# PV248 Python

Petr Ročkai and Zuzana Baranová

# Part A: Introduction

This document is a collection of exercises and commented examples of source code (in Python). All of the source code included here is also available as source files which you can edit and directly execute (we will refer to these files as the source bundle). Additionally, this section contains the rules and general guidelines that apply to the course as a whole.

The latest version of this document along with the source bundle is available both in the study materials in IS[1] and on the student server aisa:

- a PDF version of this text is called `pv248.seminar.pdf` and the source bundle is in directories `01` through `12`, `s1` through `s3` and `sol` – use the 'download as ZIP' option in the sidebar to get entire directories in one go,
- log into `aisa` using `ssh` or `putty`, run `pv248 update`, then look under `~/pv248` (this chapter is in subdirectory `00`).

We will update the files as needed, to correct mistakes and to include additional material. On `aisa`, running `pv248 update` at any time will update your working copies, taking care not to overwrite your changes. It will also tell you which files have been updated.

Each of the following chapters corresponds to a single week of the semester. The correspondence between exercises and the content of the lectures is, however, somewhat loose, especially at the start of the semester.

NB. If you are going to attend the lectures (you need to enroll separately, subject code is PV288), all you need at the start is intuitive familiarity with common programming concepts like classes, objects, higher-order functions and function closures (which can be stored in variables), as covered in e.g. PB006. You will get all the details that you may need in the lectures. On the other hand, if you are not going to attend lectures, you either need to already know all the theory, or you need to study it in your free time (this subject is purely practical).

## Part A.1: Course Overview

Since this is a programming subject, the coursework – and grading – will center around actual programming. There will be 2 types of programs that you will write in this subject: very small programs for weekly exercises (you should be able to solve these at the rate of 2-3 per hour) and small programs for homework (a few hundred lines and anything from a few hours to a day or two of work).

As you probably know by now, writing programs is hard and as a consequence, this course will also be rather hard. You will need to put in effort to pass the subject. Hopefully, you will have learned something by the end of it. Further details on the organisation of this course are in the remaining files in this directory:

- `a2_grading` – what is graded and how; what you need to pass,
- `a3_practice` – weekly 'practice' (preparatory) exercises,
- `a4_sets` – general guidelines that govern assignment sets,
- `a5_exam` – final exam (colloquium),
- `a6_reviews` – teacher and peer reviews,
- `a7_cheating` – don't.

Study materials for each week are in directories `01` through `12`. Start by reading `00_intro.txt`. The assignment sets are in directories `s1` through `s3`, one for each 4-week block (again, start by reading the intro).

The exercises for any given week will make use of the material covered in the lecture, though some weeks it will be a fairly loose fit. Especially when the lecture material is broad (like in weeks 1 and 2), the seminar will mainly include general programming exercises. Topics will get more specific and focused as the semester progresses.

The subject is divided into 4 blocks, each 4 weeks long. The first 3 blocks are during the semester, the last block is in the exam period. The topics covered are as follows:

| block | topic | date |
|---|---|---|
| 1 | 1. expressions, variables, functions | 13.9. |
| | 2. objects, classes, types, mypy | 20.9. |
| | 3. scopes, lexical closures | 27.9. |
| | 4. iterators, generators, coroutines | 4.10. |
| 2 | 5. memory management, refcounting | 11.10. |
| | 6. object and class internals | 18.10. |
| | 7. generators & coroutines cont'd | 25.10. |
| | 8. testing, profiling, pitfalls | 1.11. |
| 3 | 9. text, predictive parsing | 8.11. |
| | 10. databases, relations vs objects | 15.11. |
| | 11. asynchronous programming, http | 22.11. |
| | 12. math and statistics, recap | 29.11. |

The fourth block is relevant to you only if your ending type is 'colloquium' (see following sections for details) and does not bring any new material (since there are no lectures or seminars). It will instead feature reviews and a final test.

## Part A.2: Grading

To complete the course, you need to collect 60 points in each block. There are no other requirements.

NB. If your ending type is 'credit (z)' only blocks 1–3 are relevant for you (i.e. the fourth block in exam period, and hence the final exam, do not exist for you). If your ending type is 'colloquium (k)' then you need those 60 points in all blocks, including the 4th.

The subject entails a number of different activities, all of which are rewarded with points. The goal is to give you some freedom in what you want to focus on: the points are fully interchangeable. There are three main areas from which you can mix and match your preferences:

1. seminars: work out short exercises, attend the seminar, participate actively,
2. sets: work out more challenging exercises over a longer period of time (up to a month),
3. code review: write elegant code and help others do the same.

The points are split 2:2:1 between those areas. Please keep in mind that each block is graded separately: to pass the subject, you need to pass all four (or three, see above) blocks. In each block, the maximum is 120–150 points, while 60 are required to pass it. The available points are allocated as follows:

- max 60pt seminars (4 weeks, 15 points each week):
  - 6pt practice exercise sanity (1pt/exercise),
  - 3pt practice exercise verity (.5pt/exercise),
  - 3pt seminar attendance,
  - 3pt activity in the seminar,
- max 60pt task sets (4 tasks, 15 points each),

[1] https://is.muni.cz/auth/el/fi/podzim2022/PV248/um/

- max 30pt reviews of tasks from previous block:
  - 15pt teacher review (depending on grade and task),
  - 15pt peer reviews.

In block 4, there are no seminars nor tasks, but the final test in this block is worth 90pt.
Only points awarded within a block count toward passing it. That means, for instance, that all reviews of tasks from set 1 are counted in block 2.

## Part A.3: Practice Exercises

Each chapter in this exercise collection has 4 types of exercises: elementary, practice, regular and voluntary, (3 + 6 + 6 + 3, for a total of 18)[2], and a variable number of demonstrations (heavily commented code that illustrates a particular concept or construction).
The elementary and regular exercises come with reference solutions (in the folder `sol` in the source bundle, or in section K at the end of this document). The practice and voluntary exercises, on the other hand, focus on solving problems on your own.
The practice exercises are to be worked out and submitted before the corresponding seminar (where you will discuss your submitted solutions as a group). These exercises come with two sets of test cases:

1. 'sanity' which are enclosed (i.e. you can run them at your own leisure as you work on your solution) and
2. 'verity' which you cannot see, and will run only twice: Thursday 23:59 and then after the submission period closes on Saturday (again 23:59).

Two thirds of the points (1 point per exercise) are awarded on sanity tests alone (hence you can be sure that you have gained those points right after you submit). Test results will be visible in the notepads in the IS.
To submit the exercises, obtain a copy of the study materials using `pv248 update`, fill in the solutions, then use `pv248 submit` in the corresponding directory to submit them. Be sure to check the notepads to confirm that the submission was successful and that the tests passed.
The submission deadline is at 23:59 of the last Saturday before the corresponding seminar:

| block | unit | lecture | verity | deadline |
|---|---|---|---|---|
| 1 | 1 | 13.9. | 15.9. | 17.9. |
| | 2 | 20.9. | 22.9. | 24.9. |
| | 3 | 27.9. | 29.9. | 1.10. |
| | 4 | 4.10. | 6.10. | 8.10. |
| 2 | 5 | 11.10. | 13.10. | 15.10. |
| | 6 | 18.10. | 20.10. | 22.10. |
| | 7 | 25.10. | 27.10. | 29.10. |
| | 8 | 1.11. | 3.11. | 5.11. |
| 3 | 9 | 8.11. | 10.11. | 12.11. |
| | 10 | 15.11. | 17.11. | 19.11. |
| | 11 | 22.11. | 24.11. | 26.11. |
| | 12 | 29.11. | 1.12. | 3.12. |

Please be sure that you work out every graded exercise alone. Transgressions will be penalized (more details toward the end of this chapter).

