors obligingly provided by @dataclass, we can define a formula f. Thanks to the automatically defined repr — thanks again, @dataclass — we can visualise it as well, even if in a little verbose form.

[z] The type annotation for the attributes is generally not examined by Python itself, so we can get away with just :object.

Now let us write a function `fstr` to obtain a nicer string representation of our formulæ, using pattern matching.

```
def fstr(f):
    match f:
        case str():         return f # variable
        case BinOp(l,r):    return f"({fstr(l)} {f.symb} {fstr(r)})"
        case Not(f):        return f"-{fstr(f)}"
        case _:             raise ValueError(f)

-----------------------------------------------------------------------

>>> fstr(f)
-(x -> (y | z))
```

Now *that* is a little bit more compact and legible. Note that `BinOp` will match any object of class `And`, `Or`, and `Implies`. We could have written three **`case`** lines

```
case And(l,r):      return f"({fstr(l)} & {fstr(r)})"
case Or(l,r):       return f"({fstr(l)} | {fstr(r)})"
case Implies(l,r):  return f"({fstr(l)} -> {fstr(r)})"
```

but that would not have been very elegant. If we had many functions like this, we would need to add a line to each of them each time we add a new operator. By using the `symb` attribute instead, our function will naturally adapt to any new operator, so long as its `symb` attribute is defined.

A quick note. When writing such functions, it is a good idea to *begin* by writing something like

```
case _: raise ValueError(f)
```

If nothing matches, you get `None`. It's a pain to debug if you get `None`s you don't expect because you forgot a case. Languages like OCaml or Haskell, where structural pattern-matching is most at home, are capable of telling you when your patterns are not exhaustive, and give you examples of values that are not matched (at least when there are no guards). Python cannot do that, beyond detecting obvious capture patterns of wildcards that are not on the last **`case`**.

Let us now write a function `rmimp` to get rid of implications. It is a simple transformation: the identity everywhere except where it can apply

$$x \Rightarrow y \quad \to \quad \neg x \vee y\,.$$

We obtain:

```
def rmimp(f):
    match f:
        case str():         return f
        case Implies(l,r):  return Or(Not(rmimp(l)), rmimp(r))
        case BinOp(l,r):    return type(f)(rmimp(l), rmimp(r))
        case Not(f):        return Not(rmimp(f))
        case _:             raise ValueError(f)

-------------------------------------------------------------

>>> fstr(F := rmimp(f))
-(-x | (y | z))
```

We match `Implies(l,r)` before `BinOp(l,r)` because, of course, the latter is strictly more general than the former; if it were first, `Implies` would be shadowed, never to be matched.

We use a nice little trick in

```
case BinOp(l,r):    return type(f)(rmimp(l), rmimp(r))
```

We match a binary operator, but we don't know which, aside the fact that it cannot be `Implies`. We want to reproduce it identically, recursively transforming its children. We cannot write `BinOp(rmimp(l), rmimp(r))` because that is not even a well-defined formula. We need to know *which operator*.

So we use the fact that `type(f)` returns the class of `f`, which is callable: the constructor is called. So `type(f)(rmimp(l), rmimp(r))` becomes `And(rmimp(l), rmimp(r))` if `f` is of type `And`, and so on, which is exactly what we want.

There remains to apply De Morgan's laws and negation elimination, and we have our NNF.

$$\neg\neg x \quad \to \quad x$$

Observe that De Morgan's laws

$$\begin{aligned} \neg(x \vee y) \quad &\to \quad \neg x \wedge \neg y \\ \neg(x \wedge y) \quad &\to \quad \neg x \vee \neg y \end{aligned}$$

are of the same form apart from the exchange of $\wedge$ and $\vee$, a fact we shall exploit to factorise the two rules into one:

```
def nnf(f):
    z = nnf ; morgan = {And:Or, Or:And}
    match f:
        case Not(Not(f)):   return z(f)
        case Not(BinOp(l,r) as g):
            return morgan[type(g)](z(Not(l)), z(Not(r)))
        case BinOp(l,r):    return type(f)(z(l), z(r))
        case Not(f):        return Not(z(f))
        case str():         return f
        case _:             raise ValueError(f)
```

```
--------------------------------------------------------
>>> fstr(f)
-(x -> (y | z))
>>> fstr(F)
-(-x | (y | z))
>>> fstr(N := nnf(F))
(x & (-y & -z))
```

Note the recursive calls to `(z(Not(l)), z(Not(r)))` in the De Morgan rule: it would be wrong to write `Not(z(l))`, because then the function could forget to simplify `Not(Not(..))` patterns.

28.1 An overlong aside on naming conventions

Apart from the factorisation of the De Morgan's rules, the other neat little trick in `nnf` is `z = nnf`. What is the point of it? You do a lot of recursive calls in such functions, and the shorter they are to write, the better for the legibility of the code. With this, we have short recursive calls without sacrificing the legibility of the name of the function exposed to the user.

Furthermore, you should observe by now that all those functions on formulæ are very similar-looking. Of course they are, since they act on the same inductive structure. Thus is is often expedient to copy and paste the `case` patterns of a function to quick-start the writing of the next — it saves time and helps ensure you don't miss too many cases.

But if I copy cases from, say, `rmimp`, into my new function `nnf`, I get lines of the form

```
case Not(f):        return Not(rmimp(f))
```

and if I forget to rename every single instance of `rmimp` into `nnf`, then I get *interesting* errors, whereby `nnf` escapes into `rmimp`.

To save the time necessary to rename all those calls, and avoid that type of mistake, I personally[aa] find it convenient to take the convention of naming the recursive function `z` everywhere. It also avoids problems whenever you choose to rename your functions. . .

Sometimes also, you come to realise that you need to make your pattern-matching a subfunction to the real function of interest. For instance, in Sec. 45[p274]: "A smidgen of Computer Algebra", when you simplify an arithmetical expression, each action may introduce new opportunities for simplifications; so you gain from repeating the simplification until a fixpoint is reached. In that case, you have something like

```
def simp(e):
    def z(e):
        match e:
```

[aa] Emphasis on *I, personally*. This is not at all standard; I haven't seen many people do that.

```
            case Plus(0,e) | Plus(e,0): return e
            ...
    return fixpoint(z,e)
```

If you didn't anticipate that structure, the z convention helps you avoid, again, lots of error-prone renaming and makes the refactoring trivial.

It can also be that what needs to be called recursively is a multi-argument function, and the structural recursion is only done on one of them. For instance, in Sec. 45[p274]: "A smidgen of Computer Algebra" again, you need to evaluate an expression e for a certain value val of a variable var with a function eval(e,var,val).

The matching only happens on e, while the other arguments remain the same, always. A nice z = lambda e: eval(e,var,val)) removes quite a lot of clutter.

29 Iterables, iterators, and generators

Let us now look at how iteration works under the hood.

An **iterable** is any object that can be iterated upon; by that I mean, you can write code of the form for x in object.

Being iterable entails some other consequences: for instance the in operator is automatically defined, with the default operation "iterate until you find the element".

In collections with more efficient ways of testing membership, this default behaviour is overridden via the __contains__ special method.

There are many such patterns in Python, whereby implementing some functionality automatically and implicitly provides default implementations of other, related functionalities.

The classical way of being an iterable is by implementing the __iter__ method, called by the iter builtin function, returning an *iterator* — more on that shortly. However, that is not strictly necessary, as any indexable object supporting indexes starting at 0 implicitly becomes iterable, thanks to an implicit iteration over its indexes.

```
class X:
    def  __getitem__(s,o):  # s[o]
        if o in (0,1): return 'a'
        else: raise IndexError
x=X()
```

```
>>> 'a' in x
True
>>> 'b' in x
False
```

```
>>> list(x)
['a', 'a']
>>> for c in x: print (c)
a
a
```

Regardless of how an object became an iterable, the **iter** function returns an iterator from it. Each time an iterable is iterated over, an iterator is built from it, as it is what powers an iteration.

Iterators are objects that implement the `__next__` method, called by the **next** function. It must return a value on each call, and, optionally, at some point, it may raise `StopIteration`, in which case it *must* continue raising `StopIteration` on every subsequent calls. Failing that, the implementation is deemed broken.

Note that iterators produce each value on demand, and have no memory of their previously generated values, nor memory reserved for their future values. They are iterated upon *once*, and once only, and each call of **next** irreversibly alters their state[ab]

They are merely an object, potentially with a few attributes, waiting for the next call of a specific method — which may alter their state.

They are thus very memory efficient compared to generating all the values and storing them in memory in a list or a tuple. They can even describe infinite collections, simply by never raising `StopIteration`, and that will not become a problem unless someone actively tries to loop over all values. This is an instance of *lazy*, or *call-by-need* programming, where values are evaluated only when actually required; this opens up techniques for elegant and greatly efficient code. However, Python does not have the full power of lazy evaluation, as this works best in purely functional languages, where every value is immutable. Haskell is probably the best example of a purely functional, lazy by default language.

Iterators are iterables as well, and must implement `__iter__` so that they return themselves.

29.1 Explicit class implementation

There are cleaner ways to define iterators than by manually implementing those methods — **yield**, and generator expressions — so let us take a running example. Let us implement an inclusive range function: `r(i,j)` = $[\![i, j]\!]$, as an iterator.

```
class r:
    def __init__(s,i,j):
        s.i, s.j = i,j
        s.k = i

    def __iter__(s): return s
```

[ab]— and will do so for all finite iterators. The only iterators whose state is unaltered by **next** are those that return the same value on every call, indefinitely. Those that are *reversibly* altered are infinite and cyclic.

```
    def __next__(s):
        if s.k <= s.j:
            s.k += 1
            return s.k - 1
        else: raise StopIteration
```

We have:

```
>>> list(r(2,7))
[2, 3, 4, 5, 6, 7]

>>> r(2,7)  # different iterator objects each time
<__main__.r object at 0x7fd85ffae860>
>>> r(2,7)
<__main__.r object at 0x7fd85ffb8eb8>

>>> it = r(2,7)
>>> next(it) # each call alters state
2
>>> next(it)
3
>>> list(it)
[4, 5, 6, 7]
>>> list(it)
[]
```

This is not quite the behaviour of the usual **range** object, which can be iterated over any number of times. To achieve that, we have two different classes: a reusable iterable and the iterators it produces. Let us rename our previous version of `r` to `r_iterator` and define a new class

```
class r:
    def __init__(s,i,j):
        s.i, s.j = i,j
        s.k = i

    def __iter__(s):
        return r_iterator(s.i,s.j)
```

We have:

```
>>> it = r(2,7)

>>> list(it)
[2, 3, 4, 5, 6, 7]
>>> list(it)
[2, 3, 4, 5, 6, 7]

>>> iterator = iter(it)
>>> list(iterator)
[2, 3, 4, 5, 6, 7]
>>> list(iterator)
[]
```

This time, it works as usual.

29.2 yield and yield from

While neither version of the above is very difficult to write, it is still arguably a lot of code and boilerplate given how trivial the logic of what we are implementing is.

Fundamentally, what we want to do is loop over indexes from i to j, returning each one in turn. Of course, we cannot do that with a functions **return** statement, because that returns just one value and immediately terminates the function and discards any and all of its local variables.

If only we had a keyword much like **return**, but that does not terminate the function after "returning" a value, but instead "freezes" it, with all its local variables and execution state, so that we can later thaw it, continue looping where we left off, and thus return value after value, on demand, until the end of the loop. As it happens, that is exactly what the **yield** keyword does:

```
def r(i,j):
    while i <= j:
        yield i
        i += 1
```

Any function whose body contains **yield** actually returns an iterator, or more specifically a *generator*, which is simply what we call iterators obtained in such a manner:

```
>>> r
<function r at 0x7f81332aed90>

>>> r(2,7)
<generator object r at 0x7f81332b44c0>
```

Put another way, r has become the constructor for generators. We call it a *generator function*, though by an abuse of language that I shall avoid in this section the function itself is also often called a generator.

The produced generators behave in all manners exactly as the iterators we defined previously.

```
>>> r(2,7) # different objects each time
<generator object r at 0x7f81332b4468>
>>> r(2,7)
<generator object r at 0x7f81332b4410>
```

```
>>> it = r(2,7)

>>> next(it)  # each call alters state
2
>>> next(it)
3
>>> list(it)
[4, 5, 6, 7]
>>> list(it)
[]
```

`return` can appear in a generator function — and in fact, a `return` or, equivalently, `return None` is implicitly present at the end of all functions — and simply forces the end of the iteration, similarly to a `raise StopIteration`.

Up to Python 3.6 `return <value>` s is equivalent to `raise StopIteration(<value>)`. This behaviour changes in Python 3.7: you must now use `return` to end the iteration in a generator function.

Suppose now that you want, in a generator function, to yield values from another iterator. For instance, let us write a generator function `loop(i,j,n)` that repeats the range $[\![i,j]\!]$ n times. Our first instinct would be to write something like:

```
def loop(i,j,n):
    for _ in range(n):
        yield r(i,j)
```

but that would be wrong, because then we yield a generator each time, and not the values produced by it:

```
>>> list(loop(1,3,3))
[<generator object r at 0x7fa5bc6cd410>,
 <generator object r at 0x7fa5bc6cd3b8>,
 <generator object r at 0x7fa5bc6cd468>]
```

We cannot use `return` either, as that would break the loop. The first approach would be to iterate over `r(i,j)` and yield its values:

```
def loop(i,j,n):
    for _ in range(n):
        for x in r(i,j):
            yield x
---------------------------

>>> list( loop(1,3,3) )
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

It would be much more elegant, and probably more efficient, to have a means of saying “delegate value generation to `r(i,j)`”.

For this, Python provides another keyword: **yield from**, which delegates the control flow to a subiterator:

```
def loop(i,j,n):
    for _ in range(n):
        yield from r(i,j)
---------------------------

>>> list(loop(1,3,3))
[1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> g = loop(1,3,3)  # still an iterator
>>> next(g)
1
>>> next(g)
2
>>> list(g)
[3, 1, 2, 3, 1, 2, 3]
```

While the existence of **yield from** is always convenient, it shines particularly in advanced uses of generators, such as coroutines, which we shall not discuss here.

29.3 Generator expressions

You can now return to Sec. 24.5.1[p154]: "Comprehensions for every type; first contact with generators" and Sec. 24.5.2[p156]: "Loop nesting in comprehensions" to gain a full understanding of what a generator expression is.

29.4 Understanding deeply lazy computations

Data processing, when boiled down to its essentials, tends to take the following form:

```
data0 # our data source

data1 = f1(data0)
data2 = f2(data1)
...
dataN = fN(dataNm1)
```

Some initial data, be it from a file, a database, or whatever else, goes through a number of transformations until it is ready. Traditionally, when you write code of this form, or even of the form

```
dataN = fN(fNm1(... f1(data0)...))
```

each step must be completed before the next one begins, and each result is integrally stored in memory.

But perhaps you do not really need *all* of the resulting `dataN`, though you may not know in advance how much of it you might need. In that case, instead of the traditional "left-to-right, line-by-line" approach, you may prefer a more "vertical" mode of computation, whereby you compute *just a bit* of `data0`, and then out of it, whatever you can of `data1`, and so on, and finally look at the little bit of `dataN` that you obtain, and decide whether you have what you need, or want to repeat the process to get more.

That seems like a good idea on the surface, and the need for this is obvious; when any of the `dataK` are very large and you are looking for something in the fully transformed data, it is a terrible waste to fully compute any step or store it in memory.

But implementing this seems like it would require a lot of bookkeeping to keep track of where you are on each level of data processing, and how do you even know how much of `data0` you should process to get enough new data to progress in `data1`'s computation, and get enough to progress in `data2`'s etc?

Each level of data processing may have its own unique requirements of the previous level in order to progress in its own computation. Perhaps `f1` needs 1 byte of `data0` in order to produce 1 byte of `data1`; perhaps `f2` needs 3 bytes of `data1` in order to produce 1 byte of `data2`, except when it encounters a certain rare pattern, say, `OPEN` in which case in needs to process an arbitrary amount of data, looking for another pattern, say `CLOSE`, and produces 10 bytes; perhaps `f3`. . .

Keeping track of a multi-layered computation in that way, so as to determine how much of `data0` should be produced to obtain, say, 20 new bytes of `dataN` seems, if not impossible, as least very difficult, programmatically.

Let us change our viewpoint a bit. Each level of computation is now naturally lazy. It knows what it needs to do, but doesn't actually do any of it unless actually prompted. When it is asked to produce something, it asks the level below it for stuff, until it has enough and is ready to produce something; it then returns it, and goes back to sleep until somebody bothers it again, asking for more. Spoiler alert: this is exactly how iterators behave.

In this model, if you need 20 bytes of `dataN`, you just wake up `fN`, ask for 20 bytes, and then it is no longer your problem. You need not know or keep track of the needs of each layer. You know *your* needs, 20 bytes in that case, and each layer knows its *own* needs, and will prod the layer below it for what it needs, and no more.

`fN` will take what it needs from `fNm1`, give you the result and go back to sleep, and during that process `fNm1` will do the same with `fNm2`, and so on, all the way down to `data0`.

Let us illustrate that with some code. To simplify and help visualise the vertical aspect of the computation, each level produces one value for one value of the previous level. The value produced is simply the computation depth, starting at 0 for the data source.

```python
def data(m):
    for i in range(m):
        print(f"Data yields 0; has {m-i-1} left")
        yield 0
    print("Data is exhausted")
```

```
def chaingens(g,lvl):
    while True:
        print(f"lvl {lvl} gen asks")
        d = next(g, None)
        if d is None: break
        res = d + 1
        print(f"lvl {lvl} gen obtains {d}, yields {res}")
        yield res

def genchain(n,m):
    if n == 0:
        return data(m)
    else:
        return chaingens(genchain(n-1,m),n)

g = genchain(3,3)

for v in g:
    print(f"Final computation depth: {v}\n")
-----------------------------------------------------------
lvl 3 gen asks
lvl 2 gen asks
lvl 1 gen asks
Data yields 0; has 2 left
lvl 1 gen obtains 0, yields 1
lvl 2 gen obtains 1, yields 2
lvl 3 gen obtains 2, yields 3
Final computation depth: 3

lvl 3 gen asks
lvl 2 gen asks
lvl 1 gen asks
Data yields 0; has 1 left
lvl 1 gen obtains 0, yields 1
lvl 2 gen obtains 1, yields 2
lvl 3 gen obtains 2, yields 3
Final computation depth: 3

lvl 3 gen asks
lvl 2 gen asks
lvl 1 gen asks
Data yields 0; has 0 left
lvl 1 gen obtains 0, yields 1
lvl 2 gen obtains 1, yields 2
lvl 3 gen obtains 2, yields 3
Final computation depth: 3

lvl 3 gen asks
lvl 2 gen asks
lvl 1 gen asks
```

```
Data is exhausted
```

Play with this code until you fully understand what is going on.

Sec. 46[p294]: “Conway sequence: generating fun” proposes an exercise that relies heavily on layering lazy computations – in the form of generators – to achieve immense performance improvements. In that exercise, each level of computation requires an unpredictable amount of data from the previous one.

29.5 Iterator patterns, tools, and tricks

29.5.1 Itertools module

The itertools module implements a number of very convenient tools acting on iterators. Read the documentation for yourself.

In the next section, we discuss some common patterns, independently of their implementation in `itertools`.

29.5.2 A generator for $\mathbb{N}$ and other infinite collections

See `itertools.count`.

While $\mathbb{N}$ is an infinite set, it is countable, which means that its elements can be enumerated. Thus, we can easily make a generator function for it:

```
def N():
    i = 0
    while True:
        yield i
        i += 1
---------------------------------
>>> g = N()

>>> [ next(g) for _ in range(5) ]
[0, 1, 2, 3, 4]
```

Could we make a generator for $\mathbb{N}^2$? For $\mathbb{Z}$? For $\mathbb{N}^n$, for arbitrary $n \in \mathbb{N}$? For $\mathbb{R}$? Why?

29.5.3 Getting (up to) the n-th element

See `itertools.islice`.

29.5.3.1 *Keep the previous elements*

Iterators are generally not indexable. Is there a convenient way to get the n-th element of an iterator? Not if you care about the previous elements — in that case, convert the elements you need into a sequence type:

```
>>> g = r(1,1000)
>>> l = [ next(g) for _ in range(5) ]
>>> l
[1, 2, 3, 4, 5]
>>> next(g)
6
```

The above code is interesting, because, until Python 3.7, it highlighted the one instance where it was not actually *entirely* true that `list(<generator expression>)` is the same as `[<generator expression>]`:

```
>>> g = r(1,3)
>>> [ next(g) for _ in range(5) ]
StopIteration

>>> g = r(1,3)
>>> list( next(g) for _ in range(5) )
[1, 2, 3]  # in 3.6; StopIteration in 3.7+
```

In the context of a "free" generator expression, `StopIteration` is captured to. . . stop the iteration, whereas it bubbles up in any other context. This behaviour was changed in PEP 479, so that now the exception bubbles up in every context.

From 3.7 onward, to get the same behaviour as 3.6's `( next(g) for _ in range(i) )`, returning up to `i` elements without raising an exception even if the generator is exhausted early, you would need to write something like

```
def upto(g,i): #3.7
    for k,e in enumerate(g):
        if k >= i: return
        yield e
```

To my knowledge, it is not possible to get the same behaviour in a lone comprehension expression anymore.

29.5.3.2 *Discard the previous elements*

If you don't care about previous elements, then you can use something like that:

```
>>> g = r(1,1000)
>>> next(x for k,x in enumerate(g) if k == 5) # or k >= 5
6
>>> next(g)
7
```

Why does it work?

```
x for k,x in enumerate(g) if k == 5
```

is itself a generator expression, and you ask it to produce a value. enumerate, which we met before in the context of lists and strings, actually produces an iterator, returning a tuple (index,element) for every element of the input iterable, on demand:

```
>>> g = r(1,3)
>>> ge = enumerate(g)
>>> next(ge)
(0, 1)
>>> next(ge)
(1, 2)
>>> next(g)
3
>>> next(ge)
StopIteration
```

The generator expression will loop without producing any value until the if condition is satisfied. Then, when k == 5, for the first time, it will yield a value, which is returned by next. The generator expression is then discarded, since there is no binding to it.

This example shows, again, how multiple iterators can be layered, each only soliciting a value from the layer below when it is itself solicited.

29.5.4 Length of an iterator

Sometimes, one may wish to know how many elements are generated by an iterator. Unfortunately, len(..) is not defined on iterators:

```
>>> g = r(1,10)

>>> len(g)
TypeError: object of type 'generator' has no len()
```

It is, however, very easy to compute the length using sum(..), but note that this will consume the iterator:

```
>>> sum( 1 for _ in g )
10

>>> next(g)
StopIteration
```

As usual with iterators, there is no way to get any information on future values, including how many there are, without exhausting the iterator.

Also note that trying that on infinite iterators will of course mire you in an infinite loop.

30 Static typing

31 Parallelism and concurrency

We start by defining a few terms. Going into details is not at all the aim of this course — nor would I be competent to teach them. You will undoubtedly go much further in the Systems Programming and Network classes.

For the purposes of the final exam, you will not need to go farther than the contents of this section. For the project, however, you *may* need to dig a little deeper in some aspects.

31.1 Concurrency and parallelism

Multitasking is essential, in particular for GUI programming: if you do everything in a single thread, your GUI will become unresponsive until whatever computation the program is performing terminates. This is a problem, especially if you would like to use the GUI to cancel the operation. . .

Executing several tasks, several execution flows, simultaneously is called *concurrency*. This simultaneity may be only apparent; that is, those tasks may not actually be executed at the same time, but give the illusion thereof by having their executions interleaved, in very short intervals. This is the role of the part of the operating system called the *scheduler*. Thus, you can have effective multitasking on a single CPU core.

If tasks actually execute simultaneously over different cores, then we speak of *parallelism*.

31.2 Threads and processes

Generally speaking, a *thread of execution*, or *thread*, is simply a sequential flow of instructions that runs on a processor. The term is a bit overloaded.

There are two traditional main kinds of “execution flows”, called *processes* (“heavy weight” threads) and *light weight threads*, or just *threads*. Processes are full-fledged executing programs with their own dedicated memory. Threads exist inside a process, and share its memory with every other thread inside it.

Intermediate notions of “context of execution”, sharing specific parts of the memory — rather than all or nothing — are supported by some OS kernels, such as Linux, but the above are the standards.[ac] Keep in mind that those definitions are not universal, and some sources you might read may speak of “threads” without implying “lightweight”.

[ac] http://lkml.iu.edu/hypermail/linux/kernel/9608/0191.html

Using the terminology defined above, threads are much faster to create and to switch between than processes. They are executed concurrently “inside” a process. Which does not mean that they are necessarily being executed on the same CPU core as their parent process; only that they share its memory. In general, this raises problems of memory integrity. To avoid those, Python has a Global Interpreter Lock (GIL). The GIL ensures that the reference counting that is at the center of the runtime’s garbage-collection is done safely and efficiently. It has a global lock on all resources shared between threads, and ensures that at most one thread modifies them at any given time. This is a simple, efficient, deadlock-free solution, with the drawback of hamstringing Python ability to parallelise tasks: with this restriction, your treads might as well all run on the same CPU core.

Processes are much heavier than threads, but since they are independent, with their own copies of all the essential context of execution, they can be spread over CPU cores. Therefore, multiprocessing can be used to achieve parallelism, whereas multithreading cannot. They both have their uses, however.

31.3 Python and concurrency: the three ways

There are three main ways to implement concurrency in Python:

- **Multithreading.**

  Accessible trough multithreading and concurrent.futures.ThreadPoolExecutor.

  Distribute tasks over light-weight threads. The multitasking is said to be *pre-emptive*, because the scheduler can interrupt a task at any time, including right in a middle of incrementing a variable, and give the CPU to another.

  This is most useful with several I/O-bound tasks. That is to say, if you have several tasks that spend most of their time waiting on the memory, the SSD or hard-drive, user-driven events, or the network, rather than doing long stretches of complex computations, then using multithreading can considerably speed things up. While a task is busy waiting, the scheduler executes another.

  Though they are “lightweight”, there *is* an overhead to multithreading, as we shall see shortly in our small experiments. It is a very bad idea to use multithreading on CPU-bound tasks — that is to say tasks that perform heavy computations, with little if any time spent waiting upon I/O. Since threads are not parallelised — because of the GIL — adding the overhead of task scheduling and context switching on top of things does not improve things at all. You still get one CPU core’s worth of horsepower, minus the overhead.

  For CPU-heavy tasks, you will want to use multiprocessing instead.

- **Asynchronous I/O.**

  Accessible through the `asyncio` module and the `async/await` syntax.

asyncio implements *cooperative*, or *non-preemptive* multitasking. Everything is run in a single thread — and thus a single CPU — and each task voluntarily cedes control to the scheduler whenever it makes sense for it to do so, using the **await** keyword. The scheduler is not otherwise at liberty to take control.

Provided the code is well thought-out, this can easily be the most elegant and highest performing approach, with very minimal overhead when compared to multi-threading.

Like multithreading, it is highly suitable for wrangling I/O-bound tasks, and worse than useless for CPU-bound ones.

Within the scope of this course, we can forget about asyncio.

◇ **Multiprocessing.**

Accessible through multiprocessing and concurrent. futures.ProcessPoolExecutor.

Distribute tasks over multiple CPU. There is quite a bit of overhead there, as as the program, complete with the Python interpreter, must essentially be duplicated, and complex communication must be put into place to share objects between processes.

Essentially, data must be serialised — that is to say, converted into a bit stream — for the sake of the transfer, and deserialised by the target process. The pickle module is used internally for this task. But not everything is cleanly serialisable by pickle. Anonymous **lambda** functions, for instance, are not.

These limitations mean that getting multiprocessing to work can be much trickier, *ceteris paribus*, than the equivalent multithreading.

Also keep in mind that, while multiprocessing is the *only* way to get more performance in CPU-bound tasks, the large overhead means that N times the cores does not quite translate into N times the performance, and multiprocessing is overkill on I/O bound tasks.

31.4 Examples and performance tests

Note: *The tests below were run on an Intel Core i7-7700K CPU @ 4.20GHz, with 8 logical and 4 physical cores. It should go without saying that you may get very different results depending on your hardware.*

Note 2: *Do not use Idle for those tests; it mishandles the standard output of threads, and does not display that of processes at all. I suppose that behind the scenes, Idle interacts with the interpreter process, and knows nothing of other processes it may spawn. The same might be the case in other editors. Use the console.*

Let us try all this out on simple examples, where tasks are independent. We shall use the high-level interface provided by concurrent.futures.

Since we shall need to do some performance testing, let us begin by defining a nice decorator (cf. Sec. 25.2[p174]: "Function decorators") for that purpose:

```
from time import sleep
import time
import concurrent.futures as cf
import os

def display_time(f):
    def F(*a,**k):
         x = time.perf_counter()
         res = f(*a,**k)
         y = time.perf_counter()
         print(f"TIME ELAPSED: {f.__name__}{a,k}: {y-x:.3f}s")
         return res
    return F
```

It is not very precise, as it can be influenced by many factors including the machine's global load, and does not attempt to mitigate those fluctuations like `timeit` does, but it is more suitable for complex code, and we only need a rough idea for our purposes. As you will see, the performance differences are quite stark.

Now, let us simulate an I/O bound task, and a CPU-bound one.

```
@display_time
def io(id):
    print(f"IO-bound operation {id} START {os.getpid()}",flush=True)
    sleep(1)
    print(f"IO-bound operation {id} STOP  {os.getpid()}",flush=True)
    return f"{id}X"

@display_time
def cpu(id):
    print(f"CPU-bound operation {id} START {os.getpid()}",flush=True)
    for i in range(1000):
        x = 2**10**6
    print(f"CPU-bound operation {id} STOP {os.getpid()}",flush=True)
    return f"{id}Y"
```

The **id** argument is just an arbitrary task identifier so we can differentiate the different "instances" of this task when they run. Running them, we see they both take some time to execute:

```
>>> io(0)
IO-bound operation 0 START 61699
IO-bound operation 0 STOP  61699
TIME ELAPSED: io((0,), {}): 1.028s
'0X'

>>> cpu(0)
```

```
CPU-bound operation 0 START 61699
CPU-bound operation 0 STOP 61699
TIME ELAPSED: cpu((0,), {}): 2.559s
'0Y'
```

Note that they have the same Process ID. This is normal, as everything is running in the main — and so far, only — Python thread.

You may want to comment out the `@display_time` decorators on those two functions; we have seen what we need, and we shall have more than enough output to deal with without them.

Now let us see how to deal with a large number of *independent* tasks — let us say ten. There are three ways we can organise the work. First, we can perform the tasks sequentially; for this, we define:

```
@display_time
def normalmap(*args): return list(map(*args))
```

Why do we use a *map* function? The idea is to associate, to each ID or input, the corresponding output by the task.

For I/O-bound tasks, imagine that we have a number of URL from which to extract information, stored in a list. Then we expect as output the list of the extracted data.

For CPU-bound tasks, we could have a list of numbers as input, and do primality testing on then, yielding a list of Booleans.

Here, we input tasks IDs, and get just enough output to know the task ID and the type of operation (IO vs CPU). But the architecture is the same for any kind of task — so long as all instances are independent.

Anyway, what we have here is the sequential strategy: execute all tasks in full, one after the other. Let us test how we do on I/O-bound tasks:

```
>>> normalmap(io, range(10))

IO-bound operation 0 START 49672
IO-bound operation 0 STOP  49672
IO-bound operation 1 START 49672
IO-bound operation 1 STOP  49672
...
IO-bound operation 9 STOP  49672
TIME ELAPSED: normalmap((<function io at ..>, range(0, 10)), {}): 10.011s
['0X', '1X', '2X', '3X', '4X', '5X', '6X', '7X', '8X', '9X']
```

Predictably, ten one-second tasks, one after the other, amounts to ten seconds. Now let us use multithreading. It is actually quite simple to implement; let us replace `normalmap` by a multithreaded version of `map`:

```
@display_time
def iomap(*args):
    with cf.ThreadPoolExecutor() as e: return list(e.map(*args))
```

The idea is that the Executor will handle a pool of worker threads — the number of which it determines as a function of the number of CPU cores — and distribute the tasks among them. It will collect the results, and free all resources when exiting the **with .. as** context.

Let us see how this performs:

```
>>> iomap(io, range(10))

IO-bound operation 0 START 49672
...
IO-bound operation 9 START 49672
IO-bound operation 0 STOP  49672
IO-bound operation 2 STOP  49672
IO-bound operation 1 STOP  49672
...
IO-bound operation 8 STOP  49672
TIME ELAPSED: iomap((<function io at ..>, range(0, 10)), {}): 1.013s
['0X', '1X', '2X', '3X', '4X', '5X', '6X', '7X', '8X', '9X']
```

Note that the order in which the tasks start and stop is a bit random; it depends on which worker happens to be available, or to finish first. Depending on system load, tasks may well start in order, and if they do they are a bit more likely to finish in the same order. Whatever happens in each test, the idea is that the order cannot be relied on in general.

The Process ID is still the same for everything, because all the threads are owned by the same, main, Python interpreter process.

Performance-wise, we are pretty much in the optimal case: we execute ten tasks in the time needed to complete just one. This scales pretty well; see on a hundred tasks:

```
TIME ELAPSED: iomap(..., range(0, 100)), {}): 9.017s
```

Now let us do the same thing with multiprocessing:

```
@display_time
def cpumap(*args):
    with cf.ProcessPoolExecutor() as e: return list(e.map(*args))
```

No surprises in this implementation; the Executor abstract class is the same for processes as for threads — that's the whole point of having an abstract class in the first place. So uses of ThreadPoolExecutor and ProcessPoolExecutor are interchangeable. . . or are they? Restore the @display_time decorator on io, and try

```
>>> cpumap(io, range(10))
```

```
AttributeError: Can't pickle local object 'display_time.<locals>.F'
```

Recall what I said earlier about serialisation. Because of `pickle`'s limitations some code that multithreads just fine straight up won't run with multiprocessing,

In our case, fix that by commenting out the decorator again, and let us do it for real this time:

```
>>> cpumap(io, range(10))

IO-bound operation 0 START 49735
...
IO-bound operation 7 START 49744
IO-bound operation 0 STOP  49735
IO-bound operation 8 START 49735
IO-bound operation 1 STOP  49738
IO-bound operation 9 START 49738
IO-bound operation 2 STOP  49739
...
IO-bound operation 9 STOP  49738
TIME ELAPSED: cpumap((<function io at ..>, range(0, 10)), {}): 2.057s
['0X', '1X', '2X', '3X', '4X', '5X', '6X', '7X', '8X', '9X']
```

Observe the different Process IDs, as expected for multiprocessing: one per worker process. If you keep track of the appearing PIDs for a large enough number of tasks, you will see there are — by default — as many worker processes as you have CPU cores.

Looking at the performance, things are not *too* bad, but it takes twice as long as multitreading. The overhead is amortised for larger numbers of tasks. . .

```
TIME ELAPSED: cpumap((<function io at 0x7f9f0e8d95a0>, range(0, 100)), {}):
13.065s
```

. . . but it is still a 44% time increase compared to multithreading. This reinforces the earlier point: don't use multithreading for I/O-bound tasks. It works, but it's just not the right tool for the job.

Now let us move on to CPU-bound tasks. Sequential execution yields predictable results:

```
>>> normalmap(cpu, range(10))

CPU-bound operation 0 START 49672
...
CPU-bound operation 9 STOP 49672
TIME ELAPSED: normalmap((<function cpu at ..>, range(0, 10)), {}): 24.339s
['0Y', '1Y', '2Y', '3Y', '4Y', '5Y', '6Y', '7Y', '8Y', '9Y']
```

Let us see how multithreading fares in that situation:

```
>>> iomap(cpu, range(10))
```

```
CPU-bound operation 0 START 49672
...
CPU-bound operation 7 STOP 49672
TIME ELAPSED: iomap((<function cpu at ..>, range(0, 10)), {}): 40.133s
['0Y', '1Y', '2Y', '3Y', '4Y', '5Y', '6Y', '7Y', '8Y', '9Y']
```

Oh boy! We managed to take nearly twice as long as sequential execution! The extra 16 seconds are basically the time you spent managing your threads instead of doing the computations. The threads themselves brought nothing at all to the table, since you can only compute on one CPU core at a time, because of the GIL. The computer spent all its time trying to do the work while being forced to run like a headless chicken from thread to thread. Don't do this to your poor computer. For CPU-bound tasks, do this instead:

```
>>> cpumap(cpu, range(10))

CPU-bound operation 0 START 50000
...
CPU-bound operation 9 STOP 50008
TIME ELAPSED: cpumap((<function cpu at ..>, range(0, 10)), {}): 7.011s
['0Y', '1Y', '2Y', '3Y', '4Y', '5Y', '6Y', '7Y', '8Y', '9Y']
```

Not too bad; let us see how that scales:

```
TIME ELAPSED: cpumap((<..cpu..>, range(0, 4)), {}): 2.438s
TIME ELAPSED: cpumap((<..cpu..>, range(0, 5)), {}): 3.586s
TIME ELAPSED: cpumap((<..cpu..>, range(0, 6)), {}): 3.642s
TIME ELAPSED: cpumap((<..cpu..>, range(0, 8)), {}): 4.689s
TIME ELAPSED: cpumap((<..cpu..>, range(0, 9)), {}): 6.855s
TIME ELAPSED: cpumap((<..cpu..>, range(0, 40)), {}): 22.917s
```

You might hear your computer's fans rev up for the last one, as we are maxing out all the processor's cores.

We see a significant jump between four and five tasks, and almost none between five and six. Recall that my CPU has four physical cores: that's why. Intel's "HyperThreading" lets the CPU pretend to have eight — and `ProcessPoolExecutor` will use eight workers by default — and that makes for more efficient context switching in some circumstances, but there is only so much it can do.

Let us limit ourselves to four workers instead:

```
def cpumap(*args):
    with cf.ProcessPoolExecutor(max_workers=4) as e: return list(e.map(*args))
```

We obtain:

```
TIME ELAPSED: cpumap((<..cpu..>, range(0, 4)), {}): 2.451s
TIME ELAPSED: cpumap((<..cpu..>, range(0, 5)), {}): 4.876s
TIME ELAPSED: cpumap((<..cpu..>, range(0, 6)), {}): 4.822s
TIME ELAPSED: cpumap((<..cpu..>, range(0, 8)), {}): 4.861s
TIME ELAPSED: cpumap((<..cpu..>, range(0, 9)), {}): 7.224s
TIME ELAPSED: cpumap((<..cpu..>, range(0, 40)), {}): 24.451s
```

Note that the cutoffs are much clearer here, with the times being multiples of the 2.4 seconds it takes to do a parallel batch of four tasks.

But. . . let's also note that the times are systematically *worse* than with 8 workers. I did not predict that result. I — being a perfect innocent in matters of CPU design — would have expected hyperthreading to be irrelevant at best for a purely CPU-bound job maxing all physical cores.

How can we understand those results? How can you perform *five* 2.4s jobs on four cores in only 3.5 seconds? I don't know, so I asked Dr. Bobelin, who apparently talks about such things in fifth year (4AS and 2SU options) in his "Architecture Security for the Cloud" class. Here is the analogy he gave: suppose you have three steaks to cook, and only two grills, each only large enough for a single steak. A steak must be cooked one minute for each side, for a total of two minutes, to be done. How much time does it take to cook all steaks?

The naïve approach is to fully cook two steaks first, fully occupying the grills for two minutes, then fully cook the last one, occupying a grill for an additional two minutes. Thus, everything is cooked in four minutes. Schematically, this is the strategy:

Minutes:	m_1	m_2	m_3	m_4	
Grill 1:	1	1	3	3	
Grill 2:	2	2			
Waiting:	3	3			
Done:			1,2	1,2	1,2,3

But there is another strategy: switch a steak after one minute:

Minutes:	m_1	m_2	m_3	
Grill 1:	1	1	2	
Grill 2:	2	3	3	
Waiting:	3	2		
Done:			1	1,2,3

Everything is cooked in three minutes. So the performance gains observed with eight workers are more a matter of giving the scheduler more opportunities to spread the load evenly than a reflection on physical versus virtual cores, specifically.

There are many more considerations that apply to performance gains while multithreading/processing, involving keywords such as "ALU", "FPU", "L1 cache", and more. Dr.

Bobelin said “predicting performance gains from multithreading/multiprocessing is very difficult”.

Unless *you* are a specialist in those things, I suggest you don’t blindly tinker with `max_workers`. If you feel the need to, perform experiments.

31.5 The dangers of multithreading: race conditions and deadlocks

Even though you should not need to deal with such problems directly for this course or the Python project, you need to be minimally aware of the kind of difficulties that arise with multithreading.

The most common is *race conditions*, where the outcome is dependent upon the timing of different threads. Imagine that we have a shared resource. Each threads reads it, computes, then modifies that resources.

Let us consider threads Alice and Bob; Alice reads the resource first; while she computes, Bob reads it as well. They both have read the same value, as Alice hasn’t modified the resource yet. She finishes her computation, and writes the resource. Then Bob finishes, and writes as well. The problem is, Bob never read Alice’s work; he just overwrote the result of her work with his.

Let us illustrate that:

```
shared = 0

def increment_shared(id):
    global shared
    for _ in range(10):      # each threads does shared += 10
        x = shared           # read resource
        time.sleep(0.0001)   # computation
        shared = x+1         # write to resource

def mciomap(*args):          # lots of worker threads
    with cf.ThreadPoolExecutor(max_workers=1000) as e:
        return list(e.map(*args))

from collections import Counter
c = Counter()

for _ in range(100):         # repeat experiment
    shared = 0
    mciomap(increment_shared, range(10))
    c[shared] += 1 # ten threads; shared should == 100

print(sorted(c.items()))
-------------------------------------------------------------
[(13, 6), (14, 27), (15, 39), (16, 17), (17, 6), (18, 1),
    (19, 1), (21, 2), (22, 1)]
```

Morally, we should have seen shared take the value 100, every time. In practice, in a hundred repetitions of this experiment, we saw values between 13 and 22. Roughly 85% of all attempted incrementations were overwritten by another thread.

This is a bit of an artificial example, because we invoke sleep, which strongly encourages the scheduler to move to another thread, but still, in theory, preemption may occur at *any* time.

You may think that it cannot happen in the middle of a simple instruction such as x += 1, but it absolutely can. Let us take a look at the bytecode for that:

```
>>> from dis import dis # disassembler
>>> dis('x += 1')
  1           0 LOAD_NAME                0 (x)
              2 LOAD_CONST               0 (1)
              4 INPLACE_ADD
              6 STORE_NAME               0 (x)
              8 LOAD_CONST               1 (None)
             10 RETURN_VALUE
```

This "simple instruction" is actually quite complex for the interpreter; it may stop between any of these lines. And going further, each of these bytecode instructions may well translate into several processor steps.

You just cannot trust, in a threading scenario, that your code will not be preempted at an inconvenient point. The worst thing is, most of the times, it won't. Replace the sleep by a small computation, and you will *probably* get a perfect [(100, 100)] score. Observing race conditions is really, really rare, in most code that contains them.

I said this was the *worst* thing about this. If you find yourself thinking "surely bugs being rare is a *good* thing?", give yourself a sharp rap on the knuckles, this is a terrible way of thinking. Recall the philosophy of **assert**ions.

We *want* incorrect programs to fail; we want them to fail obviously and we want them to fail fast. A student trying to get a passing grade for shoddy work may be thankful that the bugs stayed under the rug during the demonstration; but subtle bugs like that are the bane of a developer.

An undiagnosed race condition may cause crashes, or data loss, or more generally inconsistent behaviour, obvious or subtle, weeks or months after the software is widely deployed, and can be almost impossible to trigger on purpose unless you already know exactly where the problem is. There is *some* weird problem *some*where in our 100 000 lines codebase; sometimes our radiation therapy machines spit out lethal doses of radiation, and kill the patients. Oops. Go debug that.

Here I am referring to the Therac-25 case (1985–87). This, and other cases concerning critical systems, are the motivation for the use of formal methods to obtain proofs of correctness for concurrent systems. This is the object of next year's Formal verification class. You can freely join the class and read my lecture notes on the topic if you're interested; if not, I shall see you next year. End of sponsor's message.

Back to race conditions. They can be prevented by using *locks* (or "*mutexes*"[ad], or "*semaphores*[ae]"). Those are objects whose possession gives you the right to access the resource. A thread can acquire the lock, then release it. When acquiring the lock, you have to wait until whoever owns it releases it. The lock itself is implemented in such a way that its fundamental operations are atomic (i.e. non-preemptable) so that at most one thread may hold the lock at any time. This enforces mutual exclusion: only one thread may access the resource at any given time.

So thanks to the magic of locks, problem solved, right? Right? Sure, if you are careful, and use locks every time a shared resource is involved, you won't have race conditions. Yipee. You have traded the "race condition" class of annoying, subtle, perverse problems for the class of annoying, subtle, and only slightly less perverse problems known as "deadlocks".

So you have ten threads sharing the Lock A. What happens if one of them decides not to give it back? Deadlock; nobody else can do anything. Why would a thread decide not to give a lock back? Is it *evil*? Well, maybe it has an intrinsic bug, yes. But it need not have. Imagine that it needs to acquire locks A and B, before doing its thing and releasing both. B is currently in the hands of another thread, which wants to acquire B, then A, do its thing, and release.

The first has A, waits on B. The second has B, and waits on A. There are enough resources for everyone, and everyone is willing to release what they have, nobody is doing anything intrinsically wrong, yet we find ourselves in a situation where nobody can do anything anymore. The program grinds to a screeching halt. *Deadlock*.

Making sure nothing like that ever happens is not easy. By not easy, I mean *undecidable* in the general case, and "people who can't afford to have that happen to their critical systems spend millions of € on testing and formal methods to ensure that doesn't happen". And it's not always enough. For more on that, see you next year in the Formal Verification class.

In the meanwhile, just avoid having to deal with any of that if at all possible. Race conditions are icky, so global variables + threads = Nope. Locks are tricky, so preferably, have someone else worry about them.

During the project, if you use a GUI framework, or `pygame`, or . . . , it will *probably* hand you an event queue or something to that effect. This is good, because that means it falls on the framework to make sure that the various processes and threads involved do not step on each other's toes when leaving messages.

Then you can set camp in the main event loop, read the messages, and keep your eyes down. If you do need to share things with other threads, be careful, and be afraid, be very afraid.

[ad] MUTual EXclusion

[ae] Semaphores are a bit more general, but this is not important at that point.

For projects that involve a global "world state" and several actors/threads — which is often the case — it may be a good idea to have the world state modified *exclusively* through messages sent to the main even queue, rather than having to wrangle a lock on your world state. Or maybe not. You will have to figure that out.

32 TODO list

slice assignment

cartesian iterable

```
for elem, ekey in ((e, key(e)) for e in iterable):
```

gene ngram + timeit compare

eval()

better timeit details

global nonlocal

cProfile snakeviz

GUI

next(,default) object() pattern as in zip() code

profiling

```
python3 -m cProfile -o profile.prof ./tests.py
snakeviz profile.prof
```

I raged so much because of this, vs OCaml semantics: https://stackoverflow.com/questions/2295290/what-do-lambda-function-closures-capture

imports:

```
if __name__ == '__main__':
```

image pythontutor shallow copy with transparent background instead of white.

Usage of assertion vs exception: recovery possible ?

```
if __debug__
```

calendar: proleptic

```
@contextmanager
def cd(dir):
    currdir = os.getcwd()
    os.chdir(os.path.expanduser(dir))
    try: yield
    finally: os.chdir(currdir)
```

pattern: singleton binding:

```
return sum( f(x)*(h-l) for (l,h) in partition(a,b,n) for mid in [mid(l,h)] )
```

For debugging :

```
assert all(factorial(X:=n) == forig(n) for n in range(10)), X
```

section on how to replace bash scripts by Python; os, Path libs etc.

Strange handling of exceptions in comprenhensions:

```
for a,b,c in range(-5,5):
    print(a,b,c)

Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    for a,b,c in range(-5,5):
TypeError: cannot unpack non-iterable int object
bool( True for a,b,c in range(-5,5))
True
bool( prinnt(a,b,c) for a,b,c in range(-5,5))
True
bool( print(a,b,c) for a,b,c in range(-5,5))
True
bool( assert False for a,b,c in range(-5,5))
SyntaxError: invalid syntax
bool( 1/0 for a,b,c in range(-5,5))
True
bool( 1/0 for a,b,c in range(-5,1/0))
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    bool( 1/0 for a,b,c in range(-5,1/0))
ZeroDivisionError: division by zero
```

```
'REGEXP': r'/(?!/)(\\/|\\\\|[^/])*?/[%s]*' % _RE_FLAGS,
```

random hash salting of sets

http://dabeaz.com/coroutines/ https://stackoverflow.com/questions/9708902/in-practice-

https://discuss.python.org/t/structural-pattern-matching-should-permit-regex-string-m 22700/9 Personalised str patterns (regexp, prefix, etc) MCre, MCpref, ...

Python versions history

```
l =["abc","ABC"]
zip(*zip(l)) == l
False
list(zip(*zip(l))) == l
False
list(zip(*zip(l)))
[('abc', 'ABC')]
```

Theorem ?

Part III

Python Exercises

33 Foreword

The sections that follow are your first exercises, and your entry point into this Python class. Alternate between answering the questions below and reading the lecture notes in Part II[p47].

Whenever I direct you to a specific section, make sure to read it carefully, and to seek out, in the lecture notes, whatever information you may be missing to understand those sections.

Note that you will be expected to have read and more or less understood all of Part II[p47] by the end of the semester. Points that are a bit beyond the scope of this class and will not be tested in the exam are indicated as such in the notes; you still need to read those parts, though.

We have eleven **lab classes** (three of which are in autonomy) and two **lectures**, roughly at the $\frac{1}{3}$ and $\frac{2}{3}$ marks.

During the lectures, I shall discuss the solution to as many exercises as time allows, and take your questions.

The new key insights and core competencies targeted by each question or exercise are written in this format. They should become clear *after* having solved the questions and having discussed them with me.

33.1 An open letter to Python Gods

A note to those who already know Python. Or *think* they do.

You may skip this diatribe if you *don't* think that.

Even if you are already well practised in Python, please *do not rush through the exercises at all speed*; do not skip them to get to something more "interesting" more quickly. It is quite unlikely that you already know *everything* in Part II[p47], and there is a difference between **(1)** having enough tools to be able to cobble together a purported solution to a problem, and **(2)** using the right tools to quickly produce short, efficient, reliable, and well-tested code that fully satisfies its specification. Those exercises, no matter how trivial some may seem to you, are opportunities for me to engage with you on those topics, check for bad habits you may have acquired, etc; do not neglect them.

Also note that, during the exams, I often ask questions testing your knowledge of specific Python idioms and structures, such as comprehension expressions, the limitations of sets and dictionaries, the specificity of Python's `int` type compared to that of other languages, etcetera. I had some cases in previous years of students entering the class fully confident that they already knew Python, because they had successfully completed some project in it. Thus, they paid no attention and did not put any effort into this class, and learned nothing new. They were then positively outraged to receive a failing grade in the final exam; dismayed that writing Java or C roughly translated into Python's syntax did not satisfy. Do not be them.

34 Basic data types, expressions, and functions

34.1 Conversion Celsius ↔ Fahrenheit

def <function>; return <expression>; assert for defensive programming

La formule de conversion entre ces deux unités étant

$$F = \frac{9}{5}C + 32\,,$$

nommer, écrire et documenter les deux fonctions de conversion `F_to_C` et `C_to_F` pour passer d'une unité à l'autre.

Quelles sont les conditions d'utilisation de ces fonctions ? As a reminder, the absolute zero is −459.67°F and −273.15°C.

On veillera à utiliser des assertions (cf. Sec. 22.6.6[p107]: "Assertions: cheap unit testing and preconditions enforcing") afin de tester ces conditions d'utilisation. Please read that section very carefully. In particular, do not burden your assertions with redundant error messages.

Notons que, bien que le type des arguments d'entrées fasse moralement partie des préconditions de toute fonction, on ne demande pas de le vérifier programmatiquement dans ces TDs, et il est peu idiomatique de le faire en Python.

Do not use `input()`! That is what the function's arguments are for. If I ever want user interaction, I shall ask for it explicitly. Use `print`s and, whenever possible, `assert`s in your code to test the functions on different values.

Do not confuse `return` and `print`: whenever I ask for a function, it must `return` something, so that I can use the function in later computation, not `print` it.

34.2 Floating-point comparison: almost there

float has finite precision; bool/predicates are very simple;
assert for cheap unit tests

Let us check that our two conversion functions are coherent with one another, by testing that their absolute zeros match.

However, we cannot simply test equality between floating point numbers, for reasons discussed in Sec. 22.2[p79]: "Floating-point numbers: `float`": there *may* be a loss of precision:

```
>>> F_to_C(-459.67) == -273.15
False

>>> F_to_C(-459.67)
-273.15000000000003
```

Although you may or may not observe it depending on the exact way you performed the computation:

```
>>> [ (a,b)
      for F in range(20)
      for a,b in [[5/9*(F-32), 5*(F-32)/9]]
      if a!=b ]

---------------------------------------------

[(-15.555555555555557, -15.555555555555555),
 (-12.222222222222223, -12.222222222222221),
 (-11.666666666666668, -11.666666666666666),
 ( -7.777777777777779,  -7.777777777777778)]
```

Instead of running that risk, we shall test whether the two values are *very, very close*.

(21) Define a predicate `isalmost(n,m,d=1e-13)` that tests whether n and m are at a distance at most d [af].

(22) Verify that the following assertions are satisfied:

```
assert isalmost ( F_to_C(-459.67) , -273.15 )
assert isalmost ( C_to_F(-273.15) , -459.67 )
assert all( isalmost( efc:=F_to_C(C_to_F(c)), ec:=c )
       and  isalmost( efc:=C_to_F(F_to_C(c)), c )
       for c in range(-273, 200) ), (ec, efc)
```

(Almost) always leave your assertions in your code, to prevent future regressions.

34.3 Taking root

34.3.1 Greatest root

reading a specification and enforcing valid inputs with `assert`; `None` as `null`

Écrire et documenter une fonction `greatest_root` telle que `greatest_root(a,b,c)` retourne la plus grande racine réelle du polynôme de second degré $ax^2 + bx + c$ si elle existe, et `None` sinon. See Sec. 22.5[p98]: “Nihilism: `NoneType`: expression versus statement”.

[af] Such a basic tool to manipulate floating-point numbers is of course provided in the standard library. This function is a simpler re-implementation of math.isclose.

Quelles sont les conditions d'utilisation (ou *préconditions*) de cette fonction ?

Indication: do all tuples $(a, b, c) \in \mathbb{R}^3$ describe a valid polynomial of the second degree?

Note: as mentioned in Sec. 22[p77]: "Basic data types", in Python, it is not idiomatic to test the type of parameters explicitly. While it is true that testing whether a, b, c are numerical values would be pertinent , let's not do that.

Once you have worked out what the preconditions are, make sure to enforce them through an **assert**.

Quick reminder from high school: the roots of $ax^2 + bx + c$ are given by

$$\frac{-b \pm \sqrt{\Delta}}{2a},$$

where $\Delta = b^2 - 4ac$ is called the *discriminant*. They are real if $\Delta \geqslant 0$.

The following assertions must be satisfied:

```
assert greatest_root(1,1,1) == None
assert greatest_root(1,-2,1) == 1
assert greatest_root(4,-4,-24) == 3
# we can do direct comparison because we know the results are exact,
# at least in this case, and we only want to prevent regressions
# This is overspecification, and might refuse correct versions of
# the function. However, we can deal with those problems, using
# isalmost, as they arise.
```

Feel free to add some more.

34.3.2 Real roots

carefully reading and respecting the specification
virtues of homogeneous return types and containers
unit tests on ranges of value to enforce consistency of implementations

Write a function `roots(a,b,c)` returning a tuple containing the real roots of the second-degree polynomial $ax^2 + bx + c$, in no particular order.

Note how this specification is formulated: you must *always* return a tuple. Do not return a tuple sometimes and `None` some other times.

The following assertions must be satisfied:

```
assert roots(1,1,1) == ()

assert roots(1,-2,1) in [ (1,1), (1,) ]
# I didn't specify whether single roots should be repeated,
# so both versions are valid

assert set(roots(4,-4,-24)) == {-2, 3}
```

```
# I did not specify the order of roots, hence the set test
```

Write an assertion testing, for all valid values of $a, b, c \in \llbracket -5, 5 \rrbracket$, that the outputs of `greatest_root` and `roots` are coherent.

This can be done in one or two logical lines if you can have read Sec. 24.5[p153]: "Comprehension expressions" and Sec. 23.2[p112]: "Conditional expression: `.. if .. else ..` ternary operator", but I do not *require* that at this point.

35 Dungeons and magic methods

Object Oriented Programming (OOP)
implementing classes
understanding magic/dunder methods

You will want to have taken at least a cursory look at Sec. 21.2[p62]: "A few words about Object Oriented Programming (OOP)" and Sec. 27[p183]: "Object Oriented Programming in Python".

Let us implement dice rolls, in the style of Dungeons & Dragons[ag]:

> *Dice rolls are described with expressions such as "3d4+3," which means "roll three four-sided dice and add 3" (resulting in a number between 6 and 15). The first number tells you how many dice to roll (adding the results together). The number immediately after the "d" tells you the type of die to use. Any number after that indicates a quantity that is added or subtracted from the result.*

You will need to use the `random.randint` function.

Our goal in this exercise is to implement a "function" `d` such that we can define a dice, and then roll it as many times as needed, as follows:

```
>>> d20, d6 = d(20), d(6)  # let d20 be a 20-sided die, and d6 a 6-sided one
>>> d20
d20: 14
>>> d20
d20: 4
>>> d20
d20: 7
...
```

Of course the values displayed must be uniformly random integer values between 1 and 20, inclusive, not necessarily $14, 4, 7, \ldots$.

[ag] https://www.d20srd.org/srd/theBasics.htm#dice

What's going on here is that each time `d20` is converted into a string for the purpose of representing its value (which is what happens when you put a value in the prompt >>>), the dice is rolled and the value displayed.

Then we will extend this behaviour to support addition and multiplication.

(23) Justify (i.e. write a sentence explaining) that `d` cannot possibly be a classical function of the form

```
def d(n): return random.randint(...)
```

(24) Implement a class `d(N)`, representing an N-sided die.

For instance, you can declare:

```
>>> d20, d6 = d(20), d(6)    # let d20 be a 20-sided die, and d6 a 6-sided one
```

Note: see Sec. 21.5.2[p67]: "Parallel variable assignment" if that syntax confuses you.

`d` will have an attribute `N`, giving the number of faces of the dice:

```
>>> d20.N, d6.N
(20, 6)
```

To do this, you will need to understand the rough ideas under then notions of *class*, *object*, *instance*, and *attribute*, as well as the `__init__` method. Sec. 27.4[p187]: "Constructors: beware of mutable structures" can serve as a good reference point for the latter — though, again, you will need to understand at least some of what comes before.

(25) Implement for `d` a method `.roll()` returning the value of a roll (uniformly random on $[\![1, N]\!]$).

You will use `randint` for this. Throughout this exercise, *no other method* may call `randint`.

You should observe something like:

```
>>> d20.roll()
13
>>> d20.roll()
8
>>> d20.roll()
1

>>> type(d20)
<class '__main__.d'>

>>> type(d20.roll())
<class 'int'>
```

(26) Now let's implement the behaviour presented in the introduction:

```
>>> d20
d20: 14
...
```

As well as things like

```
>>> f"I roll {d20}"
'I roll d20: 11'
...
```

(This is a string conversion in an fstring; see Sec. 22.4.10.4[p94]: "The good stuff: formatted string literals" if you don't understand this expression; it's not central to this immediate exercise.)

In other words, we want that, each time `d20` is converted into a string for the purpose of representing its value, the dice is rolled and the value displayed.

For now out the output looks like:

```
>>> d20
<__main__.d object at 0x7447d25af680>
```

To implement what we want, you will need to read and understand Sec. 27.7[p191]: "String representations `str` and `repr`".

(27) Rather than having to call `.roll()` explicitly in all circumstances where we want to use the roll value in a computation, we would like to have dice that can be converted into integers:

```
>>> int(d20)
15
>>> int(d20)
16
```

You well need to read Sec. 27.10[p197]: "Special, magic, dunder methods" and implement `__int__`. This is a long and difficult section, so for now be efficient in skimming through it to get the relevant information for the problem at hand. Read in more detail during your personal study time.

(28) Now, we would like to be able to perform arithmetic on rolls

```
>>> d20+100
106
>>> d20+100
101
>>> type(d20+100)
<class 'int'>
```

Currently, if we try that, we get

```
TypeError: unsupported operand type(s) for +: 'int' and 'd'
```

Just because we can convert to `int` does not mean that the conversion is done automatically. . .

Implement the `__add__` method to make it work as expected.

(29) So far, so good, but wait, if we try

```
>>> d20+100
108

>>> 100+d20
TypeError: unsupported operand type(s) for +: 'int' and 'd'
```

Isn't addition commutative? What's going on?

Take some time to understand the problem, which is explained in great detail in Sec. 27.10[p197]: "Special, magic, dunder methods" — but do not spend the whole lab class on this! — then implement the `__radd__` method so that things work as expected:

```
>>> 100+d20
111
>>> 100+d20
120
```

(30) Now let's do multiplication:

```
>>> 10*d6
29
>>> 10*d6
34
```

Which magic method do you need to implement?

Also note that the semantics of `10*d6` is *not* "roll a d6 once, and multiply the roll value by 10", but "roll a single d6 10 times, and add the roll values", or equivalently "roll 10 individual d6s, and sum their rolls".

You can tell because neither 29 nor 34 are multiples of 10 :-)

You will implement the relevant method in one line, using `sum`. See Sec. 24.5.3.4[p159]: "Reductions" for examples of how to use `sum`. You do not need to fully understand how that syntax works at this point — but you will later, believe that!

Now, this should work:

```
>>> -100 + 10*d20 + 40
72
```

```
>>> -100 + 10*d20 + 40
62
```

(31) Test the following code, which performs a large number of 3d6 rolls and plots the values obtained:

```
from collections import Counter

N = 100000
for v, c in (l :=sorted(Counter( 3*d6 for _ in range(N) ).items())):
   print(f"{v:2} {'='*(c//500)}")

print("\navg", sum(c*v for v,c in l)/N,
     "\nexpected avg", 3/6*sum(range(1,6+1)), "=", 3*(6+1)/2)


---------------------------------------------------------------------
 3
 4 ==
 5 =====
 6 =========
 7 =============
 8 ===================
 9 ======================
10 =========================
11 ========================
12 =======================
13 ===================
14 ==============
15 =========
16 =====
17 ==
18

avg 10.48865
expected avg 10.5 =10.5
```

You can have very minor numerical variations in the observed average `avg`, but it should be *very close* to 10.5, which is the mathematical expectancy of 3d6, and the curve obtained should approximate the form of a nice normal distribution.

Reminder: In case you where wondering, the mathematical expectancy of dN — that is to say a random variable with outcomes $[\![1, N]\!]$ with uniform probability $\frac{1}{N}$ — is given by

$$E[dN] \;=\; \sum_{k=1}^{N} \frac{1}{N} k \;=\; \frac{1}{N} \sum_{k=1}^{N} k \;=\; \frac{1}{N} \frac{1}{2} N(N+1) \;=\; \frac{1}{2}(N+1)\,.$$

Thus the expected average for 3d6 is

$$E[3d6] \;=\; 3E[d6] \;=\; 3 \times \frac{6+1}{2} \;=\; 3 \times \frac{7}{2} \;=\; 10.5\,.$$

(32) Now let's implement another type of multiplication, so that the semantics of d6*10 (as opposed to 10*d6) is indeed "roll a d6 once, and multiply the roll value by 10".

Replacing 3*d6 by d6*3 in the previous code, you should get something like this:

```
 3 =================================
 6 =================================
 9 ================================
12 ================================
15 =================================
18 =================================

avg 10.50894
```

Now 3*d6 and d6*3 are both valid rolls. The expected average is the same, but the resulting distribution is very different. . .

Moral: it's up to you what the operators of the types you define mean. Just because it's * doesn't mean it absolutely has to be commutative. Certainly there is nothing in Python to enforce that. Pick semantics that make intuitive sense, and document them properly.

Oftentimes, the functions implemented by operators correspond to clearly and explicitly named methods which can be used as less compact but more legible alternatives. For instance, we could have chosen to write things like 3*d6 as d6.sum_multiple_rolls(3) instead, or in addition.

Again, it's up to you to define operators as you like, but don't go overboard with it. Sometimes the right move is to just use explicitly-named methods and leave all the weird symbols alone.

(33) Compare the distributions of 3*d6+0 and 3*(d6+0). What's going on?

Note: while d6 represents a random variable, we chose to implement + so that d6+0 becomes an integer. This choice has consequences, and one of them is that d6 and d6+0 are very different objects.

That is fine in this context — we just want to write expressions like 10*d6 + 3 to roll the dice and play the game — but if we wanted to implement random variables more seriously we would need d6+n, where $n \in \mathbb{N}$, to be of type "random variable". Developing this is much more complicated than what we did, and beyond the scope of this exercise.

Note that later on, for instance in question (42)[p252] and others, we will be using mathplotlib to generate graphs and write magic things like

```
x = np.linspace(-2,2,100) # x varies in [-2,2], 100 uniform samples
```

and expressions such as 2 * x**2, and magically plot the curves.

What will be going on behind the scenes is the same kind of trickery we just employed, whereby things that look like simple arithmetical expressions actually hide something more abstract, here random variables / samplings.

(34) I hope you get the idea, and would be capable of supporting subtraction, division, etc, without further guidance. . .

36 We all `float` down here!

It is nice that we can solve the zeroes of a second-degree polynomial. Now let us do the same thing for the zeroes of *any* continuous function f. We shall use approximate numerical methods.

Keep in mind that floating point numbers are dangerous; don't turn your back on them. Be sure to read Sec. 22.2[p79]: "Floating-point numbers: `float`" in that regard.

36.1 The Zero Dichotomy

thinking recursively
functions are first-class citizens
less trivial effects of `float` precision loss
beware false simplicity of `int`
hide recursion from user with recursive subfunctions
why inclusive/exclusive ranges $[\![a, b[\![$ are cool
asymptotic complexity vs. small instances

Recall the classical theorem of intermediate values:

Theorem 1 (Intermediate Values)**.** *If a function $f : \mathbb{R} \to \mathbb{R}$ is defined and continuous on a real interval I, then its image $f(I)$ is also an interval.*

It has an important corollary:

Corollary 2 (Bolzano's theorem – BT)**.** $\forall a, b \in I$, if $f(a)f(b) \leqslant 0$, then $\exists z \in [a, b] : f(z) = 0$.

Proof. $f([a, b])$ is an interval; it contains $f(a)$ and $f(b)$. Therefore it contains $[f(a), f(b)]$ (or $[f(b), f(a)]$). We have $f(a)f(b) \leqslant 0$, which means that if either $f(a)$ or $f(b)$ is positive, then the other must be negative, and vice versa. Thus we have $0 \in [f(b), f(a)] \subseteq f([a, b])$.

In short: f changes sign between a and b, so its curve must cross the abscissa. □

We are going to use Bolzano's theorem to implement a **dichotomic search** — or more precisely a binary search; also known as the bisection method — for a zero; that is to say, a value z such that $f(z) = 0$. What is a dichotomic search, you ask? That is what you do when you search for a word in the dictionary — I mean a *paper* dictionary, not an online one. (If you are not old enough to have manipulated one of those, use your imagination).

The principle is simple: you open the dictionary at some place, roughly down the middle (maybe you have better guesses if the word begins by A or Z, but it matters little in the end), and you determine, using the alphabetical order, whether the word you are looking for is to the left, or to the right, or your current position. Therein lies the "dichotomy", the "bisection": you have two mutually exclusive options: left and right. If you cut exactly in the middle each time, it's a binary search.

Then you repeat that process, with either the left or right part of the dictionary, which is of course much smaller, again and again until you are close enough to your target.

This is a very efficient process: $O(\log_2 N)$ where N is the size of the dictionary — or more generally of any sorted list. Not convinced of the logarithm? if the dictionary is twice as big, you open it at the middle, and choose left or right, and you're back to the original size. One more step to handle twice the size.

Let us do that to search for zeroes: start with a and b such that f changes sign between them, cut down the middle of $[a, b]$, and then there must be a zero either left or right; choose by testing on which side there is a sign change, and repeat until you're close enough to your taste. That is to say, until $|a - b|$ is smaller than your desired precision.

(35) **TODO VH:** Separate into two questions: normal precision and max, with assertions for each.

Write a function `di(f,a,b,d=1e-16)`, where $f : [a, b] \to \mathbb{R}$ is continuous and changes sign on $[a, b]$, that returns an approximation of a zero z, ideally within a precision d.

In practice $\mathbb{R}$ will be approximated by **`float`**. You are not required to test whether the inputs satisfy the assumptions — how would one even test continuity of f?.

By "approximation of a zero z, *ideally* within a precision d", I mean any z' such that $|z - z'| \leqslant d$, if floating-point precision is sufficient to achieve that, or the best z' you can realistically get if not. Note that f is completely irrelevant to this notion of precision.

Tip: *there are two ways to write a dichotomy: recursively, and with a* **`while`** *loop. Write it recursively first, it's simpler. You'll write it with a* **`while`** *in the next question.*

Tip2: *Sec. 21.5.5[p68]: "Functions are first-class citizens".*

As an example of how it must behave, let us approximate $\sqrt{2}$ as the positive root of $X^2 - 2$:

```
from math import sqrt

def g(x): return x**2 - 2

>>> res = di(g,1,2)
1.414213562373095
```

```
>>> sqrt(2)
1.4142135623730951
>>> sqrt(2)-res
2.220446049250313e-16
```

Printing each step of the process with `print(a,m,b, b-a)`, where `m` is the middle, we have:

```
1 1.5 2 1
1 1.25 1.5 0.5
1.25 1.375 1.5 0.25
1.375 1.4375 1.5 0.125
....
1.414213562373095 1.4142135623730951 1.4142135623730954
                                            4.440892098500626e-16
1.414213562373095 1.414213562373095 1.4142135623730951
                                            2.220446049250313e-16
```

This also illustrate why I said *ideally* within a precision d. Observe that our precision objective is not actually met in the example. Indeed I could run with `d=1e-160` and get the very same result.

This is because we are dealing with floating-point numbers, and thus loss of precision. It may well be that the middle becomes impossible to distinguish from a or b — because of loss of precision — before $|a - b|$ becomes quite small enough. In that case we run the risk of entering an infinite loop, so we must return the result we have now. That is precisely what happened here: a and m are the same in the last line, so our current approximation is the best we can do.

Keep that in mind, and be sure your function doesn't enter infinite loops. Also keep in mind that you may write a function that is correct, but does not have the same precision errors as mine because you have not written the computations in the *exact* same way, and thus will behave slightly differently on the same examples. Such are the joys of working with floating point numbers...

At the end of the day, the following assertion must hold:

```
from math import nextafter as na, inf
def neigh(f): return na(f,inf), na(f,-inf)
assert all( abs(n**.5 - (r:=di(lambda x:x**2-n, 0,99,d)) ) <= d
            or r in neigh(n**.5) # result is closest float
        # for di in [di, di_while]
        for n in range(20) for d in [1e-7, 1e-16, 1e-32] ), r
```

It ensures that either the required precision is achieved, or we are in a case where there is no representable floating-point value strictly between `a` and `b`, like in the case `a=1.414213562373095, b=1.4142135623730951`, and so we lack information to choose between the two.

(36) Now write a function `di_while`, as in the previous questions, but implemented using a **`while`** loop.

For assertions, simply uncomment the line

```
for di in [di, di_while]
```

in the previous question's assertion.

(37) So we have made a big deal of **`float`**'s precision problems in the last question. Is the grass greener with **`int`**?

Write a function `find(x,l)` that returns an index of `x` in the sorted list `l`, if it exists, and `None` if `x` does not appear in `l`.

The fact that `l` is *sorted* must immediately suggest to you to use a dichotomic search! Why write a linear algorithm when you can write a logarithmic one?

If an element appears twice, any suitable index may be returned; it need not be the first or last or anything specific.

The function must implement a dichotomic search and satisfy the following assertions:

```
assert find(3,[3]) == 0

assert all( (fe:=find(ie:=i,ir:=r)) == (i if 0<=i<N else None)
            for N in range(9) for r in [range(N)]
            for i in list(r)+[-1,N] ), (ir, ie, fe)

assert all( l[fe:=find(ie:=i,il:=l)] == i
            for N in range(5) for R in [2,3]
            for l in [[ e for e in range(N) for _ in range(R) ]]
            for i in range(N)), (il, ie, fe)
```

You will code the dichotomic search recursively. Since the bounds `a,b` are not parameters of the function, you will need to hide them from the user. A good approach taken by Python's library[ah] is to pass them as optional arguments, but this has two drawbacks as a general solution for that type of need: (1) each recursive call will need to pass the same `x,l` again, which is tolerable here but lacks legibility when there are more arguments to repeat, and (2) sometimes it makes no sense to let the user see and modify the parameters you recurse upon. What you will do instead is use a recursive subfunction, here called `z`. Your function will thus be of the form:

```
def find(x,l):
    def z(a,b):
        ....
    return z(...)
```

[ah] https://docs.python.org/3/library/bisect.html

Now, on to the algorithm itself. If you code the search naïvely, you will probably use `a,b` as inclusive bounds, and you are very likely to run into infinite loop problems because... what is the integral middle of $[\![0, 1]\!]$? Whether you use floor or ceiling, you run into the same "the middle is confused with a bound" problem as in the previous question, except this time, you cannot stop immediately on grounds that maximum possible precision has been achieved.

It is quite possible to write a correct search that way, but it takes a lot of strategically placed `+1` and `-1` to make it correct, and it is not immediately obvious when reading the code that it is correct. So let's do things a bit differently, so that correctness is more obvious.

We shall take a leaf from the "modern" way of representing integer ranges, using $[\![a, b[\![$ instead of $[\![a, b]\!]$; that is to say, the lower bound is inclusive, but the upper bound is exclusive. Note that this is the convention adopted by Python's `range` and slice notation, and used in many other languages as well. It has many good properties that make working with ranges easier:

- $b - a$ is the length of the range, not $b - a + 1$,
- 0 and the length of the collection are the starting bounds, not 0 and length -1,
- cutting a range does not require $+1$ or -1 anywhere either: $[\![a, b[\![= [\![a, m[\![+ [\![m, b[\![$.

We will take advantage of that here. We shall write the computation of the middle as

```
m = a + (b-a) // 2
```

that is to say, we keep our starting point, but divide the length `b-a` of the search space by 2. We test for length 0 and 1 for our stopping conditions, and otherwise, we know that the remaining range, being of length at least 2, will split nicely. Note that it does not actually matter whether we use `//` or `floor` or `ceil` in the computation of $\frac{b-a}{2}$.

Last constraint, do not test equality at every loop. Just one inequality will do, until you only have one element left.

The moral of the story is that, despite having no precision loss problems or even integer overflows (in Python!), `int` is not necessarily simpler to handle than `float`. Some thought is required to handle bounds correctly, and the inclusive/exclusive convention makes it much easier.

Let us test the performance of our function: the code below compares the performance of `find` and that of the default `list.index` method, which performs a linear traversal of the list, as does the `x in l` construct.

```
def time_test():
    from timeit import timeit
    print(f"len\tindex\tdicho\tratio")
    for N in [10**n for n in range(9)]:
        l = list(range(N))
        i = [i*N//10 for i in range(10)]+[N-1]
        ti = sum( timeit(lambda: l.index(k), number=1) for k in i )
        td = sum( timeit(lambda: find(k,l),  number=1) for k in i )
        print(f"{N}\t{ti}\t{td}\t{ti/td}")
```

The generated text isn't pretty but can be pasted into your preferred spreadsheet software.

However you visualise it, you should get something like that:

len	index	dicho	ratio i/d
1	4.92E-06	9.13E-06	0.5388829451
10	6.00E-06	1.71E-05	0.3514938517
100	7.39E-06	1.79E-05	0.4123655766
1000	3.60E-05	2.29E-05	1.57186566
10000	4.36E-04	3.26E-05	13.38771158
100000	0.004439856	4.32E-05	102.6793683
1000000	0.030203839	4.33E-05	698.1933006
10000000	0.272337098	6.73E-05	4046.012322
100000000	2.78551715	8.81E-05	3.16E+04

From this, the following conclusions can be drawn: `index` is slightly more efficient than `dicho` for small lists. Somewhere between lists of size 100 and 1 000, (from more testing, the inflection point seems to be around 625) this changes, and `dicho` becomes orders of magnitude more efficient for large lists. This is exactly what you expect when comparing a simple linear algorithm to a more sophisticated logarithmic one. The latter is more expensive to set up, which does not necessarily pay on small instances, because there is little time there to gain anyway, but crushes the simple algorithm on large instances, where the cost of the increased sophistication is dwarfed by the time gains.

Of course, this discussion elides the question of the cost of making sure the input list is ordered in the first place. . .

If you run `time_test` and your version of `find` does not exhibit the same asymptotic behaviour, you have not correctly implemented the dichotomic search; try again!

36.2 This is all very derivative. . .

functions are nice objects, easy to pass *and return*
handling optional arguments
difference between symbolic math function and python function object
mathplotlib for interactive visualisation of data

Let's get back to our search for zeroes in continuous functions. A binary search is quite efficient, as we have seen, but we can do better. We could use the slope of the tangent line of the curve at a given point to guide the way, much more efficiently, towards the zero. But before we can do that, we must be capable of computing — an approximation of — that slope.

That is to say, we want to numerically approximate $f'(a)$, the derivative of f at point a. It is defined by the classical formula

$$f'(a) \quad = \quad \lim_{h \to 0} \frac{f(a+h) - f(a)}{h} \,,$$

and can thus be approximated by

$$f'(a) \quad \approx \quad \frac{f(a+h) - f(a)}{h} \,, \tag{36.1}$$

for small values of h. (36.1) is called the forward difference formula. There are others that can be used to the same effect:

$$f'(a) \quad \approx \quad \frac{f(a) - f(a-h)}{h}$$

is the backward difference formula. And one can average the two and get

$$f'(a) \quad \approx \quad \frac{1}{2}\left(\frac{f(a+h) - f(a)}{h} + \frac{f(a) - f(a-h)}{h}\right) \quad = \quad \frac{f(a+h) - f(a-h)}{2h} \,, \tag{36.2}$$

which is the central difference formula. We shall use mostly (36.1), because it is the most straightforward, and (36.2), because it is numerically better than the other two, as we shall see.

Let us use as running example the functions

```
def f(x): return 2*x
def g(x): return x**2 - 2
```

Notice that $f(x) = g'(x)$.

(38) Write a function `deriv(f, x, h=.01)` that returns the value $f'(x)$, as approximated through (36.1)[p251], using the provided value of h.

The following assertion must hold:

```
assert all( abs(deriv(g,x)-f(x)) <= 0.02 for x in range(100) )
```

(39) Write a function `fderiv(f, h=.01)`, that returns the derivative function f', as approximated by `deriv` (with `h` potentially overridden).

See Sec. 21.5.5[p68]: "Functions are first-class citizens" and Sec. 21.5.6[p69]: "Anonymous functions: **`lambda`**".

The following assertion must hold:

```
assert all( fderiv(lambda x:x**2)(x) == deriv(lambda x:x**2, x)
            for x in range(100) )
```

(40) Define `G = fderiv(g)`, and let us test how well our approximation performs: we should have $G(x) \approx f(x)$. Write a procedure `test_deriv()` whose invocation yields this:

```
 x    f(x)   G(x)         f(x)-G(x)
-2.0 -4.00 -3.99   -0.01000000000001755
-1.9 -3.80 -3.79   -0.009999999999995346
...
 1.9   3.80   3.81   -0.009999999999999343
 2.0   4.00   4.01   -0.009999999999888765
```

You may want to read Sec. 22.4.10[p92]: "Formatting strings" to format the output properly.

Note: you can't use **float**ing-point numbers in **range**. To understand why, please study Sec. 47[p297]: "Fear the floating-point ranges" (on your own time).

You can see that we have a precision of about 0.01, give or take. You can play with the values of h, going to $0.001, 0.0001, \ldots$ to see how it affects the precision. Does adding more zeroes always increase the precision? Why?

(41) Now go back to the original values of h and modify `deriv` to use the central difference formula (36.2)[p251] instead of (36.1)[p251]. What do you observe? Does adding more zeroes to h still increase the precision, even if just at first?

For your information: There are good reasons for the central difference formula to outperform the others. It can be shown that there are constants K, K', and K'', such that

$$\left|\frac{f(a+h)-f(a)}{h} - f'(a)\right| \leqslant hK \quad \textit{and} \quad \left|\frac{f(a)-f(a-h)}{h} - f'(a)\right| \leqslant hK' ,$$

while

$$\left|\frac{f(a+h)-f(a-h)}{2h} - f'(a)\right| \leqslant h^2K'' .$$

Of course, for h small, that means a much better precision in general for the central difference...

(42) To make this more fun, let's visualise the curves using `matplotlib`. First, you need to install the relevant packages, for your OS and for Python (via `apt`, `pacman`, or `pip3`).

Your best bet is to install the package using your package manager. It should be

```
sudo apt install python3-matplotlib
```

under Debian / Ubuntu and

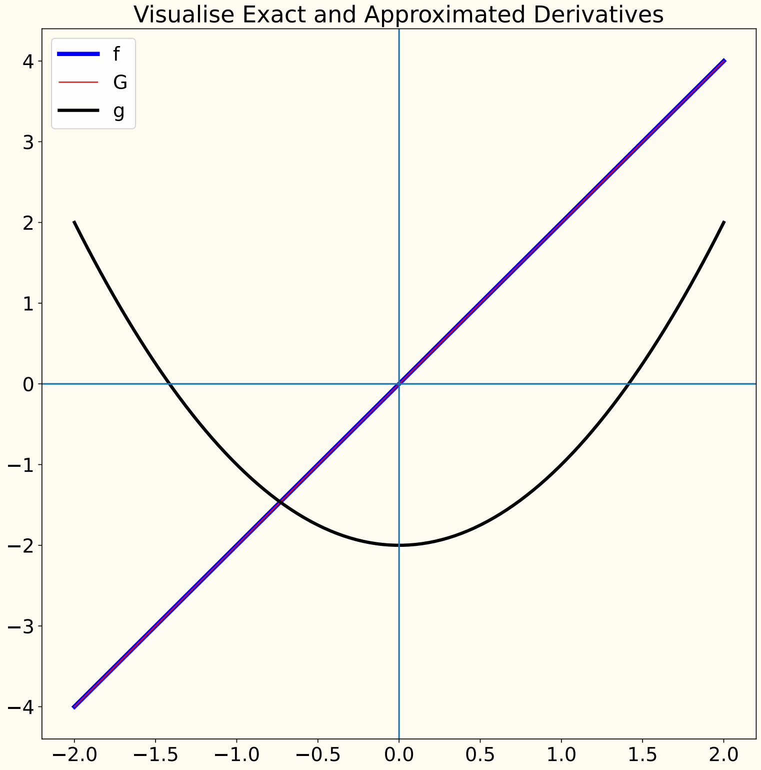


Figure 3: `matplotlib` visualisation

```
sudo pacman -S python-matplotlib
```

under Arch.

If you're not administrator on your machine or that fails for whatever reasons, run the following command:

```
pip3 install matplotlib
```

If all installs well, good. If not, you may need some libraries for your OS, which again, requires admin access. Under a Kubuntu 21.04 I installed the packages below — you may or may not need to do something equivalent. Pay attention to what `pip3` tells you.

```
sudo apt install libtiff5-dev libjpeg8-dev libopenjp2-7-dev zlib1g-dev \
    libfreetype6-dev liblcms2-dev libwebp-dev tcl8.6-dev tk8.6-dev \
    python3-tk libharfbuzz-dev libfribidi-dev libxcb1-dev
```

When all is installed properly we can start to play. Copy the following code after the function definitions:

```
import numpy as np, matplotlib.pyplot as plt
x = np.linspace(-2,2,100) # x varies in [-2,2], 100 uniform samples
npf = f(x) ; npg = g(x) ; npG = G(x) # our functions, with special object x
```

```
plt.figure(figsize=(12,12))
plt.rcParams.update({"font.size": 18 }) # I need glasses, OK?

plt.plot(x,npf,"b"      ,label="f", linewidth=4)
plt.plot(x,npG,"r"      ,label="G", linewidth=1)
plt.plot(x,npg,"black"  ,label="g", linewidth=3)

plt.title("Visualise Exact and Approximated Derivatives")
plt.legend(loc="best")
plt.axvline(0); plt.axhline(0) # draw abscissa and ordinate axes

# plt.savefig("../derivapprox.pdf", transparent=True)
plt.show()
```

When executing that, you should get an interactive version of Figure 3[p253]. The commented `plt.savefig` line is of course what I used to generate the figure — plus a run of `pdfcrop`.

Observe, by zooming on the blue line, that our central approximation, in red inside the thicker blue line, is indistinguishable from the real thing.

You may amuse yourself by trying more complicated functions.

36.3 The Newton–Raphson method

Let us come back to our problem of zeroes of f. How can we use our newfound derivation powers to get even more efficient approximations than with a binary search?

The idea is to start with a guess x_0, then to refine that guess by computing the tangent to f at x_0, then following that tangent to where it intersects with the abscissa, and wherever that is, this is our new and improved guess x_1.

After all, a tangent is simply a linear approximation of f at that point. Instead of finding the zero of f directly, we find that of an approximation.

What happens if $f'(x_0) = 0$? Well, we are stuck, since the tangent is parallel to the abscissa. Unlike the binary search, this method is not *guaranteed* to converge; however, when it does, it does so *very* fast, as we shall see.

Let us take it from the top. We have a differentiable real function f, and an initial guess x_0. The tangent line of f at x_0 is given by the equation

$$t(x) \quad = \quad f'(x_0)(x - x_0) + f(x_0)\,,$$

and crosses the abscissa at x_1, solution of $t(x_1) = 0$:

$$\begin{aligned} 0 &= f'(x_0)(x_1 - x_0) + f(x_0) \\ x_1 &= x_0 - \frac{f(x_0)}{f'(x_0)} \end{aligned}$$

Following the same reasoning, at each step we obtain the next guess by computing

$$x_{n+1} \quad = \quad x_n - \frac{f(x_n)}{f'(x_n)} \ .$$

When do we stop that process? Unlike before, we lack a well-defined search interval, so we cannot know how close we are to the solution. We do know, however, how close $f(x_n)$ gets to zero, and we shall use that as a criterion.

(43) Write a function `newton(f,x,eps=1e-15)` that computes, if possible, a zero of a differentiable function $f : \mathbb{R} \to \mathbb{R}$ using the Newton-Raphson method. The function shall trigger an assertion if it runs into a null derivative, and a guess x_n shall be considered good enough to return if $|f(x_n)| \leqslant \varepsilon$. Optionally, for debugging purposes, you can have it print each of its guesses.

Tip: this function is doable in three lines — at least in its recursive version.

With it, let us compute $\sqrt{2}$ again, with initial guess 1:

```
>>> res = newton(g,1) # here with optional printing of guesses
-> 1
-> 1.4999999999999996
-> 1.4166666666666667
-> 1.4142156862745099
-> 1.4142135623746899
-> 1.4142135623730951
>>> sqrt(2)
1.4142135623730951
>>> sqrt(2)-res
0.0
```

Note how fast it is, compared to the binary search!

Dichotomy is a general approach; specific problems may allow for more even more efficient specific approaches

37 Comprehension expressions

Read Sec. 24.5[p153]: "Comprehension expressions" with great attention. Do not neglect the examples in Sec. 24.5.3[p158]: "Common comprehension patterns".

For nearly each question, you will use a comprehension expression. For instance, if I ask for the set of all even numbers strictly less than n, then I expect to see

```
{ i for i in range(n) if i%2==0 }
```

If I ask you for a function returning the set of all even numbers strictly less than n, I expect

```
def evens(n):
  return { i for i in range(n) if i%2==0 }
```

Where a comprehension expression is possible, an answer based on usual constructions by iteration will not be suitable for the purpose of this exercise.

comprehension expressions are compact, legible, and easy to write.
you love comprehension expressions

37.1 Warm-up

(44) Write a function `cart_prod(A,B)` returning the Cartesian product of sets `A` and `B`. For instance:

```
>>> cart_prod({'a', 'b'}, {1,2,3})
{('a', 1), ('a', 2), ('a', 3), ('b', 1), ('b', 2), ('b', 3)}
```

Note that, since the result is a set, the order in which its elements are displayed is unpredictable; see Sec. 24.3[p141]: "Sets: class `set`". If you want to display the elements in a legible order, you can use **`sorted(.)`**

The following assertions should hold:

```
assert cart_prod(range(3), []) == set()
assert type(cart_prod([1],[2])) is set
assert cart_prod(range(2), range(10, 12)) == {(0,10), (0,11), (1,10), (1,11)}
```

(45) . . . now compute

```
>>> cart_prod("ab", {1,2,3})
```

instead of

```
>>> cart_prod({'a', 'b'}, {1,2,3})
```

What happens, and why? Is it a bad thing?

*If in doubt, read Sec. 23.4[p114]: "**`for .. in .. range`** loop" again, especially the part where the keywords* iterable *and* collection *appear.*

what works for an iterable type usually should work for another

(46) Write a function `squares(n)` returning the list of all square numbers (i.e. integers that are the square of another integer) in $[\![0, n]\!]$.

We deal with integers; make sure to avoid floating point computations; for instance you must not write `sqrt(i)` or `i**0.5` either. No square root or floating point number of any kind.

The following assertion must hold:

```
assert squares(100) == [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

conditionals in comprehension expressions
comprehension expressions and boolean operators

(47) Write a predicate `isprime(n)` (ℕ → `bool`) testing whether a natural integer is prime, that is to say, whether it is strictly greater than 1 and divisible only by 1 and n.

As usual in this exercise, the body must be written in one line of the form `return <expr>`.

Read Sec. 24.5.3.4[p159]: "Reductions", especially the part about `any` and `all`.

The following assertion must hold:

```
assert [ i for i in range(30) if isprime(i) ] \
      == [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

combine comprehensions with boolean operators

37.2 Palindromes and other one-liners

All of the following functions must be written in one line: that is to say, their body must be of the form `return <expr>`.

You may write a first version of them using normal loops as a draft if it helps you, but the final product must be of this form. Of course the solution will often be a comprehension expression, but sometimes it can be another simple expression.

sequence index manipulation
`any`, `all`, and `sum` reductions

(48) Write a predicate `palindrome(s)` testing whether the sequence `s` is a palindrome.

Read Sec. 22.4.6[p86]: "Slicing and dicing, concatenation, repetition", especially about negative indexes. Read Sec. 24.5.3.4[p159]: "Reductions", especially about `any` and `all`.

A *palindrome* is a sequence that can be read either left-to-right or right-to-left: `ABCBA` is an example.

To test that, you will test that all elements are equal to their mirror: the first to the last, the second to the penultimate, and so on.

The following assertions must hold:

```
assert palindrome('abba')
assert palindrome('abcba')
assert palindrome('')
assert palindrome('a')
assert not palindrome('ab')
```

(49) Write a function `inverse(s)` returning the list of the elements of the sequence `s`, in reverse order. For the purpose of this exercise, you will use a comprehension expression, not a `[::-1]` slice.

The following assertions must hold:

```
assert inverse('abc') == ['c', 'b', 'a']
assert inverse('') == []
```

(50) Write a predicate `palinv(s)` equivalent to `palindrome(s)`, but using `inverse`.

The following assertions must hold:

```
assert palinv('abba')
assert palinv('abcba')
assert palinv('')
assert palinv('a')
assert not palinv('ab')
```

(51) `for elem in collection` versus `for i in range`

Write a function `rmfrom(s,bad)` returning the list of the elements of the collection `s` that do not appear in the collection `bad`. The order of elements must be preserved if `s` is a sequence.

The following assertion must hold:

```
assert rmfrom('esope reste ici et se repose', 'aeiouy ') == \
       ['s', 'p', 'r', 's', 't', 'c', 't', 's', 'r', 'p', 's']
```

(52) laziness is a virtue: reuse previous functions

Write a function `rmspaces(s)` returning a list of the elements of the sequence `s`, in the original order, from which spaces have been removed.

The following assertion must hold:

```
assert rmspaces('esope reste ici et se repose') == \
       [ 'e', 's', 'o', 'p', 'e', 'r', 'e', 's', 't',
         'e', 'i', 'c', 'i', 'e', 't', 's', 'e', 'r',
         'e', 'p', 'o', 's', 'e']
```

(53) are you virtuous yet?

Write a predicate `palindrome_sentence(s)` testing whether the sentence described by the sequence `s` is palindrome. A sentence is palindrome is the sequence of its letters is palindrome — whitespace is abstracted away.

The following assertions must hold:

```
assert palindrome_sentence('esope reste ici et se repose')
assert not palindrome_sentence('esope reste ici et se reposes')
```

(54) **a lot of maths can be translated in Python almost directly**

Write a function `fsum(f,i,j)` such that

$$\texttt{fsum(f,i,j)} \; = \; \sum_{k=i}^{j} f(k) \, .$$

The following assertions must hold:

```
assert fsum (lambda i:i, 0,10) == 55
assert fsum (lambda i:i**2, 0,10) == 385
```

38 Our generators have character

(55) *You will need Sec. 22.4.7*$_{[p89]}$*: "Python strings use Unicode".*

Write a function `crange(a,b)` that returns a generator for all characters from `a` to `b`, in Unicode point order. This can (and must) be done in one line, using a generator expression.

It must satisfy the following assertion:

```
assert "".join(crange('A','Z')) == 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
assert next(crange("a","b")) == "a"
```

generator expressions
variadic functions

Read Sec. 24.5.1$_{[p154]}$*: "Comprehensions for every type; first contact with generators".*

(56) *This question requires the* **`yield`** *or* **`yield from`** *keywords from Sec. 29*$_{[p208]}$*: "Iterables, iterators, and generators", as well as Sec. 25.1*$_{[p171]}$*: "Variadic function definition". It is advised to use Sec. 24.6*$_{[p164]}$*: "Packing and unpacking" as well.*

Write a variadic function `charrange`($a_1, b_1, \ldots, a_n, b_n$) returning a generator for all characters of the successive ranges a_k, b_k, as defined in the previous question. It need not be written in one line.

It must satisfy the following assertions:

```
assert "".join(charrange('A','Z','a','z','0','9')) == \
    'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789'
```

```
assert next(charrange("a","b")) == "a"
assert "".join(charrange()) == ''
```

yield and yield from, and the difference between them

39 And then there were Nones...

side effects; order of evaluation

Read Sec. 22.5[p98]: "Nihilism: NoneType: expression versus statement".

Mentally execute the script below, and write down the output which you expect Python to produce.

```
2+2
print(2+2)
print(print(2+2),print(2+2))
l= [ 1+i for i in range(3) ]
pl = [ print(1+i) for i in range(3) ]
print(l,pl)
```

Execute the code. Compare what was actually produced to what you thought would be. If they do not match exactly, take the time to understand why.

40 Sets, dictionaries and slices training

If you have not already done so, read Sec. 22.4.6[p86]: "Slicing and dicing, concatenation, repetition", Sec. 24.3[p141]: "Sets: class set", and Sec. 24.4[p144]: "Dictionaries: class dict".

sets are hash tables ⟹ no mutable values

sets are unordered ⟹ not indexable

sequence slicing syntax (on indexable stuff only!)

False == 0, so problems in sets, dicts

comprehensions are loops behind the scenes, so side effects work as usual

Mentally execute the following blocs of code, and write down on a piece of paper what you think Python will display.

In cases where Python's output is not entirely predictable, be sure to note that on your answer, and explain the cause and extent of this unpredictability.

Then execute the code and confront your answer to reality.

(57) `print(set(’totto’))`

(58) `print({’totto’})`

(59) `print({{’toto’}, {’tata’}})`

(60) `print(’abcde’[-1])`

(61) `print({’abcde’}[0][1])`

(62) `print(’abcdefg’[2:5])`

(63) `print((list(’abcdefg’)*3)[2:5])`

(64) `print((list(’abcdefg’)*3)[19:22])`

(65) `print(’abcdefg’[-5:-2])`

(66) `print( list(range(12))[13:5:-2] )`

(67) `print({0:1, None:2, False:5})`

(68)

```
s = { print(i) for i in range(1,3) }
ss = { (i,print(i)) for i in range(1,3) }
sss = { (i,i,print(i)) for i in range(1,3) }
print(s,ss,sss,sep=’\n’)
```

41 What the *what*!?

Some days, in my profession, you come across code that you just *have* to share with the world. As therapy.

In this exercise, I share with you very *special* code that I have seen written sincerely, candidly, by my own students, in answer to questions in this very document, whether in class or during an exam[ai].

The trick is, I don’t tell you whether it works, or what question it is supposed to answer. It is up to you to figure out what it does — or purports to do.

Then you must correct it where necessary, and simplify it.

This section is small for now, as I only recently had this idea to systematically weaponise students’. . . *creativity* into exercises, but, with the upcoming exams, I have every confidence that it will grow fast :-)

(69) Courtesy of a student from 3A STI Apprentissage, 2020-2021, who wishes to remain anonymous:

[ai] It’s less funny during an exam; no points are awarded for being facepalm-worthy.

```
def spicy_function(X, Y):
 E = set()
 { E.add( (x,y) ) for x in X for y in Y }
 return E
```

This works, and not quite accidentally either, but it's interesting to understand *why*, and to understand what the *value* of

```
{ E.add( (x,y) ) for x in X for y in Y }
```

is, and what happens to it.

side effects ≠ the value denoted by the comprehension

(70) 3A STI, 2019-2020. This one was written all in one line. Given the limitations of the PDF / paper format, I had to wrap it.

This is sad, as some of the poetry is lost.

```
True if len(p) == 0 else not False in {True if p[j] ==
          p[len(p)-j-1] else False for j in range (len(p)//2)}
```

(C==True)==True is not better in an expression-if
"there *has* to be a better way to write this"

42 Encapsulating the sparse matrices

encapsulation: hide implementation details behind an interface
tailor your implementation to your needs
test your assumptions regarding performance

A *sparsely populated matrix* (or *sparse matrix* for short) is a matrix in which the vast majority of the values are 0, `null`, `None`, or whatever other value signifies "nothing to see here!" in the context at hand.

For instance, when implementing a game operating on a map of size 100×100, if you have only a couple hundreds of characters on the map, the matrix representing the situation would qualify as sparse. Sparseness, and its opposite, *density*, have strong implications regarding the performance of various implementations of matrices.

In this exercise we shall write two different implementations of matrices, hiding the dirty details of the implementation from the user behind an interface. This extremely common technique is referred to as *encapsulation*[aj] in Object-Oriented Programming (OOP). You will

[aj] https://en.wikipedia.org/wiki/Encapsulation_(computer_programming).

need to read and understand the basics of Sec. 27[p183]: "Object Oriented Programming in Python".

Then, we shall compare the performance of those two interchangeable implementations.

To keep things simple, we are not going to implement matrices in all generality, but only square matrices M_N of size $N \times N$, $N \in \mathbb{N}$, of the form

$$M_N \;=\; \begin{bmatrix} 0 & 0 & \cdots & 0 \\ 0 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & N-1 \end{bmatrix},$$

which is to say, diagonal matrices defined as

$$[M_N]_{ij} \;=\; \begin{cases} i & \text{if } i = j \\ 0 & \text{otherwise} \end{cases} .$$

Furthermore, we shall only implement one operation on those matrices: the computation of the sum $\sum M_N$ of their elements:

$$\sum M_N \;=\; \sum_{i,j} [M_N]_{ij} \,.$$

Though we shall not implement that, we shall allow for the possibility of the matrices being modified by the user; thus our representations and the implementation of our methods shall remain fully general, and will not take advantage of the extremely specific form of M_N. It just happens that we initialise our matrices to M_N, that's all, and every bit of code you write should work equally well if we decided to initialise to random values instead.

While this still appears very restrictive, to go back to the "game" application, this is sufficient to *simulate* populating sparse matrices and executing an operation that needs to take everything on the grid into account. With this, we shall have more than enough to draw pretty clear conclusions with regards to performance; you'll see.

Our implementations of matrices shall follow the following interface: the class constructor will take as single argument the dimension N, of default value 100, and the matrix object shall offer:

- the attribute `N`, storing the dimension,
- the attribute `m`, storing the internal representation of the matrix, which the user is not really supposed to interact with directly,
- the method `sum`, taking no argument, returning the sum of the elements of the matrix,
- and support a string representation (via `repr`, cf. Sec. 27.7[p191]: "String representations `str` and `repr`"), for which I'll provide most of the code.

(71) Let us implement a class `matrix` providing a naïve "list of list" implementation of M_N.

You might want to refresh your reading of Sec. 24.2.2.2[p134]: "Case study: nested lists/ matrices" on the initialisation of matrices.

Complete the following code to initialise the matrix to M_N:

```
class matrix():
    def __init__(s, N=100):
        s.N = N
        s.m = [ [... for ...] for ... ]

    def __repr__(s): return f"matrix({repr(s.m)})"
```

Yes, as the code indicates, populating the internal representation should be done in one (logical) line. Remember that Sec. 23.2[p112]: "Conditional expression: `.. if .. else ..` ternary operator" exists.

You should obtain

```
>>> matrix(3)
matrix([[0, 0, 0], [0, 1, 0], [0, 0, 2]])
```

and the following assertion should hold:

```
assert matrix(3).m == [[0, 0, 0], [0, 1, 0], [0, 0, 2]]
```

Aside on `repr`: Note that our `repr` does not exactly follow Python convention, in that it does not return the Python code that would produce the object:

```
>>> matrix([[0, 0, 0], [0, 1, 0], [0, 0, 2]])
TypeError: 'list' object cannot be interpreted as an integer
```

Morally, we should have returned the string `"matrix(3)"`, but we are anticipating a more general version of `matrix`, where we can initialise a matrix with whatever we want, and modify it.

(72) Now, implement the `matrix.sum` method, by completing the following code:

```
    def sum(s):
        return sum(...)
```

There again, the implementation should be in one line, following the structure of the code I provide, and the following assertion should hold:

```
assert all( matrix(N).sum() == N*(N-1)//2 for N in range(10) )
```

We're all done with our first implementation.

(73) Intermission:

Pop quiz on something completely different: in the assertion above, how do I know that `N*(N-1)//2` will be evaluated as `(N*(N-1)) // 2`, which works because maths guarantee that `N*(N-1)` is even[ak] for all `N`, and not `N * ((N-1)//2)` which introduces a rounding error if `N-1` is odd?

```
>>> [ (a,b) for N in range(10) if (a:= N*(N-1)//2) != (b:= N*((N-1)//2)) ]
[(1, 0), (6, 4), (15, 12), (28, 24)]
```

Tip: Figure 1[p111].

always think of precedence and associativity
when in doubt, don't write useless parentheses: check, and learn

(74) Now let's go back to our matrices, and provide another implementation, called `smatrix`, with a completely different internal representation. Spoiler alert, it is called `smatrix` because it is optimised for sparse matrices.

Instead of a natural "list of lists" representation, we'll use a "mapping" representation, whereby we store a mapping from coordinates of non-zero cells to their value. Any cell not appearing in the mapping is assumed to contain 0.

For instance,

$$M_3 \quad = \quad \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 2 \end{bmatrix}$$

is represented as

$$\{\,(1,1) \mapsto 1,\ (2,2) \mapsto 2\,\}\ .$$

You will implement this mapping using a dictionary. More specifically, you will use a `defaultdict`, so that any coordinate not in the dictionary is associated with 0. See Sec. 24.4.1.2[p151]: "`defaultdict`, from `collections`".

You will complete the following code:

```
class smatrix():
    def __init__(s, N=100):
        s.N = N
        s.m = defaultdict(..., { ... for ...  })

    def __repr__(s): return f"smatrix({repr(s.m)})"
```

You should get something like

[ak] The product of an even integer with any other integer is even. Of two consecutive numbers, one is even. Therefore. . .