## Part A.4: Task Sets

There are 3 sets of tasks and each has a 4-week window when it can be submitted. In total, you will have 12 attempts at every task, spread across the 4 weeks. The submission deadlines (i.e. the dates when 'verity' tests run) are at 23:59 on these days:

| set | week | Mon | Wed | Fri |
|---|---|---|---|---|
| 1 | 1 | 19.9. | 21.9. | 23.9. |
| | 2 | 26.9. | 28.9. | 30.9. |
| | 3 | 3.10. | 5.10. | 7.10. |
| | 4 | 10.10. | 12.10. | 14.10. |
| 2 | 1 | 17.10. | 19.10. | 21.10. |
| | 2 | 24.10. | 26.10. | 28.10. |
| | 3 | 31.10. | 2.11. | 4.11. |
| | 4 | 7.11. | 9.11. | 11.11. |
| 3 | 1 | 14.11. | 16.11. | 18.11. |
| | 2 | 21.11. | 23.11. | 25.11. |
| | 3 | 28.11. | 30.11. | 2.12. |
| | 4 | 5.12. | 7.12. | 9.12. |

**A.4.1 Submitting Solutions** The easiest way to submit a solution is this:

```
$ ssh aisa.fi.muni.cz
$ cd ~/pv248/s1
<edit files until satisfied>
$ pv248 submit s1_a_while
```

The number of times you submit is not limited (but not every submission will be necessarily evaluated, as explained below).
NB. Only the files listed in the assignment will be submitted and evaluated. Please put your entire solution into existing files.
You can check the status of your submissions by issuing the following command:

```
$ pv248 status
```

In case you already submitted a solution, but later changed it, you can see the differences between your most recent submitted version and your current version by issuing:

```
$ pv248 diff
```

The lines starting with `-` have been removed since the submission, those with `+` have been added and those with neither are common to both versions.

**A.4.2 Evaluation** There are three sets of automated tests which are executed on the solutions you submit:

- The first set is called syntax and runs immediately after you submit. Only 2 checks are performed: the code can be loaded (no syntax errors) and it passes mypy (strictness depending on the task at hand).
- The next step is sanity and runs every 6 hours, starting at midnight (i.e. 0:00, 6:00, 12:00 and 18:00). Its main role is to check that your program meets basic semantic requirements, e.g. that it recognizes correct inputs and produces correctly formatted outputs. The 'sanity' test suite is for your information only and does not guarantee that your solution will be accepted. The 'sanity' test suite is only executed if you passed 'syntax'.
- Finally the verity test suite covers most of the specified functionality and runs 3 times a week – Monday, Wednesday and Friday at 23:59, right after the submission deadline. If you pass the verity suite, the task is considered complete. The verity suite will not run unless the code passes 'sanity'.

Only the most recent submission is evaluated, and each submission is evaluated at most once in the 'sanity' and once in the 'verity' mode. You will find your latest evaluation results in the IS in notepads (one per assignment). You can still submit new versions after you pass 'verity' on a given task (e.g. because you want to improve the code for review). If your later submission happens to fail tests, this is of no consequence

[2] Some exercises and demonstrations are currently missing.

(the task is still considered complete).

**A.4.3 Grading** Each task that passes verity tests is worth 15 points. For those tasks, you can also get additional points for review, in the following block:

- set 1: 60 for correctness in block 1, 30 for reviews in block 2,
- set 2: 60 for correctness in block 2, 30 for reviews in block 3,
- set 3: 60 for correctness in block 3, 30 for reviews in block 4 (the exam one).

**A.4.4 Guidelines** The general principles outlined here apply to all assignments. The first and most important rule is, use your brain – the specifications are not exhaustive and sometimes leave room for different interpretations. Do your best to apply the most sensible one. Do not try to find loopholes (all you are likely to get is failed tests). Technically correct is not the best kind of correct.

Think about pre- and postconditions. Aim for weakest preconditions that still allow you to guarantee the postconditions required by the assignment. If your preconditions are too strong (i.e. you disallow inputs that are not ruled out by the spec) you may fail the tests.

Do not print anything that you are not specifically directed to. Programs which print anything that wasn't specified will fail tests. If you are required to print something or create strings, follow the format given exactly.

You can use the standard library. Third-party libraries are not allowed, unless specified as part of the assignment. Make sure that your classes and methods use the correct spelling, and that you accept and/or return the correct types. In most cases, either the 'syntax' or the 'sanity' test suite will catch problems of this kind, but we cannot guarantee that it always will – do not rely on it.

If you don't get everything right the first time around, do not despair. There are quite a few attempts to fix your mistakes (though if you aren't getting things right by the second or third attempt, you might be doing something wrong). In the real world, the first delivered version of your product will rarely be perfect, or even acceptable, despite your best effort to fulfill every customer requirement. Better get used to occasional setbacks.

Truth be told, only very small programs can be realistically (i.e. without expending considerable effort) written completely correctly in one go. Of course, what is 'very small' varies with experience: for some of you, assignments in sets will fall into this 'very small' category. Hopefully, for most of you, at least by the end of the course, the practice exercises will be 'very small' in this sense.

## Part A.5: Final Exam

If your ending type is 'colloquium (k)', you need to complete the 4th block, which takes place in the exam period. In this block, you can get points for reviews (for assignment set 3) and for the final exam, which consists of 6 exercises, each worth 15 points. Since you need 60 points to pass, that means either:

- 4 out of 6 exam exercises,
- 15+ points from reviews + 3 out of the 6 exam exercises.[3]

The exercises will come from this collection (types `p`, `r` and `v`) and so shouldn't be too hard to solve if you passed the first three blocks.[4] The main differences are:

1. you won't be able to consult anyone at the exam, for real (though this should have been the case during the semester too!),
2. no internet (and hence no googling and no `stackoverflow`), only this document (without the solution key) and the standard Python documentation.

In addition to the documentation, the exam computers will have the following software installed:

- Python interpreter (obviously) along with `mypy`,
- a selection of text editors, VS Code and Thonny.

You will have a total of 4.5 hours to solve the exercises and you will get interim 'verity' results (without counterexamples) every 90 minutes. Use of `mypy` at the exam will follow our use throughout the semester: passing `mypy` will be required (and checked by syntax tests, so you get early feedback), but strict mode will be only enabled for exercises with sufficiently simple types.

## Part A.6: Reviews

All reviews (teacher and peer) happen in the block that follows the block in which the code was written, i.e. reviews will happen in blocks 2-4. All code that passed verity tests is eligible for review (again both teacher and peer). If you submit (and pass) multiple tasks, reviewers can choose which they want to review.

While you cannot get reviews on code that failed verity tests, such solutions can be discussed in the seminar (with your tutor and with your classmates), even anonymously if you prefer. This should give you an idea where you made mistakes and how to improve in the future. Of course, this is only possible after the last deadline on the assignment passes.

**A.6.1 Peer Reviews** You may also participate as a reviewer through the peer review system (your solutions are up for peer review automatically). In addition to collecting points for the effort, we hope that the reviews you write will help you better understand how to read other people's code, and the ones you receive help you improve your own code and its understandability for others.

**A.6.2 Reading Reviews** The `pv248 update` command will indicate whether someone reviewed your code, by printing a line of the form:

```
A reviews/hw1.from.xlogin
```

To read the review, look at the files in `~/pv248/reviews/hw1.from.xlogin` – you will find a copy of your submitted sources along with comments provided by the reviewer.[5]

If you like, after you read your review, you can write a few sentences for the reviewer into `note.txt` in the review directory (please wrap lines to 80 columns) and then run:

```
$ pv248 review --accept
```

Your comments in `note.txt` will be sent to the reviewer through IS. Of course you can also discuss the review by other means.

**A.6.3 Writing Reviews** To write a review, start with the following command:

```
$ pv248 review --assignment s1_a_while
```

Substitute the name of the assignment you want to review (note that only tasks that you have successfully solved are eligible). A solution for you to review will be picked at random.

```
$ cd ~/pv248/reviews/
$ ls
```

There will be a directory for each of the reviews that you requested.

[3] Yes, in theory you can get 30 points on reviews, but don't count on it. There are too many variables. 15 is very possible though.

[4] If you are worried, you can change the ending type to 'z', at the expense of 1 credit. On the other hand, don't be worried, the exam isn't any harder than what you did during the semester, and there's plenty of time.

[5] There is also a copy in the study materials in IS, in the directory named `reviews`. Only you can see the reviews intended for you.

Each directory contains the source code submitted for review, along with further instructions (the file `readme.txt`).
When inserting your comments, please use double `##` to make the comment stand out, like this:

```
## A longer comment should be wrapped to 80 columns or less,
## and each line should start with the ## marker.
```

In each block, you can write up to 3 reviews. The limit is applied at checkout time: once you agree to do a particular review, you cannot change your mind and 'uncheckout' it to reclaim one of the 3 slots.

**A.6.4 Grading** All reviews carry a grade (this includes peer reviews), one of:

- A – very good code, easy to read, no major problems,
- B – not great, not terrible,
- C – you made the reviewer sad.

The points you get for the review depend on the grade:

- for teacher review, you get 15, 7.5 or 0 points,
- for peer review, the reviewer gets 2 points for writing the review in the first place, and a variable part that affects both the reviewer and the author of the code:
  - A: the reviewer was so impressed that they give the entire variable part of the reward to the coder (0 to reviewer, 2 to coder),
  - B: mixed bag, the reward is split (0.5 to reviewer, 1 to coder),
  - C: the reviewer keeps everything as a compensation for the suffering they had to endure (1 to reviewer, 0 to coder).