```
>>> smatrix(3)
smatrix(defaultdict(<function smatrix.__init__.<locals>.
    <lambda> at 0x7f54f0748220>, {(1, 1): 1, (2, 2): 2}))
```

which is not pretty to look at — we'll fix that later — and the following assertions must hold:

```
assert type(smatrix(3).m) is defaultdict
assert smatrix(3).m == { (1, 1): 1, (2, 2): 2 }
assert smatrix(3).m[(999, 999)] == 0
```

(75) Now, implement the `smatrix.sum` method, by completing the following code:

```
    def sum(s):
        return sum(...)
```

As usual, in one line.

For this first implementation, I add the constraint that it must be a pretty naïve implementation of $\sum_{i,j} [M_N]_{ij}$, following the structure of `matrix.sum`, explicitly asking the internal representation for the values of all the cells in the matrix.

If you see a *much* better way of doing this, don't worry, we'll get there in a couple of questions. If you don't, and wonder what I'm talking about, don't worry, your first instinct will probably be the naïve implementation I'm asking for, so all is well :-)

```
assert all( smatrix(N).sum() == N*(N-1)//2 for N in range(10) )
```

(76) Write a method `smatrix.full_matrix()` that returns the "list of lists" representation of the matrix.

You should have

```
>>> smatrix(3).full_matrix()
[[0, 0, 0], [0, 1, 0], [0, 0, 2]]
```

and the following assertion must hold:

```
assert all( smatrix(N).full_matrix() == matrix(N).m for N in range(10) )
```

translate from one representation to another

(77) Now let us fix our `repr` so it doesn't look too ugly. Change what needs to be changed so that we see

```
>>> smatrix(3)
smatrix([[0, 0, 0], [0, 1, 0], [0, 0, 2]])
```

instead of

```
>>> smatrix(3)
smatrix(defaultdict(<function smatrix.__init__.<locals>.
    <lambda> at 0x7f54f0748220>, {(1, 1): 1, (2, 2): 2}))
```

The following assertion should hold:

```
assert all( repr(smatrix(N)) == "s"+repr(matrix(N)) for N in range(10) )
```

hide dirty internal structure from the user

(78) It's time for some performance testing.

Here is some code for performance testing:

```
def test():
    from timeit import timeit
    for desc,f in [
    ("matrix init",              lambda: matrix()),
    ("matrix init+sum",          lambda: matrix().sum()),
    ("sparse matrix init",       lambda: smatrix()),
    ("sparse matrix init+sum", lambda: smatrix().sum()),
        ]:
        print(f"{desc:<25}{timeit(f,number=1000):.3f}")
```

Execute it, and you should see numbers telling a story similar to this:

```
matrix init                0.183
matrix init+sum            0.446
sparse matrix init         0.005
sparse matrix init+sum     1.208
```

At the risk of stating the obvious, these are performance numbers and will vary from machine to machine; they should lead to the same *conclusion*, however.

And what is that conclusion? We see that `smatrix` initialises massively faster than `matrix`. This is not surprising, as the former allocated all cells into memory, even if zero, and almost all of them are zero, whereas the latter only allocates memory for non-zero cells. All good.

However, we are disappointed to see that `sum` performance is actually *worse* for `smatrix`. How come?

good structure + bad algo = bad performance

(79) If you had a better idea for implementing `smatrix.sum`, now is your time to shine. If not, now is your time to think one up.

Tip: just sum over the non-zero values of the internal representation. See Sec. 24.4[p144]: "Dictionaries: class dict" to find a useful method of dict, beginning by v, that you might want to use. . .

With the new implementation, the performance profile should match

```
matrix init               0.185
matrix init+sum           0.467
sparse matrix init        0.005
sparse matrix init+sum    0.006
```

In other words, smatrix is now massively more efficient than matrix for all supported operations.

I hope you appreciated in passing how convenient it is that we can just completely change the implementation of a method at the drop of a hat, and still trust in the correctness of the code because we have unit-test assertions in place. Isn't it nice? (You *do* have assertions in place, right? Right?)

good structure + good algo = good performance

(80) What performance profile would you expect from those two implementations on *dense* matrices, that is, matrices in which most values are non-zero?

error 404: universal perfect structure not found

43 Power to the sets!

We recall the notion of **powerset** of a set S, denoted by $\wp(S)$, 2^S, $\mathbb{P}(S)$, $P(S)$, etcetera.

$$s \in \wp(S) \iff s \subseteq S \qquad \text{or} \qquad \wp(S) = \{\, s \mid s \subseteq S \,\}\,.$$

In other words, the powerset of S is the set of all (non-strict) subsets of S. For instance:

$$\wp(\{0,1\}) \;=\; \big\{\varnothing,\ \{0\},\ \{1\},\ \{0,1\}\big\}$$

Further recall the property $|\wp(S)| = 2^{|S|}$, which is one of the reasons for the use of the notation 2^S: we have $|2^S| = 2^{|S|}$; the other reason is of course the bijection between the powerset and the set of functions $\{0,1\}^S$ — both reasons boil down to the same thing, in the end.

It is interesting to see a proof of that property:

***Theorem 3** (Cardinality of Powerset). Let S be a set; then $|\wp(S)| = 2^{|S|}$.*

Proof sketch. By induction on $|S|$.

If $|S| = 0$, then $S = \varnothing$, and $|\wp(\varnothing)| = |\{\varnothing\}| = 1 = 2^0$.

Let $S = \{e\} \cup T$, with $|T| = n$, and assume $|\wp(T)| = 2^n$. Let $s \subseteq S$; then either $e \in s$ or $e \notin s$, and in either case $s - \{e\} \in \wp(T)$. So we have in total 2^n powersets containing e, and 2^n not containing e, so $|S| = 2^{n+1} = 2^{|S|}$. □

Note that the proof suggests a recursive definition of the powerset...

(81) Define a function `powerlist(l)` returning the list of the positional sublists of the list (or any other iterable) `l`, in no particular order.

By "positional sublist", I mean a list containing some elements of `l`, based on their positions, not on their values. Thus, if values are repeated in the list `l`, they are still treated as distinct.

The following assertions must hold:

```
assert powerlist([]) == [[]]
assert sorted(powerlist([1,2,3]), key=lambda x:(len(x), x)) == \
    [[], [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3]]
assert sorted(powerlist([1,1,1]), key=len) == \
    [[], [1], [1], [1], [1, 1], [1, 1], [1, 1], [1, 1, 1]]
assert all( len(powerlist(range(n))) == 2**n for n in range(5) )
```

You will implement this recursively and take care to avoid repeating redundant recursive calls.

This can easily be done in three lines (excluding function declaration).

The cardinality proof above hints at the recursion you need to apply.

You may — no, you most definitely *will* — want to apply some pattern-matching or packing/unpacking, here: see Sec. 23.6[p119]: "Pattern matching: **`match..case`**" and Sec. 24.6[p164]: "Packing and unpacking". Hint: what does `e,*l = l` or `*l,e = l` do?

using packing/unpacking for recursion on lists

(82) Now write `powerlist2`, another version of `powerlist`, implemented non-recursively.

There again, this can easily be done in three lines (excluding function declaration).

It must satisfy the same assertions as `powerlist`; you can have the assertions apply to both functions like so:

```
for powerlist in (powerlist, powerlist2):
    assert powerlist([]) == [[]]
    assert sorted(powerlist([1,2,3]), key=lambda x:(len(x), x)) == \
        [[], [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3]]
    assert sorted(powerlist([1,1,1]), key=len) == \
        [[], [1], [1], [1], [1, 1], [1, 1], [1, 1], [1, 1, 1]]
    assert all( len(powerlist(range(n))) == 2**n for n in range(5) )
```

a recursive definition can also suggest an easy loop implementation

(83) Can you implement a function `powerset(s)` in Python that naively matches $\wp(\cdot)$ and returns a `set` of `set`s? Why?

Tip: Sec. 24.3[p141]: “Sets: class `set`”, Sec. 24.3.1[p144]: “Frozen sets: class **`frozenset`**”.

(84) Implement a function `powerset(s)` corresponding to $\wp(\cdot)$, and returning a `set` of **`frozenset`**s. The parameter s may be of any iterable type, and must not be altered by the call.

Reminder: packing/unpacking works on sets, and `set.pop` exists as well.

The following assertion must hold:

```
assert all( powerset(r:=range(n)) == { frozenset(s) for s in powerlist(r) }
            for n in range(5) )
```

sets of sets are common in maths; need thought in Python

(85) Write a generator function `powergen(s)`, where s is again any iterable, than generates all sub`set`s of s.

The following assertions must hold:

```
assert type(powergen([])) is type(_ for _ in [])
assert all( type(s) is set for s in powergen(range(5)) )
assert all( set(map(frozenset, powergen(r:=range(n))))
            == { frozenset(s) for s in powerlist(r) }
            for n in range(5) )
```

You can use **`yield from`** if you want, but in that instance it is simpler not to.

recursion in generator functions

44 Let’s get primitive!

In Sec. 36.2[p250]: “This is all very derivative. . . ”, we computed numerical approximations of derivatives. Let us now do the same thing for integral calculus and primitives.

apply comprehension expressions to
solve a seemingly nontrivial problem in a few lines

Our goal is to compute a numerical approximation of a primitive (or antiderivative) of any given continuous function of type $\mathbb{R} \to \mathbb{R}$. That is to say, given a function f, we want to obtain a function F such that $F' \approx f$.

Because constants disappear during derivation, every function f has infinitely many primitives, differing only up to an additive constant. Each can be written F or $\int f(x)\,dx$.

Primitives and definite integrals, which represent the area beneath the curve of the function, are related by the following equation:

$$\int_a^b f(x)\ dx \quad = \quad F(b) - F(a)\ .$$

Thus, if we can compute a definite integral, we can obtain a suitable primitive via

$$F(x) \quad = \quad \int f(x)\,dx \quad = \quad \int_0^x f(t)\ dt\ .$$

Our first objective will therefore be to implement definite integrals. The simplest way to do that is to use Riemann integrals.

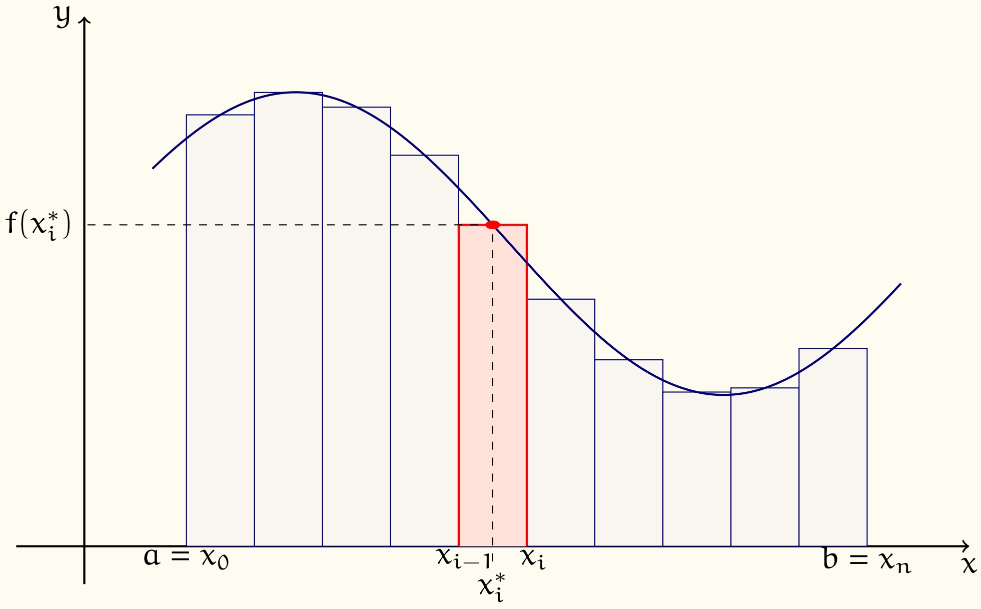


Figure 4: General idea of Riemann integration

The general idea is to approximate the area under the curve by partitioning the abscissa into many smaller intervals, and approximate the area under the curve on each smaller interval by a rectangle. Then you can sum all rectangles. The smaller each small interval, the better the approximation.

There are different possible choices regarding the selection of the smaller intervals, and of the value used for the height of each rectangle, but they all give the same result when the intervals get infinitesimally small. This is illustrated visually in Figure 4[p271].

Definition 4 (Riemann integral)**.** Let $f : [a, b] \to \mathbb{R}$ and

$$P = \left[(x_0, x_1), (x_1, x_2), \dots, (x_{n-1}, x_n)\right] ,$$

a partition of $[a, b]$. That is to say, we have

$$a = x_0 < x_1 < x_2 < \cdots < x_n = b$$

The **Riemann sum** of f on P is

$$\sum_{i=1}^{n} f(x_i^*) \, \Delta x_i \, ,$$

where $\Delta x_i = x_i - x_{i-1}$ and $x_i^* \in [x_{i-1}, x_i]$.

The **Riemann integral** is the limit of the Riemann sum as the maximal size of partitions $\|\Delta x\| = \max_i \Delta x_i$ goes to 0:

$$\int_a^b f(x) \, dx \quad = \quad \lim_{\|\Delta x\| \to 0} \sum_{i=1}^{n} f(x_i^*) \, \Delta x_i .$$

The choice of x_i^* can produce different sums, but does not matter at the limit, for the integral.

For our purposes, we shall take uniform partitions, that is to say all Δx_i are equal, and select x_i^* as the middle of its interval:

$$x_i^* \quad = \quad \frac{x_i + x_{i-1}}{2}, \qquad \forall i \, ,$$

this is called a **middle Riemann sum**.

(86) Write a function `partition(a,b,n)` that, given two floating point values a and b, and $n \in \mathbb{N}$, returns the partition of the interval $[a, b]$ into n equal parts.

Specifically, you must return a list of couples `[ (a1, b1), ..., (an, bn) ]` such that $a_1 = a$, $b_n = b$, and all intervals $[a_i, b_i]$ are of the same size.

It should be written in one or two lines using a comprehension expression, and it must satisfy the following assertions:

```
assert partition(-1, 1, 1) == [(-1.0, 1.0)]
assert partition(0,30,3)   == [(0.0, 10.0), (10.0, 20.0), (20.0, 30.0)]
assert partition(30,0,3)   == [(30.0, 20.0), (20.0, 10.0), (10.0, 0.0)]
assert partition(-1, 1, 4) == [(-1.0, -0.5), (-0.5, 0.0),
                               (0.0, 0.5), (0.5, 1.0)]
```

(87)

Applying `for a,b in l`: multiple assignment in iteration

Write a function `riemann(f, a, b, n=100)` that returns an approximation of

$$\int_a^b f(x)\,dx\,,$$

the definite integral of f on the interval $[a, b]$, performed by middle Riemann sum, with a partition of $[a, b]$ in n equal parts.

This function can and should be written in one line, of the form `return sum ( ... )`, and should satisfy the following assertions:

```
assert abs(riemann(lambda x:x, 0,1) - 0.5) < 1e-9
assert abs(riemann(sqrt, 0,1) - 2/3) < 1e-4
```

(88) Write a function `primitive(f, x, n=100)` returning the value

$$F(x) \;=\; \int_0^x f(t)\,dt\,,$$

as approximated by a middle Riemann sum on a uniform partition in n equal parts.

The following assertion must hold:

```
assert all( abs( primitive(lambda x:x, x) - (x*x / 2)) < 1e-9
                 for x in [-32, 0, 1, 2, 8, 64] )
```

(89) Write a function `fprimitive(f, n=100)` returning F, the approximated primitive function of f, as per the previous question.

The following assertion must hold:

```
assert all( fprimitive(sqrt)(x) == primitive(sqrt, x) for x in range(100) )
```

(90) Let us visualise this. Using the same example functions f and g as in Sec. 36.2[p250]: "This is all very derivative...", graph the functions g, $g + 2$, and $F = \int f(x)\,dx$, as given by `F = fprimitive(f)`.

You should obtain something similar to the nearby figure. You should, as for derivatives, see that the error due to the numerical approximation is imperceptible.

We shall come back to derivation in Sec. 45[p274]: "A smidgen of Computer Algebra", using symbolic methods instead of numerical ones. Note that computing *primitives* symbolically is a *vastly* more difficult problem than computing *derivatives* symbolically. Many primitives of rather simple functions do not even have a closed form! For instance,

$$\int e^{-x^2}\,dx, \qquad \int x^x\,dx, \qquad \int \frac{1}{\ln x}\,dx, \qquad \int \sin x^2\,dx, \qquad \int \frac{\sin x}{x}\,dx, \ldots$$

Thus, we shall *not* come back to integration from a symbolic standpoint.

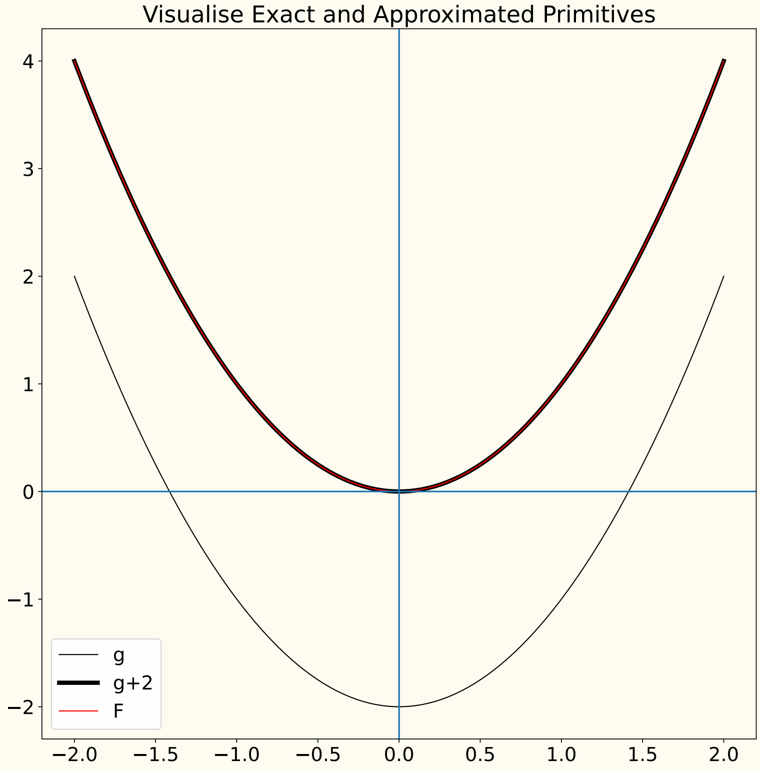


Figure 5: `matplotlib` visualisation of primitive

45 A smidgen of Computer Algebra

*This exercise requires a good understanding of Sec. 23.6[p119]: "Pattern matching: **`match..case`**", Sec. 27.5[p188]: "**`match`**ing attributes", and Sec. 28[p202]: "Advanced structural pattern **`match`**ing". For the second section, Sec. 27[p183]: "Object Oriented Programming in Python" is also required.*

apply advanced pattern matching to a rather complex problem

In Sec. 36.2[p250]: "This is all very derivative. . . ", we used numerical methods to approximate the values of derivatives at any point, which is all well and good but. . . didn't you wish we could get exact answers? For instance

$$\frac{d}{dx}\left(x^2 - 2\right) \quad = \quad 2x\,,$$

or

$$\frac{d}{dx}\left(1 + 2\ln(x^2 - 1)\right) = \frac{4x}{x^2 - 1}$$

or

$$\frac{d}{dx}\left(\ln x(3x^2 + 1)\right) \quad = \quad 3x + \frac{1}{x} + 6x\ln x\,.$$

How do we get *that* kind of answers out of the computer? Well, one way is to fork out the money for Computer Algebra Systems (CAS) such as Maple, Mathematica, or MATLAB, or install free and open-source alternatives such as SageMath[al], Axiom, or Maxima, or even just look up the solution on `https://www.wolframalpha.com/` . . . but where is the fun in *not* reinventing the wheel?

Instead, we shall implement our own rudimentary CAS. We will need to manipulate mathematical expressions symbolically to compute derivative functions.

First step, what is the formal grammar of those expressions? We shall limit ourselves to

$$e \;::=\; x \;\mid\; e+e \;\mid\; e-e \;\mid\; e\div e \;\mid\; e\times e \;\mid\; f(e) \;\mid\; e^{e} \;\mid\; -e \;\mid\; v\,,$$

where $v \in \mathbb{R}$ is a constant value, x is a variable name and f a function name. In practice we shall support only ln.

45.1 A perfect `match`

(91) implementing an inductive type definition in Python

Define a class system in the style of Sec. 28[p202]: "Advanced structural pattern `match`ing" for the type of mathematical expressions.

Note that every construct is binary, even function calls and exponentiation that are not typically thought of as "operators". Thus I suggest defining a larger type `BinExpr` to handle all of theses, and only defining $+, -, \times, \div$ as proper `BinOp`. This will enable the factorisation of some rules. Thus I propose that you copy this

```
@dataclass
class BinExpr:
    a: object
    b: object

class BinOp (BinExpr): pass

@dataclass
class UnOp:
    a: object
```

and define each operator by inheritance of those types. Variable and function names shall be strings.

For instance, you should be able to write

[al] If we had a *serious* Python project involving symbolic computation, the sane thing would be to use Sage and SymPy. SymPy is a Python library for symbolic computation, and Sage, which includes SymPy, is partly implemented in Python, and interoperates with it.

```
>>> x = "x" # our main variable name
>>> f1 = Plus(1, Mul(2,Call("ln",Minus(Pow(x,2),1))))
>>> f1
Plus(a=1, b=Mul(a=2, b=Call(a='ln', b=Minus(a=Pow(a='x', b=2), b=1))))

>>> f2 = Mul(Call("ln",x), Plus(Mul(3,Pow(x,2)), 1))
>>> f2
Mul(a=Call(a='ln', b='x'), b=Plus(a=Mul(a=3, b=Pow(a='x', b=2)), b=1))
```

Make sure to define f1, f2, and x in your source file, we shall use them as running examples.

(92) implementing recursive functions on inductive types

Ok, we have formulæ, but they are ugly to look at. Write a function estr such that

TODO ADD ASSERTIONS THROUGHOUT EXERCISE

```
>>> estr(f1)
'(1 + (2 * (ln(x^2 - 1))))'

>>> estr(f2)
'((ln x) * ((3 * x^2) + 1))'
```

Recall the trick from Sec. 28[p202]: "Advanced structural pattern **matching**" to use an attribute symb to associate a symbol to each operator.

There are still a lot of parentheses, but analysing operator precedence to get rid of some of them would be a more difficult exercise. This is good enough for our purposes.

(93) embedding semantics in inductive type definitions catching specific exceptions

We have symbolic expressions, which is nice. But at some point we want numerical results as well, if only to be able to graph them. Write a function[am] **eval**(e,var,val) that produces the numerical value of the evaluation of the expression e when the variable var is affected the value val.

For instance, you should obtain:

```
>>> eval(f2, x, 0)
inf

>>> eval(f2, x, 0.1)
-2.371662645783867

>>> eval(f2, x, 1)
```

[am] Note that a function named **eval** is already defined in Python, but it does something quite different — it evaluates a string containing Python code. There is no harm in masking its definition, as we do not use it.

```
0.0

>>> eval(f2, x, 2)
9.010913347279288

>>> eval(f2, x, 1.4)
2.3149289879539445

>>> eval(f2, "y", 1)
... an error of some sort
```

For graphing purposes, in case of division by zero or domain error — for instance ln is not defined everywhere — you will return the value `float('inf')`, which is to say ∞. For this, you will need to use `try/except`. Note that you should be precise in which exceptions you catch:

```
>>> log(0)
ValueError: math domain error

>>> 1/0
ZeroDivisionError: division by zero
```

We specifically want to intercept `math domain error`, not all `ValueError`s. `ValueError`s can arise in many other cases, in fact it is probably a `ValueError` which we want to raise if we evaluate an improper expression. To avoid catching unwanted exceptions, use something like

```
except ValueError as e:
    if str(e) =="math domain error":
        return float('inf')
    raise e
```

Thus, we do not interfere with our ability to raise `ValueError` when faced with an expression which we cannot evaluate:

```
>>> eval([],x,1)
ValueError: []
```

Note that it is possible to use the same trick as for string conversion to handle all `BinExpr` in one line. Just as we have an attribute `symb` that contains the symbol of a construct, we can have an attribute `sem` that contains its semantics.

For instance I have

```
class Plus       (BinOp):
    symb = "+"
    sem = lambda x,y:x+y
```

You can do the same thing not only for all operators, but also for Call and Pow.

There is a niggling little difficulty to it, though. In Sec. 28[p202]: "Advanced structural pattern matching", we could write directly

```
match e:
    case BinOp(l,r):    return f"({fstr(l)} {e.symb} {fstr(r)})"
```

Since sem is a function, and not a constant like symb, when BinOp (or BinExpr) is instantiated it becomes a bound method, and thus takes self as a first argument. Recall that e.sem(a,b) is a notational shortcut for BinExpr.sem(e,a,b), if e is of type BinExpr.

To avoid this, you need to get the attribute sem not from the *instance* e, but from the *type* BinExpr. Thus you will write something like type(e).sem(a,b).

If you run into the error

```
TypeError: Call.<lambda>() takes 2 positional arguments but 3 were given
```

That is probably the origin of the problem.

Another potential difficulty is the handling of Call. We restrict ourselves to things like Call("ln", e), where the left-hand side is a constant function name — and we have only need of ln, specifically, but would like to be able to extend to sin, cos, etc. You need to write a semantics attribute of the form $\lambda f, e : [\![f]\!](e)$, where, for instance, $[\![\cdot]\!]$:"ln"$\mapsto \ln$. Think carefully about how to do that.

Or, you could just write one line per operator and not have to think about any of that, but that's just no fun at all.

Recall as well the trick of using z = lambda e: eval(e,var,val) for the recursive calls, explained at the end of Sec. 28.1[p207]: "An overlong aside on naming conventions".

(94) Now that eval is all set, let us graph our functions. This time we do not get to cheat with numpy to define the functions — recall the magical x in question (42)[p252]:

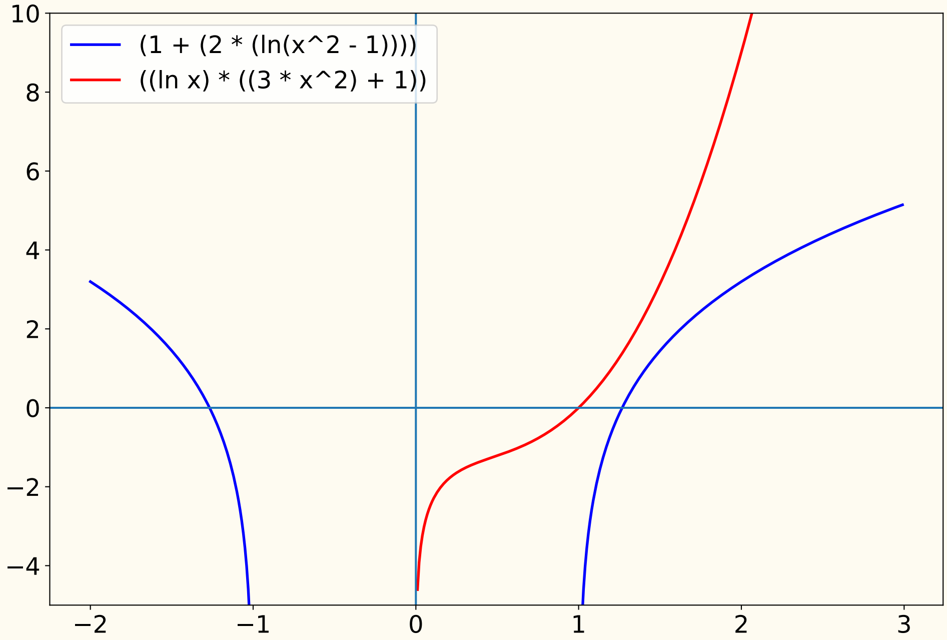
```
x = np.linspace(-2,2,100) # x varies in [-2,2], 100 uniform samples
npf = f(x)
```

— we do it the hard way instead, by generating sequences of couples (x,f(x)), and graphing that.

```
import matplotlib.pyplot as plt

plt.figure(figsize=(12,8))
plt.rcParams.update({"font.size": 18 })

X = [-2 + i/100 for i in range(500) ]
Yf1 = [eval(f1,x,X) for X in X]
```

```
Yf2 = [eval(f2,x,X) for X in X]
plt.ylim([-5, 10]) # limit the y axis
plt.plot(X,Yf1,"b",label=estr(f1), linewidth=2)
plt.plot(X,Yf2,"r",label=estr(f2), linewidth=2)

plt.legend(loc="best")
plt.axvline(0); plt.axhline(0)

##plt.savefig("../excasf1f2.pdf", transparent=True)
plt.show()
```

You should obtain this:



(95) Now we can move on to the very heart of the matter: symbolically computing the derivative. Your goal is to write a function `D(e,x)`, where `e` is an expression and `x` a variable name — in practice `"x"` — that returns an expression for $\frac{\mathrm{d}}{\mathrm{d}x}e$, the derivative of `e` with respect to `x`.

For instance, we should get

```
print("f2:", estr(f2), "\n\t->")
print(estr(D(f2,x)))
--------------------------------------------------------------------
f2: ((ln x) * ((3 * x^2) + 1))
  ->
(((1 / x) * ((3 * x^2) + 1)) + ((ln x) * (((0 * x^2)
                                           + (3 * (2 * x^1))) + 0)))
```

For now we shall not make any attempt at simplifying the expressions thereby obtained — e.g. multiplications and additions by 0 — that will be the goal of the next question.

To achieve the computation of the derivation, recall (some of) the rules of derivation.

We have:

$$
\begin{aligned}
\frac{d}{dx}c &= 0 \qquad c \in \mathbb{R} \\
\frac{d}{dx}x^n &= nx^{n-1} \\
\frac{d}{dx}\ln x &= \frac{1}{x}
\end{aligned}
$$

as well as, using f' as short for $\frac{d}{dx}f(x)$:

$$
\begin{aligned}
(cf)' &= cf' \qquad c \in \mathbb{R} \\
(f+g)' &= f' + g' \\
(f-g)' &= f' - g' \\
(fg)' &= f'g + fg'
\end{aligned}
$$

Those rules should be enough to handle f2. For f1, you will also need to support the chain rule, or composition rule:

$$(g \circ f)'(x) = g'(f(x))f'(x) \; .$$

Don't do that for now, we shall come back to it later, once we have a complete chain for f1. . .

(96) We can now compute derivatives, and they are correct, but there are many obvious simplifications left on the table. We want to define a function simp(e) to simplify the expressions e we obtain.

Simplifying mathematics equations is actually a very difficult topic in all generality, where most questions become undecidable. Indeed there are so many ways to manipulate the expressions, and no clear criterion of when the expression is “fully simplified ”. Think of all the possibilities when applying associativity and commutativity rules to all operators, and so on. That way lies madness.

We shall instead only pick up on the most obvious simplifications involving the constants 0 and 1. For instance, we want to obtain:

```
print(estr(D(f1,x)))
print(estr(Df1 := simp(D(f1,x))))
----------------------------------------------------------------------
(((1 / x) * ((3 * x^2) + 1)) + ((ln x) * (((0 * x^2)
                                             + (3 * (2 * x^1))) + 0)))
(((1 / x) * ((3 * x^2) + 1)) + ((ln x) * (6 * x)))
```

Even this is not fully straightforward to code. Consider the expression

$$(0 \times x^2) + (3 \times e) \; ,$$

where e is some sub-expression. When doing your recursive descent into the structure, you only see something of the form

```
Plus( Mul(..), Mul(..) )
```

you don't know yet that the left-hand side is zero, so you cannot simplify. You will need to come back later, from the top, and do another pass. The alternative would be to write deep patterns, like

```
Plus( Mul(0, e)), Mul(..) )
Plus( Mul(e, 0)), Mul(..) )
...
```

but the number of rules explodes exponentially with the number of simplifications you want to detect as well as the depth of detected patterns. This is not sustainable.

So, you are not going to write `simp` immediately. First, write a function `fixpoint(f,e)` that applies a function `f` on `e` repeatedly, until a fixed point e^* is reached: that is, until $f^n(e) = f^{n+1}(e) = e^*$. It then returns e^*.

In other words, `f` is applied on `e` until it can find nothing left to change.

(97) Now that we have `fixpoint`, we can code `simp`. The idea is that we shall have the architecture suggested at the end of Sec. 28.1[p207]: "An overlong aside on naming conventions":

```
def simp(e):
    def z(e):
        match e:
            case Plus(0,e) | Plus(e,0): return e
            ...
    return fixpoint(z,e)
```

The sub-function `z` does only one pass, but it is applied repeatedly until all simplifications are exhausted. For the patterns themselves, start with the simplest identity function you can write, then add the special patterns — `Plus(0,e)` etc — on top. Make it so.

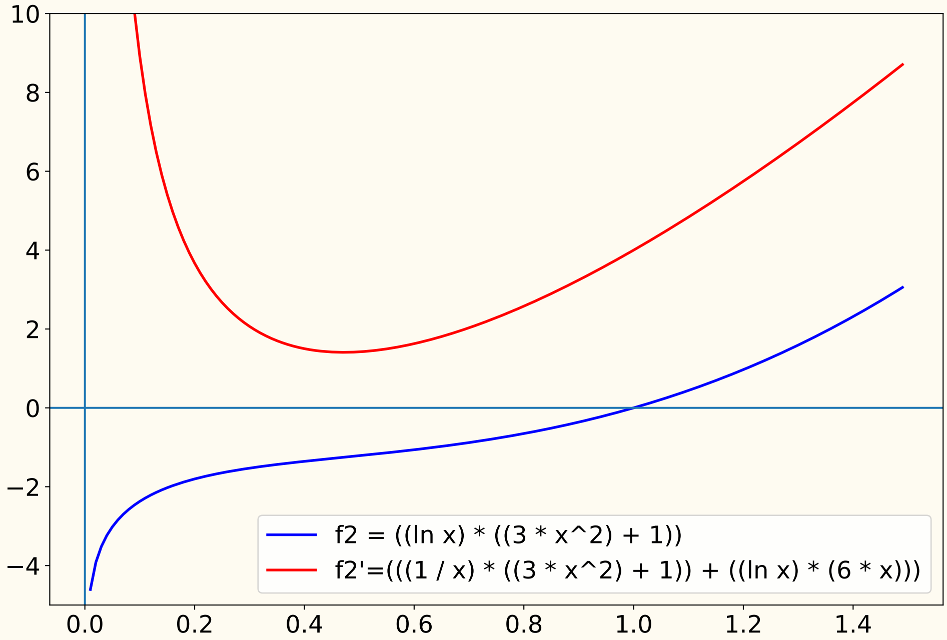
We obtain

```
(((1 / x) * ((3 * x^2) + 1)) + ((ln x) * (6 * x)))
```

for the derivative of `f2`, which matches

$$\frac{d}{dx}\left(\ln x(3x^2+1)\right) \quad = \quad 3x + \frac{1}{x} + 6x\ln x\,.$$

(98) Using our `eval`, `simp`, and `D` functions, produce a plot of `f2` and its derivative:



(99) Now let us deal with f1. Recall that $g \circ f(x) = g(f(x))$ and that we need the chain rule

$$(g \circ f)'(x) = g'(f(x))f'(x) \ ,$$

which we are going to write more compactly as

$$(g \circ f)' = (g' \circ f)f' \ .$$

How does that apply here? We have $\ln(x^2 - 1)$; $g = \ln$ and $f = \lambda x : x^2 - 1$. Let us simplify that view and consider any expression as a function — by default, a function of x.

Under that view, we have two expressions $g = \ln x$ and $f = x^2 - 1$. We already have the machinery necessary to derive either expression. There remains to implement the composition $\circ$.

It is actually very simple: $g \circ f$ it is the substitution of all instances of x in g by the expression of f. In our example:

$$g \circ f \quad = \quad (\ln x)[x \leftarrow f] \quad = \quad (\ln x)\left[x \leftarrow x^2 - 1\right] \quad = \quad \ln(x^2 - 1) \ .$$

Likewise we can compute

$$\begin{aligned}(g' \circ f)f' \quad &= \quad \big((\ln x)'\big)\left[x \leftarrow x^2 - 1\right] \cdot (x^2 - 1)' \\ &= \quad \left(\frac{1}{x}\right)\left[x \leftarrow x^2 - 1\right] \cdot 2x \\ &= \quad \frac{1}{x^2 - 1} \cdot 2x \\ &= \quad \frac{2x}{x^2 - 1} \ ,\end{aligned}$$

and indeed

$$\frac{d}{dx}\ln(x^2-1) \quad = \quad \frac{2x}{x^2-1}\,.$$

That means we have already all the tools we need except for a substitution function. Let us remedy that.

Write a function `sub(e,x,f)` that returns the expression obtained by substituting in `e` every instance of `x` by the expression `f`. For instance, we should have

```
>>> estr( sub(Minus(Mul(2,x),x), x, Plus(1, Pow(x,3))) )
'((2 * (1 + x^3)) - (1 + x^3))'
```

This is not a difficult function to write: it is the identity function, with just one more rule.

(100) Now you can extend the differentiation function `D` to support the chain rule. All you need is a single new **`case`** line.

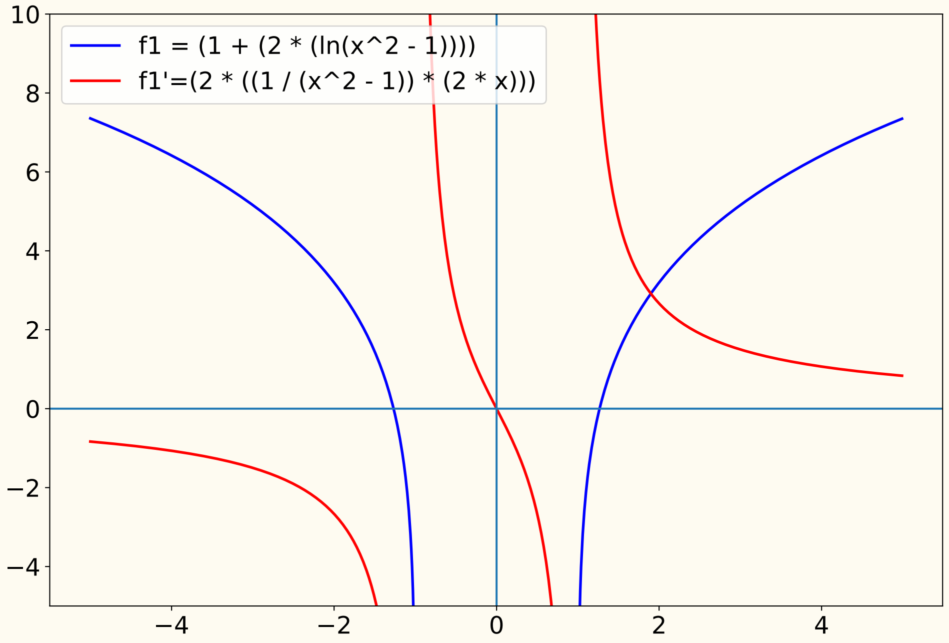
You should obtain

```
print("f1:", estr(f1), "\n\t->")
print(estr(D(f1,x)))
print(estr(Df1 := simp(D(f1,x))))
---------------------------------------------------------
f1: (1 + (2 * (ln(x^2 - 1))))
  ->
(0 + ((0 * (ln(x^2 - 1))) + (2 * ((1 / (x^2 - 1))
                                     * ((2 * x^1) - 0)))))
(2 * ((1 / (x^2 - 1)) * (2 * x)))
```

which is as expected:

$$\frac{d}{dx}\big(1+2\ln(x^2-1)\big) \quad = \quad \frac{4x}{x^2-1}\,.$$

(101) Now that all is said and done, plot `f1` and its derivative:



45.2 I object!

Reading Sec. 27$_{[p183]}$: "Object Oriented Programming in Python" is required for this part of the exercise.

OO wrappers around procedural/functional implementations

Let us make our CAS more user-friendly by setting up a layer of object-oriented syntactic sugar around it. The goal is to set up a wrapper class `F` — for **F**ormula — around our expression type, so that the user can employ the usual syntax to define symbolic expressions. For instance, we should be able to write

```
X  = F('x')                        # declare a symbolic variable
ln = lambda x: F(Call("ln",x.f))    # declare a symbolic function

F1 = F(f1); F2 = F(f2)
FF1 = 1 + 2*ln(X**2 - 1)

-----------------------------------------------------------------

>>> FF1
(1 + (2 * (ln(x^2 - 1))))

>>> F1(X,2)
3.1972245773362196

>>> FF1.D(X)
(2 * ((1 / (x^2 - 1)) * (2 * x)))
```

(102) Begin by creating a class `F` that acts as a wrapper for string conversion. We should be able to do

```
>>> f1
Plus(a=1, b=Mul(a=2, b=Call(a='ln', b=Minus(a=Pow(a='x', b=2), b=1))))

>>> F1
(1 + (2 * (ln(x^2 - 1))))

>>> F1.f  # the expression is stored internally as attribute f
Plus(a=1, b=Mul(a=2, b=Call(a='ln', b=Minus(a=Pow(a='x', b=2), b=1))))

>>> repr(F1)
'(1 + (2 * (ln(x^2 - 1))))'

>>> str(F1)
'(1 + (2 * (ln(x^2 - 1))))'
```

(103) Extend the class to support

```
>>> F1 + F1
((1 + (2 * (ln(x^2 - 1)))) + (1 + (2 * (ln(x^2 - 1)))))
```

(104) Extend the class to support

```
>>> F1 + 10
((1 + (2 * (ln(x^2 - 1)))) + 10)
```

(105) Extend the class to support

```
>>> 10 + F1
(10 + (1 + (2 * (ln(x^2 - 1)))))
```

(106) At this point imagine what the code is going to look like once you support every operator. There is some factorisation to do. Write a "dispatch" function `disp(op,s,o)` and a function `rdisp(op,s,o)` so that your implementation of + support looks like

```
class F:
    ...
    def  __add__(s,o): return  disp(Plus,s,o)
    def __radd__(s,o): return rdisp(Plus,s,o)
```

(107) Using this, quickly add support for `*, **, -`, so that we can handle

```
X  = F('x')                        # declare a symbolic variable
ln = lambda x: F(Call("ln",x.f))    # declare a symbolic function

FF1 = 1 + 2*ln(X**2 - 1)

-----------------------------------------------------------------
```

```
>>> FF1
(1 + (2 * (ln(x^2 - 1))))
```

(108) Extend the class so that we can write

```
>>> FF1.D(X)
(2 * ((1 / (x^2 - 1)) * (2 * x)))
```

(109) Extend the class so that we can write

```
>>> F1(X,2)
3.1972245773362196
```

instead of

```
>>> eval(f1,x,2)
3.1972245773362196
```

You should be getting the idea by now. . . Using these techniques, we can completely hide our underlying datatype from the end user.

(110) **(Perspectives)** The exercise stops there, but there is no end to the interesting things we could do to improve and extend our CAS. Extensive automatic simplification, handling of integration, an interactive mode where the user chooses which rules to apply to their system, LaTeX output and display, and so on, and so forth.

If you are interested in this, that can be the object of an "Application Projet" at the end of the year — one week full-time projects done in groups of four. Ask me about it if that kind of thing is your cup of tea.

46 Conway sequence: generating fun

Completing this exercise requires a good understanding of Sec. 29[p208]: "Iterables, iterators, and generators", in particular Sec. 29.4[p213]: "Understanding deeply lazy computations".

lazy evaluation = performance (often)
implementing lazy evaluation in a complex problem

In this section, we shall play with *Conway* sequences[an], also called *look-and-say* sequences. Mostly, we shall focus on the Conway sequence with seed $C_0 = 1$. Here are the first few

[an] Following Stigler's law of eponymy, Conway sequences are actually due to. . . errrr, ok, Conway really *did* invent that. The exception to the rule, I guess. Never mind, then. Carry on.

elements of this sequence:

$$\begin{aligned}
C_0 &= 1\\
C_1 &= 11\\
C_2 &= 21\\
C_3 &= 1211\\
C_4 &= 111221\\
C_5 &= 312211\\
C_6 &= 13112221\\
C_7 &= 1113213211\\
C_8 &= 31131211131221\\
C_9 &= 13211311123113112211\\
C_{10} &= 11131221133112132113212221\\
C_{11} &= 3113112221232112111312211312113211\\
&\dots
\end{aligned}$$

How is it defined? C_{n+1} is defined recursively from C_n as the sequence of numbers obtained by reading the digits of C_n out loud, organised by groups of identical digits, announcing first the number of digits, then the digit in each group.

For instance:

- 1 is read as "one 1" : 11.
- 11 is read as "two 1s" : 21.
- 21 is read as "one 2, followed by one 1" : 1211.
- 1211 is read as "one 1, one 2, and two 1s" : 111221.
- 111221 is read as "three 1s, two 2s, and one 1" : 312211.
- and so on...

We want not only to generate this sequence, but to do so efficiently, getting only the first few digits of each number up to a high rank, even though the length of C_n grows exponentially with respect to n. It is clear that the last digits of, say, C_{10} are not involved in the computation of the *first* digits of C_{11}, so if that's what we really need, why compute C_{10} all the way?

Of course, we shall also write a more traditional, sequential implementation as well, for comparison purposes.

You may not use anything from `itertools`, as I ask to to reimplement some of its functionality.

(111) **mixing `yield` and `return` in a generator function**
using `enumerate`

Write a function `upto(g,i)`, returning a generator for the first `i` elements generated by `g`.

Note: *prior to version 3.7, this could and should have been done in one line. This is no longer the case due to changes in the semantics of generator expressions.*

The following assertions must hold:

```
assert next(upto(range(3),8)) == 0

assert list(upto((x for x in range(3)),8)) == [0, 1, 2]

assert list(upto((n*n for n in range(100)),7)) == \
    [0, 1, 4, 9, 16, 25, 36]
```

(112) For fun, write a function `nth(g,n)`, returning the `n`-th element of an iterator `g`.

This can and should be done in one line.

The following assertion must hold:

```
assert all( nth(g,i) == i*i
            for i in range(7)
            for g in [(n*n for n in range(7))] )
```

(113) Write a function `powers(f,s)`, where f is a unary function, that returns a generator for the successive powers $s,\ f(s),\ f^2(s),\ f^3(s), \ldots$, where

$$\begin{aligned} f^0(x) &= \lambda x.x && \text{(identity function)} \\ f^n(x) &= f \circ f^{n-1},\ n > 0 \end{aligned}$$

The following assertion must hold:

```
assert next(powers(lambda x:2*x,1)) == 1

assert list(upto(powers(lambda x:2*x,1),7))  == \
    [1, 2, 4, 8, 16, 32, 64]
```

(114) Write a function `group(l)`, with `l` being an iterable, that returns a generator for the groups of successive identical elements appearing in `l`. Each group shall be returned as a list.

The following assertions must hold:

```
assert next(group('a')) == ['a']

assert list(group('')) == []

assert list(group('a')) == [['a']]
assert list(group('aaba')) == [['a', 'a'], ['b'], ['a']]

assert list(group('aabbbcdaaaa')) == \
    [['a', 'a'], ['b', 'b', 'b'], ['c'], ['d'], ['a', 'a', 'a', 'a']]
```

This is a simpler version of `itertools.groupby`.

(115) For our purposes, it is probably more efficient to generate the groups as couples `(length,element)` rather than as lists of identical elements. Write a function `groupn(l)`, similar to `group(l)`, but generating said couples.

The following assertions must hold:

```
assert next(groupn('a')) == (1, 'a')

assert list(groupn('aabbbcdaaaa')) == \
    [(2, 'a'), (3, 'b'), (1, 'c'), (1, 'd'), (4, 'a')]
```

(116) For fun, bridge the gap between `groupn` and `group` by writing a function `groupl` that takes as input the output of `groupn`, and converts it into the output of `group`.

This can and should be done in one line.

The following assertions must hold:

```
assert next(groupl(groupn('aa'))) == ['a', 'a']

assert all ( tuple(group(s)) == tuple(groupl(groupn(s)))
             for s in ('','a','aaba','aabbbcdaaaa') )
```

(117) Using `groupn` — since it is the most efficient — write a function `say(s)` that transforms any string `s` into its "look-and-say" version. That is to say, a function that transforms a string representing C_n into a string representing C_{n+1}.

This can and should be done in one line.

The following assertion must hold:

```
assert list(upto(powers(say,'1'),7)) == \
    ['1', '11', '21', '1211', '111221', '312211', '13112221']

assert list(upto(powers(say,'22'),7)) == \
    ['22', '22', '22', '22', '22', '22', '22']
```

(118) Write a procedure `conway(seed='1',maxrnk=100,maxlen=30)` that displays the Conway sequence of seed `seed`, up to and including rank `maxrnk`.

As elements of the sequence grow exponentially in size, we truncate their display to the first `maxlen` digits.

The output of a call of `conway()` should look like this:

```
  0   1
  1   11
  2   21
  3   1211
  4   111221
  5   312211
  6   13112221
  7   1113213211
  8   31131211131221
  9   13211311123113112211
 10   11131221133112132113212221
 11   3113112221232112111312211312 11...
...
 60   13211321322113311213212312311 2...
 61   11131221131211132221232112111 3...
```

Spoiler alert: the display should begin to slow down around rank `50`, and slow down to a crawl around rank `60`, making it impractical to go much farther.

To understand why, recall that our implementation of say needs the whole of C_n to begin computing C_{n+1}. C_{50} has 1 166 642 digits; C_{60} has 16 530 884. Keeping up with this quickly becomes impractical.

Yet, we only need a few digits from the beginning of each element, and it is clear, from the way the sequence is constructed, that those depend only on the first few digits of the previous ranks. Thus we only actually make use of an infinitesimal fraction of the digits we compute.

To exploit that fact, we shall overhaul our computation to make sure that there are generators every step of the way.

(119) Write a function `sayg(s)` playing the same role as say, except that instead of taking and returning strings, it takes an iterable and returns a generator.

This can and should be done in one line.

The following assertion must hold:

```
assert list(sayg(''))       == []
assert list(sayg('1'))      == ['1', '1']
assert list(sayg('1211'))   == ['1', '1', '1', '2', '2', '1']
```

(120) Write a function `nthpowerg(f,n,s)`, where f is a unary function, n an integer, and s a seed value, that returns

- ◇ a single-value generator for s if $n = 0$

- ◇ an iterator for $f^n(s)$ otherwise, assuming f is an iterator function.

This can be done either iteratively or recursively.

The following assertion must hold:

```
assert "".join( upto(nthpowerg(sayg,6,'1'),8) ) == '13112221'
```

(121) Write a procedure conwayg, equivalent to conway, but using generators exclusively to achieve the same result. This time, performance should not be an issue. A call to conwayg() should look like this:

```
  0  1
  1  11
  2  21
  3  1211
  4  111221
  5  312211
  6  13112221
  7  1113213211
  8  31131211131221
  9  13211311123113112211
 10  11131221133112132113212221
 11  3113112221232112111312211312 11...
 12  1321132132111213122112311311 22...
...
 60  1321132132211331121321231231 12...
 61  1113122113121113222123211211 13...
...
 99  1321132132211331121321231231 12...
100  1113122113121113222123211211 13...
```

and take no time at all.

Note that we have achieved this considerable speedup without in any way lessening the generality of our code, or increasing its complexity — excepting the fact the generators require somewhat more abstraction from the programmer.

Conventional programming could possibly achieve a similar speedup by simply computing the first digits in a fixed-length (maxlen) array. This should work, because the sequence visibly "inflates" with each step, so that the maxlen first digits of each rank can be computed with *at most* maxlen digits of the previous one.

However, we would need to prove that mathematically to have confidence in such code. Moreover, we would need to do so for all possible seeds. Furthermore, we shall see later on that this would be futile because you *cannot* prove that: experimentally, we quickly find values of maxlen for which the hypothetical property stated above is simply false. Perhaps we could show that `maxlen` plus some constant would work? Perhaps it holds for all values of `maxlen` greater than some constant M? I don't know.

And still, even if it worked, which it doesn't, it would be inefficient when asking for a large amount of digits from high ranked values, as we would compute many unneeded digits.

And of course, while such an approach *might* plausibly have worked – or could maybe be tweaked into working – for Conway sequences, it would flat out fail with any sequence deprived of anything resembling this supposed inflation property, whereas, using generators, we simply *don't have to care* about the behaviour of the sequence. We know our code is correct, in the sense that we know we are going to compute what we need, and hopefully no more.

There is a cost, of course: a little more thinking is required to manipulate generators correctly, and there is an overhead computational cost to having all those objects messaging each other saying "hey! wake up! I need a value!". If you need *all* the values *all* the time anyway, there is no point in using this; but if not, it is usually a very good investment to make your code as generator-friendly as possible.

(122) Let us quantify the gains from using generators. More specifically, supposing we want to get the first j digits of C_R, the questions are:

- ◇ How many digits need I compute, manually, to obtain that, globally and for each previous rank $k \leqslant R$?
- ◇ How many digits are actually computed by the generator-based method?
- ◇ How many digits are computed by the non-generator method?
- ◇ What is the ratio between those quantities?

First, as an example, let us see manually what is strictly needed to get the first digit of C_5:

$$\begin{aligned}
C_0 &= \underline{1} \\
C_1 &= \underline{11} \\
C_2 &= \underline{21} \\
C_3 &= \underline{121}1 \\
C_4 &= \underline{1112}21 \\
C_5 &= \underline{3}12211
\end{aligned}$$

To compute C_5's 3, we need four digits from C_4, as we cannot conclude as to the number of ones until we see a *different* digit.[ao]

[ao] Actually, we could show mathematically that no repetition longer than three can occur, and conclude upon

To compute C_4's 1112, we need 121 from C_3; indeed, without the final one from C_3, we don't know whether C_4 begins with 1112 or 1122 or 1132, etc. And so on, you get the idea.

Back to the generators. Write a procedure `perf(R,j)` evaluating the performance of generator-based versus classical approach on the computation of the first j digits of C_R. The output must look like this:

```
>>> perf(5,1)
Performance analysis: rank 5, 1 digit.
First 1 digit of C_5 = '3'
C_0 : 0 of 0
C_1 : 2 of 2
C_2 : 2 of 2
C_3 : 3 of 4
C_4 : 4 of 6
C_5 : 1 of 6
Total: 12 of 20, or 60.0%
```

For instance the line `C_4 : 4 of 6` means that generators computed four digits of C_4, whereas the classical approach computed all 6 — the classical approach computes the entirety of each rank. In total, generators computed 12 digits, whereas the classical approach computed 20. The seed is ignored in both counts.

Note that the generator approach should exactly match the manual reasoning above!

On higher ranks, you should get:

```
>>> perf(55,30)
Performance analysis: rank 55, 30 digits.
First 30 digits of C_55 = '111312211312111322212321121113'
C_0 : 0 of 0
..
C_4 : 6 of 6
..
C_10 : 5 of 26
..
C_20 : 4 of 408
..
C_30 : 4 of 5808
..
C_40 : 5 of 82350
..
C_50 : 12 of 1166642
C_51 : 12 of 1520986
C_52 : 17 of 1982710
C_53 : 20 of 2584304
C_54 : 21 of 3369156
```

seeing the third one. However, we haven't shown that, our say and sayg algorithms do not take that into account, and thus for now we don't know whether 1111 may occur, so we need to see the next digit.