Or in a tabular form:

| | A | B | C |
|---|---|---|---|
| reviewer | 2 + 0 | 2 + 0.5 | 2 + 1 |
| coder | 2 | 1 | 0 |

NB. If you are writing a peer review: please include the grade as `@A`, `@B` or `@C` at the top (first line) of the file you are reviewing.

For teacher reviews, if you get a grade other than A, you can improve your solution and submit it again. The reviewers will have about 10 days to finish the reviews, then you have 9 days to resubmit an improved solution (but please note that the solution must still pass verity tests, which run on the usual Mon-Wed-Fri cadence, to be eligible for a second round of review). The relevant dates are:

| set | deadline | review | resubmit |
|---|---|---|---|
| 1 | 14.10. | 26.10. | 4.11. |
| 2 | 11.11. | 23.11. | 2.12. |
| 3 | 9.12. | 21.12. | 6.1. |

## Part A.7: Cheating

Seriously, don't. Cheating wastes everyone's time and it reflects really poorly on you. We consider any cooperation on practice exercises (type `p`) and sets (type `s`) – i.e. anything you get points for – to be cheating. There is plenty of other material for studying together.
If you cheat and get caught, you will (in every instance separately, where an instance is one week of practice exercises or a single assignment from a set):

- lose 1/2 of the points awarded for the affected work,
- lose 10 points in the block (i.e. you will need to have earned 70 on top of the points lost above),
- lose additional 10 points from the overall score (this means that, as a special exception, you can still fail even if you pass each block – you also need 180 or 240 points total depending on ending type to get a passing grade; with both 10-point penalties applied, you need 200 or 260 points to pass despite the cheating incident... on a second incident, this increases to 220 and 280 respectively, and so on).

Especially egregious cases of cheating will earn you a failing grade (X) immediately.

# Part 1: Python 101

As we have mentioned, each chapter is split into 4 sections: demonstrations, practice (preparatory) exercises, regular exercises and voluntary exercises. The demos are complete programs with comments that should give you a quick introduction to using the constructs that you will need in the actual exercises. The demos for the first week are these:

1. `list` – using lists
2. `tuple` – lists, sort of, but immutable
3. `dict` – using dictionaries
4. `str` – using strings
5. `fun` – writing functions

Sometimes, there will be 'elementary' exercises: these are too simple to be a real challenge, but they are perhaps good warm-up exercises to get into the spirit of things. You might want to do them before you move on to the practice exercises.

1. `fibfib` – iterated Fibonacci sequence

The second set of exercises are those that are meant to be solved before the corresponding seminar. The first batch should be submitted by 17th of September. The corresponding seminars are in the week starting on 19th of September. Now for the exercises:

1. `rpn` – Reverse Polish Notation with lists
2. `image` – compute the image of a given function
3. `ts3esc` – escaping magic character sequences
4. `alchemy` – transmute and mix inputs to reach a goal
5. `chain` – solving a word puzzle
6. `cycles` – a simple graph algorithm with dictionaries

The third section are so-called 'regular' exercises. Feel free to solve them ahead of time if you like. Some of them will be done in the seminar. When you are done (or get stuck), you can compare your code to the example solutions in Section K near the end of the PDF, or in the directory `sol` in the source bundle. The exercises are:

1. `permute` – compute digit permutations of numbers
2. `rfence` – the rail fence transposition cipher
3. `life` – the game of life
4. `breadth` – statistics about a tree
5. `radix` – radix sorting of strings
6. `bipartite` – check whether an input graph is bipartite

The final set are voluntary exercises, which have sanity tests bundled, but reference solutions are not provided. They are meant for additional practice, especially when you revisit previous chapters later in the semester.

1. (this section is empty for now)

## Part 1.1: Semantics

There are two fundamental sides to the 'programming language' coin: syntax and semantics. Syntax is the easy part (and you probably al-

ready know most, if not all, Python syntax). Syntax simply tells you how a (valid) program looks; on the other hand, semantics tells you what the program means, or in the simplest interpretation of meaning, what it does. While this is a question that can be attacked formally (i.e. using math), there is no need to worry – we will only talk about semantics intuitively in this course.
There are, in turn, two fundamental aspects of semantics:

- control – given the current situation, where does the program go next? which is the next statement or expression that will be executed?
- data – what are the values of variables, what are the results of expressions? what is the program going to output when it prints the 'thing' named `x`?

Clearly, there is interplay between the two: when the program encounters an `if` statement, which statement comes next depends on the result of the expression in the conditional. Intuitively, this is obvious. Sometimes, it is useful to be explicit even about things that are obvious. Just like with syntax and semantics, one of those aspects is clearly simpler: we all understand control quite well. You know what an `if` statement does, what a function call does (though we will revisit that), what a `for` or a `while` does. So let's focus on the other one, data. That turns out to be quite a bit trickier.[6]

## Part 1.2: Values and Objects

When talking about data, it might be tempting to start with variables, but this is usually a mistake: first, we need to talk about values and cells[7]. Because right at the start, there are some problems to resolve and they generally revolve about identity.
What is a value? Well, `1` is a value. Or `0`, or `2`, or `(0, 1)`, or `None`, or `[1, 2, 3]`, or `"hello"`, or any number of other things. What have all those things in common? First of all, they can all be stored in memory – at least for now, we will not worry about bits and bytes, just that a value is a piece of data that can be remembered (it doesn't even need to be in computer memory, you can remember them in your head).
What else can we do with values, other than remember them? We can perform operations on them: `1 + 1` is `2`, `"hello" + " " + "world"` is `"hello world"` and so on. Clearly, taking some values and performing operations on them produces new values. Imagine that we had `1` and then some other `1` – if we were to compute `1 + 2`, does it make a difference which one do we use? Obviously, it does not. Equal values are interchangeable: replacing `1` with another `1` will not change a program in any way. Values do not have an identity.
What is identity? Now that is a complicated philosophical question (no, really, it's been debated for well over 2 millennia).[8] Fortunately for us, it is much easier for us: an cell is created, then it is alive for a while and at some point it is destroyed. The identity of a cell is fixed (once created, it is always the same cell) and no two cells have the same identity.[9] It does not matter how much we change the cells (the technical term is mutate), it is still the same cell. And it is still different from all other cells.

[6] As easily shown by trying to explain an `if` to a non-programmer, vs explaining variables. Variables are hard.
[7] Terminology is hard. We could use object for what we call cell here, perhaps more intuitively, but that conflicts with the other meaning of the word 'object' in the object-oriented programming context. Which we are also going to need.
[8] You can look up the ship of Theseus for a well-known example. But the question was on people's minds long before that, at the very least all the way to Plato, 5 centuries earlier.
[9] It might be tempting to associate the identity with the address of the cell – where it is stored in memory. Tempting, but wrong: even though Python does not move cells in memory, it will re-use addresses, so an address of an cell that was destroyed can be used by a different cell later. But it is still a different cell even though it has the same address. However, there is a useful implication: if two cell have distinct addresses, they must be distinct cells (they have a different identity). Beware though, this does not hold universally in all programming languages!

So what is the relationship between cells and values? In a nutshell, a cell combines a value with an identity. There are two cases where the identity becomes important:

1. the behaviour of the program can directly or indirectly depend on the identity of a cell (e.g. by using an 'are these cells the same' operator, which is available in Python as `foo is bar`),
2. a value associated with a cell can change, i.e. the cell is mutable (in Python, this depends on the type of the cell: some are mutable, but some are immutable).

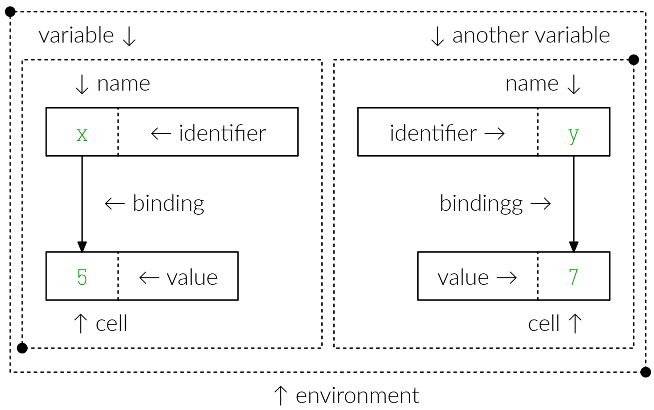
Out of these two, the latter is quite clearly much more important: in fact, the former rarely makes sense in the absence of the latter. Cell identity is only important in the presence of cell mutability. For immutable cells, we prefer to talk and think about values and disregard that they are perhaps associated with some cells (since the only other property of the cells, their identity, we do not care about).

## Part 1.3: Names and Binding

Now that we understand values and cells, we need to look at the other 'side' of variables: their names. There are no fewer than 3 important concepts that come into play:

- name itself is an identifier, usually an alphanumeric string (give or take a few chars), that the programmer uses to refer to a particular value or cell (depending on which of the two is the more important concept in the given context),
- binding associates a name with a cell (or, again, a value): you can visualise this as an 'arrow' connecting a name to its cell,
- environment is the collection of names and their bindings active at any given point in the execution of the program.

A picture is said to be worth a thousand words (note that the dashed boundaries do not necessarily represent anything 'physical' in the sense 'actually stored in memory at runtime' – they are there to delineate the concepts):




There is one more concept that perhaps clarifies the role of a name (as opposed to an identifier, which is a purely syntactic construction):

- scope is a property of a given name and gives bounds on the validity of that name: which parts of the program can refer to this name (notably, the same string can be associated with two different names, but only one of them might be in scope at any given time).

## Part 1.d: Demos

**1.d.1** [`list`] In Python, list literals are written in square brackets, with items separated by commas, like this:

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

Lists are mutable: the value of a list may change, without the list itself changing identity. Methods like `append` and operators like `+=` update

the list in place.
Lists are internally implemented as arrays. Appending elements is cheap, and so is indexing. Adding and removing items at the front is expensive. Lists are indexed using (again) square brackets and indices start from zero:

```
one = a_list[ 0 ]
```

Lists can be sliced: if you put 2 indices in the indexing brackets, separated by a colon, the result is a list with the range of elements on those indices (the element on the first index is included, but the one on the second index is not). The slice is copied (this can become expensive).

```
b_list = a_list[ 1 : 3 ]
```

You can put pretty much anything in a list, including another list:

```
c_list = [ a_list, [ 3, 2, 1 ] ]
```

You can also construct lists using comprehensions, which are written like for loops:

```
d_list = [ x * 2 for x in a_list if x % 2 == 1 ]
```

There are many useful methods and functions which work with lists. We will discover some of them as we go along. To see the values of the variables above, you can do:

```
python -i d1_list.py
>>> d_list
[2, 6]
```

**1.d.2** [tuple] Internally, tuples are immutable lists. The main difference in the implementation is that being immutable, tuples have a fixed number of elements. However, there are important use case differences: lists are usually homogeneous, with an arbitrary (unknown ahead of time) number of elements. Even when heterogeneous, they usually hold related types. Syntactically, tuples are written into parentheses and separated by commas. In many cases, the parentheses are optional, though. A one-tuple is denoted by a trailing comma,[10] while an empty tuple is denoted by empty parentheses (in this case, they cannot be omitted).

```
a_tuple = ( 1, 2, 3 )
b_tuple = 1, 2, 3 # same thing
c_tuple = ()      # empty tuple / zero-tuple
d_tuple = ( 1, )  # one-tuple
e_tuple = 1,      # also one-tuple
```

Tuples are usually the exact opposite of lists: fixed number of elements, but each element of possibly different type. This is reflected by the way they are constructed, but even more so in the way the are used. Lists are indexed and iterated (using for loops), or possibly filtered and mapped when writing functional-style code.
Tuples are rarely iterated and even though they are sometimes indexed, it's a very bad practice and should be avoided (especially when indexing by constants). Instead, tuples should be destructured using tuple binding:

```
a_int, b_int, c_int = a_tuple
```

As you might expect, a_int, b_int and c_int are newly bound variables with values 1, 2 and 3.

**1.d.3** [dict] Dictionaries (associative arrays) are another basic (and very useful) data structure. Literals are written using curly braces, with colons separating keys from values and commas separating multiple key-value pairs from each other:

```
a_dict = { 1: 1, 2: 7, 3: 1 }
```

In Python, dictionaries are implemented as hash tables. This gives constant expected complexity for most single-item operations (insertion, lookup, erase, etc.). One would expect that this also means that dictionaries are unordered, but this is not quite so (details some other day, though).
Like lists, dictionaries are mutable: you can add or remove items, or, if the values stored in the dictionary are themselves mutable, update those. However, keys cannot be changed, since this would break the internal representation. Hence, only immutable values can be used as keys (or, to be more precise, only hashable values – another thing to deal with later).
Most operations on items in the dictionary are written using subscripts, like with lists. Unlike lists, the keys don't need to be integers, and if they are integers, they don't need to be contiguous. To update a value associated with a key, use the assignment syntax:

```
a_dict[ 1 ] = 2
a_dict[ 337 ] = 1
```

To iterate over key-value pairs, use the items() method:

```
a_list = []

for key, value in a_dict.items():
    a_list.append( key )
```

You can ask (efficiently) whether a key is present in a dictionary using the in operator:

```
assert 2 in a_dict
assert 4 not in a_dict
```

Side note: assert does what you would expect it to do; just make sure you do not write it like a function call, with parentheses – that will give you unexpected results if combined with a comma.
Again, like with lists, we will encounter dictionaries pretty often, so you will get acquainted with their methods soon enough.

**1.d.4** [str] The last data type we will look at for now is str, which represents Unicode strings. Unlike lists and dictionaries, but quite like integers, strings in Python are immutable. You can construct new strings from old strings, but once a string exists, it cannot be updated.
There are many kinds of string literals in Python, some of them quite complicated. The basic variation consists single or double quotes (and there is no difference between them, though some programmers give them different semantics) enclosing a sequence of letters:

```
a_str = 'some string'
```

To access a single character in string, you can index it, like you would a list:

```
b_str = a_str[ 1 ]
```

Rather confusingly, the result of indexing a str is another str, with just one character (code point) in it. In this sense, indexing strings behaves more like slicing than real indexing. There is no data type to represent a single character (other than int, of course).
Since strings are immutable, you cannot update them in place; the following will not work:

```
a_str[ 1 ] = 'x'
```

Also somewhat confusingly, you can use += to seemingly mutate a string:

```
a_str += ' duh'
```

[10] This is a bit of a nuisance, actually, since leaving an accidental trailing comma on a line will quietly wrap the value in a one-tuple.

What happened? Well, `+=` can do two different things, depending on its left-hand side. If the LHS is a mutable type, it will internally call a method on the value to update it. If this is not possible, it is treated as the equivalent of:

```
c_str = 'string'
c_str = c_str + ' …and another'
```

which of course builds a new string (using `+`, which concatenates two strings to make a new one) and then binds that new string to the name `c_str`. We will deal with this in more detail in the lecture.
Important corollaries: strings, being immutable, can be used as dictionary keys. Building long strings with `+=` is pretty inefficient.[11] In essence, even though you can subscript them, strings behave more like integers than like lists. Try to keep this in mind.
As with previous two data types, we will encounter quite a few methods and functions which work with strings in the course. Also, the reference documentation is pretty okay. Use it. The most basic way to get to it is using the `help` function of the interpreter:

```
>>> help('')
>>> help({})
>>> help([])
```

Of course, you can also break out the web browser and point it to `https://docs.python.org/3` (unfortunately, `help` and the online docs don't always match, and the online docs are often more comprehensive; `help` is good for a quick overview, but if you don't see what you are looking for, refer to the web before despairing).