```
C_55 : 30 of 4391702
Total: 343 of 18858434, or 0.0018188148602370697%
```

Tips:

Implementing this is a bit tricky.

I advise creating lists to store the needed numbers of digits, one for generators and one for the classical approach, and writing "hacked" versions of `nthpowergp` and `saygp` so that the list concerned with generators is updated each time a digit is computed, as a side-effect.

Note that these hacked versions of `nthpowergp` and `saygp` can and should be subfunctions of `perf`.

(123) Find a simple counterexample for our earlier hopeful assertion that perhaps

> "The j first digits of each rank can be computed with *at most* j digits of each of the previous ranks."

(124) *Perspectives:* for those of you interested in going (much) farther in your understanding of lazy evaluation, I recommend implementing the Conway sequence in Haskell. Haskell is a pure functional language with lazy evaluation.

An implementation of Conway every bit as powerful as our generator version can be obtained completely transparently in just a few lines of code.

Of course, this is *far* outside the scope of this class; or of your curriculum, for that matter. You will not be taught Haskell — or OCaml, or Scheme (Lisp), or indeed any functional language — at INSA CVL. I would recommend studying this in your own time if (and only if) you wish to acquire a larger understanding of programming paradigms and techniques, and are not afraid of maths and abstraction.

Part IV

Additional Python Exercises

The following are retired exercises, made redundant by new material, as well as archives from various tests and non-INSA training sessions. They are provided in no particular order.

They can provide additional fun to any student who may prematurely run out of stuff to do during classes... Idle hands are the Devil's playthings, after all.

Part V[p339]: "DIU EIL: Récursivité" contains even *more* additional exercises, specifically geared around the concept of recursiveness, which you are encouraged to practice on.

47 Fear the floating-point ranges

Many of you, while answering question (40)[p252], wrote something like `range(-2, 2, .1)`, and were surprised to learn that range does not support floating-point steps:

```
>>> range(-2, 2, .1)
TypeError: 'float' object cannot be interpreted as an integer
```

Some students' reactions even implied that they deemed it quite lazy indeed of the Python developers not to have bothered implementing that trivial use-case.

If you are among them, I regret to say that in spite of my best efforts so far, you have not yet learned to fear floating-point numbers. Not properly. Not enough. Not yet. But you will,... you will.

Since floating point-ranges are so easy, let's implement them ourselves. . . what's the worst thing that could happen? 😇

(125) Write a generator function `frange_inc(i,f,s)` returning all the floating points numbers between the initial value i (inclusive) and the final value f (exclusive), by successive increments of the step value s.

In other words, it should generate all numbers $i \leqslant i + ks < f$, for $k \in \mathbb{N}$.

This is basically the behaviour of `range`, but extending to floating-point numbers.

For instance, `frange_inc(0, 1, 0.1)` should be expected to yield $0, 0.1, 0.2, \ldots, 0.9$. Note that 1.0 is not generated, as the upper bound is exclusive; just like for `range`.

At least that's what we hope to obtain. . .

This specific implementation will proceed in the most natural way, by incrementing the current value by s at every step, hence the `_inc` in the name, and stopping when the value exceeds f.

If you implement it correctly[ap], the following assumptions should hold:

```
assert list(frange_inc(0,0,1))    == []
assert next(frange_inc(0,1,1))    == 0
assert list(frange_inc(-5,5,1.0)) == [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
assert list(frange_inc(.7,.8,.1)) == [0.7, 0.7999999999999999]
assert list(frange_inc(0,.8,.1))  == [0, 0.1, 0.2, 0.30000000000000004,
                                     0.4, 0.5, 0.6, 0.7, 0.7999999999999999]
```

(126) Looking at the assertions of the previous question, you may notice a *slight* problem: not only are there approximation errors in several places (who could possibly have seen this coming?), but some of them even change the number of values returned:

[ap] or as correctly as *possible* given the constraints, see next question. . .

```
list(frange_inc(.7,.8,.1)) == [0.7, 0.7999999999999999]
```

where we expected `[.7]`, as .8 is excluded. But we got a value *slightly* smaller than .8, so it passed. This is not an error in the logic of the implementation of `frange_inc`, mind you; the incrementation and test were correct, it is just a raw fact that, for instance

```
>>> .7 + .1
0.7999999999999999
>>> .7 + .1 < .8
True
>>> .2 + .1
0.30000000000000004
```

What are we to do about it? The obvious solution is to detect and correct such mistakes by separately computing how many numbers are expected. The formula is fairly easy:

$$\left\lceil \frac{f-i}{s} \right\rceil . \tag{47.1}$$

In case you don't know, $\lceil \cdot \rceil$ is the *ceiling* function:

$$\lceil x \rceil \quad = \quad \min\{\, n \in \mathbb{Z} \mid n \geqslant x \,\} .$$

So, for our problematic example, we expect, by this formula, $\left\lceil \frac{.8-.7}{.1} \right\rceil = \left\lceil \frac{.1}{.1} \right\rceil = 1$ number, which is coherent with our expected result of `[.7]`.

All there remains to do is to implement (47.1) in `frange_inc`, and count how many numbers have been returned. Problem solved!

If that sound good to you, **YOU STILL DON'T FEAR FLOATING-POINT NUMBERS ENOUGH, YOU SORRY FOOL!**

Let's test our brilliant idea in Python on our example:

```
>>> ceil((.8-.7)/0.1)
2
```

Wait, what? (Please insert "record scratch" sound effect here) Is our formula wrong? Does `ceil` not implement the ceiling function?

```
>>> (.8-.7)/0.1
1.0000000000000009
```

Nope, it's just that our galaxy-brained, 300 IQ master plan had a tiny, niggling, piffling little flaw: in order to bypass the inherent inability of a floating-point computation to provide exact answers, we double-checked with *another floating-point computation*. Can you spot the flaw yet? Do you fear **`float`** yet?

How do we solve this? Easy. We don't. We just don't. We do not solve this because it is not solvable. Not without using computationally expensive computer algebra.

It is not possible to implement a floating point range function that behaves as **range** does because loss of precision makes the very act of predicting and controlling the number of returned elements unpredictable. Therefore, a **range** and a frange on the same numbers have no guarantee to return the same *number* of values, let alone the same values.

If that's not big enough of a problem for you (who cares about consistency between functions that do essentially the same thing!), try this on for size:

```
>>> list(frange_inc(10**16, 10**16+2, 1.0))
```

I did not write a return value beneath, because there is none; you've got an infinite loop. Use CTRL+C in the terminal or interactive mode to interrupt the computation:

```
KeyboardInterrupt
```

What now?

```
>>> g = frange_inc(10**16, 10**16+2, 1.0)
>>> next(g)
10000000000000000
>>> next(g)
1e+16
>>> next(g)
1e+16
...
```

It returns 10^{16} in a loop, as though it forgot to increment at all… How is this even possible?!

```
>>> 10**16 + 1.0
1e+16
>>> 10**16 + 1.0 == 10**16
True
```

Oh. Oh! Right! Precision problems! Did you see that coming? You should have. For large enough numbers, the precision loss is so great that you can lose entire *units*. Or more; wanna see a magic trick, kid? I'll make the number 10 disappear:

```
>>> 1e+18 + 10 == 1e+18
True
```

Poof! You like that? You want more? Let's do 1000!

```
>>> 1e+19 + 1000 == 1e+19
True
```

POOF! Are you afraid of big bad **float** yet?

Let's re-evaluate my earlier statement in light of this: never mind consistency, we don't have a frange function with the same behaviour as **range** because *it would randomly get into infinite loops*, depending on the respective sizes of i, f, s. And that is why **range** runs away screaming when you feed it a floating-point number.

By the way, did I ever mention that floating-point numbers are scary?

(127) Lest you believe that I sabotaged us by enforcing a specific implementation based on successive incrementations by s, let us write frange_mul(i,f,s), with the same expected behaviour, but using multiplication instead to compute the $i \leqslant i + ks < f$, for $k \in \mathbb{N}$. That is to say, you will literally compute $i + ks$ in a loop.

If this is done correctly, you should obtain:

```
assert list(frange_mul(0,0,1))    == []
assert next(frange_mul(0,1,1))    == 0
assert list(frange_mul(-5,5,1.0)) == [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
assert list(frange_mul(.7,.8,.1)) == [0.7, 0.7999999999999999]
assert list(frange_mul(0,.8,.1))  == [0, 0.1, 0.2, 0.30000000000000004,
                       0.4, 0.5, 0.6000000000000001, 0.7000000000000001]
```

Rejoice, for we have traded one set of approximation errors for a slightly different one, and fixed... absolutely nothing.

Note that no implementation will solve the `1e+19 + 1000 == 1e+19` problem.

No implementation based on an indefinite loop (**while**) can work, because you can't know when or even *whether* to stop. No implementation based on a definite loop (**for**) can work, because you can't compute how many values you should return. No implementation can work, full stop.

Floating-point ranges based on a "step" are *fundamentally* flawed, regardless of implementation.

(128) Must we give up on floating-point ranges altogether? Of course not; we merely need think about them differently. Instead of giving a "step", we'll just give the number of values we expect.

Write a function frange(i,f,n) returning n floating point numbers uniformly spaced (within approximation error) on the interval $[i, f]$.

If $n = 1$ it shall return the middle of the interval.

```
assert list(frange(0,0,1))   == [0]
assert next(frange(0,1,1))   == 0.5
assert list(frange(0,1,1))   == [0.5]
assert list(frange(0,1,2))   == [0, 1]
assert list(frange(-5,4,10)) == [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
assert all( len(list(frange(0,1,n))) == n for n in range(100) )
assert list(frange(0,1,11))  == [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6,
                                 0.7, 0.8, 0.9, 1.0]
```

This looks much nicer.

That doesn't mean we don't still have precision problems. You can rely on obtaining however many numbers you asked for, but not on them being distinct:

```
>>> len(set(frange(10**15, 10**15+10, 10)))
10
>>> len(set(frange(10**16, 10**16+10, 10)))
6
>>> len(set(frange(10**17, 10**17+10, 10)))
2
>>> set(list(frange(10**17, 10**17+10, 10)))
{1e+17, 1.0000000000000002e+17}
```

This is the best you can hope for when you only have 2^{64} floating point numbers (they occupy 64 bits) to represent $|\mathbb{R}| = \infty$ real numbers (and that's a *dense* infinity: $|\mathbb{R}| = \aleph_1 > |\mathbb{N}| = \aleph_0$ [aq]).

In other words, the above is the most sensible implementation of a floating-point range you can get. At least, that I know of. Feel free to share your brilliant ideas.

Oh,... do you fear **float**ing-point numbers yet? Just checking.

48 Let's decorate!

Be sure to read and understand Sec. 25.2[p174]*: "Function decorators" before tackling this exercise.*

(129) Sometimes, you might want to slow code down; either because you want the time to read the messages, or because some function is hammering a resource too hard, refreshing a web page 10 times a second, something like that.

To that effect, let us write a parametric decorator `slow(n)` that forces a function to wait for `n` seconds every time it is called. It must work on recursive function calls:

For instance:

```
@slow(1)
def verbose(n=100):
    if n <= 0: return
    print(n, "I do stuff and I talk about it!")
    verbose(n-1)
---------------------------------------------------
                                    # wait a second
```

[aq] https://en.wikipedia.org/wiki/Aleph_number

```
100 I do stuff and I talk about it! # wait a second
99 I do stuff and I talk about it!  # wait a second
...
```

(130) Following the same principle, write a parametric decorator `slow_scroll(n)` that allows a function to be called `n` times without delay, but then stops everything, waiting for the user to press ENTER. When he does, the function can again be called `n` times before being stopped, and so on.

Applying `@slow_scroll(3)` on our `verbose` function, we have:

```
100 I do stuff and I talk about it!
99 I do stuff and I talk about it!
98 I do stuff and I talk about it!
----------------------------------------------------------------------------
        # user presses ENTER
97 I do stuff and I talk about it!
96 I do stuff and I talk about it!
95 I do stuff and I talk about it!
----------------------------------------------------------------------------
        # user presses ENTER
94 I do stuff and I talk about it!
....
2 I do stuff and I talk about it!
----------------------------------------------------------------------------
        # user presses ENTER
1 I do stuff and I talk about it!
```

49 Rage against the virtual machine

`ex_vdisks.tex` WORK IN PROGRESS! SKIP FOR NOW !

In this exercise, we will implement functionality inspired by virtual machine sparse disks and snapshots.

Disclaimers: Note that I use VirtualBox and will stay close to its terminology, so some details and terminology may differ in other software, but the ideas here are universal. I'll also abstract away a lot of details, and my objective is not to write a reference document on Virtual Box or any other implementation, so do not take it as such.

If you have use modern virtual machine software, you have noticed that they have some pretty nifty functionalities. Among other things:

(131) Virtual disks can be much smaller on the host than on the guest (unless you select "Fixed Size" when creating them as opposed to the default "Dynamically Allocated", which you probably shouldn't, even for performance reasons.

(132) You can take *snapshots* of the current state of the machine (here we'll focus on the main disk) at any point, and restore them later. If at some point you decide to restore a snapshot, you don't even have to lose your current state: you can snapshot it as well, and then restore the earlier snapshot.

Thus you can create "alternative timelines". You can install linux, snapshot (let's call the state s), install Apache, snapshot (s'), restore s, install NGinx, snapshot (s''), and now s' and s'' represent two diverging timelines, sharing the past up to state s, and no further.

This would not be terribly impressive if the implementation just took full copies of the disks each time, but in practice, taking snapshots is virtually instantaneous and consumes very little disk space at first! However, *deleting* a snapshot takes some time, and snapshots that "live" long take more and more space.

What's going on under the hood? I think most of you should have no difficulty imagining an implementation for **(1)**, especially if you've ever met the keyword *sparse* in Maths or CompSci. The idea is obviously to only store data when it is actually written by the guest, as opposed to preallocating everything. You just need a data structure of the form "sector X → data, sector Y → data, ...". [ar] The file starts empty and fills up as sectors are written in over time. When reading, any sector not appearing in the file is considered empty. We'll call that structure a *sparse disk*.

Note that in practice, "sectors" might be 1MB pages, as for the VDI format[as], or anything else. We'll ignore that kind of gritty details.

Now, understanding what's going on with snapshots in **(2)** takes a little bit more work, but the fundamental idea is actually the same as **(1)**: only write in that which changes.

When a snapshot is taken, the current disk file goes read-only, and a new, empty, *differencing disk* is created, which takes all writes from that point on. That is to stay, it stores only the changes compared to the original disk. It actually has the same structure as a sparse disk, and we shall not concern ourselves with the distinction between the two any longer.

When reading, the machine first looks in the differencing disk, and if the sector it is looking for is not there, it looks in the "older", read-only disk — which we shall call its *parent*. Further snapshots can be created and *chained* in this way.

Whereas taking a snapshot is instantaneous, because we merely create a new, empty differencing disk, deleting one involves *merging* disks. At this point, things become non-trivial and, if Virtual Box's forums are anything to go by, there is a lot of confusion among users as to what's going on, the most common error being not understanding the distinction between *snapshots*, which are merely points in time which we have interest in preserving, and *differencing disks*. Snapshots are *deleted*, not merged. Disks are merged, and the effect of

[ar] You might notice a *slight* resemblance with a certain Python data structure. . .

[as] https://forums.virtualbox.org/viewtopic.php?f=35&t=8046

that is deleting a snapshot.

To alleviate any confusion, we shall formalise things a bit.

We see a disk as an array (or Python list) of data. Let $N \in \mathbb{N}$ be the size of our disk, $X = [\![0, N-1]\!]$ our *sectors* and Y some target data space. For instance, if N is the size in bits, then $Y = \{0, 1\}$. A *disk* is a total function $D : X \to Y$.

A *differencing disk* is a partial function $\Delta : X \nrightarrow Y$. Differencing disks Δ_1 and Δ_2 can be *chained*, which we shall write multiplicatively, with the following semantics:

$$\Delta_1\Delta_2 : \left| \begin{array}{lcc} X & \longrightarrow & Y \\ i & \longmapsto & \Delta_2(i) \text{ if defined, else } \Delta_1(i) \end{array} \right.$$

To translate the concept of sparse disk, all we need is to chain a differencing disk after the *empty disk*, defined as

$$\varepsilon : \left| \begin{array}{lcc} X & \longrightarrow & Y \\ i & \longmapsto & 0 \end{array} \right.$$

which returns 0 — or, frankly, any placeholder value, for every sector. We shall use "_" as placeholder value in the implementation. Note that ε is just an abstraction, and we do not imply that the implementation explicitly allocates and zeroes out every sector.

We'll speak of "global state" to mean not just the state of the disk at a given point in time, but the whole chain of disks and snapshots for the machine. Our default global state, with one sparse disk and no snapshot, is therefore of the form $\varepsilon\Delta_0$. In practice only Δ_0 actually occupies space on the disk, ε is just the concept of what happens when a sector is not defined in Δ_0.

Note that this, and by extension any chain of the form $\varepsilon\Delta_0 \dots \Delta_n$, describes a disk, that is to say a total function $X \to Y$, because ε is total. If none of the Δ_k has data for a sector, then in the end ε always does. By convention, when writing such chains we understand that each Δ_k corresponds to one physical file on the host.

Taking a snapshot means that Δ_0 goes read-only, and a new differencing disk is chained: $\varepsilon\Delta_0\Delta_1$. To denote snapshots, or other states of interest, like the current state, we'll use the letters s and c, respectively, and insert them in the chain. They do not change its semantics, but help us to keep track of what's going on. Thus, we would write the global state above $\varepsilon\Delta_0 s_1 \Delta_1 c$, with our snapshot named s_1, and our current state c.

The act of taking a snapshot therefore translates into "substitute $s\Delta c$ for c", with a fresh differencing disk Δ.

We now have the requisite notations to explain snapshot deletion clearly. Suppose we are in state $\varepsilon\Delta_0 s_1 \Delta_1 c$ and no longer need the snapshot: then we can combine $\varepsilon\Delta_0\Delta_1 c$ into $\varepsilon(\Delta_0\Delta_1)c$, that is to say, merge the two differencing disks into one. Note that, in terms of semantics, trivially $\varepsilon\Delta_0\Delta_1 c = \varepsilon(\Delta_0\Delta_1)c$, so we have not altered our current state, and in terms of implementation, computing a disk representing $\Delta_0\Delta_1$ is easy: it suffices to overwrite all data of Δ_0 with Δ_1. We say that we merge Δ_1 *into* its parent Δ_0.

The act of deleting a snapshot s therefore translates, in general, as replacing a pattern $\Delta s\Delta'$ in the chain by $(\Delta\Delta')$.

We will not deal with snapshot restauration or trees of snapshots in this exercise, because at this point we have all the main mechanics and the rest is just a matter of careful bookkeeping when moving the current state around. You are free to extend the exercise with this on your own, if you like.

The preliminaries are done, let's get to the implementation.

(133) We'll create a class `disk` for differencing disks (in fact, arrays or lists) with `"_"` as placeholder value for undefined sectors.

The first and only mandatory argument of the class constructor should be the size of the array. The class will use a Python dictionary as its internal data structure, and *not* store all the array in memory.

Furthermore, we need to be able to print the state of the disk, displaying its contents in extenso, what's really stored in the dictionary, and optionally a comment, all on one line.

As the start of our running example, the following code

```
D = disk(10)
D.print("ε Δ0 c (untouched)")
```

should yield:

```
__________ {}                    | ε Δ0 c (untouched)
```

That is to say, we create an empty disk of size 10, and see that internally it's just an empty dictionary, despite "containing" ten instances of `"_"`.

There is quite a lot to do to get there. You'll need an `__init__` method, of course, but for the purposes of printing, you will need several things, besides obviously a **`print`** method. First, you will need a way to query the disk for the value of a given sector. You will use index notation for this, so that for instance

```
>>> D[10]
'_'
```

For this, you'll implement a `__getitem__` method. No need to think about chaining for now, we can take care of that later. You might want to add a defensive assertion to ensure that the index is in X.

You'll implement `__repr__` and `__str__` such that

```
>>> repr(D)
'{}'

>>> str(D)
'__________'
```

Each of the two can and should be implemented in one line.

Do not duplicate the reading logic in `str`! We'll need to alter that logic later, and it's bad practice in general to duplicate logic. You have *one* method that does the reading, and that's `__getitem__`. Everything else should use it if they need to know what's on a disk sector.

Will all this, you can at last implement **`print`**.

(134) Let's implement write support. For instance we should have:

```
>>> D = disk(10)
>>> D[5] = 'A'

>>> print(D)
_____A____

>>> D
{5: 'A'}
```

For this, you'll need to implement the `__setitem__` method. The logic is trivial so this should be a very short line, not counting the assertion about the index being in X, which is probably good to have.

Since any differencing disk that is being written to is the only one that can be written to — the ancestors are read-only — the writing logic is complete, we won't have to come back to this method.

(135) For our convenience, let's implement a method `D.writeat(i, l)` that writes the elements of `l` starting at index `i` of the disk. For instance,

```
>>> D = disk(10)
>>> D.writeat(2, "blah")

>>> print(D)
__blah____
```

This should be done in one or two short lines.

Please do not duplicate the writing logic in this method: use `D[..] = ...`

(136) Let's prepare the ground for chaining. Add an optional parameter `parent=None` to `__init__`, which is to be stored as attribute.

While we're at it, let's also add an attribute `children`, initialised to `[]`. It will be useful later for the merging operation.

We're now ready to implement snapshotting... Write a method `D.diff()` that takes a snapshot of the current state.

That is to say, if our current global state is $\ldots\Delta c$, it must become $\ldots\Delta s\Delta_{\text{new}}\mathtt{c}$.

Make sure that Δ_{new} knows that its parent is Δ, and that Δ knows that Δ_{new} is among its children.

`diff()` must return the new differencing disk Δ_{new}. Taking a snapshot will basically be done by writing

```
D = D.diff()
```

This method can easily be implemented in three short lines.

(137) We are very close now; all that remains is to implement the chaining logic in `__getitem__`. Go and do so now. Any request the current disk can't immediately satisfy must be passed to its parent. If it has no parent, then the placeholder value `"_"` is returned.

If everything is done right, the following instructions

```
D = disk(10)
D.print("ε Δ0 c (untouched)")
D.writeat(5, "AA")
D.print("ε Δ0 c (written to)")
D = D.diff()
D.print("ε Δ0 s1 Δ1 c (snap just taken)")
D.writeat(4,"BB")
D.print("ε Δ0 s1 Δ1 c (written)")
D = D.diff()
D.print("ε Δ0 s1 Δ1 s2 Δ2 c (snap just taken)")
D.writeat(5,"C")
D.print("ε Δ0 s1 Δ1 s2 Δ2 c (written)")
```

should produce

```
__________ {}                   | ε Δ0 c (untouched)
_____AA___ {5: 'A', 6: 'A'}     | ε Δ0 c (written to)
_____AA___ {}                   | ε Δ0 s1 Δ1 c (snap just taken)
____BBA___ {4: 'B', 5: 'B'}     | ε Δ0 s1 Δ1 c (written)
____BBA___ {}                   | ε Δ0 s1 Δ1 s2 Δ2 c (snap just taken)
____BCA___ {5: 'C'}             | ε Δ0 s1 Δ1 s2 Δ2 c (written)
```

The method should be around three of four lines long at this point, discounting assertions.

(138) Before getting into merging, let's give ourselves a way to check the global status of the current chain. Write a method `D.history()` that returns the list of of its ancestors, in chronological order.

That is to say, our current disk Δ_2 in the chain $\varepsilon\Delta_0\Delta_1\Delta_2$ should return the list `[`Δ_0`,`Δ_1`,`Δ_2`]` for its history. For instance, after executing the instructions of the previous question, we should have:

```
>>> D.history()

[{5: 'A', 6: 'A'}, {4: 'B', 5: 'B'}, {5: 'C'}]
```

Note that the elements you see here are the `repr` of each disk, which we set earlier to that of its internal dictionary.

`history` should be written in two of three lines.

For a nicer visualisation of that history, write a method `D.printhistory()` printing, still in the same context as above, the following:

```
####################
_____AA___ {5: 'A', 6: 'A'}
____BBA___ {4: 'B', 5: 'B'}
____BCA___ {5: 'C'}
####################
```

(139) merging

50 The cheapest DBMS ever

You will need to read Sec. 23.6[p119]: "Pattern matching: ***match..case****". Sec. 22.4.10.4[p94]: "The good stuff: formatted string literals" will come in handy as well.*

So you need a Database Management System, but installing MariaDB or PostgreSQL is too easy for you? You want to implement your own? Good news, you will get to do so in the various Database courses at the end of the third year and during your fourth year.

Before those happy days come, let us ease into it by implementing an exceedingly barebones DBMS accepting a few simple commands. The aim is more to practice a few simple **match/case** statements than to think about databases. You will continue this exercise in the DB courses.

Download our tiny toy database `db.ods` from Celene, and run the command

```
pip install pyexcel-ods3
```

To import the spreadsheet into Python, begin your program with this code:

```
#!/usr/bin/env python3

from pyexcel_ods3 import get_data
dbr = get_data("db.ods") # raw database
```

If all goes well, dbr now contains a dictionary of the form

$$\text{table name (str)} \quad \to \quad \text{table (list of list)}\ .$$

The first line of each table contains the column names. We shall work with the data structure as it is.

(140) Our system must have an interactive command-line interface, and implement a *tiny* subset of SQL. Use the `input` function to implement a prompt, prefixed by `db>`. To interpret the commands in the main loop, you will use `str.split` and a `match/case`.

If a command is unrecognised, the prompt simply shows the list of words that were passed, separated by whitespace.

```
db> some arrant nonsense 10
? ['some', 'arrant', 'nonsense', '10']
```

Unlike true SQL, our language need only accept lowercase versions of the keywords. It is also case sensitive when it comes to table names, column names, etc.

(141) Let us implement the `show tables` command:

```
db> show tables
  * students
  * origins
```

(142) Now, implement a (non-SQL) `view <table name>` command, showing the contents of a table:

```
db> view students
  Name    Age    Python    TL    Origin
+-------+-----+--------+----+--------+
| Toto | 20  | 15      | 5  | L2M     |
| Tata | 21  | 8       | 10 | DUTG    |
| Titi | 20  | 15      | 18 | L3I     |
| Bibi | 20  | 12      | 15 | L2M     |
| Baba | 18  | 15      | 11 | L2M     |
+------+-----+--------+----+--------+

db> view origins
  Origin     OriginName       Tutor
+--------+-----------------+-------+
| L3I     | Licence 3 Info  | No     |
| DUTG    | DUT GEII        | Yes    |
| L2M    | Licence 2 Maths | Maybe |
+--------+-----------------+-------+

db> view InvalidTable
KeyError('InvalidTable')
```

Note that any exception raised during the execution of a command must be caught and displayed by the prompt.

Also note that whatever procedure you use to pretty-print the tables will have to be used for `select` queries as well.

(143) Finally, let use implement a very restricted version of `select <columns> from <table> where <conditions>`. We can of course use `*` to select all columns:

```
db> select * from students
  Name   Age   Python   TL   Origin
+------+-----+--------+----+--------+
| Toto | 20  | 15     | 5  | L2M     |
| Tata | 21  | 8      | 10 | DUTG    |
| Titi | 20  | 15     | 18 | L3I     |
| Bibi | 20  | 12     | 15 | L2M     |
| Baba | 18  | 15     | 11 | L2M     |
+------+-----+--------+----+--------+
```

Otherwise, column names must be separated by commas, without any whitespace. Note that this means our column names must not contain spaces either.

The reason we do not allow spaces is that doing so would make parsing the `select` command with a simple `split + match/case` impossible. The aim of the exercise is not to implement an general parser for SQL, that would be in the scope of the Languages Theory or the Compilation course, not this Python course. We can do quite enough for our immediate purposes by keeping things simple.

For instance, we have:

```
db> select Name,Python from students
  Name   Python
+------+--------+
| Toto | 15     |
| Tata | 8      |
| Titi | 15     |
| Bibi | 12     |
| Baba | 15     |
+------+--------+
```

Finally, we must implement filtering conditions. A condition is of the form `<column name>=<value>` no spaces allowed, and there may be several of them, separated by commas, with, again, no spaces allowed. For instance:

```
db> select Name,Origin from students where Python=15,Age=20
  Name   Origin
+------+---------+
| Toto | L2M     |
| Titi | L3I     |
+------+---------+
```

(144) **(optional, for DB course)** Now we shall extend the power of `select` so that it supports inner joins. We shall use a non-SQL syntax for this. Instead of a standard table name, we can write `<table name1>|<column name>|<table name2>` to join two tables on a common column.

For instance, we have:

```
db> select * from students|Origin|origins
  Name   Age   Python   TL   Origin     OriginName      Tutor
+------+-----+--------+----+---------+-----------------+-------+
| Toto | 20  | 15     | 5  | L2M     | Licence 2 Maths | Maybe |
| Tata | 21  | 8      | 10 | DUTG    | DUT GEII        | Yes   |
| Titi | 20  | 15     | 18 | L3I     | Licence 3 Info  | No    |
| Bibi | 20  | 12     | 15 | L2M     | Licence 2 Maths | Maybe |
| Baba | 18  | 15     | 11 | L2M     | Licence 2 Maths | Maybe |
+------+-----+--------+----+---------+-----------------+-------+
```

Of course we can combine this with filters and column selection:

```
db> select Name,Tutor from students|Origin|origins where Python=15,Age=20
  Name   Tutor
+------+-------+
| Toto | Maybe |
| Titi | No    |
+------+-------+
```

51 Cryptanalyse amusante

This exercise is not focused on any particular Python technique. Use what you have got, and what makes sense.

In this exercise we implement and break the 1553 Vigenère cipher — actually due to Bellaso, and misattributed to Vigenère, following Stigler's law of eponymy.

This polyalphabetic substitution cipher maintained a strong reputation for unbreakability – earning the nickname *le chiffre indéchiffrable*, "the unbreakable cipher" – from its inception until 1863, when Kasisky – a Prussian infantry officer – published a general method to break it.

Sir Charles Babbage – whose 1837 Analytical Engine pioneered the concept of a full, programmable computer – had actually broken it, even in a stronger, autokey version of it, nine years earlier, during the Crimean War, but did not publish his work, as the technique was classified as a military secret.

Even a few decades after that, it was still thought of as unbreakable by many laymen and non-specialist mathematicians — not too surprising, given that it endured for over three centuries.

You should be able to break it in *at most* three TDs :-)

The Enigma machines that famously formed the core of German military communication during the Second World War implemented a *much* stronger, but related, polyalphabetic substitution cipher. With a lot of work and more than a fair bit of luck, the Allies were able to crack it, turning the tide of the War.

51.1 Une grille de chiffrement

Ecrire une procédure pour afficher la grille suivante:

```
    A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
-------------------------------------------------------
A | A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
B | B C D E F G H I J K L M N O P Q R S T U V W X Y Z A
C | C D E F G H I J K L M N O P Q R S T U V W X Y Z A B
D | D E F G H I J K L M N O P Q R S T U V W X Y Z A B C
E | E F G H I J K L M N O P Q R S T U V W X Y Z A B C D
F | F G H I J K L M N O P Q R S T U V W X Y Z A B C D E
G | G H I J K L M N O P Q R S T U V W X Y Z A B C D E F
H | H I J K L M N O P Q R S T U V W X Y Z A B C D E F G
I | I J K L M N O P Q R S T U V W X Y Z A B C D E F G H
J | J K L M N O P Q R S T U V W X Y Z A B C D E F G H I
K | K L M N O P Q R S T U V W X Y Z A B C D E F G H I J
L | L M N O P Q R S T U V W X Y Z A B C D E F G H I J K
M | M N O P Q R S T U V W X Y Z A B C D E F G H I J K L
N | N O P Q R S T U V W X Y Z A B C D E F G H I J K L M
O | O P Q R S T U V W X Y Z A B C D E F G H I J K L M N
P | P Q R S T U V W X Y Z A B C D E F G H I J K L M N O
Q | Q R S T U V W X Y Z A B C D E F G H I J K L M N O P
R | R S T U V W X Y Z A B C D E F G H I J K L M N O P Q
S | S T U V W X Y Z A B C D E F G H I J K L M N O P Q R
T | T U V W X Y Z A B C D E F G H I J K L M N O P Q R S
U | U V W X Y Z A B C D E F G H I J K L M N O P Q R S T
V | V W X Y Z A B C D E F G H I J K L M N O P Q R S T U
W | W X Y Z A B C D E F G H I J K L M N O P Q R S T U V
```

```
X | X Y Z A B C D E F G H I J K L M N O P Q R S T U V W
Y | Y Z A B C D E F G H I J K L M N O P Q R S T U V W X
Z | Z A B C D E F G H I J K L M N O P Q R S T U V W X Y
```

Note aux “astucieux”: on ne se contentera pas de copier-coller depuis le PDF et faire `print`.

En utilisant cette grille, on peut chiffrer des messages. Par exemple "IAMASECRETMESSAGE", crypté par "IAMTHEKEY", devient "QAYTZIMVCBMQLZEQI". L’idée est de répéter la clef jusqu’à ce qu’elle soit de même longueur que le message, ce qui donne

```
IAMASECRETMESSAGE
IAMTHEKEYIAMTHEKE
```

puis de coder lettre à lettre via la grille: grille(I,I) = Q, grille(A,A) = A, grille(M,M) = Y, grille(A,T) = T, etcetera, et on obtient finalement

```
IAMASECRETMESSAGE
IAMTHEKEYIAMTHEKE
QAYTZIMVCBMQLZEQI
```

On note que, contrairement à des chiffres naïfs, où l’on substitue un symbole (par exemple une autre lettre) à une lettre, dans ce cas, une lettre peut être chiffrée de différentes manières selon sa position. Ici, les trois instances de A dans le message sont chiffrées par A, T, et E, respectivement.

Convainquez-vous que le déchiffrement est aussi facile que le chiffrement... quand on a la clef.

51.2 On automatise tout ça

Écrire une fonction pour chiffrer et déchiffrer du texte sans aucune ponctuation (on considérera uniquement les lettres A..Z, que ce soit pour le message ou la clef). La fonction utilisera un argument booléen optionnel pour passer du mode codage au mode décodage: `crypt(msg,key)` chiffre, et `crypt(cmsg,key,True)` déchiffre.

```
msg1 = "THESTUDENTSARENICEANDHARDWORKING"
key1 = "ORARETHEY"
cyp1 = crypt (msg1,key1)

>>> cyp1
'HYEJXNKILHJAIIGPGCOEDYEKKAMFBIEK'

>>> crypt(cyp1, key1 ,True)
'THESTUDENTSARENICEANDHARDWORKING'
```

On ne s’amusera pas à lire ça sur la grille de la question précédente; on ne construira pas de liste de liste ou autre grosse structure. In what follows, we shall need to crypt and decrypt thousands of messages every second to perform cryptanalysis in a reasonable time; the efficiency of this function is therefore of paramount importance.

Je conseille d'utiliser les fonctions `ord` et `chr` pour réduire ça à un peu d'arithmétique. Il faut que cette fonction soit efficace pour la suite.

On pourra s'aider du fait que, sous la représentation 0..25 pour A..Z, et en notant M, K, et C le message, la clef (répétée), et le texte chiffré, on a pour tout indice k

$$C_k = M_k + K_k \mod 26$$

pour le chiffrement, et il s'ensuit donc que

$$M_k = C_k - K_k \mod 26\,,$$

pour le déchiffrement.

51.3 Cryptanalyse. Ça ne rigole plus

C'est la guerre entre l'Empire des Méchants Professeurs (EMP), et les GentilZ Zétudiants RebellZ (ZZZ). Vous êtes un cryptanalyste des ZZZ. De nombreux Bothans sont morts[at] pour intercepter des transmission impériales; les voici (les transmissions, pas les Bothans):

```
2017:
MVUDHIVKSMREKSGMMEKOZXSVZVNMTATSLZTOITYGIROLZWMGFRIMIQCLXECSIXLASULCR

2018:
GVVMFEMMCTKYQBPZBDPYHJYYZIYSOHZRMNIOXMIQPYGBPMLUKVWZRFHAIWECJC

2019:
LXATDEMAFLIDVVFZKZHPBWARJEWXMAHSMZATGWPCJIDIWFSSVTMNAUTVJCYFDVVL

2019 bis:
LXATDEMAFLIDVVFZKZHPBWARJEZEHWQTIINWRMNUWEXMTTPMQZXHQDLIPKZSYMDHREVVXCZPX

2020:
XCZJLWVGIKEYQRBVTQNSPAFMWJEICSIHXNVRPLDIAKENVHFTAMTNGIGIDPWMPRCYUDIBKWHKW
    ...  TEWEXUCIGOCQAVS
```

Damned ! Elles sont chiffrées; et probablement avec des mots de passe différents.

On sait toutefois que l'Empire utilise la grille de chiffrement des questions précédentes, et rédige toujours ses messages en anglais. De plus, ses officiers n'utilisent jamais des clefs de plus de 10 lettres. En revanche on ne sait rien de la façon dont les clefs sont choisies; c'est probablement aussi de l'anglais, mais rien n'est certain.

[at] On ignorera les nouveaux Star Wars made in Disney pour évaluer la *coolness* de mes références de pop-culture. Merci. D'ailleurs *quels* nouveaux Star Wars ? Il n'y a pas de nouveaux Star Wars. That's crazy talk.

Ca fait tout de même

$$\sum_{k=1}^{10} 26^k = 146813779479510 \approx 1.4 \times 10^{14}$$

mots de passe possibles, ce qui est un peu trop pour tous les essayer.

Nous allons malgré tout casser ce chiffre!

La première chose à faire est d'automatiser la reconnaissance de texte en anglais (probablement) valide, par opposition à une suite aléatoire de lettres. Une bonne façon de faire est d'analyser un large corps de textes anglais, en notant la fréquence d'apparition de n-grammes (sous-mots à n lettres). Par exemple, TION est un 4-gramme apparaissant beaucoup plus souvent que AAZS.

Une telle base de données de fréquences permet alors de donner un score à du texte, qui a de bonnes chances d'être plus élevé sur du vrai texte que sur du charabia.

Construire cette base serait un exercice intéressant, mais heureusement le Célène (vaisseau-mère de la Rébellion) dispose déjà du matériel nécessaire à l'analyse des quadgrams anglais (sur la base d'une analyse d'un vaste corps de textes anglais). On l'utilise comme suit — après avoir téléchargé et extrait les fichiers[au]:

```
import ngram_score as ns
fitness = ns.ngram_score()

>>> fitness.score('THISISACOHERENTSENTENCE')
-79.75074906594747
>>> fitness.score('LKFJLSDFJIOJZOJMIOFJNZA')
-176.05856134934515
```

L'idée pour casser le code est la suivante: les effets du mot de passe étant locaux, plus on se rapproche du bon mot de passe, plus on va avoir de bonnes lettres et fragments de mots se rapprochant de l'anglais. En effet, si j'ai deux lettre consécutives correctes sur le mot de passe, alors deux lettres consécutives seront déchiffrées en clair dans le message, et ce autant de fois que le mot de passe est répété pour couvrir le message.

On va donc essayer de muter des mots de passe lettre à lettre, afin de se rapprocher du bon mot de passe. On ne va donc visiter qu'une toute petite fraction des possibilités, guidés par notre heuristique des fréquences de 4-grammes.

Supposons que la clef soit de longueur 5, et partons de AAAAA, par exemple; essayons toutes les lettres à la première position, et choisissons celle qui donne le meilleur score, par exemple SAAAA. Ensuite on passe à la deuxième, et on obtient SUAAA. Au bout du compte, on obtient SUGER. On recommence à la première lettre. Elle ne bouge pas. Puis la seconde. Puis la troisième; ah! celle-là bouge, on a SUPER. Les autres ne changent pas. On fait encore

[au] Those are due to a guy called James Lyons. (Though I'm not sure who made the 4gram database.) Since using somebody else's code and database feels a lot like cheating, we shall make our own later on...

un tour, plus rien ne bouge. SUPER est donc notre optimum local, en partant de AAAAA. On regarde le message correspondant; c'est soit le vrai message, soit du charabia qui se trouve sonner pas mal comme de l'anglais. Rien ne garanti qu'on aura la même chose en partant de BBBBB.

Dans le doute, on essaye de partir des 26 candidats AAAAA à ZZZZZ, et on garde le meilleur résultat, qui est notre candidat pour une clef de longueur 5. Reste à faire la même chose pour toutes les longueurs de clefs possibles (1..10).

On ecrira une fonction `autobreak(<cyphertext>)` qui affiche le meilleur candidat (clef, message, et score) pour chaque intervalle 1..n de longueur de clef possible. Grâce à cela, on déchiffrera le message de l'Empire.

Par exemple,

```
>>> autobreak (cyp1)
          W LCINBROMPLNEMMKTKGSIHCIOOEQJFMIO -215.1410845557442
         XW KCHNARNMOLMELMJTJGRIGCHONEPJEMHO -192.46846034100733
        HEA AUECTNDELAFABEGICCHADRAKDWMYXIXG -182.68686138209173
       XUMC KESHATYGONXYLOUNJICCGESINGADEOSI -165.81846421022996
      WYAIT LAEBERMIDONCIANTICGLHAECREOFTPIM -157.3321135527277
      WYAIT LAEBERMIDONCIANTICGLHAECREOFTPIM -157.3321135527277
    GETYXAR BULLANTCHOLDIRALNEREMSARMDMOVELM -142.23665938186278
    GETYXAR BULLANTCHOLDIRALNEREMSARMDMOVELM -142.23665938186278
  ORARETHEY THESTUDENTSARENICEANDHARDWORKING -112.42451600254101
  ORARETHEY THESTUDENTSARENICEANDHARDWORKING -112.42451600254101
```

Notons que la solution à -142 commence à ressembler fortement à de l'anglais. Si on n'a pas de chance (et on a plus de chance de ne pas avoir de chance quand le message est petit), on peut obtenir par hasard un "message" encore plus anglais que le message réel: considérons

```
>>> c = crypt('THISISATEST','K')
>>> c
'DRSCSCKDOCD'
>>> autobreak (c)
          K THISISATEST -30.663467249176005
          K THISISATEST -30.663467249176005
          K THISISATEST -30.663467249176005
          K THISISATEST -30.663467249176005
      LKKKZ SHISTRATEDS -29.063122550537994
     DZKPMJ ASINGTHEENR -27.47716321489606
    ZOKPMJD EDINGTHEASO -26.043436663213683
   KKKKZVGM THISTHEREST -25.225017010062118
   KKKKZVGM THISTHEREST -25.225017010062118
 KKOLOPRVBW THERENTINGT -25.003927573660018
```

La solution finale, "the renting T", a un meilleur score que le message original, et peut nous faire croire qu'il est question de la location de quelque-chose par un mystérieux monsieur "T". "This, the rest", est aussi un message convainquant. Un indice pour préférer "this is a test" est que quitte à choisir un mot de passe long, on choisit rarement des mots de passe avec des lettres répétées. Le reste est une question d'interprétation.

Évidemment, plus le message est long, moins on risque d'avoir ce genre de soucis.

Note en passant: on pourrait aussi utiliser des algos génétiques sur ce problème. ;-D Les résultats seraient probablement meilleurs, mais ça serait plus compliqué à mettre en place.

51.4 n-gram analysis

It is advised to read Sec. 24.4[p144]: "Dictionaries: class `dict`", and in particular Sec. 24.4.1.3[p153]: "`Counter`, from `collections`", quite carefully before proceeding. Some material from Sec. 26[p181]: "Reading and writing files" will also be necessary.

Han Solo is loosely allied with the ZZZ RebelZ, and shares their need for codebreaking tools, but he goes his own way in the traditional lonesome cowboy fashion, and is far too proud to ask for their help.

Consequently, he needs to decrypt the messages, but has no access to ZZZ's n-gram database and scoring function. He has a bad feeling about this, because that means he must make his own. While he does not have access to a vast corpus of texts on which to train his database, he does have access to a copy of Tosltoï's War and Peace, pilfered from Chewbacca 's nightstand.

He also has access to *you*, and goes to take a nap while you do all the work. Typical. You still have the code you wrote for the ZZZ, but you can't use the ngram database or the `ngram_score` code previously provided. Better get going.

(145) Download War and Peace (`WP.txt`) to your R2D2 unit.[av]

(146) Your first task will be to clean it up for ngram analysis. We only take alphabetic characters into account, so everything else must go.

To prepare for this, write a predicate `isalpha(c)` testing whether `c` is a character of `A..Za..z`.

It must satisfy the following assertions:

```
assert all ( isalpha(chr(n)) == chr(n).isalpha() for n in range(128) )
assert all (not isalpha(chr(n)) for n in range(128,256) )
```

The reason we do not use `str.isalpha` directly is that it accepts accented characters.

(147) Will shall need to iterate on the cleaned text ngram by ngram. Write a function `gramiter(s,n=4)` that returns an iterable over each successive ngram in the string `s`.

Preferably, return a generator.

[av] You'll find it on Celene, as usual.

(To do things optimally, we would accept any iterable; thus we could accept a generator as well, and lazily process ngrams as we read the file. However, I have not written the section presenting the tools for that yet, so for now we'll put the file in a string first and process that in a second time. It's less elegant, but it works.)

In any case, the following assertions must be satisfied:

```
assert tuple(gramiter("ATTACK")) == ('ATTA', 'TTAC', 'TACK')
assert tuple(gramiter("ATTACK",n=1)) == ('A', 'T', 'T', 'A', 'C', 'K')
```

(148) We have all the tools we need to create our database. Write a procedure

```
process(text="WP.txt", out="quads.txt"):
```

which processes the file `text`.

That means loading it up to memory in a string (again, there are more elegant ways, but this works) containing only the alphabetic characters — all in uppercase. You can use `str.upper()` for uppercase conversion.

Then, you can count the number of occurrences of each 4gram in the text, and write them to a file `out`. Each line of that file should be of the format

```
<4GRAM> <nb of occurrences>
```

and the lines should be sorted in order of decreasing occurrences.

The beginning of the output file should therefore look like this:

```
THAT 8285
THER 8187
WITH 6538
DTHE 6295
NTHE 5728
OTHE 5590
...
```

Friendly Tip: `Counters` seem like wonderfully topical things, don't you think? So does their `most_common` method... Just sayin'.

Run `process()` once, creating the database file `out`. Then comment out the call, for loading the database from `out` will be much faster than recomputing it from scratch each time.

(149) Write a function `load_grams(fname="quads.txt")` that returns a counter containing all 4gram/occurrences data in the file.

Define a variable `C = load_grams()`

(150) Define a function `score(s,C=C)` returning the sum of all 4gram occurrences in the string `s`, according to the counter `C`. Since Python does not suffer from integer overflow problems, and we only intend to sum over relatively short sentences, we can affort that computation. Of course, any 4gram that does not appear in the database is interpreted as having `0` occurrence.

We now have a function returning a numerical value which can rightly be expected to increase with the "typicality" of the text — length being equal:

```
>>> score('THISISACOHERENTSENTENCE')
13537
>>> score('BLAHIBLAHBLOBYAAKNOWAHH')
1623
>>> score('LKFJLSDFJIOJZOJMIOFJNZA')
0
```

We are done! Huzzah! Replace your old fitness function by this one, and get codebreaking again. How does it work out for you? *Spoiler alert: not great.*

Note: test on the Empire's messages, and on the provided `cyp1` example.

This fitness function does not seem up to snuff. Why do you think that is?

(151) Let's inject a little maths into this party and come up with a better score function.

We have a database of K $n$grams $g_1, \ldots, g_K$, of respective occurrences $o(g_1), \ldots, o(g_K)$. Let

$$N = \sum_{k=1}^{K} o(g_k)$$

be the total number of 4gram occurrences.

Under the – very, *extremely* questionable – modelling hypothesis that 4grams occurrences are independent in text, the probability of an $n$gram is therefore given by

$$\mathbb{P}(g_k) = \frac{o(g_k)}{N}$$

and the probability of a string s is given by

$$\mathbb{P}(s) = \prod_{g \in \mathtt{ngrams}(s)} \mathbb{P}(g) .$$

However, that would mean that any 4gram that does not appear in our database would instantly set the probability of the sentence to zero. Zero (and one) are *very* strong probabilities — in Laplacian/Bayesian[aw] terms, they mean literally *infinite* evidence

[aw] Following Stigler's law of eponymy, Bayesian probability theory is really due to Laplace.

against or for a proposition. We know our database is woefully incomplete. Therefore we assign a small probability

$$\mathbb{P}_0 \quad = \quad \frac{1}{100N}$$

to any 4gram not appearing in the database.

Alter your score function to return $\mathbb{P}(s)$. You should get something like this:

```
>>> score('THISISACOHERENTSENTENCE')
9.52447211241367e-85
>>> score('BLAHIBLAHBLOBYAAKNOWAHH')
7.907084778567561e-146
>>> score('LKFJLSDFJIOJZOJMIOFJNZA')
9.26233860365592e-169
>>> score('LKFJLSDFJIOJZOJMIOFJNZA'*2)
0.0
>>> score('THISISACOHERENTSENTENCE'*4)
0.0
```

Test the codebreaker with this.

Spoiler alert: disappointment awaits you.

This probability-based approach is not an altogether bad one in principle. There is a trick to implementing it *properly*, however. If you kept your ears open during class, and read the part of this document on floating point numbers, you know that I am more than a tad skittish about them. Here, we are blithely *multiplying* a whole *bunch* of very, *very* tiny floating point numbers.

How trustworthy are those infinitesimal results, really? Given the danger of arithmetic underflow, not very much. As a reminder, the smallest representable floating-point value in Python is

```
>>> sys.float_info.min
2.2250738585072014e-308
```

Mathematically, what will happen to $\mathbb{P}(s)$ as s grows longer, regardless of how "English" that sentence may be? How will it play out with `sys.float_info.min`?

(152) Let us keep the very same method, but use a mathematical trick to alleviate the loss of precision. If only we could *add* numbers, instead of multiplying them, the overall loss of precision would be much less. If only there was a well known mathematical Jedi mind-trick we could use to turn those $\times$ into $+\ldots$

Oh wait, there *is*:

$$\log(xy) \quad = \quad \log x + \log y \ .$$

Let's select base 10 arbitrarily – just so you can compare your numerical values to those of others:

```python
from math import log10 as lg
```

Now let us transform our probabilities into log probabilities: we have

$$\mathbb{L}(g_k) \quad = \quad \lg(\mathbb{P}(g_k)) \quad = \quad \lg\left(\frac{o(g_k)}{N}\right)$$

and the log probability of a string s is given by

$$\mathbb{L}(s) \quad = \quad \lg(\mathbb{P}(s)) \quad = \quad \sum_{g \in \mathrm{ngrams}(s)} \mathbb{L}(g)\ .$$

Let us note in passing that log probabilities have uses far beyond a "mere" numerical computation trick; they are well-known in probability theory and in information theory, as they represent (minus) the information content of an even. As we are now discovering, they are central to natural language processing.

Update your `score` function to use $\mathbb{L}(s)$. You should get something like:

```
>>> score('THISISACOHERENTSENTENCE')
-84.02115908547061
>>> score('BLAHIBLAHBLOBYAAKNOWAHH')
-145.10198360473777
>>> score('LKFJLSDFJIOJZOJMIOFJNZA')
-168.03327934653242
```

Test the codebreaker again. And rejoice! This time, it is supposed to work fairly well, though not quite as well as it did when we had the larger database.

Speculate on *why* that matters for some messages and not others.

(153) To confirm that the difference in codebreaking performance stems from the database and not your implementation of the scoring function, without touching the code, swap your Tolstoï-based database for the previous one, `english_quadgrams.txt`: it is written in exactly the same format.

You should obtain the same codebreaking performance as before. Not only that, but you should obtain exactly the same numerical values for the score, as we now use exactly the same scoring principles as before.

```
>>> score('THISISACOHERENTSENTENCE')
-79.75074906594747
>>> score('BLAHIBLAHBLOBYAAKNOWAHH')
-119.3170079814685
>>> score('LKFJLSDFJIOJZOJMIOFJNZA')
-176.05856134934515
```

If that is not the case, there is something wrong with your code.

(154) This is not a question, but thoughts on how to improve the method.

- ◇ We could extend the database both in size and in breadth of styles of English which it represents. As a test, I used a Python web scrapper to extract the text of the humongous Worm web-serial, written in modern, casual American English, and combined it with War and Peace — Worm is about thrice as long as War and Peace, so this substantially reinforced the database. Judging by the number of occurrences of the most common 4grams, we are still a long way off.

  For comparison, the War and Peace database offers `THAT` as the most common, at 8285 occurrences. The War and Peace + Worm database offers `THER` at 31 921 occurrences, while the original database offers `TION` at a whopping 13 168 375 occurrences; obviously, it must have been made with a very large corpus of text. (I still don't know who is the original source for that file. Anyone finding out, please tell me!)

  The addition of Worm did improve the quality of the partially cracked keys, but not to the point of cracking them completely. Equaling the quality of the larger database would mean processing a few thousand books.

- ◇ We could move up to 5grams. The received wisdom in that domain is that 4grams are a sweet spot beyond which the marginal increase in performance does not justify the significant increase in database size.

- ◇ If improving the database is not realistic, perhaps we can improve the scoring function instead. The formula

  $$\mathbb{P}(s) \quad = \prod_{g \in \mathtt{ngrams}(s)} \mathbb{P}(g)$$

  rests on the assumption of independence of the successive 4grams. I described this assumption as *extremely questionable*, which was a polite way of pointing out that it is, plainly, false. Convenient, yes. A decent approximation, certainly. But false in glaringly obvious ways.

  A better approximation would be a Markov process. Consider ATTACK: the first 4gram is ATTA; having read that, the next is necessarily of the form TTAx, for some x. Thus the probability of any 4gram not of that form being the second one is zero. Literally zero this time, not "a very small probability".

  The probability of TTAx, given that the previous 4gram was ATTA, or, put another way, the probability of the next letter being x given that we just read ATTA, is given by

  $$\mathbb{P}(x \mid \mathrm{ATTA}) \quad = \quad \mathbb{P}(\mathrm{TTA}x \mid \mathrm{ATTA}) \quad = \quad \frac{o(\mathrm{TTA}x)}{\displaystyle\sum_{\mathrm{TTA}y \in D} o(\mathrm{TTA}y)},$$

where D is our database. $\mathbb{P}(s)$ is then obtained by multiplying the successive conditional probabilities, starting from the probability of the first 4gram:

$$\mathbb{P}(s) \;=\; \mathbb{P}(g_0 g_1 \dots g_n) \;=\; \mathbb{P}(g_0)\prod_{k=1}^{n}\mathbb{P}\big(g_k \mid g_{k-1}\big)\,.$$

Of course, we would still require a log probability implementation. I haven't had the time to test that, but I assume that this new version of `score` would be noticeably slower, but much more precise. I would expect (hope?) the codebreaker to succeed on all messages, even using only the War and Peace database.

◇ We could also simply test more keys in the same timeframe by making use of multiple processors. This would offset our more complicated scoring method.

I shall get around to writing a section on that eventually.

52 Be there or be square!

In this section, we focus on the computation of the integer square root `isqrt`$(n) = \lfloor\sqrt{n}\rfloor$, which we shall perform in many different ways, and compare with respect to performance.

Rappel: $\lfloor k \rfloor$ est la partie entière inférieure de k.

52.1 The easy way: `isqrt_builtin`

enforcing consistency of multiple implementations

Write a function `isqrt_builtin(n)` using `math.sqrt` and `math.floor`. It shall serve as a reference implementation.

The following assertion must hold:

```
assert [ isqrt_builtin(n) for n in range(30) ] == \
    [ 0, 1, 1, 1, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3,
      3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 5, 5, 5, 5, 5 ]
```

In the next questions, for every subsequent function `f` computing the integer square root — you will implement `isqrt_hard`, `isqrt_dicho`, and more. . . — the following assertions must hold:

```
for n in range (100):
    assert f(n) == isqrt_builtin(n), n
```

Note that the error message n will enable you to know on which value `f` failed, as it will be displayed in the interactive mode.

Note that you could factorise all those tests, like so:

```
for f in (isqrt_hard, isqrt_dicho,...): # replace by your functions
    for n in range (100):
        assert f(n) == isqrt_builtin(n), (f,n)
```

No other use of `isqrt_builtin` is allowed in the next questions. Nor can you use `math.sqrt`, or `pow(n,0.5)`, or `n ** 0.5`, or `n ** (1/2)`, or any other direct way of computing the root. You must implement the algorithms suggested by the question, **and use integers exclusively**.

52.2 Racine carrée entière, the hard way!

Note: On utilisera la syntaxe `x * x` *pour coder x^2, car c'est plus efficace que* `x ** 2`*.*

Écrire une fonction `isqrt_hard(n)` calculant $\lfloor\sqrt{n}\rfloor$, pour tout entier naturel n.

On utilisera impérativement une boucle `while`, testant les carrés $0^2, 1^2, 2^2, 3^2, \ldots$ successivement, jusqu'à trouver la bonne valeur.

52.3 Racine carrée entière, 20% cooler!!

If I am not there to help you or this takes too much time, skip this section. The important thing is to get to the empirical comparison of your solutions, at the end — it does not change much if you have one fewer version of `isqrt` *to compare.*

Let us compute a closed form[ax] of the sum S_n of the n first odd numbers:

$$\begin{aligned} S_n &= \sum_{k=1}^{n}(2k-1) &= 2\sum_{k=1}^{n}k - \sum_{k=1}^{n}1 \\ && = 2\frac{1}{2}n(n+1) - n \\ && = n(n+1) - n \\ && = n(n+1-1) \\ && = n^2\,. \end{aligned} \tag{52.1}$$

This gives us an idea to compute `isqrt` more efficiently. What if, instead of doing a multiplication at each iteration, we replaced that multiplication by additions — additions are less expensive, in processor time, than multiplications.

The idea is to use the same logic as `isqrt_hard`, except that instead of computing the next square from scratch at each iteration of the main loop, you can take advantage of Equation (52.1) to use the previously computed n^2 to obtain $(n+1)^2$; you just need to increment your previous result by the next odd integer. That integer itself can be maintained from one iteration to the next by adding 2. Thus you replace a multiplication, which can be fairly

[ax] A *closed form* is an expression using only a fixed number of straightfoward operations, such as +, −, × etc, but no $\sum$, $\prod$, $\int$ etc. Put another way, "compute the value of S_n as a function of n", or "solve the sum".

expensive in terms of computation time, by two additions, which *may* be less expensive — further testing will determine how well that works out in practice.

With this in mind, write a function `isqrt_sum(n)` computing $\lfloor\sqrt{n}\rfloor$, behaving fundamentally like `isqrt_hard`, but using the ideas above to be (hopefully) more efficient.

52.4 Racine carrée entière, par dichotomie

dichotomy on `int` has its own logical difficulties
using a recursive subfunction

On est toujours, jusqu'à présent, dans un nombre d'opérations linéaire en la taille de n. Écrivons donc maintenant une fonction `isqrt_dicho(n)` calculant $\lfloor\sqrt{n}\rfloor$ de façon similaire à la version "hard" mais en procédant cette fois par recherche dichotomique au lieu de tester toutes les valeurs. On s'attend dont à un nombre logarithmique d'opérations.

You will write two versions: the first one using **`while`**, and then the second one, `isqrt_dicho_rec(n)`, using recursion. Note that both versions must present the same interface to the user. Unlike in question (35)[p246], where the search interval $[a, b]$ was expected as arguments, you have no room in the function's signature for that.

Tip: *Use a recursive subfunction.*

Both versions must be tested.

52.5 Empirisme forcené: Ultimate Showdown of Ultimate SQRT

performance testing of multiple implementations
recognising the futility of reinventing the wheel and premature optimisation

Utilisez la fonction `timeit` du module éponyme pour tester si les versions par somme et dichotomie sont réellement meilleures (ou pires ?) que la version brutale, et à quel point – on utilisera également, comme base de comparaison, la fonction `isqrt_builtin(n)`. .

On expliquera brièvement mais clairement ce qu'on fait, pourquoi, ce que l'on obtient, ce que ça veut dire, et si ça nous surprend.

Si l'on perçoit une morale à cette exercice, on l'exprimera succinctement.

Pour illustrer les résultats, on produira (au moins) un joli petit graphe montrant le comportement de nos quatre approches en fonction de la taille de n. Le graphe pourra être fait avec `matplotlib`, `gnuplot`, Libre Office, Word, Google Sheets, ce que vous voulez, mais vous en fournirez dans tous les cas une version PDF ou PNG, lisible par tout un chacun.

Note: on utilisera l'interface Python du module `timeit` — par opposition à la ligne de commande — dont la documentation est ici.

Begin by importing the function:

```
from timeit import timeit
```

The `timeit` function executes a small snippet of code a large number of times, and returns the total time taken, in seconds, as a **`float`**. Execution times of small snippets of code are highly variable, depending on system activity; repeating the execution ensures that measures are reasonably precise.

`timeit` has two interfaces; one using strings, and one using *callables* (via lambdas), which is what we shall use.

The "callable" interface expects a nullary function, whose execution is timed, and an optional argument `number`, defaulting to 10^6. Timeit returns the total time taken by the execution of the callable, repeated `number` times.

For instance, let us say we want to time the execution of `isqrt_hard(1000)` and `isqrt_hard(1000000)`:

```
>>> N = 10000                           # repetitions

>>> timeit(lambda: isqrt_hard(1000) , number=N) / N
9.853858899987244e-06                   # average time, in seconds

>>> timeit(lambda: isqrt_hard(1000000) , number=N) / N
0.00029428713969991804
```

53 Prime numbers and sieve of Eratosthenes

In this section, you cannot reuse the results of previous questions unless explicitly mentioned, since the goal is to do the same thing with many different approaches.

more complex comprehension expressions

(155) Écrire un prédicat `isprime(n)` ($\mathbb{N} \to$ `bool`) testant si un entier naturel n est premier, c'est à dire s'il est strictement supérieur à 1 et divisible seulement par 1 et n.

The body must be written in one line of the form **`return`** `<expr>`.

Read Sec. 24.5.3.4[p159]: "Reductions", especially the part about `any` and `all`.

The following assertion must hold:

```
assert [ i for i in range(30) if isprime(i) ] \
       == [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

(156) **Reminder:** Composite numbers are positive integers that are not prime. Put another way, they are formed by the multiplication of two smaller numbers, other than 0 or 1.

With a comprehension, build the set `comp` of composite numbers in $[\![1, n]\!]$, using the following method: use a filter to retain only those integers which have a divisor. Again, do **not** reuse `isprime` for this.

Note that I merely ask for a set, written with a comprehension expression, not for a function returning a set. Assume that `n` is defined; for instance `n = 100`.

The expression will thus be of the form

```
comp = { i for i in range(...) if any(......) }
```

or

```
comp = { i for i in range(...) for j in range(...) if ... }
```

For bonus points, try and write a version for each of those two forms. Might there be a notable difference in efficiency between the two for large values of `n`?

(157) Again, build the set `comp2` of composite numbers in $[\![1, n]\!]$, but this time build the multiples of 2, then the multiples of 3, and so on.

For this question, recall that the multiples of `i` can easily be computed through `range(i, n+1, i)`, thanks to the optional third argument of `range`.

Thus the expression will be of the form

```
{ j for i in range(...) for j in range(...) }
```

(158) *Again*, build the set `comp3` of composite numbers in $[\![1, n]\!]$, but this time by noting that they are the set of numbers of the form $i \times j$, for i, j in suitable ranges.

Thus the expression will be of the form

```
{ i*j for i in range(...) for j in range(...) }
```

The following assertion must hold:

```
assert comp == comp2 == comp3, (comp^comp2, comp^comp3)
```

Note that the error message gives you the symmetric difference between those sets, making debugging easier.

(159) Build the tuple `primes` of prime numbers in $[\![1, n]\!]$. Use `comp`, as defined in previous questions; do **not** use `isprime`, however.

The following assertion must hold:

```
assert primes == tuple( k for k in range(1, n+1) if isprime(k) )
```

54 For me it was a Tuesday...

In any calendar-related question, it is forbidden to use the `datetime` *module or anything similar, unless explicitly required by the question.*

The only permitted use of `datetime` *is to verify your answers through assertions, as shown in question 165[p333].*

In this section, we shall represent dates as triplets of integers y, m, d, for years, months, and days, respectively. Months are represented as integers in $[\![1, 12]\!]$.

In some later questions, it might be useful to note that, with this representation, the order between two dates can be tested naturally through

```
(y,m,d) < (Y,M,D)
```

thanks to the lexicographical order on tuples (cf. Section 22.4.9[p92] and Equation 24.1[p129]).

54.1 Suis-je bissextile ?

Dans le calendrier grégorien, qui a pris effet le 15 octobre 1582, une année bissextile est une année qui est divisible par 4 mais pas par 100, ou alors qui est divisible par 400.

Nommer, documenter, et écrire une fonction pour tester si une année est bissextile. Comme toujours, on réfléchira aux conditions d'utilisation...

Obviously, this must be a predicate (the return value is a Boolean). Try to write it in a single line of the form

return <Boolean expression>

If you really feel the need to use an **if**, by all means do so at first, but then spend time rewriting it into the single-expression form.

54.2 Le mois le plus long

On écrira (nommera, documentera, testera, etcetera) une fonction donnant le nombre de jours dans un mois. On rappelle que février a 28 jours dans une année normale et 29 dans une année bissextile.

Tip: you can use the **in**/tuple syntax (cf. Sec. 22.6.2[p100]: "**in** and **is**"):

```
x in (1,3,7,10)
```

replaces

```
elif x == 1:.. elif x == 3.. etc
```

A **match/case** can also serve:

```
match x:
    case 1|3|7|10: ...
```

Depending on how you choose to write the function, an expression-**if** / ternary operator (cf. Sec. 23.2[p112]: "Conditional expression: `.. if .. else ..` ternary operator") may also be convenient.

54.3 Aide aux jours invalides

Bien que toute date soit représentable par un triplet d'entiers y, m, d, tout triplet ne représente pas une date valide. Par exemple 2015, 29, 2 et 2015, 2, 29 sont toutes deux invalides, pour des raisons différentes.

On écrira une fonction `is_valid_date` pour tester cela.

There again, this is a simple predicate, and should be written in a single line of the form

return <Boolean expression>

Now you can — nay, *must* — use this convenient predicate in every function taking dates as inputs, as a precondition assertion.

54.4 Comptons les jours, approximativement

On veut compter les jours entre deux dates.

Écrire une fonction `days_between_approx(y,m,d,Y,M,D)` renvoyant le nombre (approximatif) de jours entre les deux dates représentées dans les paramètres.

Elle doit donner un résultat cohérent quelque-soit l'ordre dans lequel les dates sont données. Il est utile de penser à cela comme une *difference* entre deux dates.

On ignorera (le temps de cet exercice) les questions d'années bissextiles: on considérera qu'un an a 365.2425 jours en moyenne, et que les mois, au nombre de 12 par ans, ont, aussi en moyenne et approximativement, le même nombre de jours.

54.5 `days_between`, exact version

We shall now perform an exact computation. `days_between(y,m,d,Y,M,D)` has the same parameters and meaning as `days_between_approx` , but must return the *exact* number of days.

This question is *quite* difficult, and it is advised to spend some time on paper breaking the difficulty into subproblems before starting to code. Do not hesitate to define additional helper functions for clearly-identified sub-problems.

The following assertions must be satisfied:

```
assert (days_between(1985,10,21, 1985,10,21) ==      0)
assert (days_between(1985,10,20, 1985,10,21) ==      1)
assert (days_between(1985,10,21, 1985,10,20) ==     -1)
assert (days_between(1985,10,21, 2017,9,19)  ==  11656)
assert (days_between(2017,9,19,  1985,10,21) == -11656)
assert (days_between(1999,12, 5, 2000,3,1)   ==     87)
```

54.6 weekday, the hard way

Sachant que le premier janvier 1900 était un lundi, écrire, à l'aide de la fonction `days_between`, une fonction `weekday(y,m,d)` permettant de déterminer le jour de la semaine de n'importe quelle date du calendrier grégorien — y compris avant 1900.

On utilisera les entiers suivants pour représenter les jours de la semaine:

Dimanche	Lundi	Mardi	Mercredi	Jeudi	Vendredi	Samedi
0	1	2	3	4	5	6

Note: la fonction doit bien renvoyer ces entiers, et non imprimer les noms des jours.

Here are a few tests that your function must pass:

```
assert weekday(1900,1,1)   == 1
assert weekday(1985,10,21) == 1
assert weekday(2017,9,19)  == 2
assert weekday(1899,12,31) == 0
assert weekday(1700,1,1)   == 5
assert weekday(2019,9,14)  == 6
```

54.7 Impression calendrier

À l'aide de la fonction précédente, on écrira une procédure `cal` imitant le comportement du programme `cal` d'Unix (qui, évidemment, affiche un calendrier). En particulier, un appel à `cal(2018,9)` doit imprimer ceci:

```
  Septembre 2018
di lu ma me je ve sa
                   1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30
```

54.8 Merci, Delambre !

La formule de Delambre est une manière plus directe de calculer le jour de la semaine. Une date y, m, d correspond, selon cette formule, au jour de la semaine

$$K = d + \overline{m} + \left\lfloor \frac{21}{4} y'' \right\rfloor + \left\lfloor \frac{5}{4} y' \right\rfloor + 2 \mod 7$$

où

$$y'' = \left\lfloor \frac{y}{100} \right\rfloor$$

est la partie séculaire de l'année, et

$$y' = y \mod 100$$

en est l'année dans le siècle courant; de plus, le code du mois $\overline{m}$ est donné par

m	1	2	3	4	5	6	7	8	9	10	11	12
$\overline{m}$, année normale	4	0	0	3	5	1	3	6	2	4	0	2
$\overline{m}$, année bissextile	**3**	**6**	0	3	5	1	3	6	2	4	0	2

Par exemple, pour le 21 octobre 1985, on a

$$\begin{aligned} K &= 21 + 4 + \left\lfloor \frac{21}{4} 19 \right\rfloor + \left\lfloor \frac{5}{4} 85 \right\rfloor + 2 \mod 7 \\ &= 232 \mod 7 \\ &= 1 = \text{lundi}\,, \end{aligned}$$

ce qui est correct.

Implement a function `weekday_delambre`, as a replacement for `weekday`.

Vérifier que les deux méthodes implémentées pour calculer le jour de la semaine donnent bien les mêmes résultats pour tous les jours de 1900 à 2100.

Adaptez `cal` de manière à ce qu'elle accepte, comme argument optionnel, la fonction déterminant le jour de la semaine. Un appel à cal prendra donc la forme, par exemple:

```
cal(1985,10, weekday=weekday)
```

ou

```
cal(1985,10, weekday=weekday_delambre),
```

54.9 Approximating the approximation error

(160) Vérifiez qu'entre le 21 octobre 1985 et aujourd'hui et, l'erreur encourue par l'usage d'approximations dans la fonction `days_between_approx` est de l'ordre de 0.01%.

By this I mean that the following must hold:

```
def approxrat(*p):
    ex = days_between(*p)
    ap = days_between_approx(*p)
    #print(ex,ap,ap/ex)
    return ap/ex

assert isalmost (approxrat(1985,10,21,2020,9,19) , 1 , 0.0001)
```

Of course, `isalmost` is the function defined in question 21[p237], which you can write again or import.

(161) Verify that, between 1900 and 2100, the error is of the order of 0.0005%.

That is to say, it must hold that

```
assert isalmost (approxrat(1800,1,1, 2100,1,1)   , 1 , 0.000005)
```

(162) En y réfléchissant, proposez et testez un intervalle pour lequel l'erreur due à l'approximation sera beaucoup plus importante.

In fact, you can – and must – find two dates so that the following holds:

```
assert approxrat(....) > 1.7
```

54.10 Effects of approximations on `weekday`

A question in this section requires some understanding of Sec. 29[p208]: "Iterables, iterators, and generators".

Sec. 24.6[p164]: "Packing and unpacking" can also be quite useful on occasion.

Your `weekday` *function must imperatively be correct for that part.*

Since the approximation is so good, did we really need to go to the trouble of writing the exact `days_between` function, or could we just have used the approximation `days_between_approx` instead?

(163) Write a function `weekday_approx`, similar to `weekday` but using `days_between_approx` instead of `days_between`, and rounding the result in an integer. See whether it fails any assertions which its exact counterpart satisfies.

Spoiler alert: it does.

(164) Let us quantify the degree to which the approximation fails to yield the correct day by computing the ratio of weekdays correctly computed through approximation over a few centuries.

Write a generator function `daysgen(y,m,d,Y,M,D)` generating all successive days – as triplets – starting from `y,m,d` and ending just before `Y,M,D` — the end point is excluded.

The following assertions must pass:

```
assert next(daysgen(1899,12,31,1900,1,4)) == (1899, 12, 31)

assert  list(daysgen(1899,12,31,1900,1,4)) == \
    [(1899, 12, 31), (1900, 1, 1), (1900, 1, 2), (1900, 1, 3)]

assert list(daysgen(2020,2,28,2020,3,2)) == \
    [(2020, 2, 28), (2020, 2, 29), (2020, 3, 1)]

assert list(daysgen(2019,2,28,2019,3,2)) == \
    [(2019, 2, 28), (2019, 3, 1)]

assert sum(1 for _ in daysgen(1985,10,21, 2017,9,19)) == 11656
```

(165) Check that the following assertion holds, that checks the correctness of `weekday` quite exhaustively:

```
from datetime import date

assert all( (date(*t).weekday()+1)%7 == weekday(*t)
            for t in daysgen(1800,1,1, 2100,1,1) )
```

For once you can comment this assertion out after verifying it, as it might take a second or two to execute.

(166) Extend `daysgen` to accept a seventh optional argument, defaulting to `False` and which, if `True`, causes the function to produce not just date triplets `t`, but couples `t, weekday(t)`. For performance reasons, this should be done with only one invocation of `weekday`, regardless of the number of days generated.

All previous assertions must remain satisfied, along with the following:

```
assert list(daysgen(1899,12,31,1900,1,4,True)) == \
    [((1899, 12, 31), 0), ((1900, 1, 1), 1),
     ((1900, 1, 2), 2), ((1900, 1, 3), 3)]

assert list(daysgen(1899,12,31,1905,1,4,True)) == \
       [ (t,weekday(*t)) for t in daysgen(1899,12,31,1905,1,4) ]
```

(167) Verify that the approximation computes about 49% of weekdays correctly in the interval `1985,10,21, 2017,9,19` and about 35% in the interval `1800,1,1, 2100,1,1`.

Those percentages may vary greatly depending on the way in which you choose to round `days_between_approx`; for instance it may be 38% instead of 49%, and so on. So long as you find something *roughly* in the same ballpark, it's okay.

The assertions that must hold are thus of the form

```
def approxdayrat(*p):
    N = days_between(*p)
    n = sum( 1 for (t,d) in daysgen(*p,True) if d == weekday_approx(*t) )
    #print(n,N,n/N)
    return n/N

assert isalmost( approxdayrat(1985,10,21, 2017,9,19), .49, .001 )
assert isalmost( approxdayrat(1800,1,1, 2100,1,1),    .35, .01 )
```

(168) If you like, write a concluding haiku about the treacherousness of floating-point approximations, even slight, when applied to integral, exact computations.

55 Dichotomie

On veut réaliser une recherche de zéro d'une fonction continue f sur l'intervalle $[a, b]$, i.e. résoudre $f(x) = 0$. On suppose que f change de signe entre a et b.

L'étudiant Toto propose le code suivant pour résoudre le problème par dichotomie:

```
def sign(x):
    return 1 if x >= 0 else -1

def di(f,a,b):
    m = (a+b)/2
    if f(m) = 0:
        return m
    if sign(a) == sign(m):
        return di(f,m,b)
    else:
        return di(f,a,m)
```

Ce code contient (au minimum) trois erreurs: une d'ordre syntaxique (sans se prononcer quant à savoir si Python la détecte en tant que SyntaxError), une d'ordre numérique, et une d'ordre logique.

(169) Quelles sont-elles ? On expliquera le problème succinctement (une ligne max par erreur).

(170) Proposer un code corrigé pour `di`.

(171) Le `if` dans `sign` et ceux dans `di` jouent-ils le même rôle syntactique pour Python ? Le(s)quel(s) ? (Réponse en deux lignes au plus).

56 En sommes. . .

Nous allons écrire différentes versions de la fonction sum, déjà prédéfinie en Python. On a

$$\texttt{sum}([e_1, \dots, e_n]) \quad = \quad \sum_{k=1}^{n} e_k \quad ,$$

et la fonction s'applique en fait à n'importe quel type d'itérable fini; une propriété que l'on souhaiterait (optionellement) préserver dans nos implémentations.

(172) sum_while: on utilisera une boucle **while**.

(173) sum_for_range: on utilisera une boucle **for** utilisant un **range**.

(174) sum_for: on utilisera une boucle **for** n'utilisant pas de **range**.

(175) Ces implémentations sont-elles équivalentes ? Pourquoi ? (Donner l'argument en une ligne.)

(176) Donnez la sortie de Python lors de l'exécution du bloc de code suivant:

```
l = list(range(1,6))
s = set(l)

print(l, sum_while(l), sum_for_range(l), sum_for(l))
print(s, sum_while(s), sum_for_range(s), sum_for(s))
```

(177) Définir une fonction reduce, applicable à tout itérable fini $[e_1, \dots, e_n]$, telle que

$$\texttt{reduce}([e_1, \dots, e_n], e_0, f) \quad = \quad f(\cdots f(f(e_0, e_1), e_2), \dots, e_n)$$

(178) Compléter le code suivant pour obtenir une implémentation de sum en une ligne.

```
def sum_reduce(l):
    return reduce(## COMPLETER ##)
```

On rappelle l'existence de la construction

$$\texttt{lambda}\ x_1, \dots, x_n : f(x_1, \dots, x_n)$$

en Python pour coder une fonction "anonyme" de paramètres $x_1, \dots, x_n$, renvoyant $f(x_1, \dots, x_n)$ – où $f(x_1, \dots, x_n)$ peut être n'importe quelle expression Python utilisant (ou pas) les variables $x_1, \dots, x_n$. Cette construction joue le rôle syntaxique d'une expression.

(179) Implémenter en une ligne (sans compter l'entête **def** union(l):) une fonction union réalisant l'union des ensembles contenus dans un itérable fini:

$$\texttt{union}([S_1, \dots, S_n]) \quad = \quad \bigcup_{k=1}^{n} S_k \quad .$$

(180) Donner la sortie de Python pour le code:

```
print(union([set('abc'), set('baba'), set('coucou')]))
```

57 Enter the Matrix: find your paths!

Note:

I gave this exercise in a second-sitting examination, at the end of the year. Consequently I made use, for flavour, of notions of Graph Theory which are only tackled during the second semester. Those notions are neither advanced nor strictly necessary to answer the questions, but the exercise will undoubtedly be more fun for you if you understand what concrete problem we are solving.

I would therefore advise you to take a cursory look at the notions of "directed graph" and "adjacency matrix" before proceeding with the questions.

Soit par exemple la matrice d'adjacence suivante, entendue pour un graphe de nœuds A, B, C:

$$\mathbf{M} = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Nous allons jouer à calculer tous les chemins de longueur n sur des graphes. J'espère que cela vous évoque des souvenirs...

It is advised to recall the existence of the builtin `sum` function, as well as that of the `union` function defined in question 179[p335]. Both might come in handy in this exercise.

(181) Définir une variable Python `M` telle que `M[i][j]`= $\mathbf{M}_{ij}$. De plus, chacune des cases de `M` doit être mutable.

(182) Définir une procédure `mprint` pour afficher une matrice; par exemple

```
>>> mprint(M)
[0, 1, 1]
[1, 0, 1]
[0, 1, 0]
```

(183) Définir une fonction `check_squares` prenant deux matrices carrées (de même dimensions $n \times n$) en argument, et renvoyant n. Si les matrices ne satisfont pas cette propriété, `check_squares` doit provoquer une `AssertionError`.

(184) Compléter le code suivant afin de réaliser une fonction `mmul` permettant de multiplier deux matrices carrées contenant des valeurs numériques.

```
def mmul(A,B):
    n = check_squares(A,B)
    res = ## completer ##
        ## completer ##
        ## completer ##
            res[i][j] = ## completer ##
    return res
```

On rappelle que le produit matriciel est défini par

$$[\mathbf{AB}]_{ij} \quad = \quad \sum_{k=1}^{n} \mathbf{A}_{ik}\mathbf{B}_{kj} \; .$$

De plus, on ne pourra pas rajouter des lignes de code en le complétant. Par là j'entends que chaque instance de `## completer ##` doit être remplacée par du code qui tient naturellement en une ligne.

(185) Donner une fonction `letter` qui à un nombre de $0..25$ associe la lettre de $A..Z$ correspondante. (Une ligne, hors entête.) On définit pour la suite $\Sigma = \{A, \ldots, Z\}$.

(186) Donner une procédure `to_path_mat` qui transforme sur place une matrice d'adjacence $\mathbf{M}$ classique en une matrice $\mathbf{P}$ contenant des ensembles de mots telle que $\mathbf{P}_{ij} = \varnothing$ si $\mathbf{M}_{ij} = 0$ et $\{\texttt{letter}(i)\texttt{letter}(j)\}$ sinon. Dans le cas de notre $\mathbf{M}$, on a

$$\mathbf{P} = \begin{bmatrix} \varnothing & \{AB\} & \{AC\} \\ \{BA\} & \varnothing & \{BC\} \\ \varnothing & \{CB\} & \varnothing \end{bmatrix}$$

(187) Nous appelons ceci des *matrices de chemins*, pour lesquelles nous définissons la *concaténation de chemins*, notée $\odot$:

$$[\mathbf{A} \odot \mathbf{B}]_{ij} \quad = \quad \bigcup_{k=1}^{n} \mathbf{A}_{ik} \odot \mathbf{B}_{kj} \; ,$$

où $\odot$ est définie sur les langages et les mots (chemins) comme suit:

$$L \odot M = \{ l \odot m \mid l, m \in L, M \} \qquad ua \odot av = uav, \; \forall a \in \Sigma, \; u, v \in \Sigma^*$$

Écrire en deux lignes une fonction `pconcat` qui réalise $\odot$ sur les mots. Elle doit déclencher une `AssertionError` si les chemins ne sont pas compatibles.

(188) Écrire en une ligne une fonction `psetconcat` qui réalise $\odot$ sur les langages.

(189) Écrire une fonction `pmul` qui réalise $\odot$ sur les matrices. Elle doit comporter le même nombre de lignes que `mmul`.

58 Générateur de nombres premiers et autres

(190) Ecrire une fonction `allints(n=0,step=1)` renvoyant un générateur pour $\mathbb{N}$ par défaut et pour les entiers $\{\, n + \mathit{step} \times k \mid k \in \mathbb{N} \,\}$ en général.

On n'utilisera pas la fonction `count` *du module* `itertools`*, car on est en train de la redéfinir :-P*

(191) Ecrire une procédure `testgen(g,n)` imprimant la liste des n premiers éléments du générateur g. On doit avoir par exemple:

```
>>> testgen(allints(4,3),10)
[4, 7, 10, 13, 16, 19, 22, 25, 28, 31]
```

On supposera que g contient assez d'éléments.

(192) Écrire un prédicat `isprime(n)` ($\mathbb{N} \to$ `bool`) testant si un entier naturel n est premier, c'est à dire s'il est strictement supérieur à 1 et divisible seulement par 1 et n. (On ne demande pas de documentation ni d'`assert` dans cet examen.)

(193) Écrire un générateur produisant tous les nombres premiers supérieurs ou égaux à m. ($m \in \mathbb{N}$).

Part V

DIU EIL: Récursivité

Cette partie sert de support pour le cours de *Récursivité* du Diplôme Inter-Universitaire Enseigner l'Informatique au Lycée (DIU EIL) que j'assure à Orléans (2019–2021). The introduction on terminology also serves during the "Mise à niveau mathématique: Induction" classes for third-year apprentices.

Étudiants INSA CVL:

Le fait que vous n'en soyez pas les destinataires principaux ne doit pas vous vous empêcher d'aborder ces exercices. Certains d'entre eux — par exemple sur la mémoisation de la suite de Fibonacci, ou les relations de récurrence linéaires — sont d'ailleurs tombés en examen.

Étudiants DIU EIL:

Les exercices marqués par ♠ seront traités en séance, en mode TP. Si vous êtes en avance, vous pouvez donc passer à l'exercice marqué suivant.

Ceux marqués par ♣ seront survolés en séance, sous la forme d'un cours magistral.

Le reste peut servir de base au travail personnel, à plus long terme, et pourra être abordé en séance le temps le permettant.

59 A Point on Terminology

> *"To understand recursion, one must first understand recursion."*
>
> — Some wise guy.

The topic is *recursion*. There are two other, closely related words which we shall encounter quite often when exploring the topic: *induction*, and *recurrence*.

I find it helpful to begin by clarifying their meaning; or at least making a good attempt at that. Unfortunately, while the general idea of what those terms mean is quite easy to grasp, their use is not always consistent across all authors. There are also slight differences between the English and French use of the terms, specifically for *recurrence*.

Thus, the definitions I give below are not to be construed as universal or authoritative; they merely represent *my* best attempt at putting the concepts into neat boxes that fit *most* of the literature that *I* encountered. Your mileage may vary.

59.1 Recursion: self-reference, no strings attached

Recursion is the most general term. Anything that is defined with reference to itself, any function that calls itself, directly or indirectly, is recursive. There are no restrictions on what those self-references do, no demand for well-founded orders or base cases or termination. The resulting object may not even be well-defined. Even when it is, it may be very "hairy" and difficult to deal with.

59.2 Induction: well-founded structured recursion

Induction is a much more selective subset of recursion, applied to **(1)** recursive definitions of sets of objects (or *types*), which are then said to have inductive structure, **(2)** definitions of functions upon said structure, and **(3)** proofs — usually concerning the functions in question — that rely on said structure.

59.2.1 Defining Inductive Types

The first class of inductive definition is a recursive definition whereby, from given atomic objects, more complex objects are formed.

For instance, let us define the set E of arithmetic expressions by saying that any number is an arithmetic expression, and that any two such expressions, separated by $+$, form an arithmetic expression. (Use your imagination for other operators.)

We can formalise that by saying that E is the smallest set, with respect to inclusion, such that

$$\mathbb{R} \subseteq E \quad \text{and} \quad \varphi, \psi \in E \;\Rightarrow\; \varphi + \psi \in E\,.$$

Note that $+$ here is just a symbol; we are working with the syntax of the expression. Giving them semantics is another task.

We would instead write a formal grammar:

$$E \;\to\; \mathbb{R} \;\mid\; E + E$$

With $\mathbb{R}$ a non-terminal coding real numbers. Other communities favour a "deduction rule"-like syntax:

$$\frac{r \in \mathbb{R}}{r \in E} \qquad \frac{x \in E \qquad y \in E}{x + y \in E}\,.$$

Note that induction must proceed bottom-up[ay], constructing more and more complex objects.

Of particular interest for us is the inductive definition of integers, often referred to as the Peano definition:

$$\frac{}{0 \in \mathbb{N}} \quad \text{and} \quad \frac{n \in \mathbb{N}}{n + 1 \in \mathbb{N}}\,.$$

[ay] *morally*, at least; deduction rules build terms top-to-bottom, on the paper.

Here $n + 1$ is not to be understood as the application of some binary operator $+$, but only as a syntax for "the successor of n". Our usual notations are shortcuts:

$$\begin{aligned}1 &= 0 + 1\\ 2 &= (0 + 1) + 1\\ 3 &= ((0 + 1) + 1) + 1\\ \dots\end{aligned}$$

To avoid any confusion, in a very low-level discussion of Peano integers we shall often use

$$\frac{}{0 \in \mathbb{N}} \quad \text{and} \quad \frac{n \in \mathbb{N}}{S(n) \in \mathbb{N}}$$

instead, with $S(n)$ standing for "the successor of n". Thus we have:

$$\begin{aligned}1 &= S(0)\\ 2 &= S(S(0))\\ 3 &= S(S(S(0)))\\ \dots\end{aligned}$$

Then, using this unambiguous syntax, we can define a binary operator $+$ and later show that $n + 1 = 1 + n = S(n)$, as a theorem, and henceforth the syntactic distinction can be dropped without fear.

0 and S, or 0 and $+1$, are referred to as *constructors*, because that is what they do: they construct new values of the type. $0()$ is a nullary, or atomic, constructor, it depends on no previously constructed values and constructs... itself, essentially. $S(\cdot)$ is a unary constructor: it takes a existing value and constructs a new one, more complex. The 0-rule is an *axiom*, and the S-rule is an *inductive rule*.

A *derivation* is an application of several rules generating a value:

$$\cfrac{\cfrac{\cfrac{\cfrac{}{0 \in \mathbb{N}}}{S(0) \in \mathbb{N}}}{S(S(0)) \in \mathbb{N}}}{\begin{array}{c}\vdots\\ S(\cdots S(0)\cdots) \in \mathbb{N}\end{array}}$$

Put another way, it is a deduction of the fact that, say, $S(S(0)) \in \mathbb{N}$, in the inference system defined by the construction rules.

There are many other interesting inductive types. Consider the type $\alpha\ell$ of (linked) lists of elements of type α:

$$\frac{}{[] \in \alpha\ell} \quad \text{and} \quad \frac{a \in \alpha \qquad l \in \alpha\ell}{a : l \in \alpha\ell}\,.$$

Through longstanding tradition, staring with LISP, I believe, the empty list “[]” is often called *Nil*, and the list constructor “:” is written *Cons*, because it is the seminal constructor from which the line of thinking and the terminology presented in these pages are generalised.

Let us build an interesting list: $[1, 2, 3]$:

$$\dfrac{1 \in \mathbb{N} \qquad \dfrac{2 \in \mathbb{N} \qquad \dfrac{3 \in \mathbb{N} \qquad \dfrac{}{[\,] \in \mathbb{N}\ell}}{3 : [\,] \in \mathbb{N}\ell}}{2 : 3 : [\,] \in \mathbb{N}\ell}}{1 : 2 : 3 : [\,] \in \mathbb{N}\ell}$$

Here we have employed the usual notations and taken the existence of the integers as hypotheses, but we can fully develop that proof tree using the rules of both types:

$$\dfrac{\dfrac{\dfrac{}{0 \in \mathbb{N}}}{S(0) \in \mathbb{N}} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{0 \in \mathbb{N}}}{S(0) \in \mathbb{N}}}{S(S(0)) \in \mathbb{N}} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{0 \in \mathbb{N}}}{S(0) \in \mathbb{N}}}{S(S(0)) \in \mathbb{N}}}{S(S(S(0))) \in \mathbb{N}} \qquad \dfrac{}{[\,] \in \mathbb{N}\ell}}{S(S(S(0))) : [\,] \in \mathbb{N}\ell}}{S(S(0)) : S(S(S(0))) : [\,] \in \mathbb{N}\ell}}{S(0) : S(S(0)) : S(S(S(0))) : [\,] \in \mathbb{N}\ell}$$

Interestingly, that proof tree is, visually, surprisingly evocative of a Spanish Galleon, seen in profile. Or maybe I'm just tired.

The type of lists $\alpha\ell$ can easily be generalised to the type $\alpha\tau$ of binary trees with nodes and leaves of type α:

$$\dfrac{a \in \alpha}{a \in \alpha\tau} \quad \text{and} \quad \dfrac{a \in \alpha \qquad t_1, t_2 \in \alpha\tau}{a(t_1, t_2) \in \alpha\tau} \,.$$

For instance, we build:

$$\dfrac{4 \in \mathbb{N} \qquad \dfrac{3 \in \mathbb{N} \qquad \dfrac{1 \in \mathbb{N}}{1 \in \mathbb{N}\tau} \qquad \dfrac{2 \in \mathbb{N}}{2 \in \mathbb{N}\tau}}{3(1, 2) \in \mathbb{N}\tau} \qquad \dfrac{5 \in \mathbb{N}}{5 \in \mathbb{N}\tau}}{4\big(3(1, 2), 5\big) \in \mathbb{N}\tau}$$

59.2.2 Defining Functions and Operators on Inductive Types

Now that we have *inductive types*, what can we do with them? We can define functions acting on objects of those types; they will follow the rules though which those objects have been constructed, and break them down, recursively dealing with the more primitive sub-objects.

Let us assign a semantics to our arithmetic expressions:

$$\forall r \in \mathbb{R},\ [\![r]\!] = r, \qquad \forall e, e' \in E,\ [\![e + e']\!] = [\![e]\!] + [\![e']\!] \,.$$

Less trivially, let us equip our Peano integers with an addition. To avoid notational confusion between addition and successor, we shall write $+(x, y)$ instead of the addition $x + y$, and $S(x)$ instead of the successor $x + 1$:

$$\begin{aligned} +(x, 0) &= x \\ +(x, S(y)) &= S(+(x, y))\,. \end{aligned}$$

This is not the most trivial inductive definition on Earth, because we have two parameters. Fortunately, in that case, using the inductive structure of either one of them, leaving the other alone, suffices.

As an example on lists, let us define a function $\Sigma : \mathbb{R}\ell \to \mathbb{R}$ that sums the elements of a list of real values:

$$\Sigma[\,] = 0, \qquad \Sigma(r : l) = r + \Sigma l\,.$$

Finally, for trees, let us define $\Sigma : \mathbb{R}\tau \to \mathbb{R}$ that sums all the nodes of a tree:

$$\Sigma a = a, \qquad \Sigma\big(a(t_1, t_2)\big) = a + \Sigma t_1 + \Sigma t_2\,.$$

All those belong to the second type of inductive definitions: functions acting on inductive types, defined along the inductive structure. Note that those definitions are top-down instead of bottom-up: you take an existing structure and break it down into smaller ones, eventually finding an atom and stopping.

59.2.3 Implementing Such Types and Functions

There are languages that are particularly well-suited to the manipulation of inductive types and functions, such as OCaml and Haskell. Unfortunately, we use Python in this course.

Fortunately, however, since version 3.10, Python supports structural pattern-matching: see Sec. 23.6[p119]: "Pattern matching: **match..case**" and Sec. 28[p202]: "Advanced structural pattern **match**ing". Following the principles outlined in those sections, implementing those types and functions is fairly straightforward:

```python
class Zero: pass
Z = Zero()

@dataclass
class S:
    i: object

def plus(n,m):
    match n,m:
        case n, Zero() : return n
        case n, S(m)   : return S(plus(n,m))
```

59.2.4 Proving Stuff on Inductive Types

The third use of induction is in proofs that rely on inductive structures. Just as the construction rules of the induction definition of types dictates how functions must be built, so do they enforce the structures of proofs. For Peano integers, built by the rules

$$\frac{}{0 \in \mathbb{N}} \quad \text{and} \quad \frac{n \in \mathbb{N}}{S(n) \in \mathbb{N}},$$

we have the following inductive proof pattern, for any property $P(n)$ of integers $n \in \mathbb{N}$:

$$\frac{P(0) \qquad \forall n \in \mathbb{N},\ P(n) \Rightarrow P(S(n))}{\forall n \in \mathbb{N},\ P(n)}. \tag{59.1}$$

This theorem is not arbitrary; it follows mechanically from the inductive definition of $\mathbb{N}$, and similar theorems can be derived for any other inductive type. For instance, it holds that for every property P of α-lists, built by rules

$$\frac{}{[] \in \alpha\ell} \quad \text{and} \quad \frac{a \in \alpha \qquad l \in \alpha\ell}{a : l \in \alpha\ell},$$

we have the inductive proof pattern

$$\frac{P([]) \qquad \forall a \in \alpha,\ \forall l \in \alpha\ell,\ P(l) \Rightarrow P(a : l)}{\forall l \in \alpha\ell,\ P(l)}. \tag{59.2}$$

For binary trees $\alpha\tau$, defined by

$$\frac{a \in \alpha}{a \in \alpha\tau} \quad \text{and} \quad \frac{a \in \alpha \qquad t_1, t_2 \in \alpha\tau}{a(t_1, t_2) \in \alpha\tau}.$$

we have

$$\frac{\forall a \in \alpha,\ P(a) \qquad \forall a \in \alpha, \forall t_1, t_2 \in \alpha\tau,\ P(t_1) \wedge P(t_2) \implies P\big(a(t_1, t_2)\big)}{\forall t \in \alpha\tau,\ P(t)}. \tag{59.3}$$

59.2.5 General Forms of Induction

We have seen three inductive types, functions defined on them, and the corresponding proof patterns. I have said that the proof patterns could be mechanically derived from the inductive rules, but not how.

Let us now take a more abstract view, which will enable us to express that.

An inductive type I is defined by a finite number $N \in \mathbb{N}$ of deduction (or construction) rules of the form

$$\frac{\mathbf{x}_k \in \mathbf{X}_k \quad \mathbf{i}_k \in \mathbf{I}_k}{C_k(\mathbf{x}_k, \mathbf{i}_k) \in I}$$

where $\mathbf{X}_k$ is a product of non-inductive types (at least from the perspective of the type I), and $\mathbf{I}_k = I^{n_k}$ for some $n_k \in \mathbb{N}$. That is to say, a rule has for premisses the existence of a number of objects $\mathbf{x}$ and $\mathbf{i}$, some of different, already existing types ($\mathbf{X}$), and some of its own type ($\mathbf{I}$).

A rule where $\mathbf{i}$ is empty, that is to say, that does not depend on previously constructed members of the type I, are called *axioms*, the others are said to be *inductive rules*.

An inductive type must have at least one axiom.

Take a minute to see how the three types we have defined so far all fit in this framework.

Given those rules, functions defined on those types will generally be given by N lines of the form

$$f\big(\mathbf{a}, C_k(\mathbf{x}_k, \mathbf{i}_k)\big) \ ,$$

where $\mathbf{a}$ is a number of other arguments that do not require induction, but the definition can of course be more complex if several arguments require simultaneous induction, or if some require nested patterns like $C_i(C_j(\dots))$.

Given the rules, the proof pattern is derived as

$$\frac{\forall k \in [\![1, N]\!],\ \forall \mathbf{x}_k \in \mathbf{X}_k,\ \Big(P(\mathbf{i}_k) \implies P\big(C_k(\mathbf{x}_k, \mathbf{i}_k)\big)\Big)}{\forall i \in I,\ P(i)} \ ,$$

In other words, for each constructor rule C_k, we assume that, for all possible inputs of the constructor, it preserves the property. That is, if all "smaller" elements going into the constructor satisfy the property, then the newly constructed element does as well. As that is true of course no matter what non-inductive elements are involved in the construction.

Note that in the case of axioms, the corresponding premise reduces to

$$\forall \mathbf{x}_k \in \mathbf{X}_k,\ P(C_k(\mathbf{x}_k)) \ .$$

59.2.6 Aside: Induction vs. Deduction

Those of you with a background in philosophy — or with friends with such background — may come across contexts where induction is opposed to deduction.

It bears mention that, in such contexts, the word induction *(inductive reasoning) has a completely different meaning to that which is presented here.*

In philosophy, where deduction means "applying general laws to a particular case", induction means "drawing reasonable inferences for a general law, on the basis of particular observations". Philosophical induction may yield wrong conclusions even if the premises are true; deduction may not.

Our kind of induction, which is referred to as mathematical induction *in the context of proofs, is very much a* deductive process. *The deduction rules* (59.1)[p344], (59.2)[p344], *and* (59.3)[p344] *are just that, deductions, theorems. If the premises are true, the conclusion does follow, every time.*

In computer science, there is seldom any ambiguity about which kind of induction we use: we do not deal with philosophical induction as such. That is not to say we never deal with imperfect knowledge or with notions of "reasonable inference". We do, in various fuzzy or probabilistic logics. But those are deductive *systems, where we reason deductively about our own gaps in knowledge and uncertainties, and produce conclusions qualified by our degrees of certainty.*

For instance, in Bayesian logic, let us say that $\mathbb{P}(\text{"the Butler did it"}) = .6$*, that is to say, you rather believe the Butler might have done it; it's more likely than not.*[az] *Then it* must *follow, deductively, on pain of paradox, that* $\mathbb{P}(\text{"the Butler did not do it"}) = \mathbb{P}(\neg\text{"the Butler did it"}) = 1 - .6 = .4$*. Not .35, not .5, nothing but .4 does it.*

The contexts where you are most likely to be exposed to possible confusion are works that define logical systems to formalise some aspects of inductive reasoning. For instance a quick search in E. T. Jaynes' Probability Theory, The Logic of Science *shows many uses of the word induction/inductive, a majority of which refer to mathematical induction, but a significant proportion refers to the philosophical notion.*

Thus, as this might crop up if you read about decision theory, AI, business intelligence, etc, it bears keeping in mind to avoid potential confusion.

Back to the topic at hand. . .

59.3 Recurrence: Induction on $\mathbb{N}$

Recurrence refers to a restricted kind of recursion, whereby an integer sequence or a proof[ba] at rank n refers to previous ranks of itself, and *only* to previous ranks. The Fibonacci sequence, for instance, is defined by recurrence:

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

Recurrence therefore simply refers to the inductive structure of Peano integers, and is therefore a subset of induction.

[az] I don't want to make an aside in an aside (that would be *too* recursive) so I'll just quickly mention that Bayesian logic deals with an agent's (rational) belief in a proposition, not with limits of relative frequencies in outcomes of a hypothetical infinity of trials (the "frequentist" approach).

[ba] mostly in the French phrase "raisonnement / preuve par récurrence"; the English usage mostly uses "proof by induction".

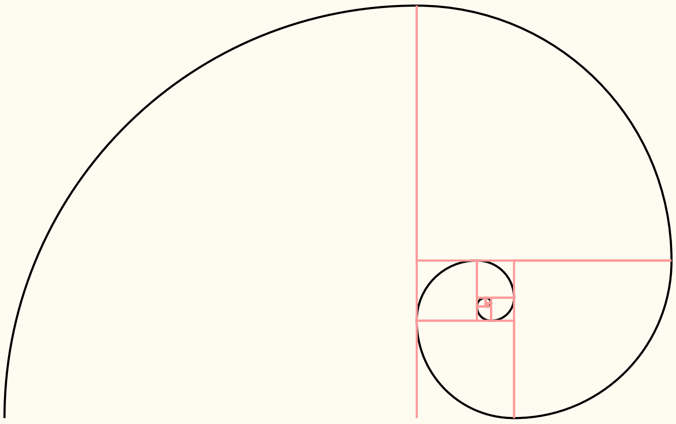


Figure 6: Spirale de Fibonacci

60 Les différents types de récursivité

Écrire les fonctions suivantes, de manière récursive:

(194) ♠ `factorial`, telle que $n \in \mathbb{N} \Rightarrow$ `factorial(n)` $= n!$, où

$$n! \;=\; \prod_{k=1}^{n} k \;=\; 1 \times 2 \times \cdots \times n$$

On note que $0! = 1$, élément neutre multiplicatif.

(195) `power`, telle que $a \in \mathbb{R}, b \in \mathbb{N} \Rightarrow$ `power(a,b)`$= a^b$. (sans utiliser `**` ou `pow`)

(196) ♠ `fibonacci`, telle que $n \in \mathbb{N} \Rightarrow$ `fibonacci(n)` $= F_n$, où la suite $(F_n)_n$ est définie par la relation de récurrence suivante: $F_0 = 0,\; F_1 = 1,\; n > 1 \Rightarrow\; F_n = F_{n-1} + F_{n-2}$.

(197) `even` et `odd`, testant la parité d'un entier $n \in \mathbb{N}$.

On utilisera des définitions mutuellement récursives.

(198) `ackermann`, telle que $m, n \in \mathbb{N} \Rightarrow$ `ackermann(m,n)` $= A(m,n)$, où

$$A(m,n) \;=\; \begin{cases} n+1 & \text{si } m = 0 \\ A(m-1,1) & \text{si } m > 0 \text{ et } n = 0 \\ A(m-1, A(m, n-1)) & \text{sinon.} \end{cases}$$

Calculer à la main $A(1,1)$. Sur machine, calculer également $A(1,2)$, $A(2,2)$, $A(3,2)$, $A(4,2)$. Que se passe-t-il ?

Modifier la fonction de manière à mettre en évidence tous les appels récursifs.

Nous reviendrons sur cette fonction dans la question 238.

(199) `syracuse` (ou `collatz`) telle que $n \in \mathbb{N} \Rightarrow$ `syracuse(m,n)` $= S(n)$, où

$$S(n) \quad = \quad \begin{cases} 1 & \text{si } n \leqslant 1 \\ S\left(\frac{n}{2}\right) & \text{si } n \equiv 0 \pmod 2 \\ S(3n+1) & \text{sinon.} \end{cases}$$

Calculer les 100 premières valeurs de S.

Formuler une conjecture. Tester la conjecture sur les 10^5 premiers entiers. [bb]

(200) ♠ Can you think of any legitimate use for a function calling itself without modifying its arguments in any way?

For instance:

```python
def f(args):
  ...          # no side effects on args
  f(args)
```

(201) What do you think of the list `l`:

```python
>>> l=[1]
>>> l.append(l)
```

Think about it and experiment, then read Sec. 24.2.2.4[p136]: "Infinitely deep lists".

Note that some language, such as Haskell, allow infinite structure without any fuss (so long as you do not seek to consume them exhaustively). For instance

```haskell
l = 1 : l
```

is a licit list definition in Haskell.

61 Combinatoire amusante — et récursivité

(202) ♠ Les arrangements ordonnés, ou permutations, de k objets parmi n, notés $\mathbf{A}_n^k$, sont donnés par les formules suivantes:

$$\mathbf{A}_n^k \quad = \quad \frac{n!}{(n-k)!} \quad = \quad n(n-1)(n-2)\cdots(n-k+1) \qquad k \leqslant n\,.$$

Donner une version directe de `permut(k,n)` $= \mathbf{A}_n^k$, en utilisant `factorial`, suivant la première formule donnée — attention au type de retour.

[bb] La preuve de cette conjecture est trop longue pour le bas de page de ce TD. Et puis elle est ouverte depuis 1937, et est considérée un des problèmes les plus difficiles des mathématiques. Ça n'aide pas. :-)

Donner une version récursive, suivant la seconde formule donnée – qui évite des calculs inutiles.