**1.d.5** [`function`] While not normally thought of as a data type, functions are an important category of entities that appear in programs. As will eventually become apparent, in Python, we would be justify to also call them objects (in other languages with first-class objects, this could be a bit confusing).
Functions are defined using the `def` keyword, or using the `lambda` keyword. The main difference is that the former is a statement while the latter is an expression (the other difference is that the former has a name, unlike the latter). The main thing that you can do with functions (other than defining them) is calling them. The mechanics of this are essentially the same in Python as in any other programming language that you may know. Frames (invocation records) are kept on a call stack, calling a function creates a new such record and returning from a function destroys its frame[12] and continues executing the caller where it left off.
As the existence of `lambda` foreshadows, we will be able to create functions within other functions and get closures. That will be our main topic the week after the next. For now, we will limit ourselves to toplevel functions (we will see methods next week).
The one possible remaining question is, what about arguments and return values? Let's define a function and see how that goes:

```
def foo( x ):
    x[ 0 ] = 1
    return x[ 1 ]
```

One conspicuous aspect of that definition is the absence of type annotations. We will address that next week – for now, we will treat Python like the dynamic language it is. Anything goes, as far as it works out at runtime. So how can we call `foo`? It clearly expects a list (or rather something that we can index, but a list will do) with at least 2 items in it.

```
a_list = [ 3, 2, 1 ]
two = foo( a_list )
```

Now depending on the argument passing mechanism, we can either expect `a_list` to remain unchanged (with items 3, 2, 1) or to be changed by the function to `[ 1, 2, 1 ]`. You are probably aware that in Python it is the second case. In fact, argument passing is the same as binding: the cells (objects) of the actual arguments are bound to the names of the formal arguments.[13] That's all there is to it.

```
assert a_list == [ 1, 2, 1 ]
```

However, you might be wondering about the following:

```
def bar( x ):
    x = 1

a_int = 3
bar( a_int )
```

If you think in terms of bindings, this should be no surprise. If you instead think in terms of 'pass by reference', you might have expected to find that `a_int` is `1` after the call. This is a mistake: if `3` was passed 'by reference' into `bar`, the `=` operator behaves unexpectedly. Again, if you remember that `=` is binding (as long as left-hand side is a name, anyway), it is clear that within `bar`, the name `x` was simply bound to a new value. To wit:

```
assert a_int == 3
```

To drive the point home, let's try the same thing with a list:

```
def baz( x ):
    x[ 0 ] = 2
    x = [ 3, 2, 1 ]
    x[ 0 ] = 1

b_list = [ 3, 3, 3 ]
baz( b_list )
```

Now we have 3 possible outcomes to consider:

- `[ 3, 3, 3 ]` – we can immediately rule this out based on the above,
- `[ 2, 3, 3 ]` – this would be consistent with all of the above,
- `[ 1, 2, 1 ]` – if `int` was actually handled differently from lists (spoiler: it isn't).[14]

```
assert b_list == [ 2, 3, 3 ]
```

We can demonstrate that all types of objects are treated the same quite easily using `id`, which returns the address of an object, and an `int` object:

```
x = 2 ** 100
y = 2 ** 101

assert id( x ) == id( x )
assert id( x ) != id( y )

z = x
assert id( z ) == id( x )
```

[11] CPython being what it is, of course there is a special case in the interpreter for `+=` on strings, when the reference count on the left-hand side is 1 (or rather 2, for complicated technical reasons). This even holds for `+` if it appears in a statement that looks like `foo = foo + bar`. Of course, relying on this optimisation is brittle, because you might have an unintended reference to the original, or you may introduce one long after you wrote the code with `+=`.

[12] This is a simplification, as we will see two weeks from now. Like many dynamic languages, Python allocates frames in the garbage-collected 'heap' and they are reclaimed by the collector (which would normally mean by reference count – so after all, they usually do get destroyed immediately… we will talk about this too, eventually).

[13] You will probably encounter this being labelled as 'call by reference'. This is not a great name for what is happening, but it's less bad than some of the other common misconceptions about function calls in Python. If you want to think of argument passing as being 'by reference', try to remember that everything is passed 'by reference' in this sense (mutable and immutable values alike).

[14] You might have read on the internet, or heard, that Python passes 'immutable values by value and mutable by reference'. This is not the case (in fact, it could reasonably be called nonsense).

```
z = y
assert id( z ) == id( y )
```

So how do we check what is passed to a function? If the object is the same on the inside and the outside, id will return the same value n both cases. Observe:

```
def check_id( x, id_x ):
    assert id( x ) == id_x

check_id( x, id( x ) )
```

## Part 1.e: Elementary Exercises

1.e.1 [fibfib] Consider the following sequences:

```
s[0] = 1 1 2 3 5 8 13 21 …
s[1] = 1 1 1 2 5 21 233 10946 …
s[2] = 1 1 1 1 5 10946 2.2112·10⁴⁸ 1.6952·10²²⁸⁷ …
s[3] = 1 1 1 1 5 1.6952·10²²⁸⁷ …
```

More generally:

- s[0][k] = fib(k) is the $k$-th Fibonacci number,
- s[1][k] = fib(fib(k)) is the s[0][k]-th Fibonacci number,
- s[2][k] = fib³(k) is the s[0][s[1][k]]-th Fibonacci number,
- and so on.

Write fibfib, a function which computes s[n][k].

```
def fibfib( n, k ):
    pass
```

## Part 1.p: Practice Exercises

1.p.1 [rpn] In the first exercise, we will implement a simple RPN (Reverse Polish Notation) evaluator.
The only argument the evaluator takes is a list with two kinds of objects in it: numbers (of type int, float, or similar) and operators (for simplicity, these will be of type str). To evaluate an RPN expression, we will need a stack (which can be represented using a list, which has useful append and pop methods).
Implement the following unary operators: neg (for negation, i.e. unary minus) and recip (for reciprocal, i.e. the multiplicative inverse). The entry point will be a single function, with the following prototype:

```
def rpn_unary( rpn ):
    pass
```

The second part of the exercise is now quite simple: extend the rpn_unary evaluator with the following binary operators: +, -, *, /, ** and two 'greedy' operators, sum and prod, which reduce the entire content of the stack to a single number. Think about how to share code between the two evaluators.
Note that we write the stack with 'top' to the right, and operators take arguments from left to right in this ordering (i.e. the top of the stack is the right argument of binary operators). This is important for non-commutative operators.

```
def rpn_binary( rpn ):
    pass
```

Some test cases are included below. Write a few more to convince yourself that your code works correctly.

1.p.2 [image] You are given a function f which takes a single integer argument, and a list of closed intervals domain. For instance:

```
f = lambda x: x // 2
domain = [ ( 1, 7 ), ( 3, 12 ), ( -2, 0 ) ]
```

Find the image of the set represented by domain under f, as a list of disjoint, closed intervals, sorted in ascending order. Produce the shortest list possible.
Values which are not in the image must not appear in the result. For instance, if the image is $\{1, 2, 4\}$, the intervals would be $(1, 2), (4, 4)$ – not $(1, 4)$ nor $(1, 1), (2, 2), (4, 4)$.

```
def image( f, domain ):
    pass
```

1.p.3 [ts3esc] Big Corp has an in-house knowledge base / information filing system. It does many things, as legacy systems are prone to, and many of them are somewhat idiosyncratic. Either because the relevant standards did not exist at the time, or the responsible programmer didn't like the standard, so they rolled their own.
The system has become impossible to maintain, but the databases contain a vast amount of information and are in active use. The system will be rewritten from scratch, but will stay backward-compatible with all the existing formats. You are on the team doing the rewrite (we are really sorry to hear this, honest).
The system stores structured documents, and one of its features is that it can format those documents using templates. However, the template system got a little out of hand (they always do, don't they) and among other things, it is recursive. Each piece of information inserted into the template is itself treated as a template and can have other pieces of the document substituted.
A template looks like this:

```
template_1 = '''The product '${product}' is made by ${manufacturer}
in ${country}. The production uses these rare-earth metals:
#{ingredients.rare_earth_metals} and these toxic substances:
#{ingredients.toxic}.'''
```

The system does not treat $ and # specially, unless they are followed by a left brace. This is a rare combination, but it turns out it sometimes appears in documents. To mitigate this, the sequences $${ and ##{ are interpreted as literal ${ and #{. At some point, the authors of the system realized that they need to write literal $${ into a document. So they came up with the scheme that when a string of 2 or more $ is followed by a left brace, one of the $ is removed and the rest is passed through. Same with #.
Your first task is to write functions which escape and un-escape strings using the scheme explained above. The template component of the system is known simply as 'template system 3', so the functions will be called ts3_escape and ts3_unescape. Return the altered string. If the string passed to ts3_unescape contains the sequence #{ or ${, return None, since such string could not have been returned from ts3_escape.

```
def ts3_escape( string ):
    pass

def ts3_unescape( string ):
    pass
```

1.p.4 [alchemy] You are given:

- a list of available substances and their quantities,
- a list of desired substances and their quantities,
- a list of transmutation rules, where each is a 2-tuple:
  - first element is the list of required inputs,
  - the second element is the list of outputs,
  - both input and output is a tuple of an element and quantity.