On pourra utiliser une *sous-fonction* récursive, ou déduire et mettre à profit une expression récursive de $\mathbf{A}_n^k$.

(203) Les combinaisons non-ordonnées de k objets parmi n, notées $\binom{n}{k}$, sont données par la formule bien connue (qu'on ne demande pas d'implémenter)

$$\binom{n}{k} = \frac{\mathbf{A}_n^k}{k!} = \frac{n!}{k!(n-k)!} = \prod_{i=1}^{k} \frac{n-i+1}{i} \qquad 0 \leqslant k \leqslant n\,.$$

On rappelle également la formule du binôme de Newton:

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k} = \sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k\,.$$

Utiliser le théorème du binôme et l'identité évidente

$$(1+x)^n = (1+x)^{n-1}(1+x)$$

pour trouver une expression récursive de $\binom{n}{k}$. On pourra aussi procéder par dénombrement combinatoire, en isolant un élément et en comptant les cas où il est pris ou laissé. On peut *aussi* connaître la relation de Pascal par cœur, mais c'est moins drôle.

(204) Utiliser cette expression pour écrire une fonction `binom` telle que `binom(k,n)` $= \binom{n}{k}$. Dessiner l'arbre des appels pour $\binom{4}{2}$.

(205) Prédire l'ordre et le nombre des appels récursifs. **Indication:** en Python, les opérandes d'une addition sont exécutées dans l'ordre: i.e. pour $x+y$, x est calculé avant y.

(206) Modifier `binom` de manière à vérifier cette prédiction.

(207) Quels sont les types d'opérations élémentaires réalisées par `binom(k,n)` ? Dénombrer le nombre d'instances de chaque type d'opération durant l'exécution de `binom(k,n)`, et en déduire la complexité en temps $T(k,n)$, que l'on exprimera comme un Θ d'une expression simple. (On utilisera le modèle de coût uniforme: les opérations élémentaires ont toutes le même coût).

(208) Quelle est la complexité en espace de `binom(k,n)` ? On pourra attendre d'avoir traité la question (228)[p353].

(209) Après avoir complété la question (236)[p356], mémoïsez `binom` à l'aide d'un décorateur.

(210) Quelle est la nouvelle complexité en temps de `binom`, après mémoïsation ?

(211) Quelle est la nouvelle complexité en espace de `binom`, après mémoïsation ?

(212) `GOTO` q. (238)[p356]

62 Tris et récursivité

Vous avez déjà vu des tris itératifs en $O(n^2)$ en moyenne et dans le pire des cas. Les meilleurs algorithmes de tris, utilisés en pratique, sont basés sur des procédés récursifs. Les deux principaux sont traités ici. On peut aussi mentionner le tris par tas, ou *heapsort*.

(213) Écrire une fonction `dicho` telle que `dicho(e,l)` soit équivalent à `e in l`, si l est une liste triée. On procédera par recherche dichotomique, et on utilisera des *slices* pour créer les sous-listes dans les appels récursifs.

(214) Écrire une nouvelle version de la recherche dichotomique, telle que `dicho(e,l,a,b)` effectue la recherche entre les indices a et b, inclus, et retourne `None` si $e \notin l$, et un indice k tel que $e = l[k]$ sinon.

(215) ♠ Le tri rapide, ou *quicksort*, de complexité moyenne $O(n \log n)$, pire des cas $O(n^2)$, repose sur les observations suivantes, en notant τl un tri d'une liste l:

- La liste vide $[]$ est déjà triée: $\tau[] = []$.
- Toute liste singleton $[e]$ est déjà triée: $\tau[e] = [e]$.
- Soit une liste $[p, e_1, \ldots, e_n]$; alors la liste

  $$\tau[e_i \mid e_i \leqslant p] \;\; + \;\; [p] \;\; + \;\; \tau[e_i \mid e_i > p]$$

  est triée, et contient les mêmes éléments que l. On appelle p le *pivot*.

Donner une fonction `qsort` qui renvoie une version triée de la liste passée en argument. On n'hésitera pas à utiliser la syntaxe en compréhension de Python, cf. Sec. 24.5[p153]: "Comprehension expressions".

Exemple: `[2*i for i in range(5) if i != 2]` renvoie `[0, 2, 6, 8]`.

On pourra également utiliser le *packing* `*` pour décomposer la liste: `p,*l` $= [p, e_1, \ldots, e_n]$ donne `p` $= e$ et `l` $= [e_1, \ldots, e_n]$. Voir aussi l'*unpacking*, question 227. Pour plus de détails, voir Sec. 24.6[p164]: "Packing and unpacking".

(216) Écrire une fonction `merge` permettant de fusionner deux listes déjà triées en une nouvelle liste triée. On procédera par induction structurelle sur les listes.

Note sur les listes: ce que Python appelle "listes" correspond classiquement plutôt à des tableaux dynamiques. Une liste classique est définie inductivement comme étant soit

- la liste vide $[]$
- un doublet $\langle e, l \rangle$ contenant un élément e (le "premier") et une liste l (le "reste")

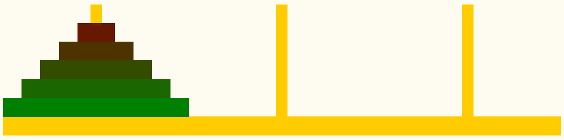
La liste `[1, 2]` est donc classiquement construite comme $\langle 1, \langle 2, [] \rangle\rangle$. L'implémentation classique est la liste chaînée, une structure contenant un élément et un pointeur vers une autre liste — ou la même liste, si on veut une liste infinie.

Même si cette structure inductive ne correspond pas à celle implémentée par Python, il est utile de la garder à l'esprit pour écrire des algorithmes récursifs sur les listes. On pourra écrire `l[0]` pour le premier élément, `l[1:]` pour le reste (ou utiliser le packing), `[e]+l` pour ajouter un nouvel élément au début, et enfin tester si la liste est vide avec `if l` ou `if not l`.[bc]

(217) Le tri fusion, ou *merge sort*, de complexité moyenne et pire des cas $O(n \log n)$, repose sur le principe suivant: pour trier une liste, on la coupe en deux morceaux de tailles égales (± 1), on trie chaque morceau, et on les fusionne. Écrire une fonction `msort` qui réalise cela. Il va sans dire qu'on devra utiliser `merge`.

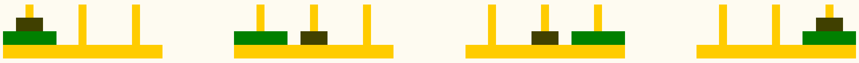
63 Les tours de Pizzanoï

Il est dit qu'au commencement du temps, le Monstre Spaghetti posa trois pieux, et empila soixante-quatre délicieuses pizzas (imputrescibles et indestructibles jusqu'à nouvel ordre) de diamètres décroissants sur le premier pieu.



Et en vérité il dit aux moines affamés: "Vous pourrez manger les pizzas lorsque vous les aurez transférées, une par une, sur le troisième pieux. Mais sachez-le, vous ne devez poser une pizza que sur un pieu vide, ou sur une pizza plus grande. Pourquoi ? Parce que."

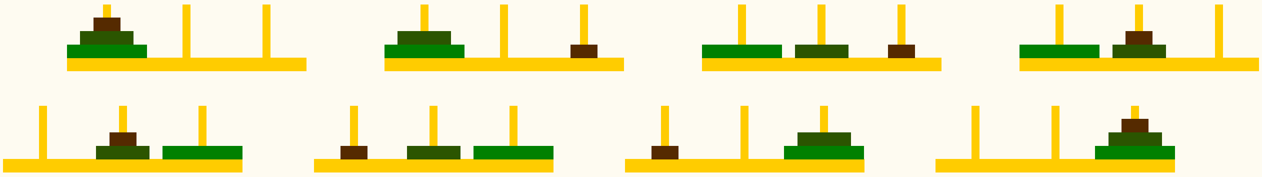
Les moines ronchonnèrent et ruminèrent et réfléchirent et résolurent le problème pour de petits nombres n de pizzas; par exemple pour $n = 2$:



et $n = 3$:

[bc] Dans le contexte d'un test, Python traduit une valeur non-booléenne en booléen selon le principe: "si c'est vide, c'est faux, sinon vrai". `if l`, `if len(l) > 0`, et `if l != []` sont donc *presque* équivalents. La différence est que les deux premiers tests seront faux pour un tuple ou un dictionnaire vide, alors que le troisième sera vrai. `if l` est donc la façon la plus agnostique du point de vue du type de tester si une structure conteneur est vide.

For more information, cf. Sec. 22.6.5[p104]: "The semantics of **and** and **or**, & implicit Boolean conversion".



(218) ♠ Pour quelles valeurs de n le problème admet-il une solution ?

Donnez un algorithme récursif.

(219) ♠ Lorsque le problème admet une solution, donnez une borne supérieure T_n au nombre d'opérations nécessaires, exprimée par une relation de récurrence.

(220) ♠ Montrer que la borne supérieure est aussi une borne inférieure – du moins si vous avez trouvé la bonne solution.

(221) ♠ Donner une forme close de T_n, par toute méthode appropriée.

(222) En comptant en moyenne 10 secondes pour déplacer une pizza d'un pieu à l'autre, dans combien de temps les moines pourront-ils passer à table ?

(223) Écrire un programme Python pour générer et représenter les étapes de résolution du problème.

64 Suite de Fibonacci: piles & mémoisation[bd]

Soit la fonction suivante, qui implémente la suite de Fibonacci:

$$F_0 = 0,\ F_1 = 1, \quad n \geqslant 2 \ \Rightarrow \ F_n = F_{n-1} + F_{n-2}.$$

```python
def f(n):
    print("call f", n) # afin d'imprimer une trace des appels recursifs
    return n if n <= 1 else f(n-1) + f(n-2)
```

Note: closed form

It will be useful in some of the following questions to keep the rate of growth of F_n in mind, which is clearer when seeing a closed form. Let

$$\varphi \ = \ \frac{1+\sqrt{5}}{2} \ \approx \ 1.6180339887\ldots$$

be the famous "golden ratio"; then we have

$$F_n = \left[\frac{\varphi^n}{\sqrt{5}}\right], \qquad \textit{for } n \geqslant 0\ ,$$

[bd] Non, l'absence de "r" n'est pas une typo.

where $[x]$ stands for "round x to nearest integer". In fact, as n grows the rounding error becomes vanishingly small.

Of course there are other closed forms, without any rounding or truncation, but this one is the simplest, and serves well to illustrate the relevant fact: F_n increases exponentially in n.

64.1 Arbre, pile, et nombre d'appels

(224) ♠ Sans utiliser la machine, prédisez ce que va afficher Python lorsqu'on exécute `print(f(5))`.

(225) Soit C_n le nombre total d'appels à f lors du calcul de $f(n)$ — i.e. , le nombre de lignes "call" affichées. Exprimez C_n par une relation de récurrence.

(226) Exprimez la complexité en temps de calcul de `f(n)` comme un Ω d'une expression simple, et en déduire qu'il est (au moins) exponentiel.

(227) ♠ Simulons la pile[(be)] d'appels avec le code suivant:

```
stack = []

def fstack(n):
    stack.append(n) ; print(*stack)
    r = n if n <= 1 else fstack(n-1) + fstack(n-2)
    stack.pop() ; print(*stack)
    return r

print(fstack(3),stack)
```

Sans utiliser la machine, écrire ce que Python afficherait si l'on exécutait ce code.

Note: l'* dans `print(*stack)` correspond ici simplement à un *unpacking*, où les éléments d'un itérable sont passés en argument de la fonction. Le code `print(*[1,2,3])` correspond à `print(1,2,3)` et produit l'affichage `1 2 3`. Voir Sec. 24.6$_{[p164]}$: "Packing and unpacking" pour plus de détails.

(228) Quelle est la profondeur maximale de la pile d'appel au cours de l'exécution ? En déduire la complexité en mémoire de `f(n)`. On peut maintenant traiter la question 208.

64.2 Mémoïsation: vers la programmation dynamique

La grande complexité en temps vient de ce que les mêmes calculs sont effectués de très nombreuses fois. On veut rendre la fonction plus efficace, spécifiquement pseudo-linéaire[(bf)],

[(be)] Une pile – pensez à une pile de copies à corriger – est une structure de données suivant la discipline LIFO (rien à voir avec le Laboratoire d'Informatique Fondamentale d'Orléans; cela signifie Last In, First Out). On peut déposer un élément sur la pile, ou retirer l'élément sur le dessus de la pile. Et c'est tout.

[(bf)] . . . c'est à dire linéaire en la *valeur* de l'entrée, par opposition à linéaire en la taille du *codage* de l'entrée, ce qui est la vraie mesure de complexité algorithmique.

en mémorisant les résultats des calculs déjà faits. On utilisera à cette fin un dictionnaire.

A chaque appel, la fonction vérifiera si le résultat est déjà précalculé dans le dictionnaire; s'il y est, elle renvoie cela directement au lieu de refaire le calcul; sinon, elle fait le calcul et ajoute ce nouveau résultat au dictionnaire avant de le renvoyer.

Ceci décrit le principe général de la *mémoïsation* — on se laisse des memos, ou memoranda, "choses qui doivent être retenues", des calculs précédents.

(229) ♣ Écrire une nouvelle version de la fonction (en partant de la version précédente) suivant ce principe; on complétera le code suivant

```
memo = {}

def ff(n):
    print("call ff", n, memo)
    ...
```

Lorsqu'on invoque `print(ff(5))` deux fois de suite, on doit obtenir

```
call ff 5 {}
call ff 4 {}
call ff 3 {}
call ff 2 {}
call ff 1 {}
call ff 0 {1: 1}
call ff 1 {1: 1, 0: 0, 2: 1}
call ff 2 {1: 1, 0: 0, 2: 1, 3: 2}
call ff 3 {1: 1, 0: 0, 2: 1, 3: 2, 4: 3}
5
# second call
call ff 5 {1: 1, 0: 0, 2: 1, 3: 2, 4: 3, 5: 5}
5
```

(230) ♣ Explain this call trace by means of a traversal of the call tree.

(231) ♣ L'utilisation d'un dictionnaire limite-t-elle l'applicabilité de la méthode de la question précédente à d'autres fonctions que Fibonacci ? Est-ce une restriction inhérente au principe de la mémoïsation ou contingente à nos choix d'implémentation ? Dans ce dernier cas, y a-t-il des alternatives ? Quelles propriétés seraient changées ?

(232) On veut encore une autre implémentation pseudo-linéaire de la suite de Fibonacci, encore plus élégante et rapide, tenant compte du fait qu'il s'agit du cas particulier d'une suite récurrente linéaire. L'idée est que nous n'avons besoin que des deux dernières valeurs, et que nous pouvons donc transformer la récursivité double sur une valeur en une récursivité simple sur un doublet.

Complétez le code suivant

```
def fff(n):
    print("call fff", n)
    if n == 0:
        return (0,1)
     .......
```

de manière à ce que fff(n)[0] = f(n), pour tout $n \in \mathbb{N}$. Lorsqu'on invoque `print(fff(5)[0])`, on doit obtenir

```
call fff 5
call fff 4
call fff 3
call fff 2
call fff 1
call fff 0
5
```

64.3 Foncteurs et décorateurs de mémoïsation

Python supporte les fonctions d'ordre supérieur, c'est à dire les fonctions qui prennent en argument ou renvoient des fonctions. Nous nous intéressons ici aux transformations fonction vers fonction. On parle parfois de foncteurs ou, dans le cas de Python, de décorateurs.

(233) Écrire un décorateur `memoize`, tel que pour toute fonction monadique `f` d'argument mutable, `memoize(f)` renvoie une version memoïzée de `f`. En exécutant

```
f = memoize(f)
print(f(5))
print(f(5))
```

on doit obtenir

```
call f 5
call f 4
call f 3
call f 2
call f 1
call f 0
5
5
```

(234) Commenter le code de test précédent, et le remplacer par

```
g = memoize(f)
print(g(5))
print(g(5))
```

Obtient-on la même chose ? Pourquoi ?

Alerte: Autant certains aspects de la programmation – par exemple la sémantique d'un **if** – relèvent dans une grande mesure du sens commun[bg] et sont transposables d'un langage à l'autre, voir d'un paradigme à l'autre, autant certains peuvent varier et être assez subtils.

En l'occurrence, le comportement de `memoize` dépend de certains choix de *language design*: clôtures lexicales & portées lexicales ou dynamiques, espaces de noms mutables ou immutables, . . . qui seront mieux compris avec de l'expérience sur plusieurs langages différents. Ne passez donc pas trop de temps sur le "pourquoi ?" pour l'instant.

(235) ♣ Commenter (ou écraser) la définition de `g` de la question précédente, et tester le code suivant:

```
@memoize
def g(n):
    print("call g", n)
    return n if n <= 1 else g(n-1) + g(n-2)
```

En déduire comment Python interprète l'annotation `@memoize` et pourquoi on parle de "décorateur".

(236) Nous avons supposé pour l'instant que la fonction à mémoïser `f` est monadique. Altérez le décorateur `memoize` de manière à ce qu'il supporte toute fonction variadique d'arguments non-mutables.

Indice: utilisez le packing (q. 215) et l'unpacking (q. 227). Dans le corps d'une définition de fonction `def f(*x)`, `x` est le tuple des arguments passés à la fonction. Voir Sec. 24.6[p164]: "Packing and unpacking" pour plus de détails.

(237) `GOTO` q. 209.

(238) Appliquez maintenant le décorateur `memoize` à la fonction `ackermann` de la question 198. Comme `binom`, c'est une fonction à deux arguments entiers. Obtient-on les mêmes gains de complexité ? Pourquoi ?

65 Dynamic Programming for Difference Equations

The Fibonacci sequence is only the best known instance of a much larger class of equations know as **linear recurrence relations**, or **linear difference equations**[bh].

[bg] Et encore, même pour le **if** il convient de distinguer les *instructions* des *expressions* conditionnelles. . .

[bh] Note to be confused with *differential* equations, of course.

A linear recurrence relation with real coefficients of order K is an equation of the form

$$u_n \quad = \quad a_0 + \sum_{i=1}^{K} a_i u_{n-i} \, ,$$

where $a_0, \ldots, a_K \in \mathbb{R}$, $a_K \neq 0$. At least K initial values must be provided to define a function $\mathbb{N} \to \mathbb{R}$.

When $a_0 = 0$, it is said to be homogeneous.

(239) ♣ Can any linear difference equation be memoised?

(240) ♣ **Dynamic programming** refers to the general idea of solving a complex problem recursively by breaking it down into simpler subproblems. If an optimal solution can be found by combining the optimal solutions of the subproblems, then the problem is said to have optimal substructure, and is well-suited for the approach.

Memoisation can be thought of as a top-down approach to dynamic programming: the problem is broken down and, at the final steps of the recursion, the values are calculated and stored.

Can you imagine what a bottom-up approach would look like? Apply this intuition to provide a linear implementation of the following function, *without* using recursion.

```
def lin_diff_eq(n, init, *a):
    """This variadic function returns the list of the n first terms
      of the linear recurrence relation

        f(0) = init[0], ..., f(m) = init[m],
        f(n) = a[0]*f(n-1) + a[1]*f(n-2) + ... + a[k-1]*f(n-k) + a[k]
                                                            , for n > m

    where k = len(a)-1 and m = len(init)-1. If init is too small,
    an AssertionError should be raised.

    For instance, lin_diff_eq(10, [0, 1], 1, 1, 0) corresponds to
    the first terms of the Fibonacci sequence

        f(O) = 0,  f(1) = 1,  f(n) = f(n-1) + f(n-2), for n >= 2 .

    Furthermore, an *efficient* implementation, in O(n) time, is required.
    """
```

Here are a few test cases; let us start with our good friend Fibonacci:

$$u_0 = 0, \; u_1 = 1, \quad u_n = u_{n-1} + u_{n-2}$$

```
>>> lin_diff_eq(10, [0, 1], 1, 1, 0)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Let us check the efficiency of our function by computing u_{999}, an impossible feat with a naïve implementation:

```
>>> round(log10(lin_diff_eq(1000, [0,1], 1,1,0)[999]))
208
```

Now for a non-homogeneous variant of Fibonacci, which is relevant for a question in another exercise (but I'm not telling you which one!):

$$u_0 = 0,\ u_1 = 1, \quad u_n = u_{n-1} + u_{n-2} + 1$$

```
>>> lin_diff_eq(10, [0, 1], 1, 1, 1)
[0, 1, 2, 4, 7, 12, 20, 33, 54, 88]
```

A simple geometric sequence:

$$u_0 = 1, \quad u_n = 2u_{n-1}$$

```
>>> lin_diff_eq(10, [1], 2, 0)
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

and a staggered, order 2 variant:

$$u_0 = 0,\ u_1 = 1, \quad u_n = 2u_{n-2}$$

```
>>> lin_diff_eq(10, [0,1], 0, 2, 0)
[0, 1, 0, 2, 0, 4, 0, 8, 0, 16]
```

Finally, an order 4 variant of Fibonacci:

$$u_0 = 0,\ u_1 = 1,\ u_2 = 0,\ u_3 = 0, \quad u_n = u_{n-1} + u_{n-2} + u_{n-3} + u_{n-4}$$

```
>>> lin_diff_eq(18, [0,1,0,0], 1, 1, 1, 1, 0)
[0, 1, 0, 0, 1, 2, 3, 6, 12, 23, 44, 85, 164, 316, 609, 1174, 2263, 4362]
```

Note for mathy types: The ultimate implementation would of course solve the equation into a closed form, which is always possible in the linear case, but **(1)** requires quite a bit of legwork (and specialised knowledge) and **(2)** won't work in the *non*-linear case, as non-linear difference equations are often unsolvable (they have no closed form).

(241) ♣ Our difference equations have two major restrictions: the coefficients being constants, and the linearity of the recursive expression. The first prevents us from dealing with, for instance, the factorial sequence:

$$f_0 = 1, \quad f_n = nf_{n-1}\ ,$$

and the second excludes very important sequences, such as the logistic map, which is a quadratic difference equation:

$$x_{n+1} \quad = \quad rx_n(1-x_n)\ . \tag{65.1}$$

Do you think those restrictions are important for our application of dynamic programming?

An aside on the logistic map:

Consider a population P. *At what rate does it grow over time? Given unlimited space and resources — that is to say, whatever pressures are at play to encourage reproduction or death do not depend on the population — the average offspring per capita and death per capita will boil down to a constant* r*, which means that the growth is proportional to the current population, and we have*

$$\frac{dP}{dt} = Pr\ ,$$

which solves into

$$P(t) \quad = \quad P_0 e^{rt}$$

Where P_0 *is the initial population. This is the Malthusian (Thomas Robert Malthus, 1766–1834) model of population, which is quite restrictive since it boils down to either one of three behaviours: unlimited exponential growth if* $r > 0$*, eventual extinction if* $r < 0$*, or endless stagnation if* $r = 0$*.*

Let us now assume, as did Pierre François Verhulst (1804–1849), that there are other forces at play — limited resources — so that the greater the population, the lesser its growth rate. More specifically, let us say that each new individual in the population increases everybody's *mortality by some amount* s*. We obtain*

$$\frac{dP}{dt} = P(r - sP)\ ,$$

which can be rewritten (rescaling P by $\frac{s}{r}$*) in the more convenient form*

$$\frac{dP}{dt} = rP(1 - P)\ .$$

What is the connection between this differen<u>*tial*</u> *equation and the quadratic differen*<u>*ce*</u> *equation* (65.1)*?*

As humans, we are used to the idea of population growth happening continuously, as our generations overlap. That is not true of all species. Consider an insect population that breeds then dies, leaving eggs which hatch much later — an example of this is Dawson's burrowing bee.

Parents never live to meet their offspring, generations are nonoverlapping. In that case, growth still happens, but in discrete steps: P is a sequence, where P_n represents the population at generation n. We have, for unlimited ressources, an equation of the form

$$P_{n+1} = rP_n \ .$$

Though we adjusted the meaning of the constant for the sake of simplicity — our r here would intuitively correspond to $1 + r$ in the continuous version — the same arguments as before apply, leading to

$$P_{n+1} = P_n(r - sP_n) \ ,$$

and, by taking $x_n = \frac{s}{r}P_n$, to equation (65.1) again.

The logistic sequence not only models real-world populations very well, but it also exhibits extremely interesting, unintuitive behaviours which are, sadly, out of the scope of this course.

Let us just note that it is quite chaotic for some values of r, to the extent that is was used as a very simple pseudo-random numbers generator in early computers — though it is not quite up to modern standards for that use case.

The logistic map therefore stands as a good introduction to the basic observation at the heart of chaos theory: the fact that non-linear systems, though simple to define, can have extremely complex and unpredictable dynamic behaviours, and that this is every bit as true for discrete recurrence relations as it is for continuous differential equations.

Of course, equation (65.1) cannot even be solved — that is to say, expressed as a non-recursive, closed form — except for some few fixed values of r. I believe the only closed forms are for $r \in \{-2, 2, 4\}$.

(242) ♣ In all generality, a **recurrence relation of order** k[bi] is an equation of the form

$$u_n = \varphi(n, u_{n-1}, u_{n-2}, \dots, u_{n-k}) \quad \text{with} \quad n \geqslant k \ ,$$

where φ is a function of type $\mathbb{N} \times S^k \to S$ for some set S. Given at least k initial values, this defines a function $u : \mathbb{N} \to S$.

Give a general dynamic programming implementation of recurrence relations as a function of the form

```
recrel(n, init, φ, k=None)
```

[bi] Also often called simply a **difference equation**. This usage can be ambiguous, though, as some authors apply the term *difference equation* exclusively to some specific forms of recurrence relations, involving differences of successive terms in a sequence. In this document, I subscribe to the more general terminology.

that efficiently yields the list of the first `n` elements of the sequence, given a list `init` of initial values, and φ as above. We shall assume that `init` contains exactly k values, which enables the function to determine the order without needing to analyse the arity of φ[bj]. Otherwise, the order of the relation must be passed explicitly as the parameter k.

Let us try it on a few interesting relations... First, the factorial equation

$$f_0 = 1, \quad f_n = nf_{n-1}$$

becomes:

```
def rr_fac(n,u): return n*u

>>> recrel( 10, [1], rr_fac)
[1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880]
```

The Fibonacci sequence:

```
def rr_fib(n,u,uu): return u+uu

>>> recrel( 10, [0,1], rr_fib)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

A quadratic variant of the Fibonacci sequence:

$$F_0 = 0,\ F_1 = 1, \quad n \geqslant 2 \quad \Rightarrow \quad F_n = F_{n-1}^2 + F_{n-2}^2.$$

```
def rr_fibs(n,u,uu): return u**2 + uu**2

>>> recrel( 10, [0,1], rr_fibs)
[0, 1, 1, 2, 5, 29, 866, 750797, 563696885165, 317754178345286893212434]
```

And finally, the logistic map. Since the logistic map has a parameter, we have to be careful with the signature of φ. We cannot include r as a parameter of φ itself, as that would confuse `recrel`. The solution is to use a higher-order function:

```
def logistic(r):
    return lambda n,u : r*u*(1-u)
```

which would admittedly be much more elegant in a language supporting currying[bk], such as OCaml or Haskell, but hey, you do with what you got, as they say.

We can play with different values of r to exhibit some of the sequence's fun behaviours:

[bj] This could be done using `inspect.getargspec`, but playing with introspection is not the goal of this exercise.

[bk] Currying is a way of reducing functions with multiple arguments into unary higher-order functions. That is, the correspondence between $f : X \times Y \to Z$ and its curried version $f' : X \to (Y \to Y)$. Functional languages of the ML family handle multiple arguments that way.

```
# two values oscillation
>>> list(map(lambda x:round(x,1), recrel( 10, [.99], logistic(3) ) ))
[1.0, 0.0, 0.1, 0.2, 0.5, 0.7, 0.6, 0.7, 0.6, 0.7]
>>> list(map(lambda x:round(x,1), recrel( 10, [.3 ], logistic(3) ) ))
[0.3, 0.6, 0.7, 0.6, 0.7, 0.6, 0.7, 0.6, 0.7, 0.6]

# four values oscillation
>>> list(map(lambda x:round(x,1), recrel( 10, [.01], logistic(3.5) ) ))
[0.0, 0.0, 0.1, 0.4, 0.8, 0.5, 0.9, 0.4, 0.8, 0.5]

# chaotic behaviour
>>> list(map(lambda x:round(x,2), recrel( 100, [.01], logistic(3.99) ) ))
[0.01, 0.04, 0.15, 0.51, 1.0, 0.01, 0.05, 0.19, 0.6, 0.95, 0.18, 0.58,
          ..., 1.0, 0.01, 0.05, 0.18, 0.58, 0.97, 0.11, 0.4, 0.96, 0.16]
```

(243) ♣ Can you apply `recrel` to the Ackermann function (198)[p347]?

(244) ♣ Can you apply `recrel` to the Collatz function (199)[p348]?

66 Dérécursivation — si, si, c'est un mot, ça.

Tout algorithme récursif peut se réécrire de façon itérative, à l'aide d'une pile (et vice-versa). En pratique, c'est comme cela que nos programmes récursifs tournent sur machine: la pile d'appel gère les appels récursifs.

Si l'on en ressent le besoin – par exemple si l'on écrit un compilateur, ou si l'on se trouve limité pas des problèmes de stack overflow[bl] – on peut donc "dérécursiver" un algorithme.

66.1 Récursitivé terminale

Le cas le plus intéressant est celui de la récursivité terminale, ou *tail recursivity* — *tailrec*. Une fonction est récursive terminale si elle n'effectue aucune opération après un appel récursif.

Dans ce cas, on peut très facilement éliminer la récursivité. En effet, comme on n'a jamais besoin de revenir à un contexte d'exécution antérieur, gérer une pile d'appel est inutile.

Notons que de nombreux compilateurs détectent la récursivité terminale et effectuent cette transformation automatiquement. C'est en particulier le cas de tous les langages fonctionnels, tels que OCaml, Haskell, Lisp, Scheme, et cetera. Ce n'est malheureusement pas le cas de Python.

De manière abstraite, un algorithme récursif (simple) est de la forme suivante, en pseudo-code style Python:

[bl] En Python, on peut augmenter "à l'arrache" la taille du stack par, e.g. , `sys.setrecursionlimit(99999)`. Dans d'autres langages ça peut être géré au niveau de l'OS.

```
def A(x):
  if C:   I ; A(τx) ; F
  else:   T
```

- ◇ A est l'algorithme sous considération
- ◇ x est l'argument – ou la liste des arguments – de A
- ◇ C est une condition portant sur x
- ◇ τ est une transformation des arguments
- ◇ I, F, T sont des traitements, initial, final, et terminal, dépendant de x.

Pour se simplifier la vie, on supposera que C, I, F, T, τ ne font pas d'appels à A.

Si F est vide, alors l'algorithme est récursif terminal, et est équivalent à

```
def A(x):
  while C:  I ; x := τx
  T
```

(245) ♠ La fonction `factorial` de la question (194)[p347] est-elle récursive terminale ? Pourquoi ? Réécrivez-la pour mettre sa structure C, I, F, T, τ en évidence.

(246) ♠ Transformer `factorial` de manière à effectuer le traitement des données dans une sous-fonction récursive terminale.

(247) ♠ Réécrivez la sous-fonction pour mettre sa structure C, I, T, τ en évidence.

(248) ♠ En appliquant la transformation générale donnée dans cette section, dérécursiver la sous-fonction.

(249) ♠ Intégrer la sous-fonction dans le corps de la fonction.

66.2 Récursitivés complexes: simuler la pile d'appel

Lorsque les récursivités sont trop complexes pour pouvoir être transcrites en récursivités terminales de cette manière[bm], il faut simuler les appels récursifs à l'aide d'une pile. Une autre façon de voir les choses est que la récursivité n'est qu'une écriture astucieuse d'un empilement.

Quoi qu'il en soit, en pratique, lorsque votre programme récursif tourne sur un ordinateur, c'est en réalité un algorithme itératif utilisant une pile qui s'exécute, la récursivité n'étant pas une caractéristique primitive des architectures matérielles actuelles (ou passées).

[bm] Il existe une approche générale pour tout convertir en récursivité terminale: **C**ontinuation-**P**assing **S**tyle. Il s'agit d'une technique avancée de programmation fonctionnelle, souvent utilisée dans les compilateurs. Comme sa petite cousine impérative, **S**ingle **S**tatic **A**ssignment (Form), elle est rarement utilisée directement par un programmeur.

La version dérécursivée avec une pile de l’algorithme A, non récursif terminal, est comme suit:

On définit `new` et `end`, deux symboles distincts, indiquant si l’on empile le début d’un nouvel appel, ou le retour d’un ancien.

```
def A(x):
  if C:   I ; A(τx) ; F
  else:   T
```

devient

```
def A(x):
    calls = [new,x]  #pile d'appels.
    ret = None   # valeur de retour
    while calls:
        x = calls.pop()  # on recupere les arguments de l'appel
        state = calls.pop()
        if state == new:   # nouvel appel; on traite le debut de A
            if C:
                I
                calls.extend((end,x))
                    # quand on aura fini l'appel recursif qui suit,
                    # il faudra terminer cet appel: il reste F
                    # On stocke le contexte d'ex\'ecution
                    # ici, le parametre de la fonction
                calls.extend((new,τx))
                    # On lance un nouvel appel A(τx)
            else:
                T
                  # les return sont traites
                  # comme une affectation a ret
        elif state == end:  # fin d'ancien appel a terminer
            F # meme remarque que pour T
    return ret
```

(250) ♣ Dérécursiver la fonction `factorial` de la question 194 en utilisant le patron ci-dessus.

(251) ♣ Que faire pour les récursivités multiples, mutuelles, ou imbriquées ?

Part VI

Python Project 2024–2025: AIge of EmpAIres

Short version: *Implement a bare bones "Age of Empires"-style RTS engine for AI-vs-AI battles.*

67 What is an RTS / AoE?

The astute reader might notice that this projet is related to, though distinct from, Part X[p399]: "Archived Python Project 2021–2022: Age of Cheap Empires". If you are not familiar with Real Time Strategy (RTS) games, especially in the Age of Empires (AoE) series, you are invited to get some idea of the general principles.

It should go without saying that you do not need to buy or play any of these games. You can get all relevant information by watching videos of AoE I and II on the internet, and consulting the wiki or the official site:

◇ https://ageofempires.fandom.com/wiki/Age_of_Empires

For a more hands-on look on that genre, you can install 0 A.D., a game originally made as a mod for Age of Empires II, now entirely free software. You can install it on most modern Linux distributions with a single command; for instance:

```
sudo apt install 0ad   # Debian-based
sudo pacman -Syu 0ad   # Arch-based
```

0 A.D. is its own thing, with its own unique mechanics, but remains extremely similar in look and feel to AoE I and II, so it can give you a general idea of how a game like that plays.

For large screens, you may want to create a file `~/.config/0ad/config/local.cfg` containing a line of the form `gui.scale = "1.875"`, to scale up the GUI elements, including the fonts — here by a factor of 1.875, to replace by what works for you.

The remainder of this document assumes basic familiarity with the concepts of those games.

68 Scope of the Project

Unlike the 2022 project, the aim is *not* to implement an RTS game. I want to avoid giving too much focus to technically uninteresting aspects such as user interface and gameplay mechanics. We shall have the bare minimum of buildings and units, and focus entirely on AI-vs-AI matches, with no direct input from the user during the matches. Your main tasks will therefore be to **(1)** develop a game engine, and **(2)** fine-tune various AI profiles (defensive, aggressive, etc).

You will demonstrate the quality of the final product by showing large battles between multiple AIs deploying sophisticated strategies.

68.1 List of Requirements

AoE2 is my reference point, and we shall take a minuscule subset of rules from it.

(1) The game takes place on a **map**, which is a grid of size $N \times M$ tiles. The absolute minimum size you must be able to handle is 120×120, which corresponds to a "tiny" map in AoE2.

(2) The **maps will be randomly generated** — a fact that you must be able to demonstrate.

(3) You will support at least two different **types of randomly generated maps**, each with strategic and tactical implications: for instance one with generous resources dotted across the map (take the Arabia map from AoE2 as reference for what that looks like) and one where all the gold is at the centre of the map.

(4) **Population limit** = maximum number of units per player: 200.

Actual limit in play determined by houses and town centres, within that maximum.

(5) The following **resources**:

a. Wood (W), 100 per tile (tree)

b. Food (through farms only) (F), 300 per farm

c. Gold (G), 800 per tile

(6) **Units**:

a. Villager: v
Cost 50F, 25 HP, Training time 25s,
2 attack (1 attack per second for all units), speed 0.8 tile/second.

Can build buildings.

The *nominal building time* t of a building given below is the time required for one Villager to construct a building alone.

If n Villagers are used, and t is the nominal building time remaining, the *actual* building time will be $\frac{3t}{n+2}$.

Can collect resources at rate of 25/minute, can carry 20.

b. Swordsman: s
Cost 50F+20G, Training time 20s,
40HP, 4 attack, speed .9.

c. Horseman: h
Cost 80F+20G, Training time 30s,
45HP, 4 attack, speed 1.2.

d. Archer: a
Cost 25W+45G, Training time 35s,
30HP, 4 attack, 4 range (Euclidean distance; norm 2), speed 1.

(7) Buildings:

a. Town Centre: T
Cost 350W, Build time 150 seconds,
1000 HP, 4x4, Spawns Villagers, Drop point for resources
Population: +5.

b. House: H
Cost 25W, Build time 25 seconds,
200 HP, 2x2, Population: +5.

c. Camp: C
Cost 100W, Build time 25 seconds,
200HP, 2x2, Drop point for resources

d. Farm: F
Cost 60W, Build time 10 seconds,
100HP, 2x2, Contains 300 Food.
Note: this is the only walkable building, cf. AoE2.

e. Barracks: B
Costs 175W, Build time 50 seconds,
500HP, 3x3, Spawns Swordsmen

f. Stable: S
Costs 175W, Build time 50 seconds,
500HP, 3x3, Spawns Horsemen

g. Archery Range: A
Costs 175W, Build time 50 seconds,
500HP, 3x3, Spawns Archers

h. Keep: K
Costs 35W, 125G, Build time 80 seconds,
800HP, 1x1,
Fires arrows: Attack 5, range 8

(8) Starting conditions:

a. Lean: 50F, 200W, 50G,
Town Centre, 3 Villagers

b. Mean: 2000(F,W,G)
Town Centre, 3 Villagers

c. Marines: 20000(F,W,G)
3 Town Centres, 15 Villagers, 2 (Barracks, Stable, Archery Range)

(9) Agile implementation.

Do not try and implement all units and buildings in one go, before starting work on the systems that depend on them.

First implement only villagers and town centres, give them large starting resources and implement villager wars.

Then, once the game loop is shown to work, go back and implement more variety of resources and units.

If at the end of the day you do not have a complete product in the sense that some units and buildings are missing, but you can otherwise demonstrate that what you *have* does work as expected, that won't be ideal, but the outcome will still be honourable. You'll indeed have made a game AI, albeit a limited one.

On the other hand, if you tell me that you have, technically, implemented every unit and every building, but none of them actually moves of builds or fights, then you don't actually have anything of any worth.

(10) Bare-bones main "menu".

Do not spend much time implementing a sophisticated game menu with a pretty background and music etc. I could hardly care less.

I care about functionality. You must be able to create a new game with various parameters, save and load a game efficiently. Whether you do this through a GUI or entirely through command line options, it must be easy and fast to use during the demonstration.

I would not accept, for instance, that your game can only have one (or three, or five) save file(s) because of limitations in your menu GUI. If you must have a menu GUI, use the standard file open dialogues for such purposes. TkInter is fine for that type of tasks.

(11) Map visualisation: terminal

Just like the game of chess is independent of the gameboard you use — wooden or glass pieces? on the table or on a computer or purely via chess notation through email or snail mail or . . . — the logic of your game must be independent of its graphical visualisation.

This is extremely important to understand, with far-ranging consequences on development time, quality and reliability of code, etc.

To force you to separate the logic of the game (the rules, its state in the abstract) from how it is visualised, I ask you to provide two visualisations.

The first one is a terminal view. It does not need to show everything, but should still be sufficient to get a general idea of what's going on for small games on very small maps.

You will use the letters given for each building / tile to represent the map. For instance

```
s  s   W   WWWWWW
          WWWW  WWWW
 vFTTTT   W W  WWWW
 FFTTTT
   TTTT    GGv CC
   TTTT     GGvCC K
Hv v
   H
```

represents a small village with a Town Centre, a gold mine to the east, guarded by a Keep, woods to the north-east, two soldiers to the north-west, a farm to the west, with a villager walking on it (presumably building it or farming), and two other villagers to the south-west, next to houses (presumably building the houses).

One must be able to pause the game with P, to scroll the map using ZQSD and directional keys (+Maj to go fast).

Pressing TAB will pause the game and open a webpage (generated HTML file) listing all units in the game and their stats (HP, position, etc) and current tasks, as well as any relevant data on the states of player AIs. (This is purely a snapshot of the current state; it need not be regenerated as the game progresses!)

For instance, one could learn that the soldiers have lost HP, and the farmer is currently building the farm.

You will not spend too much time making the page look *pretty*, but do give some thought to making it readable and searchable (collapsible sections) etc.

(12) Map visualisation: 2.5D

Separately from the terminal visualisation, you shall provide a graphical, top-down 2.5D (isometric, sprites-based) visualisation of the game map, in the style of AoE.

That means you can start or load a game either in terminal of GUI mode, or even switch between them on the fly, using the F12 button.

You may the use the sprites of AoE or other games, if you can extract or download them. It's a programming project, not an art class.

You will need a graphical framework for this task. Various possibilities include:

- PyGame, `https://www.pygame.org` is the most common choice among students for this type of projets.
- the Arcade Library `https://api.arcade.academy` `https://learn.arcade.academy`.
  Very fresh, but active; a few groups used it and had a good experience.
- TkInter,
- PySimpleGUI (with TkInter backend only; simpler to begin with)
- PyQt5 or PyQt6 (more powerful, more complex, external requirements[bn])
- wxPython, `https://www.wxpython.org`
  Bindings to wxWidgets, similar to PyQt.
- PursuedPyBear, `https://ppb.dev`
  This one seems very fresh out of the oven, and not documented.
- Kivy, `https://kivy.org`
- or anything that works with Python, really, I'm not picky, what matters is the result.

Test the different possibilities, and choose wisely.

Do not attempt to do this in full 3D.

(13) Minimap (2.5D):

The 2.5D view should include a minimap for fast navigation — this will be of great use during the demonstration.

This can take the form of an actual minimap always present in the corner as in AoE, or be a global, zoomed-out view activated through the M key. You can also have both, if time allows.

(14) Resources visualisation:

One should be able to see the stored resources (Food, Wood, Gold) for all players at all times. Additional information such as number of each unit types, etc could be useful as well. If all this obscures too much of the screen, use F1 through F4 to activate / deactivate some or all of the displayed information.

[bn] `https://pypi.org/project/PyQt5/`; cf. `http://doc.qt.io/qt-5/examples-graphicsview.html` pour de la documentation C++. C'est à adapter à la version Python, car PyQt5 est juste une bibliothèque de liens (bindings) vers Qt5.

(15) Save and load.

Those games can be long. You must be able to save the game state whenever you want, and load it without loss of information. Note that,if you have an AI, that includes what the AI knows about the world, and more generally its state of mind (planning an attack, game plan, etc).

You must be able to handle an arbitrary number of saves, manipulating them as standard files.

This will be extremely important for the defense, as you will not have time to play several full games during the demonstration. Instead you will load saved games, taken at interesting points of various games, to show off big battles, AI gameplan, etc.

F11 = quick save
F12 = quick load

Part VII

Python Projet: Practical Modalities V2

`pyproj_modalities.tex` **WORK IN PROGRESS !**

VERSION 2

69 Groups: size and composition

This project is done in groups of 5 or 6.

Groups will be determined “randomly”, not chosen by students. The aim is both to save time and avoid reproducing the usual cliques.

It is recommended that each group designate a “project secretary”, whose tasks include keeping track of who does what; he should have a global vision of the state of the project, and be able to inform me of it efficiently. He will probably be the main writer of the final report, so pick somebody who likes to write (French or English, I don’t care).

None of this should take much time, so only a *slight* reduction in overall programming or design tasks is acceptable for the group secretary.

Nor is he automatically the taskmaster, bossing others around. If that’s what you want, why not, but how you organise yourselves in the group is entirely up to you.

70 Evaluation

At the very end of the semester, each group will:

- Hand over a **short report**, a couple of days before the defence, summarising which requirements have been met, the individual contributions of each member, and the individual score of each member, agreed upon by group consensus.
- Present their work (15 minutes). This is referred to as "the **defence**" / "la soutenance". It is mostly a live demonstration of the work, with the help of a few *very specific* slides.
- Hand over the **git history** of the whole project (all source code and assets). Immediately before the defence.
- Hand over the **slides** used for the defence. Immediately after.

An individual mark shall be given to each student on the basis of all that.

Neither the report not the defence are marked in and of themselves. They are tools to communicate and assess the scope and quality of your work, and *that* is what we endeavour to evaluate.

71 Short report

The report must be a single PDF file, not exceeding 5Mb, titled "<group number> python report.pdf". Only *one* must be uploaded on Celene for the group.

It must contain the following things:

(1) **A *recent* photograph of each member of the group**, with the corresponding names, arranged so that all fits on one A4 page, vertical / portrait.

(2) **A screenshot of the 2.5D view** of your project, showing as many things as possible. Again, must fit on a single A4 portrait page.

(3) **A synthetic list of requirements**:

The project has a number of requirements, numbered in the document, each with a title in bold.

For each requirement, in order and using the same numbering scheme as me, and using the same titles in bold as me, you will state whether it is fully met (green), partially met (yellow), or not met (red), with a few lines of text explaining, where applicable, how your implementation deviates from the requirements.

(4) A detailed description of the contributions of each. It must be clear enough, in conjunction with the presentation, to enable the jury to mark the work of each student.

Each member must write a paragraph listing their useful contributions to the project.

The whole group must read every such paragraph, and a consensus must be reached that they are accurate. If a consensus cannot be reached — which will reflect poorly upon the whole group — a dissenting opinion must be written in a paragraph *below* the offending paragraph.

For instance, suppose that X claims to have done all the GUI, and Y and Z think they have meaningful contributions to it, and the rest of the group has not followed what happened in that part of the code.

Y and Z protest X's claims in his contributions paragraph when the group reads it. X refuses to modify his paragraph. Then Y and Z should add a dissenting opinion under X's paragraph, explaining what they disagree about. X cannot modify their dissenting opinion, anymore than Y and Z can modify X's contributions paragraph.

The final report must of course bear the imprimatur of all group members, but this is especially vital for those paragraphs.

(5) A zero-sum scoring of the contributions of each group member.

Il vous est demandé de pondérer la quantité de travail (utile, justifiable) de chacun par consensus du groupe. Par exemple: tout le monde a fourni le même travail, sauf X qui a travaillé 2 fois plus (fourni deux fois plus d'"utilité", pas seulement "remué deux fois plus vent") que les autres. Ces pondérations affecteront la note individuelle.

Qu'entend-on par travail utile, justifiable ?

Le plus facile à évaluer est la quantité de fonctionalités conçues et implémentées, pondérées par leur importance pour le projet.

Des aspects plus indirects ou flous de l'ordre du managérial ou "aide à la cohésion du groupe" sont à prendre en compte avec modération et beaucoup de prudence. Ne donnez pas un poids élevé à "ce gars a maintenu notre moral en faisant des blagues hénaurmes toutes les 5min" ou même au plus sérieux "il a joué le Boss et fait les diagrammes de Gantt de tout le monde" – sauf si c'était vraiment très utile, finement détaillé techniquement, et a vraiment eu une influence forte sur le groupe. Mais même dans ce cas, c'est un travail d'ingénieur qu'on note, pas de manager. S'il a fait les deux c'est un bonus, mais s'il n'a fait que le manager le score doit être faible, car ce n'est pas ce qui est demandé.

On note les "résultats", pas juste le temps passé. Quelqu'un qui bosse jour et nuit mais fait surtout des bêtises ou dessine 50 versions des icônes dont personne n'a besoin pendant qu'il reste des bugs urgents doit avoir un bas score. Quelqu'un qui fait ça alors que le groupe *insiste* pour qu'il fasse autre chose, mais est ignoré, doit avoir un très bas score.

Notons que “résultat” n'implique pas que, si ça n'apparaît pas dans le produit fini, ça ne compte pas. Le débuggage d'un bug complexe est un travail à valoriser dans le score, même si au final la partie du code qui a été debuggée a fini par être retirée du projet pour d'autres raisons.

La question est “au moment où le travail a été entrepris, était-il pertinent étant donné les connaissances du groupe à ce stade”.

Par exemple, un travail de recherche en profondeur est tout à fait valorisable, même si le résultat final n'est pas à la hauteur des espérances – mais évidemment c'est toujours beaucoup mieux s'il l'est !

De même, aider un camarade est aussi valorisable – là aussi dans la mesure du raisonnable.

Le rapport doit obligatoirement fournir la pondération de la manière suivante, obtenue par *consensus*[bo] au sein du groupe:

Chaque membre du groupe i est assigné un score / une pondération $p_i \in \mathbb{N}$, de manière à ce que le ratio $p_i/_{p_j}$ reflète bien la proportion de travail utile fourni par i par rapport à j. **Vous devez donc CONCRETEMENT rendre une liste d'association “membre du groupe $\mapsto$ score (nombre entier)”.**

(Note: ***use your full name, of the form “FAMILY Given-name”*** for this, not just your first name. My lists are sorted by family name, and I don't know by heart who “Bob” is. Bonus points (morally, at least) if you sort by family name.)

Par exemple, si l'on a $p_{Basil} = 2$, $p_{Cunégonde} = 1$, et $p_{Quetzalcoatl} = 8$ cela signifie que Basil a en gros été deux fois plus productif que Cunégonde, mais bon globalement Quetzalcoatl est un dieu et a bien porté le groupe, ayant fourni

$$\frac{p_{Quetzalcoatl}}{p_{Basil} + p_{Cunégonde} + p_{Quetzalcoatl}} = \frac{8}{11} = 73\%$$

du travail total – en supposant un groupe de trois.

Pour discuter des scores de chacun, il peut être utile d'utiliser des nombres de points “intuitifs”.

Par exemple, si $\sum_i p_i = 100$, i.e. on a 100 points au total à distribuer entre tous les membres, alors p_i représente la proportion du projet (produit fini ou travail de recherche valide) attribuable au travail de i, exprimée en pourcents.

On peut aussi partir de $p_i = 100$ pour chacun (tout le monde est égal est moyen) et ajuster en rajoutant des points aux membres moteur (par exemple, Machin est à 120%

[bo] pas *majorité*; ce n'est pas un vote. On en discute ensemble jusqu'à ce que tout le monde tombe d'accord. Voir plus bas si l'on n'y arrive pas.

par rapport au membre moyen, donc $p_{Machin} = 120$) et en enlevant aux membres qui ont été plus tirés par le groupe $p_{Truc} = 80$, en essayant de maintenir l'invariant $\sum_i p_i = 600$ (pour un groupe de 6), afin de préserver l'idée que 100 représente le score du membre moyen du groupe. (Même si ça fait chaud au cœur de dire "tout le monde est au dessus de la moyenne du groupe", mathématiquement ça ne marche pas. L'utilisation d'un score "zero sum" évite ce biais.)

Having $\sum_i p_i$ be a nice, round number is not strictly necessary, but it helps me check that I have copied the numbers correctly on my spreadsheet. In any case, tell me what $\sum_i p_i$ is supposed to be, so that it can serve as a checksum of sorts.

Note: ce score *ne doit pas être ajusté par le groupe pour prendre en compte des excuses, valides ou non*. Si A et B ont objectivement moitié moins avancé que la moyenne (notée à 100) alors tous deux doivent avoir un score de 50. Le fait que A a passé la moitié du semestre à jouer à Minecraft alors que B a passé la moitié du semestre à l'hôpital suite à une attaque de Vélociraptor (non-provoquée) ne doit pas intervenir. Les excuses valides d'ordre médical ou autres sont prises en compte *par le corps enseignant* à divers niveaux; en ce qui concerne l'auto-évaluation, ce n'est en aucun cas votre problème.

Note: this score must must be computed *with respect to the whole group*, not wrt. subgroups. For instance, one group was broken into 3 pairs with different tasks, and each pair was given an equal number of points to distribute between them. This is *not valid*, as it bypasses the hard work of evaluating the value and difficulty of each task.

Note: Consensus ≠ Vote:

A way some groups have "achieved consensus" in previous years is by averaging or summing scores given (sometimes anonymously !) by all members to each member. This has a chance of being a meaningful metric *only if* everybody is very well-informed about every other member's contributions. Otherwise it tends to produce noise, which tends to yield poorly differentiated scores. You may use such techniques if you wish, but it must be a mere *starting point* that is then *discussed* by the group until nobody is shocked by any mark.

I consider vote-based methods a bit of a "cheat code" when it comes to achieving consensus. Votes are a conflict-breaking tool, not a truth-finding tool. The only consensus truly achieved by taking a vote about X is the meta statement "the outcome of the vote about X, whatever it is, holds value." This actually says nothing about X or whether that outcome is correct.

That's fine if the question is "what colour should the bike shed be", because to the extent that there might be a right answer here, it's probably "whichever colour most people want" anyway. Even if many (or even all[bp]) people dislike the result, the most important thing is not to waste more time on the issue, and to avoid fighting over it. Let's just pick a colour and move on.

[bp] If you average the result of a vote on colour, you'll probably get a vomit-inducing khaki nobody wanted :-)

The situation is quite different when matters of fact must be decided, with real stakes and decidedly right (correct, accurate,. . .) or wrong answers. The scientific method does not proceed through votes.

Votes should only be used when there is a conflict to break, not before, and this only if there is a pressing need to coalesce to a decision. Whenever possible, a consensus obtained through rational discussion of all available evidence must be preferred.

Do not use voting as a clever tool to circumvent rational, possibly heated, discussion, and avoid having to actually formulate, defend, and change your opinions.

I want a consensus on the quality of the work of each, obtained through thorough discussion — and I know it can be hard — not *a consensus on a hack to avoid really having said discussion.*

The last thing I want is for a group (I take an extreme theoretical case) where everybody wants *all* the points to average anonymous votes and come up with the same score for everybody. . .

Si la pondération donnée par le groupe est manifestement fausse, c'est tout le groupe qui sera pénalisé.

Par exemple si le groupe dit "ben tout le monde a travaillé pareil, 100 à chacun", alors que pendant la soutenance on voit bien qu'il y a une personne qui sait répondre à toutes les questions, que ce soit sur la vue d'ensemble ou sur le fonctionnement du code, et une autre qui découvre la sujet et le logiciel le jour de la soutenance, ça va mal se passer. Soit tout le monde était tellement à la masse pendant le projet que personne n'a remarqué les grosses différences entre membres, soit le groupe est trop disfonctionnel pour avoir une conversation bilan honnête entre ses membres.

Les étudiants se plaignent souvent – et à raison ! – que les notes des travaux de groupe sont injustes; c'est l'occasion de rectifier le tir et, les enseignants n'ayant pas le budget pour une boule de cristal ou de chevaux de Troie dans vos ordinateurs, vous êtes encore les mieux placés pour le faire.

Si un consensus ne peut pas être atteint au sein du groupe (essayez, quand-même, parce que ça n'amusera personne de gérer ça et risque de pénaliser le groupe globalement) proposez plusieurs pondérations (e.g. celle soutenue par A, B, D, et E, selon laquelle C et F sont des glandeurs, et celle soutenue par C et F, selon laquelle ils ont tout fait) et nous en discuterons calmement.

Si le groupe atteint tant bien que mal un consensus mais qu'un (ou plusieurs) membre (ou sous-groupe) n'est pas satisfait, mais pas tout à fait au point de refuser entièrement le consensus (i.e. "J'accepte, mais pas content !", versus "Je refuse ! Révolution !"), ce membre peut joindre au rapport, sous le consensus, une *opinion dissidente* expliquant ce qui le chiffonne un peu dans le consensus tel qu'il est.

Le rapport peut également mentionner si le consensus a été obtenu facilement ou s'il a été difficile à négocier.

FAQ : comment la pondération donnée par le groupe affectera-t-elle la note individuelle, exactement ? Y a-t-il une formule ?

Nous noterons au mieux, dans un monde imparfait, avec les informations dont nous disposons.

There is indeed a formula that is being (somewhat) systematically applied. Following Goodhart's law [bq], I will not share it.

I would just note that in 2018–2019, the maximal difference between the worst and best marks within a group was 9 points out of 20. (The minimal intra-group difference was 0.1 points. The average intra-group difference was 4 points.)

The upshot is that you should not expect to get a good mark merely because other people are working and the end product is good. You have to contribute to it.

Conversely, if you are unlucky and end up in a disfunctional group, this does not automatically mean your mark will be terrible, so long as you can show meaningful contributions.

Overall, this system, while imperfect, proved much better, meaning much *fairer*, than handing out the same mark to everybody in each group, as was the case previously.

The cost is to force the group to confront and to *evaluate* the very real differences of skill and investment within the group, and confront one's autoevaluation to that of the group, which are very socially difficult exercises, without a doubt, but necessary ones.

72 The defence

Il y aura une journée de soutenances à la fin du semestre, où chaque groupe présentera très rapidement ses travaux, en fera une démonstration, et répondra à des questions.

Les modalités exactes sont comme suit:

72.0.1 Horaires de passage

Le planning sera en ligne sur Celene.

Deux salles seront réservées pendant les soutenances: c'est le jury qui passe d'une salle à l'autre.

De cette manière chaque groupe peut s'installer tranquillement pendant que le groupe précédent soutient dans l'autre salle.

[bq] *When a measure becomes a target, it ceases to be a good measure.*

72.0.2 Timing

Each group will have a 30min slot. You will have 15min to talk without interruption. The remainder of the time will be for questions / remarks.

Your talk will spend

- ◇ 2 minutes max on slides — In the next paragraph I'll tell you exactly what your slides are going to be. You will open with this.
- ◇ 13 min of live demonstration.

Be extremely strict on time. I shall interrupt you mercilessly the instant you go overboard either on the slides of the global duration.

You will be ready to switch back and forth between the slides and the demonstration during the questions.

72.0.3 The slides

Each slide will have your group number somewhere.

You will have *exactly* the following slides:

(1) A synthetic table of requirements met (green/yellow/red), like in the short report, but with fewer details so that it all fits on a single slide.

You will speak for about 1m30 **maximum** to summarise the important points, especially to make us understand the scope of the limitations regarding the yellow/red requirements.

(2) A single slide with a *recent* photograph of each member, their name, their individual score (as in the report — the scores must of course be the same as in the report !), and a few keywords as to the nature of their contributions.

About 30s maximum will be spent on this; the aim is the have a rough idea how uniformly (or not) the work was allocated before beginning.

We may use the slides as support during the questions phase.

72.0.4 The live demonstration

The demonstration must convince the jury that every requirement you claim to have met is indeed met.

Prepare it and rehearse it well in advance, like a theatre play, using functionalities of your project to help you, such as save and load. In your discourse, use the same keywords as in the requirements list, and quickly state the corresponding requirement number, to make it clear what requirement(s) you are demonstrating.

The demonstration **must be live**, we will not accept prerecorded videos or screenshot slide shows.

During the questions phase, you must be ready to let the members of the jury interact directly with your project.

While every member of the group must be present and answer direct questions from the jury, not every member needs to talk equally (or indeed at all) during your 15min. Again there is no mark for the defence itself; the aim is to convey the scope and quality of your work. Apportion the presentation time among your group in order to maximise the clarity of the defence.

Les démonstrations peuvent se dérouler soit sur l'ordinateur de la salle (celui connecté au vidéo projecteur), soit sur votre ordinateur portable personnel.

Dans tous les cas, évitez les temps morts dus à des contraintes techniques. Do everything on a single computer rather than spending time fighting the video projector each time you switch machines.

Vérifiez aussi la connexion de l'ordinateur servant à la démo avec le vidéo-projecteur avant le jour J.

Part VIII

Archived Project 2019–2020 & 2023–2024: Evolutionary Game of Life

73 Generalities

The aim is to create a "Game"[br] centred around a visual and interactive simulation of natural selection.

The serious purpose behind the game is to provide a refresher on evolutionary mechanics, both as a matter of general culture and as a prelude to the study of AI and metaheuristics in general.

In particular, genetic – and, more broadly, evolutionary – algorithms are a vast class of approaches simulating aspects of Darwinian biological evolution for the purpose of solving complex problems for which brute force or analytic approaches are unsuitable.

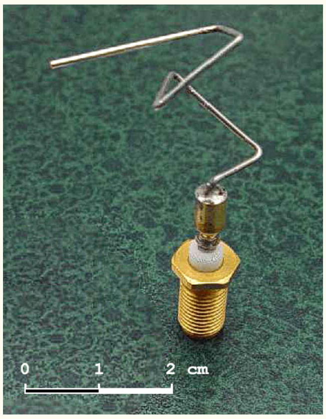


Figure 7: *NASA's ST5 satellite antenna, designed via evolutionary algorithms to meet the stringent and very specific requirements of the mission. It is the first such object to have been sent to space (2006).*

The results thus obtained can be very good, but are often a bit "alien-looking", as evolutionary processes, whether "real" or simulated, often defy human intuition, aesthetics, and engineering principles. (A look at deep-sea creatures should convey that quite well.)

For instance, the NASA's ST5 antenna in Fig. 7 was obtained by evolutionary processes. It performs quite well compared to human-designed antennae. The figure actually shows the

[br] I couldn't come up with a fun name for it, preferably with a pun or a gratuitous insersion of "INSA" in it, so I'm just calling it "the Game", for now.

second version of the antenna: the first was more tree-like. A minute change in mission parameters resulted in two completely different antenna layouts, where humans would come up with more incremental changes.

Of course, no human intervention was really needed to adjust the design the second time: the simulation was already set up, so entering the new constraints and pressing a button was all that was needed (plus a few days of computing time on a supercomputer).

A challenge with evolutionary processes (from both the points of view of computer science metaheuristics and biology) is how difficult it is to accurately predict the effects of a change in parameters, and how easy it is to come up with plausible-sounding "just-so" stories that don't pan out in the end.

The Game should provide a fun experimental platform to play with populations by altering several aspects of the creatures and their environment, and seeing how they react and evolve. It should generate high-quality graphs showing the evolution of those characteristics over time.

To be clear, while the process of *programming* the Game is in itself a way to teach evolutionary processes to INSA students, the final product should be a good tool to help present those notions to, say, high school students and such.

Compared to the related Clockwork (Part XIII[p425]: "Archived Project 2017–2018: Mon(s)tres") and Automata (Part XII[p416]: "Archived Project 2018–2019: Automata GUI") projects, this one is rather less open-ended and ambitious, as you are not required to actually design, set up, and deploy metaheuristics on a given open problem, but merely to simulate some very specific traits in a very specific simulation.

However, I shall be a bit more demanding when it comes to feature-completeness, stability, and usability of the final product. Furthermore, there is room for the more ambitious groups to considerably expand the scope of the project once they have covered the basics.

I give in this document some design choices that must be respected, so as to make it easier to compare the projects of different groups.

74 Specifications

The Game simulates a *world*. Since our divine powers are quite limited, our worlds shall be quite modest: a grid of size $N \times M$, with $N = M = 100$ by default (but this should be an easily modifiable parameter; every numerical value I give should be so; the values will be tweaked as your groups progress and experiment with them, so that they yield relatively stable populations.).

The Game should represent this world graphically, at the very least with a view from the top.

I strongly recommend setting up a terminal-only graphical engine *first* — take full advantage of the fact that the terminal can display colours and special characters. This should be fast and easy to make.

For the final product, additional support for a 2.5D isometric view – in the style of "old" games like Diablo I & II, Age of Empires I & II, etc – is also strongly recommended.

It should be reasonably cool-looking, without taking you too much time to make.

It is probably not a good idea to try and support full 3D rendering, unless some members of your group already have strong experience in that domain. I'm not even sure what a good library for that would be. Perhaps Panda3D?

The Game should of course have a GUI allowing the user to play with all the parameters of the simulation.

Here are a few possible choices of framework:

- PyGame, `https://www.pygame.org` is the most common choice among students for this type of projets.
- the Arcade Library `https://api.arcade.academy` `https://learn.arcade.academy`.
  Very fresh, but active; a few groups used it and had a good experience.
- TkInter,
- PySimpleGUI (with TkInter backend only; simpler to begin with)
- PyQt5 or PyQt6 (more powerful, more complex, external requirements[bs])
- wxPython, `https://www.wxpython.org`
  Bindings to wxWidgets, similar to PyQt.
- PursuedPyBear, `https://ppb.dev`
  This one seems very fresh out of the oven, and not documented.
- Kivy, `https://kivy.org`
- or anything that works with Python, really, I'm not picky, what matters is the result.

Note that the execution of **the simulation should be independent from the rendering**, as rendering takes a lot of processing power. Thus not only should the two run in separate threads, but it should be possible to simply activate and deactivate rendering at will while a simulation is running. Specialised "game-oriented" frameworks should take care of this fairly straightforwardly, without requiring you to do much system programming.

In the world, there live creatures; they are all named Bob. There are initially $P = 100$ of them.

[bs] `https://pypi.org/project/PyQt5/`; cf. `http://doc.qt.io/qt-5/examples-graphicsview.html` pour de la documentation C++. C'est à adapter à la version Python, car PyQt5 est juste une bibliothèque de liens (bindings) vers Qt5.

Bobs should be graphically represented by sprites that present as clearly as possible their individual attributes (speed, size, memory, etc). The representation should be as user-tweakable as possible to allow emphasis on whatever characteristics they are interested in at the time. For instance, I should be able to set things up so that faster creatures are bluer, and bigger creatures are redder, resulting in various hues of blue, red, and purple. Then I should be able to change it completely. Bobs' size should at least support a representation acting on the size of the sprites.

Each Bob spawns in a random cell in the world grid at the beginning of the simulation.

The simulation proceeds by time increments, or *ticks*: at each tick all Bobs perform an action, for instance walking to the next cell.

How complex their tasks and actions are depends on the number of characteristics which will be simulated.

74.1 Basic level: food hunting

Let's say that a *day* amounts to $D = 100$ ticks. Each day, a total quantity $F = 200$ of *food points* spawns randomly in the World, each containing $E_F = 100$ *energy*. There is nothing preventing several instances of food from spawning in the same cell, in which case the energy values add up.

Any food left uneaten disappears at the end of the day, just before the new food is spawned in.

Each Bob has an *energy* level, starting at $E_{spawn} = 100$, when they spawn, and capping out at $E_{max} = 200$.

This energy goes down by 1 each time they move, and they move at random each tick, going to an adjacent cell. Diagonal moves are not allowed (because they are at a greater distance; we'll talk about that later).

When a Bob finds food (is in the same cell as food), it instantly eats as much of it as it can, gaining all its energy. If its energy caps out in the process, there are leftovers. It can then stay stationary so long as there is food, but each tick spent stationary still consumes 0.5 energy.

If two Bobs access the same food at the same tick, one of them gets all, arbitrarily.

If a Bob's energy level ever falls at or below zero, it dies.

If a Bob's energy level caps out, it reproduces via parthenogenesis, spawning a new Bob in the same cell (several Bobs can occupy the same cell). The new Bob spawns at $E_{birth} = 50$ energy, while the "mother" Bob loses $E_{mother} = 150$ energy (putting her at $E_{max} - E_{mother} = 50$ energy).

Experiment with this and tweak the values to see what effects the different parameters have, until the values yield a stable population with interesting rates of births and deaths.

74.2 Velocity

At the basic level, all Bobs were identical. Let's change this, and start introducing individual characteristics that can be acted on by natural selection.

Let's start by adding *velocity* to the simulation.

The default Bobs all had a velocity of 1, moving 1 cell in 1 tick, at a cost of 1 energy.

Now, each Bob B has its own velocity, initially $B_v = 1$, but each Bob that spawns may mutate a bit in that respect: say that C is born of B. Then C_v should fall uniformly in the range $[B_v - \mathrm{Mut}_v, B_v + \mathrm{Mut}_v]$, where $\mathrm{Mut}_v = 0.1$ is the mutation rate for velocity. ($B_v = 1$ for initial Bobs)

Note that since velocity is not an integer quantity, handling it properly is not trivial. At each tick, the fastest moves first, and eats all it can. If a Bob has a non integer movement velocity, say, 1.6, then there are two ways to handle it:

- *partial actions:*[bt] on the next tick, it will move through an entire cell (eating all), and half a cell, eating 60% of the food energy there. On the next tick, it will eat the remainder (if nobody beat him to the punch), and move through another whole cell.
- *speed buffer:*[bu] on the next tick, it will act with a velocity of $\lfloor 1.6 \rfloor = 1$, and store the excess 0.6 into a "velocity buffer". On the following tick, the buffer will be consumed and added to the velocity before the process is repeated: thus this Bob will functionally have velocity $1.6 + 0.6 = 2.2$ for this tick, acting as though it were of velocity 2 and storing 0.2 in the buffer for subsequent ticks.

The buffer proposal is probably the best, as it avoids having to attempt giving a meaning to all partial actions: what is "partial reproduction", for instance?

Speed has a cost, however. We are going to follow physics roughly, here. The kinetic energy of an object is given by

$$\frac{1}{2}mv^2 ,$$

so, following that, on each tick, B will consume $B_c = B_v^2$ energy to move. For instance, double the speed means quadruple the energy cost. (It's a bit more complicated in real life, because that only accounts for the cost of acceleration, but this formula is sufficient to introduce clear, intuitive trade-offs in our simulation.)

To avoid Bobs becoming immortal simply by virtue of not moving at all, we say that a Bob consumes, at a minimum, $E_{\mathrm{tmin}} = \frac{1}{2}$ energy per tick. The actual energy consumption is therefore

$$B'_c \;=\; \max(E_{\mathrm{tmin}}, B_c)$$

[bt] This was the original proposal.

[bu] This is the revised proposal after discussions with students.

This applies for all other versions of B_c below.

Run the simulation and see how speed evolves over time.

The impact of each of characteristics such a this should be configurable in real time – for instance if I want to remove speed from the equation I should be able to set $\text{Mut}_v = 0$ whenever I want.

74.3 Mass

Now let's add the possibility for Bobs to evolve some serious muscle mass. Mass will be, like velocity, a genetic characteristic. Our default Bobs had mass $B_m = 1$. Like velocity, mass is genetic, and mutates in the same way. We do not model a cost on birth, however, mass has a cost on mobility. Following the kinetic energy model, mass multiplies the movement cost on each tick: a Bob will therefore consume

$$B_c = B_m B_v^2$$

energy on each tick for movement.

Mass should be reflected in the size of the sprites in the graphical representations. Note that mass is proportional to volume, and volume is proportional to the cube root of mass, so the size of the sprites should scale like $\sqrt[3]{B_m}$.

What is the benefit of being bigger? You can eat (much) smaller creatures. If Big Bob B and small bob b meet (are in the same cell at the same tick, even if one of them is only "partially" there due to non-integer speeds), and the difference in size is important ($\frac{b_m}{B_m} < \frac{2}{3}$) then, Big Bob can, and will, eat small bob. Small bob dies. Big Bob's energy level B_e is updated according to the equation

$$B_e := B_e - \frac{1}{2}\frac{b_m}{B_m}b_e + \frac{1}{2}b_e = B_e + \frac{1}{2}b_e\left(1 - \frac{b_m}{B_m}\right)$$

The idea is that the fight takes some energy out of Big Bob, although less and less the bigger is he in proportion to small Bob, and he gains up to half of small bob's energy level. In this model, he always gains *some* energy from eating another Bob.

It should be easy to change this model within the code.

If $\frac{b_m}{B_m} \geqslant \frac{2}{3}$, they ignore each other.

74.4 Perception

Now that there are predators and prey in the world, it help to be able to avoid the ones and pursue the others.

Each Bob initially had a perception score $B_p = 0$ meaning that they are blind as bats – but without any cool echolocation either. B_p measures of course the distance at which they can determine whether a cell contains noting, food, or Bobs. It is the radius of the circle of detection.

This is a genetic characteristic. Unlike speed and mass, we are going to keep that value integral, and mutate it by 0 or ± 1, equiprobably.

Since we are in a discrete world, some care will have to taken to compute those circles properly. Given that diagonal moves are disallowed, the notion of distance that matters is not actually that of the standard Euclidean geometry (ℓ^2 norm), but the Manhattan distance (ℓ^1 norm); in dimension 2, which is what we shall work in, it is computed as

$$d_1(A, B) \quad = \quad |x_B - x_A| + |y_B - y_A|$$

Compare to the Euclidean distance:

$$d_2(A, B) \quad = \quad \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}.$$

Circles in Manhattan geometry actually look like squares do in Euclidean geometry – with a $\frac{\pi}{4}$ rotation, so, pointy-end up.

At $B_p = 1$, all adjacent cells are detected.

Up till now, The Bobs' decisions have been purely random. Now, whenever they detect food, they make a beeline for it. Unlike in Euclidean geometry, there are many different shortest paths to take. Take the algorithm: so long as you're not there, reduce either your x or y-distance, with same probability.

Of course, if at any time Bob detects a new food source, while on its way to another, it should reevaluate its plan and make a beeline towards the closest one. If it detects several food sources, it should favour the bigger ones.

By food sources, we mean both spawned food and unlucky smaller Bobs. Since how good a food source a prey Bob is depends on its energy level, and that a prey Bob may run away, a predator Bob will always favour stationary spawned food to other Bobs.

When several prey Bobs are detected, the smallest ones will be favoured.

When a Bob detects a larger Bob, it moves to maximise distance between the two. When it detects several larger Bobs, it moves to maximise the distance to the closest one, and, *ceteris paribus*, to the others.

When a Bob detects prey and predators simultaneously, its prey behaviour overrides its predator behaviour: survive first, hunt second.

Perception cost is not affected by either mass or velocity. It does require eyes and a big brain, though, which requires constant energy expenditures. (A human brain consumes about 20% of the body's total energy.)

Let's say that for each point of perception radius, there is a flat $\frac{1}{5}$ penalty (again, the GUI must allow this to be modified at will) to energy each tick. The consumption becomes:

$$B_c = B_m B_v^2 + \frac{1}{5} B_p$$

74.5 Spacial memory

A big brain has other uses. An important one is to remember the existence of stuff that's not visible right now.

Currently, unless they are making a beeline for some kind food, or running away, Bobs have a $\frac{1}{4}$ probability of going back where they just came from, which is not optimal to find spawned sources.

Now, let's introduce the inheritable characteristic *memory space*. Each Bob B has currently $B_{\text{mem}} = 0$ memory points. Again, this will remain an integral value, and mutate by $0, \pm 1$, equiprobably.

A Bob has several (mutually exclusive) ways to use its memory points:

A Bob can remember the $2 \times B_{\text{mem}}$ cells it visited, and hereafter avoid them, unless doing so would lengthen a path to food or escape.

With higher priority than that, a Bob can use its memory points to remember a place where it saw spawned food not currently in its view. This can happen if a Bob detects two food sources at opposite ends of its perception, and goes towards the one, therefore losing sight of the other one. Or if it sees food while running away from a predator.

It uses one memory point to remember one food location (therefore forgetting two of its oldest visited locations). It remembers the respective energy levels of the food.

It only uses this point upon leaving sight of the food. The point is freed the instant a remembered food comes into sight again.

While not pursuing food currently in its view (that has the priority, even if it remembers a larger food source farther away), or running away, a Bob will make a beeline for the largest food source it remembers seeing.

The cost is again a flat penalty per point.

$$B_c = B_m B_v^2 + \frac{1}{5} B_p + \frac{1}{5} B_{\text{mem}}$$

74.6 Other characteristics

Each group should invent, define, and implement a few other characteristics; put your own spin on the subject. They should be explained in the report, and well-specified.

What I outline explicitly in this document should be construed as the bare minimum as expect from each group. Once that basis is assured, be creative. Your features must be deactivable, so that the jury can compared your projet to other groups under the same constraints.

The is best achieved in the same style as the features listed above: set things up so that your features are governed by constants that, for some values (0, 1,. . .) remove the feature from consideration.

I would be delighted to see additions tackling the evolution of altruism, or aggression strategies (hawks vs doves), etc.

It would be advisable to plan the more ambitious "freestyle" features somewhat in advance, and to consult me before investing significant time in them.

74.7 Sexual reproduction

So far, Bobs have reproduced solely by parthenogenesis. Now, let's add sex to the mix – though we will consider that Bobs are hermaphrodites, like snails, and not attempt to distinguish Bobs and Bobettes.

On top of still having the option of parthenogenesis, when two Bobs B and C meet, and don't eat each other, and have high energy levels $B_e, C_e \geqslant 150$, they mate, losing 100 energy each, and creating a new Bob D at initial energy $D_e = 100$.

Note that, compared to parthenogenesis, more total energy is invested into the new Bob, and it starts out with a higher energy level, but less is required from *each* parent. This is meant to model the advantages of shared parenthood.

Of course all those values should be easily modified parameters.

D has all his genetic characteristics set to the average of its parents, and then mutation is applied as usual.

For those characteristics, such as perception, which are integral, the cleanest way to handle this is probably to actually store floating point values in the "genetic code" of Bobs, and round them to the nearest integer whenever you actually use them — but not during reproduction. That way, if B and B' reproduce, with $B_p = 1$ and $B'_p = 2$ then their offspring C will have, before mutation, $C_p = 1.5$. In practice, its perception shall be rounded to 2. But if C then reproduces with B, the offspring D will have perception 1.25; in practice, it will be rounded to 1. And so on.

Sexual reproduction can be toggled on and off; parthenogenesis as well.

It will be interesting to see the effects of this addition on the speed of adaptation to changing conditions.

75 Save & Load

The user must be able to save to file, and load from file,

- the parameters of the game (N, P, etc)
- the state of the game, that is to say, the state of each and every Bob in the game, including their perception, memory, strategy, etc.

Those two aspects must be handled separately; one may want to change the parameters in bulk while the game is running. Probably the "state" save should also include parameters.

Part IX

Archived Python Project 2022–2023: Avé INSA!

The aim of the project this year is to implement a city-buildling (CB) game in the style of the *Caesar* game series, which I take to include *Caesar I, II, and III, Pharaoh, Zeus, and Emperor* by studio Impression Games, published by Sierra at the turn of the millennium.

Specifically, we shall aim to produce little clones of (a subset of) *Caesar III* with the Augustus engine, (or maybe, for some groups, Pharaoh, Zeus, or Emperor). We shall endeavour to keep the subset thereof which is cloned as close as possible to the original.

The features and level of polish expected will vary enormously between the FISA and FISE versions, with the FISA version being a *much* shorter project of smaller scope — five weeks in total, I believe, ending mid-December.

FISE have, approximately, until the end of January.

76 Resources and brief overview of game principles

76.1 Resources

- Teacher-provided resources: `http://files.vhugot.com/Restricted/Caesar3`
- Game on Steam: `https://store.steampowered.com/app/517790/Caesar_3/`
- Game on GOG: `https://www.gog.com/en/game/caesar_3`

- The Augustus engine: `https://github.com/Keriew/augustus`
- Fan site: `https://caesar3.heavengames.com/`
- FAQ on `c3_model.txt` (modding & game data extraction):
  `https://gamefaqs.gamespot.com/pc/63635-caesar-iii/faqs/14466`
- Sprites Extractors:
  `https://github.com/bvschaik/citybuilding-tools`
  `https://github.com/lclarkmichalek/sgreader`
  `http://pecunia.nerdcamp.net/downloads/utilities` (.exe)
  Take note that the build instructions need updating.
  Fill in `TARGET = sgreader` in `.pro` file and use `qmake-qt4`.
- References for housing levels:
  `https://impressionsgames.fandom.com/wiki/Housing_(Caesar_3)`
  `https://web.archive.org/web/20060713205809/http://caesaralan.co.uk/strategy/houselevels.html`
- Blog by developers of a similar game, with interesting technical insights:
  `https://nepos.games/nebuchadnezzar/blog`

76.2 Running the game

This game is a bit complex, and I don't think there is a good substitute to playing its tutorial missions and reading the game's manual and the in-game documentation.

After obtaining the game files through whatever means suit you most, note the welcome presence of the `Manuals` folder. Then install Augustus following the instructions on github. We shall use Augustus as the reference implementation.

The sprites used by the game are minuscule on today's high-pixel-density screens. In the options, set "Display scale" to something suitably high. On my 1440p display, I set it to 205%; on my 1080p, 155%. Avoid round values like 200%, because that obviates the need for smoothing, and make everything more pixelated than it needs to be.

Leave all gameplay options on the defaults settings, which are closest to the original game. We shall focus on the buildings and mechanics from the original game, not those introduced in Augustus. An exception to that is the Roadblock building (present in later games, i.e. Pharaoh for roadblock, and Emperor for selective roadblock) and, as you wish, global labour pool (idem, starting with Zeus).

76.3 Overview of game principles

The general principle is this: they are CB games inspired by historical civilisations. For the sake of this overview, we shall focus on the Roman civilisation and the mechanics of Caesar III, which is the main target of this project.

The player starts with an empty terrain, and can buy buildings to place on the map. The aim is to build a bustling city.

The challenge revolves around the evolution of houses. When you buy houses, they are not immediately built. First, the terrain is marked as residential, and if the city is not unattractive, people will come from outside the map, to erect small tents and live there. "People" manifest as *walkers*, that appear at a specific point at the edge of the map, and follow the road to their destination.

Tents provide room for few people, but those people have few needs, and you can put them to work. When you place a building that provides a service, it requires a number of workers. The unemployed in your population, if any, will apply to work. When the building has enough workers, it provides the service (possibly less efficiently if not fully staffed).

To provide a service, a building usually generates one or several walkers, who follow the roads randomly for some distance, then go back to base. While they walk, they provide the service to nearby houses / buildings.

For instance, a market will generate a merchant; when the merchant walks near a house, it replenishes the house's stores of whatever they need, and depletes the stores of the market. Likewise, a doctor's office generates a walker (doctor) that reduces the probability of plague in serviced housing, and so on.

There are a few exceptions, where buildings directly influence an area around them, with no walkers as intermediaries. Gardens positively affect desirability in a small radius; wells make water available to nearby housing, etc.

The gameplay structure is as follows:

◇ set some houses, or rather *tents*; the people populating them can staff a few basic services.

◇ once some of those services are available to the houses, they become more attractive, and *evolve*, for instance the tent may become a mud hut, which occupies the same space, but has room for more people.

◇ those new people can staff new services and the industries they depend upon, thereby making parts of the city even more attractive, and causing the residences to evolve again.

◇ the process repeats with a set sequence of needs, until the tents have been replaced by luxurious and dense residences, assuming all goes well.

All does not always go well. Less than perfect coverage means some parts of the city could catch fire or fall victim to plagues, or go without some basic resource and regress. If some crucial industry is disrupted, then a need may no longer be met once stores run out. When that happens, housing may brutally devolve into lesser states, and evict many citizens for want of space.

And if *that* happens, then suddenly the city may no longer have enough workers to staff some *other* essential industry, thereby leading to a second wave of devolution, then a third. Homelessness favours criminality, which causes more fires and vandalism if the police coverage is inadequate. All of this makes the city unattractive, causing more citizens to leave and preventing new settlers from immigrating even if there is the space to do so.

A badly built city may collapse like a house of cards at the first crisis, unless judicious emergency actions are taken (like shutting down luxury industries to ensure a steady supply of workers for the necessities).

Building a city well means taking a lot of constraints into account when placing buildings, to ensure that all required walkers have predictable paths and that there is a level of redundancy.

It would be a shame to see a city collapse because a fireman chose to go left three times in a row, thereby allowing a fire to kickstart a death spiral for the city. Since the luxury villas house far fewer people than the high-density housing they replace, even a successful transition towards a patrician neighborhood, when badly-handled, may kickstart shortages that may spiral out of control!

And then there is the money; you spend it to make buidings, pay your workforce, throw festivals, import what you can't or would rather not produce yourself, and you get it back through taxes and trade with nearby cities. If you run out of it, it's bad; if your taxes are too high, the people will grumble and eventually leave.

There are, in Caesar III, dozens of building types and services.

Quoting from the manual:

> *To reach its highest level, housing needs access to a nearby market supplied with four different foods, pottery, oil, furniture and two varieties of wine. Regular visits by workers from a bathhouse, a doctor's clinic, a barber's shop, a priest of each god's temple, and representatives of a school, academy, library, theater, amphitheater, colosseum and hippodrome are also required. If you can supply all of these goods, and access to all of these buildings, then reaching the highest values is simply a matter of enhancing desirability. Right- click on housing to discover why its growth is stagnant. The panel that appears shows what the house lacks, or the nearest negative influence on its desirability.*

Furthermore, there is a sophisticated administration interface (that gets even more sophisticated in later games!) to control the priorities of the various building types in terms of staffing: who gets the available workers in case of shortage, and in what proportions? Buildings may also be "paused" on a case by case basis. It all gets rather complex.

This is, in the end, an exercise in Python programming, not in game design. Keep things relatively simple, at least at first. If you can make the game work with a couple of services and house types, then you can theoretically make it work with a dozen; the technical challenge is essentially the same. Only take the time to refine the "game" aspect once the technical ones are solid.

Stick with Caesar III and its mechanics, with few exceptions like maybe convenient features from later games such as selective roadblocks, and implement them in order: basic needs first, and the industries required to sustain them. Use the game's sprites for graphics. Do not skip steps in those needs, or oversimplify them. This will simplify the comparison of how far different groups went.

If I see patrician villas, I should know that you have implemented grapes and wine industries and distribution, because that is a requirement in the original game.

However, you may simplify or omit things outside of the "main path" of the game, which I consider to be housing. For instance, the mood of individual gods, or Caesar's favour, may be done without. The need for water or oil may not.

The precise needs for different levels of housing are given in `c3_model.txt`.

77 Expected features

Features marked as **A** or **E** are required of FISA or FISE, respectively. Features marked as **a** or **e** are partly required, or optional but recommended.

The main difference between what's expected of FISE and FISA is the complexity of the game mechanics. The the game works with just one type of houses and one service, it's enough for a good project in FISA, but will be considered the bare minimum in FISE.

The text may offer more nuance on what should be implemented and suggest simple approaches to satisfy the requirement without too much pain, or more advanced approaches for those who want to show off their skills.

Both simple, conservative choices, implemented very competently, and more ambitious ones, even not *fully* realised in the implementation — but realised enough to showcase their potential — can result in excellent marks. Choose tactically.

Any feature not marked as required is entirely optional.

(1) **AE: Done in Python 3.**

In case there was any doubt, this is a *Python* project. . .

You may need to use some older version of Python (3.7, 3.8,. . .) for compatibility with some libraries or dependencies that may not be updated very frequently, like PyGame.

(2) **AE: GUI framework of your choice**

The game should of course have a GUI.

You may choose

- PyGame, `https://www.pygame.org` is the most common choice among students for this type of projets.
- the Arcade Library `https://api.arcade.academy` `https://learn.arcade.academy`.
  Very fresh, but active; a few groups used it and had a good experience.
- TkInter,
- PySimpleGUI (with TkInter backend only; simpler to begin with)
- PyQt5 or PyQt6 (more powerful, more complex, external requirements[bv])
- wxPython, `https://www.wxpython.org`
  Bindings to wxWidgets, similar to PyQt.
- PursuedPyBear, `https://ppb.dev`
  This one seems very fresh out of the oven, and not documented.
- Kivy, `https://kivy.org`
- or anything that works with Python, really, I'm not picky, what matters is the result.

Test the different possibilities, and choose wisely.

Based on previous experiences, and the nature of this project — especially the RTS aspect — I think PyGame is the safest choice for this project, in that most groups in previous years have used it and done satisfactory things.

PursuedPyBear seems too fresh.

PyGame is far from perfect, so I'm exceedingly interested in hearing from you if you make something else work.

I haven't seen wxWidgets used in projects, but it ought to be nearly as mature and usable as PyQt.

(3) AE: Graphical representation of the map.

The map must be represented graphically, ideally in a near identical way to Caesar III if possible.

The view may be 2D, top-down, using sprites (by far the simplest option — you can even have a terminal-based one as a fallback and to develop the logic before a more sophisticated rendering engine is available), isometric 2.5D (as in Caesar III; similar views have been consistently achieved by most groups year after year in similar projects, so it is really recommended), or full 3D (do not even attempt this unless you

[bv] `https://pypi.org/project/PyQt5/`; cf. `http://doc.qt.io/qt-5/examples-graphicsview.html` pour de la documentation C++. C'est à adapter à la version Python, car PyQt5 est juste une bibliothèque de liens (bindings) vers Qt5.

already have *considerable* experience with a suitable framework. Tests have been done last year with Ursina and Panda3D, but neither has been found suitable for use in this project).

If you go the 2D route, note that fan sites have a collection of 2D sprites which can probably be downloaded and used.

For 2.5D, the original sprites of the game should be used — note that, for copyright reasons, this severely restricts the diffusion of your project.

(4) **aE: Sprite Upscale x2**

If you use the original sprites of Caesar 3, you will notice that they are very small by today's standards, and can't really be used at native resolution.

I recommend using an AI upscaler such as `waifu2x-ncnn-vulkan`, and whatever image manipulation operations you deem necessary, to double their size.

This has the added benefit, from my point of view, of making sure you cannot simply use the Julius and Augustus projects as rendering backends, since they won't work on differently-sized sprites.

(5) **AE: Save and load. Pause.**

Those games can be loooong. You must be able to save the game state whenever you want, and load it without loss of information. That includes the state of every walker and building etc.

This will be *extremely* important for the defense, as you will not have time to play several full games during the demonstration. Instead you will load saved games, taken at interesting points of the life of your cities, and from those specific points you can add or remove buildings, shutdown or activate industries, and thus demonstrate the impact of those actions upon your city.

The quality of your defence plays a huge role in your final mark, not because it is evaluated as such, but because it determines what the jury understands of your project. Consequently, you must anticipate features of the game which make showing it off easy. Save and load is one of those. Being able to pause the game would be another. You may think of other things in the same vein. Do not hesitate to implement them and discuss it with me.

(6) **AE: Faithful Caesar III mechanics and sprites.**

I would really appreciate it if you would all use the same sprites as the original game, as much as possible, and implement clear subsets of the mechanics of the game. This is not to stifle your creativity, but to facilitate quick understanding of what you are doing.

If all groups use the same kind of units, buildings, and mechanics, with somewhat the same balance, it becomes easier to understand at a glance what's going on, and how two groups differ. This is especially crucial during the defence.

You have a very short time in which to show off your work — if much of that time is spent explaining that your drones need to mine unobtainium to build starships, rather than mine clay in order to make pottery, in the end that is time not spent showcasing how many features you support and how fast the game runs, which is probably not to your advantage.

Instead, if everybody uses the same basic concepts and vocabulary, some level of background knowledge can be assumed and we can quickly concentrate on the aspects that truly differentiate the groups. Water and oil and clay all play the same role for everybody, because everybody copies Caesar III, and we all know what we are looking at.

That being said, if you absolutely yearn to make a space-themed CB game, or a fantasy-themed game, or a horror-themed game (please, not that one), I will not explicitly *forbid* you from doing so, but be aware that you will need to communicate your mechanics extremely efficiently, and convince the jury that they are more or less equivalent what is expected.

Either way, sprites should be as recognisable as possible. I shall try and provide them in a convenient package.

Of course, as stated before, the point is not to re-implement *everything*. It is to implement a well-chosen subset, avoid having to reinvent game balance from scratch, and facilitate communication. Start with the basic aspects of the game, following the tutorial, and taking values (price of buildings, number of workers required, distance walked by the walkers,...) from gamefiles and wikis. Tweak only when you need to. Again, this is not a game design course; the game is a pretext for coding.

Do not needlessly deviate from Caesar III mechanics and values: this is a complex game, whose balance rests on carefully tweaked numbers. Balancing all the constraints may simply not be possible or fun if some numbers change. Experimenting with many sets of gameplay parameters is not a tactical investment of your energy.

Again, if that tickles your fancy to be original and go with different units, I'll *technically allow* you to do that, in the same sense as I'll allow you to shoot yourself in the foot. At the end of the day, it's *your* foot. I just don't recommend that course of action.

If you simply *must* differentiate your project from the other groups, perhaps targeting a faithful copy of another game of the same series (Pharaoh, Zeus, Emperor) would be the safest bet.

The main risk I see here is that it is still more explaining to do to the jury, in particular the "*candide*", who might just be getting familiar with C3, but not with the other games. The second risk I see is that I have fewer resources to provide to help, and there is nothing equivalent to like Julius or Augustus for those games. Those risks are still manageable overall.

Regardless, if you plan on deviating from C3, please talk to me about it during the lab classes.

(7) AE: *Real-Time* Aspects.

Note that such a game is inherently "real time" in the sense that there are no discernible turns. The many buildings and citizens and walkers of the city live their lives simultaneously, and the city must remain reactive to any action taken by the player at any time.

This real time aspect can be tricky to implement correctly. You will need a well thought-out architecture and probably some form of concurrency (multithreading, multiprocessing,. . . Sec. 31[p219]: "Parallelism and concurrency") which you will have to figure out on your own, since that is mostly taught during the second semester.

You can also try doing most everything in one thread, but it has its own difficulties: if the main game loop is not fast enough, you will have terrible input lag, dropped inputs, and more.

Libraries like PyGames usually offer examples and propose standard ways of handling inputs, event queues etc. By starting from pertinent examples, you should be able to get things working.

I thought this aspect ambitious last year, when I gave my first RT-based project (the previous ones were either turn-based or radically different types of work), but to my surprise and delight, no group had much trouble with it.

FISA:

Keep the game very simple and lean so that any architecture can be made to work. For instance, if animating your sprites becomes a problem, then just don't animate them.

(8) aE: Walker visualisation

Caesar III offers a variety of views of your city, each focused on one industry or service. For instance, one view lets you see water coverage, and related walkers, while hiding all other aspects. You must (FISA: should) implement such a view for each of the aspects you introduce in your project. Again, this is essential (FISA: useful) for a good defence, as it is otherwise difficult to quickly showcase one particular mechanic — especially if you have implemented many of them.

(9) e: Variable Speed

Saved games help you show off specific game configurations, but to really get an idea of the flow of the game, it is best to see the city evolve. But we don't have time for that in a defense.

Thus, it would be helpful if the speed of the game could be changed — specifically *increased* — so that we can see the town react on fast-forward.

Of course, how fast the game can run when at full speed will depend upon both your machine and the quality of your implementation. Thus, for the defense, you can also prepare time lapse videos showing the relevant points.

Note that this can be complex to get right, as it divorces real time from game time. If you want that feature, you should plan for it from the start; adding it *post facto* is liable to break things badly.

(10) No procedural map generation.

Caesar III has fixed maps, designed in an editor. For that type of game, where one stays hours on the same map, and the exact shape of the map is an important factor to the difficulty, this is enough. There is no need to implement procedural map generation.

(11) e: Trading

Have neighbouring cities with which to trade specific goods, as in CIII.

(12) e: Administration

Implement some degree of global control over your city to adjust tax levels, industry priorities, visualise service coverage (this is the most important part, as discussed previously), and so on.

(13) Disasters and Invasions/War

There are *some* combat mechanics in CIII; you can implement them if you want.

(14) Gods and favour of Caesar

Beyond providing services, temples also serve to appease the gods, who might send a disaster your way if they feel neglected. Likewise, Caesar may attack you if you mishandle your treasury too much.

It would be interesting to implement those aspects, but they are not at all essential. Focus first on more visible services and industries, and their walkers.

Part X

Archived Python Project 2021–2022: Age of Cheap Empires

The aim of the project this year is to implement a strategy game in the style of the *Age of Empires* game series — specifically, the first two. The first one being simpler, it seems a better starting point.

The features expected will vary enormously between the FISA and FISE versions, with the FISA version being a *much* shorter project of smaller scope — five weeks in total, I believe, ending mid-December.

FISE have, approximately, until the end of January.

78 Resources and brief overview of game principles

78.1 Resources

If you are familiar with those games, splendid. If not, it should go without saying that you do not need to buy or play any of these games. You can get all relevant information by watching videos of AoE I and II on the internet, and consulting the wiki or the official site:

- https://ageofempires.fandom.com/wiki/Age_of_Empires
- https://ageofempires.fandom.com/wiki/Buildings_(Age_of_Empires)
- https://ageofempires.fandom.com/wiki/Units_(Age_of_Empires)
- https://ageofempires.fandom.com/wiki/Technology_(Age_of_Empires)
- https://www.ageofempires.com/tech-tree/greek/

The wiki shall serve as a reference document on technologies, unit types and statistics, etc.

For a more hands-on look on that genre, you can install 0 A.D., a game originally made as a mod for Age of Empires II, now entirely free software. You can install it on most modern Linux distributions with a single command; for instance, for any Debian-based system,