The sum of the quantities on the right hand side of the list is strictly less than that on the left side. Decide whether it is possible to get from the available substances to the desired, using the given rules: return a boolean. It does not matter whether there are leftovers. Rules can be used repeatedly.

```python
def alchemy(available, desired, rules):
    pass
```

The rules from tests in a more readable format, for your convenience:

- 3 chamomile, 4 water, 1 verbena, 2 valerian → relaxing concoction
- 7 ethanol → elixir of life
- 4 water, 2 mandrake, 2 valerian, nightshade → elixir of life
- 5 tea leaves → tea tree oil
- 2 primrose oil, 2 water, 1 tea tree oil → skin cleaning oil
- 1 iron, 1 carbon → steel
- 1 footprint → 1 carbon
- 6 ice → 5 water
- 3 steel → 1 cable
- 10 lead, philosopher stone, 2 unicorn hair → 10 gold

---

1.p.5 [chain] In this exercise, your task is to find the longest possible word chain constructible from the input words. The input is a set of words. Return the largest number of words that can be chained one after the other, such that the first letter of the second word is the same as the last letter of the first word. Repetition of words is not allowed. Examples:

- { goose, dog, ethanol } → 3 (dog – goose – ethanol)
- { why, new, neural, moon } → 3 (moon – new – why)

```python
def word_chain( words ):
    pass
```

1.p.6 [cycles] You are given a graph, in the form of a dictionary, where keys are numbers and values are lists of numbers (i.e. it is an oriented graph and its vertices are numbered; however, note that the numbering does not need to be consecutive, or only use small numbers).
Write a function, `has_cycle` which decides whether a cycle with at least one even-numbered vertex is reachable from vertex 1.
Hint: look up Nested DFS. Essentially, run DFS from vertex 1 and when you backtrack through an even-numbered vertex (i.e. in DFS postorder), run another DFS from that vertex to detect any cycles that reach the (even-numbered) initial vertex of the inner DFS. All the inner searches should share the 'visited' marks. Be careful to implement the DFS correctly.

```python
def has_cycle( graph ):
    pass
```

## Part 1.r: Regular Exercises

1.r.1 [permute] Given a number n and a base b, find all numbers whose digits (in base b) are a permutation of the digits of n.
Examples:

$(125)_{10} \to \{ 125, 152, 215, 251, 512, 521 \}$
$(1f1)_{16} \to \{ (1f1)_{16}, (f11)_{16}, (11f)_{16} \}$
$(20)_{10} \to \{ 20, 2 \}$

```python
def permute_digits( n, b ):
    pass
```

---

1.r.2 [rfence] In this exercise, you will implement the Rail Fence cipher algorithm, also called the Zig-Zag cipher.
The way this cipher works is as follows: there is a given number of rows ('rails'). You write your message on those rails, starting in the top-left corner and moving in a zig-zag pattern: ↘↗↘↗↘↗ from top to bottom rail and back to top rail, until the text message is exhausted.
Example: `HELLO_WORLD` with 3 rails

```
H...O...R..
.E.L._.O.L.
..L...W...D
```

The encrypted message is read off row by row: `HOREL_OLLWD`.
Your task is to write a function which, given the plain text and the number of rails/rows, returns the encrypted text:

```python
def encrypt(text, rails):
    pass
```

And another, which deciphers the text back to the plain text:

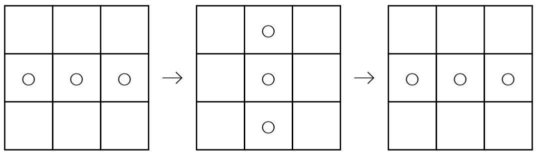
```python
def decrypt(text, rails):
    pass
```

---

1.r.3 [life] The game of life is a 2D cellular automaton: cells form a 2D grid, where each cell is either alive or dead. In each generation (step of the simulation), the new value of a given cell is computed from its value and the values of its 8 neighbours in the previous generation. The rules are as follows:

| state | alive neigh. | result |
|---|---|---|
| alive | 0–1 | dead |
| alive | 2–3 | alive |
| alive | 4–8 | dead |
| dead | 0–2 | dead |
| dead | 3 | alive |
| dead | 4-8 | dead |

An example of a short periodic game:

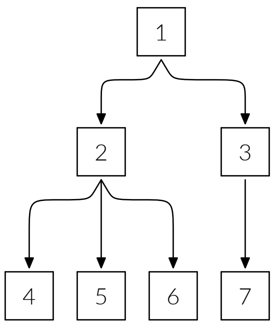


Write a function which, given a set of live cells, computes the set of live cells after n generations. Live cells are given using their coordinates in the grid, i.e. as `(x, y)` pairs.

```python
def life( cells, n ):
    pass
```

---

1.r.4 [breadth] Assume a non-empty tree with nodes labelled by unique integers:




We can store such a tree in a dictionary like this:

```python
def example_tree():
    return {1: [2, 3],
            2: [4, 5, 6],
            3: [7],
            4: [], 5: [], 6: [], 7: []}
```

Keys are node numbers while the values are lists of their (direct) descendants. Write a function which computes a few simple statistics about the widths of individual levels of the tree (a level is the set of nodes with the same distance from the root; its width is the number of nodes in it). Return a tuple of average, median and maximum level width.

```
def breadth(tree):
    pass
```

**1.r.5** [radix] Implement the radix sort algorithm for strings. Use a dictionary to keep the buckets, since the 'radix' (the number of all possible 'digits') is huge for Unicode. To iterate the dictionary in the correct order, you can use:

```
sorted( d.items(), key = lambda x: x[ 0 ] )
```

NB. Make sure that you don't accidentally sort the whole sequence using the built-in sort in your implementation.

```
def radix_sort( strings ):
    pass
```

**1.r.6** [bipartite] Given an undirected graph in the form of a set of 2-tuples (see below), decide whether the graph is bipartite. That is, whether each vertex can be assigned one of 2 colours, such that no edge goes between vertices of the same colour. Hint: BFS.
The graph is given as a set of edges $E$. For any $(u, v) \in E$, it is also true that $(v, u) \in E$ (you can assume this in your algorithm). The set of vertices is implicit (i.e. it contains exactly the vertices which appear in $E$).

```
def is_bipartite(graph):
    pass
```

# Part 2: Objects, Classes and Types

Starting this week, some of the exercises require static type annotations that can be checked with `mypy --strict` (this is the default: exercises, where static typing is tricky or burdensome will start with `# pragma mypy relaxed` – in those, you can use `Any` or even omit types entirely). The same principle will be true for tasks in task sets. In exercises and tasks where `mypy` is required, neither explicit `Any`, nor the `type: ignore` pragma can be used. These restrictions are enforced by the tests upon submission.
Demonstrations:

1. `mypy` – annotation basics
2. `class` – creating objects, with class
3. `generic` – polymorphic types and mypy
4. `type` – classes at runtime
5. `anno` – working with type annotations

Elementary exercises:

1. `geometry` – define basic types for planar geometry

Practice exercises:

1. `dsw` – Day, Stout & Warren balance binary trees
2. `ts3norm` – template system 3, normalization
3. `ts3render` – template system 3, rendering into strings
4. `bool` – boolean expression trees
5. `intersect` – computing intersections in a plane
6. `list` – linked list with generic type annotations

Regular exercises:

1. `json` – recursive data types without gross hacks
2. `rotate` – traversing a tree using rotations
3. `ts3bugs` – more fun with template system 3
4. `treap` – randomized search trees
5. `distance` – shortest distance between two 2D objects
6. `istree` – finding cycles in object graphs

Voluntary exercises:

1. (this section is empty for now)

## Part 2.1: Types and Type Systems

Before we get to classes and objects, we should talk about types. You have surely heard classifications like 'weakly/strongly typed' and 'statically/dynamically typed' (about programming languages, mainly). This is a bit of a distraction, because for a change, there is a simple definition of a type that works in many different contexts:

> A type is a (possibly infinite) set of values. A type system in turn is a set of types.

And that's basically it (with the caveat that we need to know what our values are). There is, however, one major downside of this otherwise very nice and very simple definition: we cannot do a whole lot with a type defined this way.
There is one thing that we can do though: we can test membership. The question 'is value `val` of type `T`?' turns out to be a very important one. It also readily generalizes to 'is any value that might appear here of type `T`?' – this is important, because we often do not know the exact value we are interested in (this information is only available at runtime). If you point at a variable, or more generally any expression, the set of values it can take is hard to pin down[15].
Before we proceed: why do we need to know whether some value, or expression, belongs to some type? Because we can use this information to argue about correctness of a program; in particular, we can talk about operations which only work for certain types of values. For instance, we could say that 'modulo' (remainder after division) is only defined for integers (this is a statement about types: 'integers' is a set of values, let's call this set `int`).
In languages where functions are values, we can also say things like 'set of all functions that accept an `int` and return an `int`' (we could call this type `int → int`). Clearly, this is a set of values, or in other words, a type. If we know that `f` is a function that belongs to `int → int` and we know that `not_a_number` is a value that is not of type `int`, we also know that `f( not_a_number )` is wrong.
Of course, treating types and type systems as just sets is not very practical. We will use these terms with the understanding that we need computational machinery to decide questions about types and how values relate to types, that we perhaps need syntax for annotations, and so on. But do keep the above 'core definition' in mind, even when we use the terms somewhat freely.
With that in mind, our definitions are sufficient to tentatively define the basic type system categories:

- a static type system can decide whether the value of a given expression (any value that it can take in the context of a fixed valid program) belongs to a given type,[16]
- a dynamic type system cannot do this (in general), but can still answer this question for a particular value at runtime.

And the other axis:

[15] The question whether a given variable can take a given value is, in general, undecidable (this should be quite obvious). We can also formulate the same observation this way: the exact set of values that a variable may take is not recursive (again, in general).

[16] There is obvious tension between the claim that this is, in general, an undecidable question, yet we claim that a static type system can provide an answer. This is because static type systems quite severely restrict what is considered a valid program.

- a strong type system will always give a correct answer to the membership question,
- a weak type system may fail to give a correct answer.

This latter classification seems a little puzzling. In practice, it is not binary: essentially all actual type systems are 'weak' in this sense, because they will allow the programmer to lie about types. However, the context in which an answer can be wrong is important – conventional 'weak' type systems can get the answer wrong even if the programmer didn't lie (but it is a little hard to give precise meaning to 'didn't lie', so we will leave it be).

Type systems that are both dynamic and weak are actually pretty rare, because they usually cannot erase[17] type information, and hence can use runtime type information to provide exact answers.

There are a few more interesting properties of type systems worth discussing:

1. Are types disjoint? If yes (each value belongs to exactly one type), the system is monomorphic. This is uncommon. Almost all systems allow some forms of polymorphism (values can belong to more than one type):
   - Is it true that, given two types $S$ and $T$, that either $S \subseteq T$ or $T \subseteq S$? In these cases, we say that a type system has subtyping and $T \subseteq S$ means that $T$ is a subtype of $S$ (conversely, $S$ is a supertype of $T$).
   - Parametric polymorphism is technically more difficult, but intuitively, a function is parametrically polymorphic if a single definition (function body) admits infinitely many types that conform to some 'type schema'.
   - Ad-hoc polymorphism (including 'duck typing'): neither of the above is true. Rules vary from language to language.
2. Can users define their own types? (The universal answer here seems to be 'yes'.) Given this is possible, how are new types constructed? There are two main categories:
   - algebraic data types: new types are created as products and disjoint sums of existing types (recursion is often allowed, where a type appears in its own definition),
   - inheritance: new types are created as subtypes of existing types.
3. Annotations: in which cases does the user need to explicitly mention types? Answers vary wildly depending on many factors:
   - every expression and variable (function parameter) needs annotations: an extreme that is basically never used in practice, but has some theoretical use (typed lambda calculus),
   - all variables/bindings (this extends to function parameters) need type annotations, and so do function return values, but types of expressions are inferred (languages like C or older versions of C++),
   - type information for local variables is not (usually) required, but function parameters and function return values require annotations (TypesSript and many other modern languages, newer revisions of C++, `mypy` in strict mode) – this is known as local inference,
   - only when values are explicitly constructed (literals usually carry an implicit type): see point 4 below.
4. There are 3 main categories of languages without (or with entirely optional) type annotations:
   - dynamic languages (traditional Python and many other high-level languages),
   - statically-typed languages with global inference (ML, traditional Haskell),
   - gradually-typed languages, where annotations are optional and they are checked statically when provided (to the degree that is possible: missing annotations are taken to mean that the type is dynamic, and all operations are considered valid on dynamic values) – the poster child of this category is `mypy` in non-strict mode.

There is a lot more to say about type theory, but most of it is irrelevant for Python, so we are going to skip all of it.

## Part 2.2: Classes and Objects

Like we have seen earlier with values and variables, it can be hard to pin down what objects and classes really are. Different languages give different semantics to what initially appears to be the same concept. Let's try to unravel the mess, of course with the semantics of Python firmly in our crosshairs.

In Python, objects and cells (from last week) are essentially the same thing. Outside of Python, the main difference is that objects bundle up operations (methods) in addition to data, but:

1. functions are really just values, so the semantic difference isn't huge (they can naturally appear as sub-values of a value stored in a cell),
2. there are no cells without methods in Python anyway – even traditional 'simple data' like integers come with methods.

While this may seem uncharacteristically simple, worry not: there is enough magic associated with Python objects (and we will get to it), but the magic is universal in the sense, that all (Python) cells are really objects.

So what about the other ingredient, classes? There are multiple ways to look at classes that are somewhat orthogonal (their applicability to different programming languages varies):

1. first and foremost, classes are types (mind you, not all types need to be classes, though in Python classes and types are almost the same thing) – that is, we can take a value (object) and decide whether it is an instance[18] of a particular class (i.e. it is a member of the type to which the class corresponds),
2. what more, classes are structured types – they really are products, in the sense that an object has multiple fields or sub-values that all exist at the same time (as opposed to sum types); in this sense, classes are like tuples,
3. classes describe a protocol: the names of methods, their signatures, and so on, that all instances of the class conform to (this is not quite the case in Python, but we can ignore that for now),
4. classes provide an implementation of this protocol:[19] method bodies appear in the definition of the class, and these bodies are shared by all instances:
   - this saves space, because there is no need to store each method in each instance,
   - it goes a long way toward consistency of behaviour: if a class only prescribes a protocol (which is only syntax), its instances are free to disagree on the behaviour – the semantics of a particular method,
5. classes are factories: they provide means to create new instances (i.e. to create new objects),
6. classes may be objects in their own right, with data and methods,

[17] Types are said to be erased if, at runtime, no information about types is attached to values. This is common in statically-typed languages, since they can resolve any questions about types at compile/analysis time. Maintaining type information at runtime can be quite costly, but in this case would provide little benefit.

[18] We will use the term instance to refer to objects in relation to 'their' class. Not all objects are necessarily instances in all programming languages, but in many languages, each object is an instance of some class.

[19] In some languages, there are special types of classes (abstract classes, interfaces) and/or methods (abstract methods, pure virtual methods) that do not provide an implementation. This is normally not needed in Python, because of duck typing. If required, such methods can be implemented as `raise NotImplemented()` – we will deal with exceptions some other day.

like any other objects; they may even be instances of some class (in Python, they are instances of the class type… by default) – that class is then known as a metaclass and, you guessed it, we will get to that some other day.

Especially interesting is reviewing points 1-5 in the light of point 6. What are the implications?

1. since classes are types, and classes are also objects, that means that types are objects and that means that they exist at runtime – there is no type erasure in Python (well, we already knew this),
2. perhaps more interestingly, this means that we can do type-level reflection at runtime 'for free', simply by calling methods on the class,
3. since classes exist as objects, instances can keep a (runtime) reference (this is what happens in Python), and delegate to this object in case they do not have a method 'of their own' (this is how implementation sharing happens); this is also part of the reason why methods have an explicit self argument,
4. because function calls are an operator and operators are defined by objects, calling a class is a perfectly legitimate thing to do: this simply runs a specific method of the class – this is how objects are created (but it gets complicated and involves metaclasses; we are shelving this for now),
5. inheritance is also implemented as delegation (unfortunately, because Python supports unrestricted multiple inheritance, the algorithm for lookup is ungodly).

## Part 2.3: Type Annotations

Though we established that types and classes are basically the same thing in Python, annotations are normally called type annotations and not class annotations, even though they can in fact be anything-annotations, as far as Python is concerned. This is valid Python:

```
def f() -> 3: pass
```

The code simply annotates the function's 'return type' to be 3 (not the return value mind you; it's an annotation, not the result). The interpreter only really cares that this is a valid expression that it can (and will) evaluate at runtime (at the time the function is defined)… try this:

```
def f() -> print( 'hello' ): pass
```

In any case, it is the job of an external type checker to make sense of the annotations (and of course, both of the above will be rejected by any sensible type checker). A typical example is mypy (and to some extent, it is the standard type checker, but there are others, e.g. pyright).
Now in the context of type systems, mypy is really 2 distinct systems with somewhat distinct properties. First, let's consider strict mypy, which is a more or less standard static type system with both subtyping and parametric polymorphism; it is also somewhat weak in the sense that:

1. it has cast so that the programmer can (accidentally) lie about types,
2. it can get things wrong on its own, because it relies on data flow analysis[20] to restrict union types. If you call g in the following program, it dies with a TypeError, but mypy --strict will accept it as correct:

```
x: None | int = 1

def f() -> None:
    global x
    x = None

def g() -> None:
    if x is not None:
        f()
        print( x + 1 )
```

```
Certainly, this is not the outcome we have hoped for. Without
the backing of Python's strong dynamic type system, we could
have ended up in real hot water.
```

Now the other mypy mode, without the --strict switch, is much weaker than the above. This is because any unannotated function or variable is quietly accepted as correct, including any expressions that make use of it. In non-strict mode, mypy will accept the following:

```
def h( x ):
    return x + 3

h( 'foo' )
```

Which is not to say that mypy, even in the non-strict mode, is useless: but it is important to understand the limitations. A compromise solution is using --strict but allowing oneself to use Any as an annotation, which is a way of saying 'I do not want to annotate this function, but I am aware the type is not going to be checked'. The latter two modes (non-strict and strict with Any) are examples of gradual typing.