```
sudo apt install 0ad
```

will do the trick. 0 A.D. is its own thing, with its own unique mechanics, but remains extremely similar in look and feel to AoE I and II, so it can give you a general idea of how a game like that plays.

For large screens, you may want to create a file `~/.config/0ad/config/local.cfg` containing a line of the form `gui.scale = "1.875"`, to scale up the GUI elements, including the fonts — here by a factor of 1.875, to replace by what works for you.

I am hoping that enough students will have played an AoE game before — or at least *some* similar RTS game — that I can affect one per group, and that they can serve as "RTS experts" within the group. If not, too bad, but it should not be much of a problem.

78.2 Overview of game principles

The general principle is this: they are real-time strategy games (RTS) inspired by historical civilisations and battles, where players control units on a map. Several players share a map; the last one standing wins.

You start with villagers and a Town Center.

Villagers can acquire resources: they can forage, hunt, fish, or farm for food, fell trees for wood, and mine for gold or stone — all of which takes time. Once they drop resources at the town center (or other compatible drop points), the player can use them.

For instance, for 50 food, the town center can create a new villager — a process that takes 20[bw] to 25[bx] seconds, and during which the town center cannot create other units or research technologies. A technology is researched at a building, and changes some aspect fo the civilisation, generally for the better: given access to new buildings or types of units, improve existing units, etc.

Villagers can also build new buildings[by] — which costs wood, and even stone for things like castles. They are poor fighters, though. If you want a fighting force, build a Barracks, and train soldiers there. They cost food, along with wood for archers, pikemen etc, and some gold. The more powerful units are usually more gold intensive. Cavalry units (the horse is not treated as separate) are also quite food-intensive. There is a kind of rock-paper-scissors relationship between the different unit types. For instance archers are generally good against infantry, that tends to die before getting to them and be vulnerable to shoot and run tactics, but weak against cavalry. Some units have special bonuses against others; for instance, in AoE II, the dirt-cheap Pikemen infantry line has massive bonuses against cavalry, reflecting the historical fact that cavalry charges fare poorly against rows of long, pointy sticks.

There is a population cap: 50 in AoE I, 200 in AoE II — and I believe it can be set up to 500 in AoE II Definitive Edition, but I never heard of anybody plays with those settings. The

[bw] AoE I

[bx] AoE II

[by] In AoE, military units cannot build anything — with the exception of the Sicilian Serjeant, a unique unit introduced in 2021.

In 0 A.D., from what I have seen, military units are also builders — and only they can build military buildings. I'd prefer sticking with AoE villagers in your game.

difficulty of the game is to balance your economy (villagers gathering resources) and your army.

There are limited resources on the map — how much exactly depends on the map type and on random generation. Most maps include a gold and a stone mine not far from your starting position. Others you find by exploring the map; or other players find them before you. They contain a fixed quantity of the resources; when they are depleted, that's it. Tree don't regrow — though there are a lot of them, — nor do the berry bushes, or the animals.

Farms can be planted, that generate a lot of food (though still a fixed quantity) for a modest wood investment. They are the main source of income starting midgame.

In a typical game, stone and gold run out late-game. Wood doesn't, typically, and if wood doesn't, neither does food. But it may on some map type: in a map comprised of very small islands, there may not be a lot of wood available, which constrains the use of a large navy. . .

The games offer a fairly large number of civilizations. They are, however, *symmetric* RTS, since the all civilisations share the same units and technologies, with only minor variations (and one or two unique units in AoE II). This is opposed to asymmetric RTS like Starcraft, with fewer factions, each with entirely different building, units, resources, and strategies. [bz]

In our small project, we can pretty much ignore the "multiple civilisations" aspect. One is enough, provided it has access to enough unit archetypes — e.g. infantry, cavalry, archery, siege, priests/monks[ca], perhaps a navy — to enable different strategies. At least two of those would be nice.

The most important technology available to a player is "going up an age". AoE I for instance has the Stone Age (starting age), Tool Age, Bronze Age, and Iron Age. Moving up the ages is expensive, but essential, as in unlocks a large number of buildings, units, and technologies, as well as upgrading existing buildings and units.

You may include such a mechanic if you have time, but it is not absolutely essential. This is, in the end, an exercise in Python programming, not in game design. Keep things relatively simple, at least at first. If you can make the game work with one age and two military unit types, you can also make it work with four ages and five types: the technical challenge is essentially the same. Only take the time to refine the "game" aspect once the technical ones are solid.

[bz] AoE IV is not released yet, but it seems to be designed as an asymmetric RTS, to a large extent. Which I approve, because having the same "armored guys with two handed swords" as infantry in European, Arabic, and Mesoamerican civilizations was a bit jarring. It was not a problem in the original release, with only European civilisations, but twenty years and twenty-two new civilisations later, it looks a bit weird (even if the Unique Units help in that regard).

[ca] They can convert enemy units. . .

79 Expected features

Features marked as **A** or **E** are required of FISA or FISE, respectively. Features marked as **a** or **e** are partly required, or optional but recommended.

The text may offer more nuance on what should be implemented and suggest simple approaches to satisfy the requirement without too much pain, or more advanced approaches for those who want to show off their skills.

Both simple, conservative choices, implemented very competently, and more ambitious ones, even not *fully* realised in the implementation — but realised enough to showcase their potential — can result in excellent marks. Choose tactically.

Any feature not marked as required is entirely optional.

(1) AE: Done in Python 3.

In case there was any doubt, this is a *Python* project. . .

You may need to use some older version of Python (3.7, 3.8,. . .) for compatibility with some libraries or dependencies that may not be updated very frequently, like PyGame.

(2) AE: GUI framework of your choice

The game should of course have a GUI.

You may choose

◇ TkInter,

◇ PySimpleGUI (with TkInter backend only; simpler to begin with)

◇ PyQt5 or PyQt6 (more powerful, more complex, external requirements[cb])

◇ PyGame, `https://www.pygame.org`

◇ wxPython, `https://www.wxpython.org`
Bindings to wxWidgets, similar to PyQt.

◇ Kivy, `https://kivy.org`

◇ the Arcade Library `https://api.arcade.academy`
`https://learn.arcade.academy`

◇ PursuedPyBear, `https://ppb.dev`
This one seems very fresh out of the oven, and not documented.

[cb] `https://pypi.org/project/PyQt5/`; cf. `http://doc.qt.io/qt-5/examples-graphicsview.html` pour de la documentation C++. C'est à adapter à la version Python, car PyQt5 est juste une bibliothèque de liens (bindings) vers Qt5.

◇ or anything that works with Python, really, I'm not picky.

Test the different possibilities, and choose wisely.

Based on previous experiences, and the nature of this project — especially the RTS aspect — I think PyGame is the safest choice for this project, in that most groups in previous years have used it and done satisfactory things.

PursuedPyBear seems too fresh, and I have never had any group try Kivy or the Arcade — but then again, I have never explicitly proposed them before.

PyGame is far from perfect, so I'm exceedingly interested in hearing from you if you make something else work.

I haven't seen wxWidgets used in projects, but it ought to be nearly as mature and usable as wxWidgets.

(3) AE: Graphical representation of the map.

The map must be represented graphically.

The view may be top-down, using sprites (by far the simplest option), isometric 2.5D (à la Baldur's Gate; an ambitious choice, though not too complex compared to 2D if done well), or full 3D (do not even attempt this unless you already have *considerable* experience with a suitable framework. Tests have been done last year with Ursina and Panda3D, but neither has been found suitable for use in this project).

(4) AE: Save and load.

Those games can be long. You must be able to save the game state whenever you want, and load it without loss of information. Note that,if you have an AI, that includes what the AI knows about the world, and more generally its state of mind (planning an attack, game plan, etc).

This will be extremely important for the defense, as you will not have time to play several full games during the demonstration. Instead you will load saved games, taken at interesting points of various games, to show off big battles, AI gameplan, etc.

(5) ae: AoEI-adjacent "antique" theme and units.

I would appreciate it if you would stay with roughly the same theme an units as AoE I. This is not to stifle your creativity, but to facilitate quick understanding of what you are doing.

If all groups use the same kind of units, buildings, and mechanics, with somewhat the same balance, it becomes easier to understand at a glance what's going on, and how two groups differ. This is especially crucial during the defense.

You have a very short time in which to show off your work — if much of that time is spent explaining that your drones need to mine unobtainium to build starships, which are weak to laser attacks but strong against Psykers, ... in the end that is time that is not spent showcasing how smart your AI is and how fast the game runs, which is probably not to your advantage.

Instead, if everybody uses the same basic concepts and vocabulary, some level of background knowledge can be assumed, we can quickly concentrate on the aspects that truly differentiate the groups.

That being said, if you absolutely yearn to make a space-themed game[cc], or a fantasy-themed game, or a horror-themed game (please, not that one), by all means do so, but be aware that you will need to communicate your mechanics extremely efficiently, and convince the jury that they are more or less equivalent what is expected.

Either way, sprites should be as recognisable as possible. If you can find AoE I or II's own sprites somewhere, whether from the original or "Definitive" editions , that's perfect.

You can and should also dip into the wiki, linked above, for ideas as to units and technologies.

Of course, the point is not to re-implement everything. It is to avoid having to reinvent game balance from scratch. Pick a few units, lift the health, attack values, cost, creation time etc from the wiki, and start from there. Tweak only when you need to. Again, this is not a game design course; the game is a pretext for coding.

You can go nuts with game design when your engine is solid; spending too much time pondering how many hit points and armor your hoplites should do is not a tactical investment of your energy.

Again, if that tickles your fancy to be original and go with different units, I'll allow you to do that. I just don't recommend it.

(6) aE: Human vs. AI match.

This is a game of war. What is war without an opponent? Since multiplayer is an entirely different kettle of fish[cd], you will need to implement some kind of AI.

For FISE, it is expected — or at least *hoped* — that an AI can take control of a civilisation, exactly as the player does — preferably without cheating, but if you must, you can give it extra resource or a discount... If you do, however, you must include an option or difficulty setting to disable that.

Regardless, it must have a recognisable game plan: gathering resources, building stuff, researching technologies, training units, mounting assaults.

[cc] Star Wars Galactic Battleground (2001) is precisely AoE II with a Star Wars skin.

[cd] It is quite possible that you will be called upon to add multiplayer functionality to the game during the next semester, as part of the Network Project. I haven't talked to Mr. Toinard about this yet.

To understand what it means to have a game plan, you can learn from good AoE players: there is a general consensus on what you should do at the beginning of a game to ensure a good position — a concept similar to overtures in Chess. What should the villagers gather in priority, what should they build and when, to optimise the early economy. For instance, this video covers the beginning of the game for AoE II. You can try and do something similar for your own game, and implement it as an AI strategy.

For FISA — or for FISE who are in over their heads — you should probably keep things simpler, and have an opponent that plays by different rules. You could have savages or raiders or demons or what have you spawn from special buildings — let us call them Hell Gates — at an increasing rate, and from time to time they all converge upon your buildings and kill everything in sight, then, for a change, burn everything in sight. No resource management needed, nor any particular intelligence beyond untrammelled aggression — and maybe the good sense not to leave to Gates undefended.

The aim of the game becomes to find and destroy the Gates before being overwhelmed or running out of resources.

In what follows, we shall speak of "AI" only if the computer emulates a human player, playing by the same rules (+/- mild cheats).

(7) ae: *Real-Time* Strategy.

FISE:

The real time aspect can be tricky to implement correctly. You will probably need a good architecture and some form of concurrency (multithreading, multiprocessing,. . . Sec. 31[p219]: "Parallelism and concurrency") which you will have to figure out on your own, since that is mostly taught during the second semester.

You can also try doing everything in one thread, but it has its own difficulties: if the main game loop is not fast enough, you will have terrible input lag, dropped inputs, and more.

That is not to say it is impossible for you to get an RTS working, only that it seems ambitious. Libraries like PyGames usually offer examples and propose standard ways of handling inputs, event queues etc. By starting from pertinent examples, you should be able to get things working.

However, let us note that it is in fact the first time I propose a Python project that is not intrinsically turn-based. That makes *you* pioneers, boldly going where no third-year, first semester STI student has gone before. That, or guinea pigs. Depending how you look at it.

If you fancy yourself neither a pioneer nor a lab rat, you may want to adapt the game to be somewhat turn-based.

FISA:

Keep the game very simple and lean so that any architecture can be made to work. For instance, if animating your sprites becomes a problem, then just don't animate them.

Or, you can go with a barebones turn-based adaptation of the game, played with much fewer units on a much smaller map; a bit like a chess game.

(8) Variable Speed — if RTS

Saved games help you show off specific game configurations, but to really get an idea of how your AI works, it is best to see a game in full. But we don't have time for that in a defense.

Thus, it would be helpful if the speed of the game could be changed — specifically *increased* — so that we can see the AI build its town, mount its assaults, and react to your own, all at a reasonable pace.

Of course, how fast the game can run when at full speed will depend upon both your machine and the quality of your implementation. Thus, for the defense, you can also prepare time lapse videos showing the relevant points.

Note that this can be complex to get right, as it divorces real time from game time. If you want that feature, you should plan for it from the start; adding it *post facto* is liable to break things badly.

(9) ae: Procedural map generation.

AoE generates its maps randomly, following a set of constraints regarding starting positions, resource allocations, terrain and elevation, etc.

You should probably do something like that. That said, it is not essential, so you should start by creating a single map, and getting things working on it first.

Map generation is not essential, but is a nice addition and should be very simple to implement and integrate, while adding spice to the game — especially combined with fog of war. Thus it is recommended, if you have time.

(10) e: Fog of War

AoE uses a fog of war: the map is revealed by your units, and enemy units are only visible while in the line of sight of one of your own.

This is nice to have, but not essential to the game.

Note that if it is implemented, then the AI must obey the same rules as the player, and not gain any information from what happens outside its visibility. This is not trivial to implement correctly, and even less so to assess in a defense — how does one differentiate between AI *knowledge* and *guesses*? ... especially while the AI is itself hidden under our fog of war.

(11) Replay games

Likewise, beyond save games, that preserve the state of the world at a given time t, you could try saving “replays”, that save the game at every time, and enables us to replay it like a video, but in the game engine.

Of course, the implementation would not make 100 normal saves a second. The idea instead would be to save the commands sent by each player during the course of the game, and resend them with exact timing.

That relies on the game engine being entirely deterministic. Not such a simple assumption, especially combined with real time aspects. Even if perfectly implemented, such recordings break at the slightest change in game logic — notice that nearly every game offering such features announces with each update that old replays are incompatible with the new version — AoE II and Tekken 7 are examples of such games.

That means that if you fix a bug in, say, the AI, just before the defense, your replays may — and likely will — break.

For those reasons, pursuing this feature is not recommended.

(12) e: More than 2 players.

The minimum that must be implemented is Human vs AI, and since we eschew multiplayer, there is at most one human player. AoE games support up to 8 players. You can do the same if you wish. Human vs 7 AI can be fun.

If you do so, implementing diplomacy to create factions would be relevant.

`https://ageofempires.fandom.com/wiki/Diplomacy`

If you do that, you will probably need to have different map sizes.

(13) e: AI vs. AI. Let them fight!

Gaming is hard work. Sometimes, you just want to watch a good war, but leading an army is too much work.

If you have a working AI, there should be nothing preventing you from creating a game with only AIs, and watch them duke it out.

(14) Civilisation variety.

AoE games implement a great many civilisations, each with different technology trees, bonuses, and unique units (in AoE II).

As discussed in the overview, I do not advise emulating that. One civilisation is quite enough, provided it is a bit versatile.

(15) ae: Ages.

“Aging up” is the most important technology in the AoE games. Gameplans revolve around the trade off of aging up now for a power boost, at the cost of resources and time, or using the resources to mount a raid against your opponent, in the hope of ruining their economy and delaying their own capability to age up.

Having something like that would be nice, but is not absolutely essential for the project.

If you do implement ages, just go with two of them rather than four or more. At least that is my advice unless you already have an extremely solid engine.

(16) aE: Starting resources.

You should be able to start a game with varying quantities of resources, not just the default. Ideal, you could even tweak this per player, making this a very cheap, but very effective way to control game difficulty.

Letting the AI start with lots of resources can be a good way to showcase its game plan.

(17) AE: Cheats and AI commands.

AoE has many fun cheats. Cheat codes in games are not just there for the benefit of players, however. They are useful to test the game. Here are the cheat codes that you should implement (lifted directly from AoE I and II)

- ◇ **NINJALUI:** get 10000 of each resource.
- ◇ **BIGDADDY:** spawn very powerful unit at town center.
- ◇ **STEROIDS:** training, building, research, foraging, farming, and mining times are instantaneous. . . for all players, not just you.
- ◇ **REVEAL MAP:** Reveal the entire map, exactly as though you had explored it, which means that enemy units are not revealed, while trees, and stone and gold mines are. Enemy buildings are not revealed. Issuing the command again toggles effect.
- ◇ **NO FOG:** Remove the fog of war. Any explored part of the is therefore fully visible as though under the line of sight of your units. Issuing the command again toggles effect.

Additionally, you should have a few commands to influence the AI, force it to launch an immediate attack, check its state of mind, put your civilisation on autopilot, or simply exchange the civilisations you and the enemy AI control.

The idea is that you should have commands that enable you to debug and show off your AI. What commands those are will depend on your AI.

Part XI

Archived Python Project 2020–2021: Dungeons & Dunces

80 Overview

The aim is to create a Computer Role-Playing Game (cRPG) loosely based on an extremely simplified version of the D&D 3.5 or Pathfinder first edition ruleset, in the style of games such as *Baldur's Gate*, *Neverwinter Nights*, *Pathfinder: Kingmaker*, etc.

The following links offer exhaustive references to the rulesets in question.

https://www.d20srd.org/index.htm

https://www.d20pfsrd.com/

The Pathfinder system is preferred where these differ, but the DnD page is linked as well because it is a bit simpler to read and navigate in my opinion. They are very similar in any case.

If you are completely unfamiliar with those systems, or with RPGs in general, take a few minutes to read up on the general principles, or better yet talk with a fellow student with experience in such games before reading on.

The focus will be on a dungeons crawling for a small (3 max), low level (level $\approx$ 1-5) party, with classes covering at least the three archetypes: Fighter, Mage, Rogue.

81 Features, Required and Optional

Any feature not marked as optional is required.

(1) Done in Python 3.

In case there was any doubt, this is a *Python* project. . .

(2) GUI framework of your choice

The game should of course have a GUI.

TkInter (simpler to begin with) and PyQt5 (more powerful, more complex, external requirements[ce]) are two possibilities for building it. I hear good things about PyGame as well.

(3) Graphical representation of the map.

The maps must be represented graphically.

The view may be top-down, using sprites (by far the simplest option), isometric 2.5D (à la Baldur's Gate; an ambitious choice, though not too complex compared to 2D if done well), or full 3D (do not even attempt this unless you already have *considerable* experience with a suitable framework).

(4) Grid or no grid, that is the question.

The game will need to solve whether two units are in contact, how much space they occupy, whether they are in range of some effect or ability, and how they move.

To do so, you can use a grid, which can traditionally be square or hexagonal, or simply compute the Euclidean distance within a floating point Cartesian coordinate system.

The latter is recommended. Grids are only used in tabletop games because gamers are not keen on computing squares and square roots by hand multiple times every round. With the help of a computer, this is no longer a problem.

(5) Level / Campaign editor.

The user must be able to build his own dungeons / maps, possibly involving multiple levels and connectivity between them, and them play them.

It is *not* required that the user can implement complex logic in the campaign, e.g. set triggers, handle quests etc.

It *is* required that levels can be built, connected to one another, and populated with enemies, NPCs, traps, and treasure, fixed or random to some extent. In that case, the campaign is simply a dungeon crawl.

(6) Save and load

Of course, the player must be able to save his game – at least outside of combat – and load the game later.

(7) Random level / maze generation. (Optional but strongly recommended)

One approach can be to build a fairly impressive, static campaign to show off the game engine's capabilities.

Another would be to generate levels randomly, Roguelike-style.

[ce] `https://pypi.org/project/PyQt5/`; cf. `http://doc.qt.io/qt-5/examples-graphicsview.html` pour de la documentation C++. C'est à adapter à la version Python, car PyQt5 est juste une bibliothèque de liens (bindings) vers Qt5.

The two approaches can be mixed as well; for instance, a campaign could well be static, but include, at some point, a maze that is procedurally-generated, to catch returning players off-guard.

(8) **Exploration and line of sight.**

Maps can be pre-explored, under fog of war, or completely unexplored when the party first enters it. The unexplored map is revealed as the party moves, taking into account the lines of sight of all.

Mobile units are only visible if a member of the party can see them.

Of course, you need to provide a map / minimap of the current level to the player, respecting fog of war rules.

For bonus points, allow the player to consult the maps of previously visited levels the party is not currently in, and to put their own notes and points of interest on the map.

(9) **Last known position (Optional).**

If the party loses sight of a mobile unit, there may be an option to display a "last known position" marker for that unit on the map.

If you go for "advanced AI", you will need at least enemy units to remember where they last saw the party, so as to mount ambushes etc.

Note that, for all intents and purposes, the player party is treated as a single unit for the purpose of line of sight, knowledge of the map and enemy positions, etc. That is to say, for instance, whatever the Fighter knows or sees, the Mage is aware of as well.

(10) **Talking NPCs and merchants.** The game will be focused on dungeon-crawling, but it must be possible to talk to and trade with some NPCs.

(11) **Turn-based or RTWP.**

It is strongly recommended to make the game Turn-Based.

Implement at least standard and move actions; preferably full-round attacks as well, if your game supports classes or levels with more than one attack per round.

If you are feeling adventurous, you *can* attempt to go with Real Time With Pause (RTWP), but it can get more messy.

(12) **Area of effect imprint.**

Some spells and abilities, such as Fireball, affect a wide area; the game should let you see the "imprint" of the spell before you cast it, so you can position it exactly.

(13) **Single player + AI.**

The game must support a single player mode, facing off against AI-controlled enemies.

Those enemies must be able to detect and engage the player party, making use of their basic attacks and abilities, even if they may lack good judgment in applying them, and display no sense of self-preservation.

If one creature detects the player, it may be acceptable that all of them immediately shares that knowledge.

(Sadly this describes the AI of most cRPGs. . .)

(14) Advanced AI (Optional).

It would be great if the AI could be more sophisticated.

In that case, each monster should have its own knowledge of the party and other allied monsters, and be able to run away from the party to alert its friends – possibly bringing an army back on its heels! This necessitates modeling at least basic communication between enemies.

It should use discernment in avoiding dangerous Areas of Effect, whether the party's or his own.

This intelligence can be conditional on the creature having a high enough INT stat; some creatures may be designated as leader of a group of its lesser kin or subordinates, and coordinate their actions so long as it as able to communicate with them, making it a priority target.

Different creatures may have different level of aggressiveness and courage or morale, leading them to either fight to the death of flee as the slightest hint they are outmatched. This of course may be influenced by the presence of a designated leader or of powerful paragons of their race on the battlefield.

(15) Classes.

Of course the point is not to implement 50 classes or esoteric D&D mechanics.

Just pick the main features of Fighter or Barbarian, Sorcerer or Wizard (henceforth called "Mage), and Rogue.

You will implement all attributes (STR, DEX, CON, INT, WIS, CHA), and their basic effects on attack and damage rolls, armor class, etc. Use your common sense and do not try to implement *every* feature in the rule set. Start with those features which are most important for combat: Hit Points, armor, attack rolls, saves, etcetera.

Each class should support a few levels with one or two feats, and two or three spells for the mage. Of course XP gain must be implemented. XP may be granted by killing monsters or by reaching deeper levels of the dungeon.

Rogues should implement Sneak Attack.

Mages should preferably be modelled after Sorcerer rather than Wizard due to simpler casting mechanics and higher usefulness in dungeon-crawls. Their spell-casting progression can be copied from Wizard, on odd levels instead of even ones, so as to boost their usefulness at low level. (Spellcasters are weak at low levels and shine at higher levels; since this game will stay at low level, Mages need all the help they can get). Cantrips shall follow the pathfinder model and be at-will.

If you introduce diseases, poisons, curses, or other mechanics, implementing Cleric as well is a must.

Enemies may have custom classes, or they may simply use the same ones as player characters, just with different attributes and bonuses.

(16) **Multiclassing. (Optional)**

You may allow multiclassing; that it to say, to take levels in more than one class, for instance, the first level as Rogue, and the second as Fighter, the third as Rogue again, etc.

Note that this may complicate the character sheet, so if you choose to implement this, plan ahead rather than trying to add the feature after the fact.

(17) **Races. (Very optional)** You don't need to implement multiple races. Humans will suffice. But of course you can provide other races if you want, and there is nothing more urgent to do.

(18) **Skills.**

Only a small subset of available skills need be implemented. Among them are Stealth, Disable Device, and Perception (as in Pathfinder), essential for the Rogue class.

Everything else is very optional. If you implement poisons, you will need Heal as well.

Use Magic Device is not necessary; assume the Mage and maybe the Rogue can use the wands they find.

Of course, the full complexity of each implemented skill need not be present; the basics will suffice.

Skill points per levels will need to be adjusted depending on how many skills you implement.

(19) **Resting mechanics.**

Of course, resting mechanics must allow characters to regain HP and spells. To simplify, have characters regain everything in a single rest.

There must be a risk of being attacked during the rest if enemies are nearby. This can be handled as new enemies spawning out of nowhere or as enemies making their way from other parts of the level to attack you, the latter being more advanced.

(20) **Items.**

There should be a few items available in the level or and/or from vendors: swords, and other weapons, armors, robes, wands, potions (in particular healing potions), rings, amulets, etc.

Just one or two different items for each category will suffice.

(21) Customisable hotbars and keyboard shortcuts

The GUI must provide hotbars where each character may organise their abilities, spells, etc.

The player must be able to customise the keyboard shortcuts used to access those abilities.

(22) Combat logs.

The game must provide extensive combat logs. For instance every attack must show what attack score was rolled against what AC, and it must be possible to find out where the attack and AC scores come from. That is to say, if I see my Fighter is rolling against an AC of 19, I must be able to find out how much of this AC comes from a Dexterity bonus, how much from an amulet, what kind of bonus it is each time, etc.

This will be crucial to debugging the game and checking that abilities and items work as intended.

(23) Detailed character sheet.

Likewise, the player's character sheets must detail the computations involved in the character's AC, Attack bonus, saves etc.

(24) Inventory management.

The inventory management may be based solely on weight, or on weight and grid geometry, with larger objects occupying more slots (in the style of Diablo or Neverwinter Nights).

(25) Multiplayer (Optional)

If you are *really* in need of a challenge, you can implement a multiplayer mode so that your friends can connect to your game and take control of some characters in the party from their own machines.

82 For apprentices

(1) There is much less time dedicated to that project: 5 weeks in total. (end: at the end of december).

(2) The level editor need not be fully functional. Do whatever you need so that *you* can use it to create a few levels, but it need not be usable by the end user.

(3) The DnD aspect can be toned down considerable, or even done away with entirely, as you will not *(edit a year later: I never finished that sentence and somehow nobody noticed)*

Part XII

Archived Project 2018–2019: Automata GUI

83 Vue d'ensemble

Le but du projet est de réaliser une application – avec interface graphique – pour la construction de graphes orientés et de diagrammes sagittaux[cf] d'automates d'états finis, avec des facilités automatiques et semi-automatiques pour produire des diagrammes naturels, plaisants, et lisibles, exportés au format LaTeX/Ti*k*Z. Voyez les figures émaillant ce document pour quelques exemples.[cg]

Cette application a pour vocation de venir en aide à la minorité peu considérée des professeurs de théorie des langages, qui ont très envie d'illustrer leurs diapositives et polycopiés avec de nombreux et beaux diagrammes, mais qui n'ont pas le luxe de passer à chaque fois une heure à se reconnecter avec leur artiste intérieur afin de d'imaginer comment l'automate devrait être disposé pour être "joli", et encore une heure à se battre avec LaTeX et Ti*k*Z pour coder une version approximative de cette vision.

Ce document donne des pistes quant à ce qui est attendu, mais ne doit pas être abordé comme une spécification exhaustive. Posez des questions !

Ceci est à réaliser en Python 3; le choix est ouvert pour la partie graphique, PyQt5 étant le choix par défaut[ch].

Un prototype, réalisé dans le cadre du projet d'application en fin d'année par vos prédécesseurs, est mis à votre disposition pour démarrer — mais rien ne vous empêche de recommencer à zéro. C'est même fortement recommandé.

84 Qu'est-ce qu'un automate ?

On abordera le module de théorie des langages en seconde période. Pendant la première moitié du projet, on verra donc les automates purement comme des diagrammes, et je ne

[cf] i.e. diagrammes avec des ronds et des flèches. Même racine latine, *sagitta*, "flêche", que le signe Sagittaire... ici dans un contexte plus scientifique.

[cg] Sources: mes polys, `http://www.texample.net/tikz/examples/feature/automata-and-petri-nets/`, `https://www3.nd.edu/~kogge/courses/cse30151-sp18/Public/Assignments/tikz_tutorial.pdf`, `https://tex.stackexchange.com/questions/148158/make-a-tikz-automata-edge-pass-outside-the-automata`,. . .

[ch] cf. `http://doc.qt.io/qt-5/examples-graphicsview.html` pour de la documentation C++. C'est à adapter à la version Python, car PyQt5 est juste une bibliothèque de liens (bindings) vers Qt5.

donnerai (à l'oral) qu'une vague intuition de leur sémantique – ce qu'ils "veulent dire".

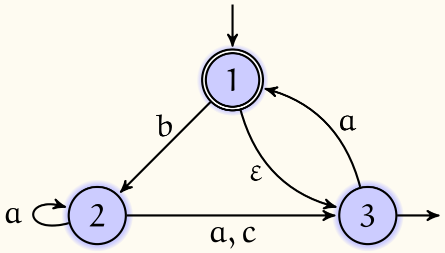
On aura malgré tout besoin de la définition formelle de la structure, afin d'avoir le vocabulaire nécessaire pour communiquer.

Un **automate fini non déterministe** est un 5-uplet $A = (\Sigma, Q, I, F, \delta)$ où:

- Q: ensemble fini d'états
- Σ: alphabet fini
- $I \subseteq Q$: états initiaux
- $F \subseteq Q$: états terminaux
- $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$: relation de transition

$$(p, a, q) \in \delta \overset{\text{notation}}{\equiv} p \overset{a}{\to} q \qquad \delta(p, a) = \{q \mid p \overset{a}{\to} q\}$$

Voici un petit diagramme sagittal d'automate, illustrant les différentes conventions de représentation:



L'alphabet est l'ensemble des étiquettes de transition, i.e. les lettres lues par l'automate. Le symbole ε ne fait pas partie de Σ, il est utilisé pour une transition qui se déclenche sans rien lire. On a

$$\Sigma = \{a, b, c\}$$

Les états sont les ronds:

$$Q = \{1, 2, 3\}$$

Les états initiaux sont indiqués par une flèche entrante déconnectée:

$$I = \{1\}$$

Les états finaux sont indiqués par un double cercle ou par une flèche sortante déconnectée:

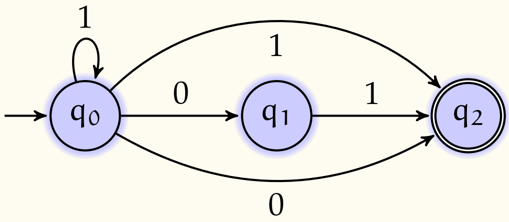
$$F = \{1, 3\}$$

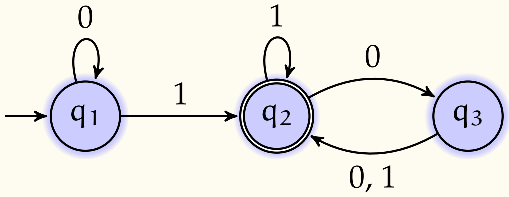
Notons qu'il n'est pas *du tout* classique d'utiliser ces deux conventions dans le même automate; ou même dans le même document. L'automate ci-dessus commet donc une faute de goût.

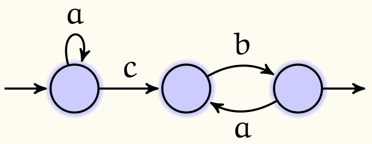
Restent les transitions:

$$\delta = \{\,(1, b, 2), (1, \varepsilon, 3), (2, a, 2), (2, a, 3), (2, c, 3)\,\}$$

Faites le même exercice avec les automates suivants:







85 Partie graphique / manuelle

Difficulté modérée; progression incrémentale; programmation objet; algorithmique simple; travail de documentation bibliothèques Python, LaTeX, TikZ. Groupe de 3 recommandé.

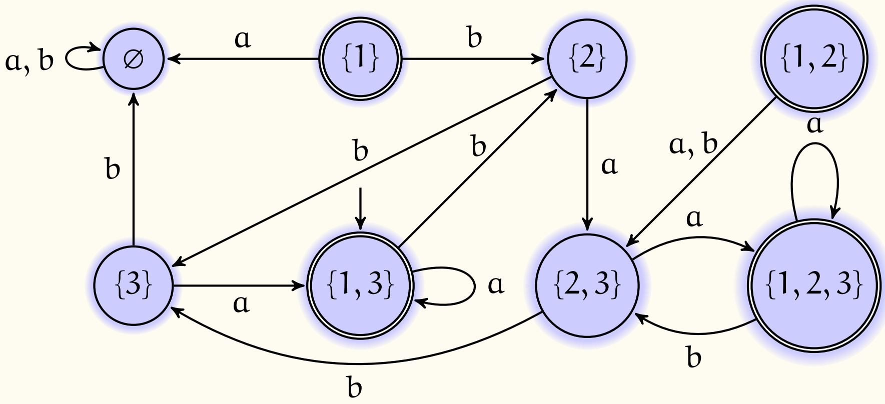
Réaliser un diagramme sagittal d'automate est largement une activité de dessin, régie en partie par des considérations purement esthétiques et en partie par des conventions et habitudes liées au domaine scientifique et favorisant la lisibilité des diagrammes.

Le point de départ de l'application est donc *in fine* un logiciel de dessin vectoriel assez ordinaire, grandement simplifié par le faible nombre de figures de base dont on a besoin: flèches, droites ou incurvées, cercles simples et doubles, étiquettes en langage mathématique (entrée en LaTeX), et c'est tout.

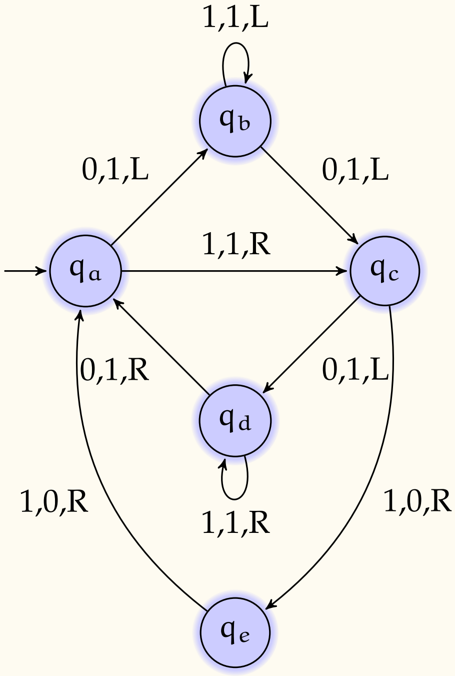
Le logiciel utilisera le vocabulaire “automates” et non le vocabulaire “dessin”, par exemple “état” et non “cercle”. Si un jour il nous vient l’envie d’avoir des états rectangulaires, cela doit être possible...

Placer des états et définir des transitions doit être aussi simple et rapide que possible pour l’utilisateur .

Notons que, bien que les noms des états soient arbitraires, on voudra parfois les nommer par des formules. Par exemple, l’automate suivant est obtenu via un algorithme de déterminisation, et il est essentiel de pouvoir faire apparaître les sous-ensembles dans les états afin d’illustrer la démarche:



De même, dans l’automate suivant, ce sont les transitions qui portent des étiquettes un peu “compliquées”, et pas seulement une seule lettre — pour ceux que ça intéresse il s’agit ici en fait d’une machine de Turing:



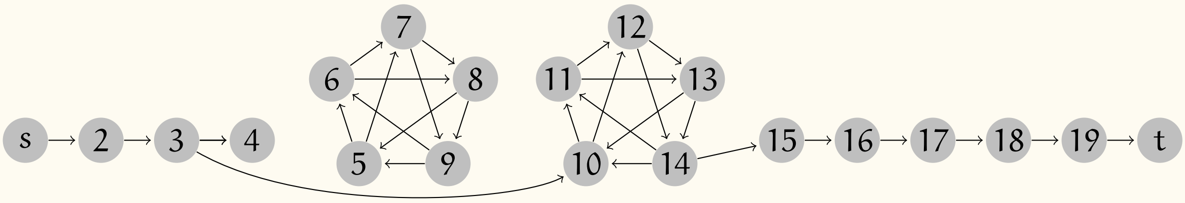
Il n’est pas *totalement nécessaire* que les formules mathématiques soient rendues correctement dans l’interface graphique – le code LaTeX peut être conservé tel-quel jusqu’à l’export vers

LaTeX/TikZ. Ce serait toutefois appréciable si les formules pouvaient être rendues directement dans l'interface. Si c'est le cas, il faut toutefois que l'utilisateur puisse désactiver cette fonctionnalité, car le rendu dépend du contexte d'évaluation (LaTeX est essentiellement un langage de programmation) et le code de l'utilisateur peut donc ne prendre sens qu'au sein de son document.

Ceci peut poser certains problèmes au moment d'évaluer la taille des états, par exemple. A vous de proposer des solutions pour gérer ces cas de la manière la plus automatique et flexible possible.

Ces briques de base, ronds et flèches, doivent pouvoir être manipulées à la fois finement et semi-automatiquement.

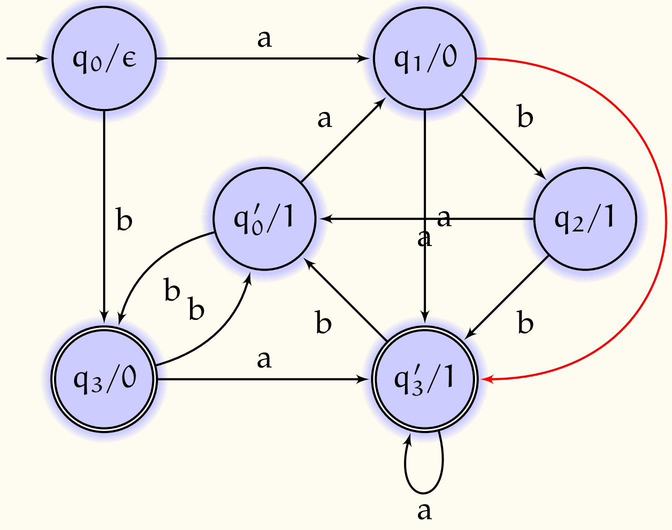
Par "semi-automatiquement", j'entends par exemple l'alignement sur une grille – ou plusieurs grilles – et la disposition de tout ou partie des états selon certains schémas; par exemple, voici un automate contenant plusieurs sous-figure régulières: deux en disposition linéaire, et deux en cercles (ou pentagones en l'occurrence).



L'interface doit permettre à l'utilisateur de réaliser semi-automatiquement ce type de figure. Spécifiquement l'utilisateur doit être capable de sélectionner des groupes de noeuds et de les arranger selon des figures géométriques régulières telles que des lignes, des cercles, et cetera.

Les arrangements faisant partie d'une telle figure géométrique – que l'on appellera un *groupe* – doivent pouvoir être sélectionnés et déplacés collectivement. On doit aussi pouvoir régler les paramètres de chacun; par exemple le diamètre d'un groupe déjà défini doit pouvoir être modifié par la suite. De plus, plusieurs groupes doivent aisément pouvoir être uniformisés – par exemple, leur donner le même diamètre, placer leur centre de gravité à des endroits alignés horizontalement ou verticalement, etc.

Les éléments doivent aussi pouvoir être manipulés finement; par exemple, on a finement déformé la transition de 3 à 10 pour éviter le premier pentagone; dans l'automate suivant, on doit pouvoir déformer la transition de $q_1/0$ à $q'_3/1$ de manière à éviter le croisement, obtenant ainsi la nouvelle transition en rouge.

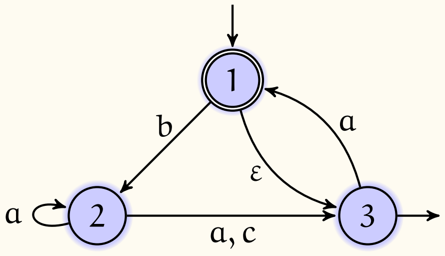


86 Import et export

L'application devra pouvoir sauvegarder et restaurer ses diagrammes dans un format au moins partiellement lisible par l'être humain et proche du formalisme $A = (\Sigma, Q, I, F, \delta)$ – avec bien sur quelques informations supplémentaire d'ordre cosmétique, telles que couleurs, courbure, localisation (x, y), etc.

Au delà de cela, il est absolument essentiel qu'elle soit capable d'exporter au format Ti*k*Z. La qualité de l'export Ti*k*Z est un facteur important, car l'utilisateur aura peut-être envie de retoucher le code généré.

Comme exemple de code Ti*k*Z, l'automate



est codé par

```
\begin{tikzpicture}[fst]
\node[state, initial above, accepting] (1) {$1$};
\node[state, below left of=1] (2) {$2$};
\node[state, below right of=1, accepting right] (3) {$3$};
\draw
```

```
  (1) edge[above] node{$b$} (2)
  (1) edge[below, bend right, left=0.3] node{$\eps$} (3)
  (2) edge[loop left] node{$a$} (2)
  (2) edge[below] node{$a, c$} (3)
  (3) edge[above, bend right, right=0.3] node{$a$} (1);
\end{tikzpicture}
```

qui est assez lisible et descriptif. Le paramêtre `fst` (pour Finite-State Transducer) est un style personnalisé défini dans ce document.

Il y a bien d'autres façons de coder cette figure, telle qu'un matrice, ou des positions (x, y) dans le plan explicites. Une grande flexibilité au niveau de la génération du code final est attendue.

À vous de vous documenter sur LaTeX et TikZ pour voir les différentes possibilités offertes par ces langages. L'application doit en tirer parti autant que faire se peut, afin que le diagramme final puisse être ajusté le plus finement possible avant l'export, et minimiser les besoins en retouches au niveau du code généré.

Au delà de la figure, on doit aussi pouvoir exporter du code LaTeX pour la définition formelle de l'automate sous forme $A = (\Sigma, Q, I, F, \delta)$. Les transitions doivent pouvoir être générées sous forme d'ensemble, comme plus haut, ou sous forme de tableau de transition.

87 Partie automatique

Difficulté élevée (pour obtenir de bons résultats); algorithmique poussée; recherche documentaire; réflexion poussée. Groupe de 3 recommandé.

Même avec une bonne interface graphique, tous ces choix et ajustements esthétiques prennent énormément de temps. Du point de vue de l'utilisateur final, la partie "dessin manuel" de l'application est donc à réserver pour les figures importantes, qui doivent être réalisées avec une qualité parfaite.

Dans la plupart des cas, l'utilisateur veut juste définir son automate "en vrac" et en quelques clics, appuyer sur un bouton et obtenir instantanément une (ou plusieurs) proposition de diagramme, calculées programmatiquement, qui soient d'une qualité passable. Ceci implique de minimiser les croisements (pas toujours possible à éviter entièrement – on verra au second semestre la notion de graphe planaire, et le fait que tous les graphes ne sont pas planaires), de disposer les états à intervales réguliers de façon à assurer un "niveau de gris" homogène dans la figure, etc.

Il existe de nombreuses techniques spécifiques au dessin automatique de graphes, ainsi que des métaheuristiques générales potentiellement applicables, telles que les algorithmes révolutionnaires / génétiques, le recuit simulé, et les colonies de fourmis, pour n'en citer que quelques-unes.

Mettre en oeuvre ces méthodes nécessitera un gros travail de recherche documentaire et de réflexion poussée pour comprendre des concepts difficiles au niveau 3A.

À ceci s'ajoute une difficulté supplémentaire: il ne s'agit pas de n'importe-quels graphes, mais d'automates; les conventions de dessin ne sont pas les mêmes; utiliser un algorithme de dessin arborescent ou fractal donnerait généralement des résultats peu appropriés, même s'ils peuvent être "jolis". On voudra généralement partir de l'état initial, aller plutôt de gauche à droite et de haut en bas, disposer les états sur une grille, sauf certains cycles et autres motifs à détecter et à mettre en exergue, et ainsi de suite.

Il n'y a pas (ou peu) de littérature disponible sur le sujet spécifique "comment dessiner des automates". Les dessins sont faits par les scientifiques "au feeling", influencés par l'habitude et le mimétisme sans que les conventions soient verbalisées.

Dans cette partie du projet, il vous appartiendra d'inférer une partie de ces conventions à partir d'exemples, de les verbaliser, et de les programmer.

Un mode interactif, permettant à l'utilisateur de fournir certaines informations sémantiques à l'application pour guider ses choix (par exemple, ce groupe d'états va ensemble, cet état est important, etc), ou bien de choisir entre plusieurs alternatives (algorithme évolutionnaire avec sélection (semi-)manuelle, est à considérer.

De plus, ces outils doivent pouvoir être appliqués semi-automatiquement au cours de l'utilisation de l'interface pour faciliter la vie de l'utilisateur – par exemple, en détectant automatiquement quand une nouvelle transition est créée si elle croise quelque-chose, auquel cas elle sera créée courbe et non droite, ou bien en offrant des suggestions d'agencements dans la marge, applicables en un clic, etc.

La partie du rapport consacrée à ces réflexions doit être assez détaillée.

88 Répartition des tâches

Il est conseillé de procéder par groupes de 6, *par exemple* en deux trinômes, un pour la partie manuelle et l'import / export, et l'autre pour la partie automatique.

Dans ce cas, les trinômes doivent évidemment communiquer et intégrer leur code au projet commun de façon très régulière – les tâches ne sont pas indépendantes et la répartition ci-dessus n'est qu'une suggestion.

Il a été noté que l es deux partie sont de difficultés et de natures différentes, la partie manuelle étant colorée "développeur" et l'autre "recherche".

Choisissez bien la partie sur laquelle vous travaillerez en fonction de vos goûts et de vos capacités.

La partie automatique peut potentiellement déboucher sur une très bonne solution (et donc une note stratosphérique) en deux semaines de réflexion et 200 lignes de code,... mais vous pouvez tout aussi bien passer un semestre à pondre 10 000 lignes de code produisant une bouillie infâme et inexploitable, de valeur 0.

En revanche, la partie manuelle est difficile à rater si on travaille régulièrement, (mais il faudra pondre beaucoup de lignes que l'on ait les bonnes idées ou pas).

Pour minimiser le risque d'avoir une mauvaise note individuelle sur la partie automatique en cas d'échec, il serait une bonne idée de contribuer un peu à l'interface graphique. Les cas simples comme la détection de croisement sont idéaux pour cela.

Il est bon également que tout membre du groupe ait une vue d'ensemble de l'application, même des parties écrites par les autres.

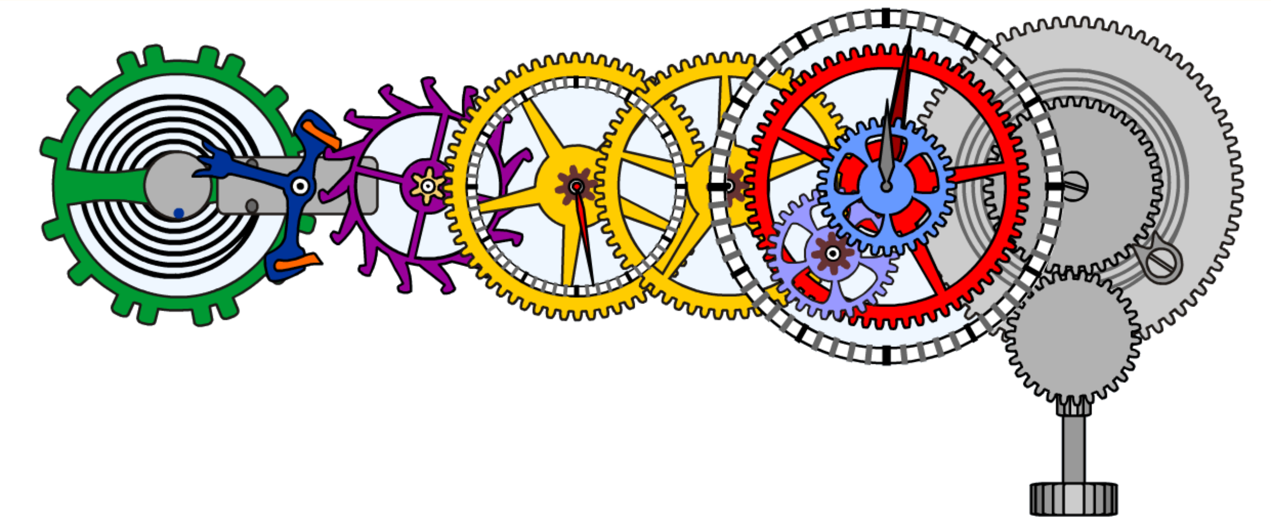


Figure 8: De droite à gauche: remontoir, roue de couronne, rochet et son cliquet, barillet, diverses roues, certaines portant des aiguilles, roue d'échappement, ancre de balancier, balancier et son ressort. Voir l'animation sur https://scratch.mit.edu/projects/26004123/#fullscreen, et la vidéo https://www.wimp.com/1949-how-a-watch-works/.

Part XIII

Archived Project 2017–2018: Mon(s)tres

89 Vue d'ensemble

Le but de ce projet, à réaliser en Python 3 par groupes de 4 à 6, est de réaliser un générateur automatique de mécanismes horlogers pour montres mécaniques doté, de plus, d'une

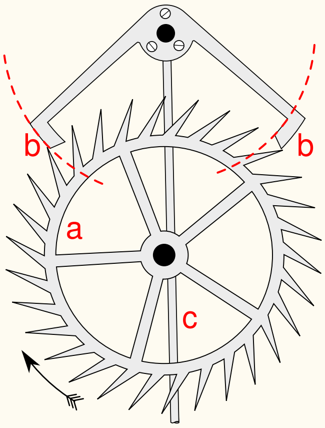


Figure 9: Une alternative au balancier: l'échappement à ancre, utilisant un pendule.

fonction de visualisation.

La génération se fera au moyen d'algorithmes évolutionnaires, une classe de métaheuristiques qui imite l'évolution biologique pour résoudre des problèmes complexes pour lesquels une approche directe (résolution d'équations) n'est pas faisable, et où l'espace de recherche est trop grand pour une approche par force brute. Plus spécifiquement, on utilisera des algorithmes génétiques.

Le principe est le suivant: nous générons aléatoirement une population de montres... ou plutôt de *monstres*, car un mécanisme généré aléatoirement a peu de chances d'être utile pour savoir l'heure... Cette population est ensuite soumise à des conditions de sélection, telles que les monstres qui sont de meilleures montres – i.e. qui sont plus utiles pour mesurer le passage du temps, ou plus efficaces – ont de meilleures chances de se reproduire, et moins de périr. Quelques mutations peuvent aussi avoir lieu. Après de nombreuses générations, on doit obtenir d'excellentes montres.

90 Les sous-problèmes

90.1 La modélisation des monstres

On impose une abstraction au niveau des pièces d'horlogerie: nos monstres évoluent dans une soupe primordiale de remontoirs, roues dentées (faisant office de rochet, couronnes, etc. On considère que toutes les roues ont entre 3 et 1000 dents, identiques pour toutes les roues sauf l'échappement[ci]), roues d'échappement (si l'on veut la considérer séparément, ce qui est conseillé), ancres, balanciers, ressorts (on pourra considérer un ressort comme un ressort moteur ou un ressort de balancier), tiges (pouvant faire office d'aiguilles ou de tiges de pendule). Le boîtier est considéré comme une pièce toujours présente. On ignorera totalement les frottements.

Selon les ambitions de votre groupe, vous pourrez choisir d'ignorer totalement les considérations d'espace:

90.1.1 Mode connexion libre

Chaque pièce peut se connecter à n'importe-quelle autre pièce à certains points de connexion: par exemple, une roue dentée peut se connecter sur son axe, et sur ses dents. Si une roue dentée se connecte à une autre sur leurs dents, on s'attend à ce que l'une entraîne l'autre; si elles se connectent sur leurs axes, on obtient un pignon. Rien n'empêche une troisième roue de se connecter sur le même axe.

[ci] On pourra tout de même aussi considérer la roue d'échappement comme une roue dentée ordinaire dans un premier temps, pour simplifier.

Une tige peut se connecter aux deux bouts; par exemple, une tige connectée au boîtier à un bout et à n'importe-quel autre objet, par exemple une roue, à l'autre bout, forme un pendule. Une tige connectée à un axe de roue dentée d'un côté, et libre de l'autre, fait office d'aiguille. En revanche, connecter une tige aux dents d'une roue stoppe le mécanisme.

Une ancre peut se connecter aux dents (à une roue d'échappement), et sur son axe, par exemple à un balancier ou à une tige.

Un ressort spirale peut se connecter en son centre et à son extrémité; par exemple, un ressort connecté à deux roues dentées coaxiales, l'une par le centre, l'autre par les dents (on imagine que le ressort se connecte *sous* les dents), forme un début de moteur.

Un remontoir peut se connecter sur l'axe ou sur les dents d'une roue, avec les mêmes effets.

A vous de définir le reste des règles de connexion entre les pièces, et de prévoir leurs effets.

90.1.2 Mode 2D$\frac{1}{2}$:

Le mode précédent ignore un certain nombre de problèmes. Il n'est en réalité pas immédiat de connecter n'importe quelle pièce à n'importe quelle autre. Par exemple, les roues dentées ne sont pas de taille arbitraire: comme les dents doivent être compatibles pour les roues connectées entre elles, le nombre de dents est proportionnel à la circonférence, et il n'est donc pas possible d'en varier la taille arbitrairement. Pour connecter deux roues distantes entre elles, on doit souvent avoir recours à des pignons fous. Le mode précédent, ignorant de tous ces aspects, va donc résulter en des constructions dignes d'Escher, impossibles à mettre en oeuvre sans rajouter des composants pour assurer les liaisons.

En mode 2D$\frac{1}{2}$, les pièces posséderont des dimensions largeur/hauteur, et on les placera sur un plan; on aura également besoin d'une dimension de profondeur, discrète. Par exemple, une roue avec pignon occupe deux plans superposés: un pour la roue, et un pour le pignon. Un moteur occupe au moins trois étages, un pour chaque roue et un pour le ressort. Les connexions se feront lorsque les pièce sont situées approximativement au bon endroit pour se toucher.

90.1.3 Que choisir ?

Le premier mode sera a priori plus aisé à mettre en œuvre du point de vue des algorithmes évolutionnaires – il y moins de possibilités – mais obtenir des visualisations intéressantes sera nettement plus difficile.

90.2 Mutation, sélection, reproduction

Pour utiliser des algorithmes génétiques, il ne suffit pas de savoir comment représenter les monstres eux-mêmes (le phénotype), mais il faut aussi décider d'un codage génétique de cette représentation: le génotype d'un monstre. C'est ce génotype qui est l'objet des croisements et mutations qui donnent son pouvoir à la méthode.

La choix du génotype vous est laissé. Pour des connections libres, une simple matrice d'adjacence peut faire l'affaire. Dans tous les cas, on peut aussi avoir une représentation en chaîne binaire, éventuellement découpée en chromosomes – c'est la méthode classique.

Il faudra ensuite définir comment fonctionnent les accouplements (enjambements en un point, en deux, uniformes, semi-uniformes ?), les mutations, et la sélection (par rang, proportionnelle,... taille de la population, etc).

Il est très fortement conseillé d'implémenter vos propre algorithmes plutôt que d'utiliser une bibliothèque toute faite. Tous ces choix de modélisation devront être faits quoi qu'il arrive, et vous devrez savoir les argumenter dans tous les cas.

Il est essentiel que le phénotype puisse être calculé assez rapidement à partir du génotype, car il faut bien l'évaluer (section suivante), et ce de très nombreuses fois. Il s'agit là d'une étape de calcul qui peut gagner énormément à être parallélisée (sur machine multi-processeur; faire ça sur de multiples machines est hors cadre ce semestre).

90.3 La mesure d'un monstre

La survie d'un monstre dépend de son utilité en tant qu'instrument de mesure du temps. Il s'agit de simuler un environnement hostile aux monstres qui ne sont pas ponctuels; c'est ici que nous allons injecter quelques connaissances d'horlogerie et nos conceptions subjectives de ce qui fait une bonne montre. On doit écrire une fonction – utilité, ou fitness – attribuant un score à chaque monstre, qu'on cherche à maximiser.

Il est important que ce score soit assez fin pour détecter de petites améliorations.

On supposera toujours la présence d'un boîtier, mais un boîtier seul ne sert strictement à rien.

Une roue dentée, connectée au boîtier sur son axe, est un peu meilleure. On peut lui donner une impulsion et attendre qu'elle s'arrête pour mesurer un intervalle de temps. C'est pénible mais très légèrement mieux que rien.

Un ressort tout seul peut jouer un rôle similaire; on attend qu'il finisse de vibrer.

Un pendule est nettement mieux: on peut compter les secondes tranquillement, pendant un temps raisonnable.

Un moteur avec un système d'échappement est nettement mieux qu'un pendule, car il va durer beaucoup plus longtemps (durée à calculer). Il est aussi moins encombrant, et peut se déplacer.

Mais tout ça est limité, car il faut garder l'œil sur le dispositif, seconde après seconde, et c'est peu lisible. Si une aiguille tourne avec, c'est mieux. Si plusieurs aiguilles tournent à des rythmes différents, c'est mieux. Si leurs rythmes se rapprochent d'une rotation par minute, heure, et jour, c'est encore mieux. Si en plus on a aussi semaine / mois, etc, c'est encore mieux.

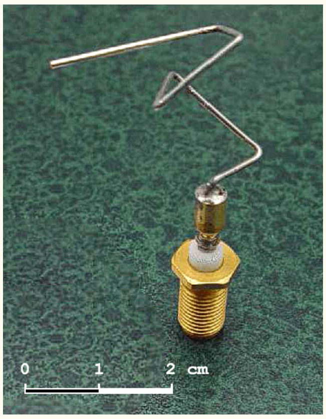


Figure 10: Antenne ST5 de la NASA, obtenue via algorithmes évolutionnaires; premier objet créé de cette manière à aller dans l'espace (2006).

Si on a moins de pièce, ceteris paribus c'est mieux. En particulier, on ne veut pas 50 moteurs à remonter. Mais attention, ceci doit rester un critère secondaire, sinon aucun mécanisme complexe n'aura le temps d'évoluer.

De même, si le mécanisme est plus compact, ceteris paribus c'est aussi mieux (en particulier pour la représentation 2,5D) – une roue à 1000 dents n'est pas indiquée pour une montre-bracelet.

Et ainsi de suite.

En revanche, on ne trichera pas en récompensant des étapes intermédiaires pour des engrenages complexes qu'on a déjà en tête. La NASA n'aurait pas obtenu l'antenne ST5 (cf. figure) s'ils avaient dressé leurs algos à imiter leurs ingénieurs.

L'exercice consiste donc principalement d'une part à simuler le comportement d'un mécanisme donné (calculer la vitesse de rotation des pièces), et d'autre part à détecter les configurations qui nous plaisent et déplaisent, en choisissant judicieusement les poids pour chaque critère.

Ce calcul va être réalisé de très nombreuses fois; il est donc impératif qu'il soit très efficace. Encore une fois, on va négliger les frottements et l'usure.

90.4 Stockage des populations

Les calculs étant longs, il est nécessaire de pouvoir les interrompre et les reprendre sans crainte. Il faudra donc pouvoir sauvegarder (et charger) la population en cours et quelques statistiques pertinentes des générations précédentes. Par exemple, les stats de fitness au cours du temps, et les meilleurs individus de chaque génération.

Il serait aussi bon de pouvoir fusionner des populations enregistrées pour former une nouvelle population initiale.

90.5 Visualisation graphique des monstres

Il va bien falloir jeter un œil aux monstres pour évaluer et exploiter les résultats produits, car personne ne va s'amuser à lire une matrice d'adjacence ou un code binaire. Ce n'est donc pas du tout optionnel.

Un graphe indiquant la connexion entre les pièces est un strict minimum, mais l'on attend plutôt un schéma ou un dessin décrivant le mécanisme de façon aussi claire et proche de la réalité que possible; cf. les deux premières figures. Ceci est bien plus aisé dans le mode 2.5D; on peut imaginer un découpage niveau par niveau. Si on peut faire ça en vraie 3D, encore mieux.[cj] C'est interactif ? Mieux. Animé ? Encore mieux.

La visualisation doit absolument être capable d'indiquer la vitesse de rotation de chaque composant, et ce de manière claire et lisible; cette fonction doit pouvoir être activée et désactivée.

90.6 Interfaces graphique et ligne-de-commande

Il faut absolument avoir une interface ligne de commande claire, robuste et bien documentée permettant de lancer, sauvegarder, et charger des évolutions, ainsi de de visualiser les meilleurs individus d'une population (en ouvrant une fenêtre graphique et/ou en générant un fichier graphique).

Il est aussi bon que le programme ait une GUI[ck] permettant de visualiser en temps réel l'évolution de la population et ses meilleurs individus, et si possible d'ajuster les poids de certains critères en temps réel, tels que l'importance de la compacité du mécanisme.

91 Répartition des tâches

Il est conseillé de procéder par groupes de 6, par exemple en trois binômes focalisés sur les trois parties principales:

- Algos génétiques (choix de génôme, mutation, crossover, parallélisation des calculs (multi-processeurs)...),
- Évaluation des monstres (horlogerie / simulation, fonction de fitness)
- Interface graphique, interactivité, et visualisation.

D'autres tâches annexes sont à répartir entre les membres: e.g. stockage et gestion des populations, ligne de commande. . .

[cj] Uniquement si on n'a rien d'autre à faire. VPython est une bonne bibliothèque, mais n'est pas compatible avec les dernières versions de Python3. On ne s'engagera dans la 3D que si on a déjà une bonne visualisation par ailleurs, ou alors si on a déjà beaucoup d'expérience de la 3D en Python.

[ck] PyQt, PyGTK, TkInter,. . . au choix

Les binômes doivent communiquer et intégrer leur code au projet commun de façon très régulière – les tâches ne sont pas indépendantes et la répartition ci-dessus n'est qu'une suggestion. Le choix de modélisation des monstres doit être fait au plus tôt, et concerne tout le monde.