## Part 2.d: Demos

**2.d.1** [mypy] In this unit (and most future units), we will add static type annotations to our programs, to be checked by mypy. Annotations can be attached to variables, function arguments and return types. In --strict mode (which we will be using), mypy requires that each function header (arguments and return type) is annotated. e.g. the function divisor_count takes a single int parameter and returns another int:

```
def divisor_count( n: int ) -> int:
```

Notice that variables, in most cases, do not need annotations: types are inferred from the right-hand side of the initial assignment.

```
    count = 0

    for i in range( 1, n + 1 ):
        if n % i == 0:
            count += 1
    return count

def test_divcount() -> None: # demo
    assert divisor_count( 5 )  == 2 # 1 and 5
    assert divisor_count( 6 )  == 4 # 1, 2, 3 and 6
    assert divisor_count( 12 ) == 6 # 1, 2, 3, 4, 6 and 12
```

For built-in types, including compound types, we will use the actual builtins for annotations (for simple types, like int and str, this has worked for a long time; for compound, types, like list or dict, it works from Python 3.9 onwards).
Compound types are generic, i.e. they have one or more type parameters. You know these from Haskell (they are everywhere) or perhaps C++/Java/C# (templates and generics, respectively). Like in Haskell but unlike in C++, generic types have no effect on the code itself – they are just annotations. Type parameters are given in square brackets after the generic type.

```
def divisors( n: int ) -> list[ int ]:
```

[20] For the more theoretically inclined: data flow analysis is clearly undecidable in general. Of course it's always possible to try some heuristics and if the result is 'don't know' simply reject the program as ill-typed. Unfortunately, that makes the system awfully restrictive, inconvenient and basically useless to a working programmer.

As mentioned above, for variables, mypy can usually deduce types automatically, even when they are of a generic type. However, sometimes this fails, a prominent example being the empty list – it's impossible to find the type parameter, since there are no values to look at. Annotations of local variables can be combined with initialization.

```
    res: list[ int ] = []

    for i in range( 1, n + 1 ):
        if n % i == 0:
            res.append( i )

    return res

def test_divisors() -> None: # demo
    assert divisors( 5 )  == [ 1, 5 ]
    assert divisors( 6 )  == [ 1, 2, 3, 6 ]
    assert divisors( 12 ) == [ 1, 2, 3, 4, 6, 12 ]
```

Finally, it is quite common in Python that a particular name (variable, function parameter) can accept values of different types. For these cases, mypy supports union types (in a direct reference to the 'types are sets' idea, those are literally unions of their constituent types, when understood as sets).

Before Python 3.10, the preferred way to write unions was to use helper classes from module typing: the more general Union[ S, T ] denotes the union of arbitrary two types (e.g. Union[ int, str ]). However, there is one very common union type, namely Union[ T, NoneType ] which can either take values from T or it can be None. Since it is so common, it can be written as Optional[ T ]. However, the new (Python 3.10) syntax is considerably nicer, and doesn't need extra imports:[21] S | T for a generic union and None | T for optional.

For example:

```
def maybe_push( stack: list[ int ], value: None | int ) -> None:

    if value is not None:
```

Notice how mypy accepts this code even though it is ostensibly ill-typed: on the face of it, value is not an int (the annotation above says it could be None), but stack only accepts elements of type int.

The code is accepted because the following line is guarded: the condition of the above if statement means that the branch is only taken if value is an actual int. In addition to conditionals, mypy will also understand assert statements of this sort.

```
        stack.append( value )

def push_either( int_stack: list[ int ], str_stack: list[ str ],
                 value: int | str ) -> None:
```

Of course is None is a pretty special case: normally, we will not want (or be able) to enumerate all possible values of a given type. However, mypy also understand isinstance:

```
    if isinstance( value, int ):
```

In this branch, mypy takes value to be of type int, since it is guarded by an isinstance.

```
        int_stack.append( value )
    else:
```

Of course, else branches are understood as well. In this case, value can be anything in int | str that isn't int, which just leaves str:

```
        str_stack.append( value )

def push_if_int( stack: list[ int ],
                 value: int | float | str ) -> None:
```

To make things more intuitive, isinstance actually lies about types. The original meaning of isinstance( x, c ) is 'is object x an instance of class c?'. But check this out:

```
    if isinstance( value, str | float ):
        pass
```

Clearly, value is not an instance of whatever type that union is, because str | list[ int ] does not evaluate to an actual superclass of str: there is only one, and that's object. And object is also a superclass of int, so things would go haywire there.

What on Earth is going on? Well, metaclasses, that's what. When you write isinstance( x, c ), the metaclass of c is consulted about the matter, and can 'claim' x as an instance of c even if they are completely unrelated (in the sense of inheritance). We will get back to this in a later chapter.

```
    else:
```

There is one last thing to illustrate: the else branch excludes both str and list[ int ], leaving only int:

```
        stack.append( value )

def test_pushes() -> None: # demo
    int_stack: list[ int ] = []
    str_stack: list[ str ] = []

    push_if_int( int_stack, 3 )
    push_if_int( int_stack, 'xoxo' )
    assert int_stack == [ 3 ]

    push_either( int_stack, str_stack, 'xo' )
    assert int_stack == [ 3 ]
    assert str_stack == [ 'xo' ]
```

Before we conclude this demo, there is one other case where variables need annotations, and it is when they are actually of union types:

```
def find_min( values: list[ int ] ) -> None | int:

    min_val: None | int = None

    for v in values:
```

Notice that the second operand of or is also treated as a proper branch by mypy: if it is ever evaluated, we already know that min_val is not None and hence is an int, which means it can be compared with v (which is also an int).

```
        if min_val is None or v < min_val:
            min_val = v

    return min_val
```

---

**2.d.2** [class] Up to a point, classes in Python follow the standard model derived from Simula and established by the likes of C++ and Java. As outlined in the introduction, classes are types (in the sense they are sets of values), while they also provide an interface and functionality to their instances.

The first major deviation from C++-like languages is that instance attributes are not listed in the class definition itself. They are instead created when an instance is initialized in the __init__ method. Like this:

```
class Base:
    def __init__( self, value ):
        self.value = value
```

[21] To make the same syntax work with Python 3.9, you can use from __future__ import annotations. This has the additional benefit of making forward references possible. The mechanism here is that with this __future__ import, annotations are stored as strings and they must be eval'd upon inspection to get the actual annotations. Since Python 3.10, the inspect module provides a helper to do that, get_annotations.

```
    def add( self, amount ):
        self.value += amount
```

You will immediately notice another difference: the self argument is explicit in all methods. What more, we also have to explicitly access all attributes (and methods) through self. This might take some getting used to. Let's define a derived class to observe a few more issues:

```
class Derived( Base ):
```

Another observation: self already exists at the time __init__ is called, so it is not a 'true' constructor. We will get into the woods of how objects are created and initialized some other day. And while there is a certain degree of magic in __init__ (it does get called automatically on new instances, after all), the amount of magic is quite limited. More specifically, __init__ methods of superclasses are not automatically called before the __init__ of the one in the current class.

```
    def __init__( self, value ):
        self.other = 2 * value
```

```
def test_classes(): # demo
```

To make an instance, simply 'call' the class itself, as if it was a function. Parameters passed to this call are forwarded to the __init__ method above (the self parameter is added by the internal machinery of object construction; clearly, at the point of the call, the object does not exist, so we can't pass it in explicitly even if we wanted to).

```
    base = Base( 3 )
```

Python being dynamic and duck-typed, we can use a builtin function, hasattr, to check whether an object has a particular attribute (notice that the second argument is a string, not an identifier; hasattr isn't that magic). Let's see:

```
    assert     hasattr( base, 'value' )
    assert not hasattr( base, 'other' )

    assert base.value == 3
```

Now for the derived class. We expect an other attribute to be present, but value to be missing, since we did not call __init__ from Base:

```
    deriv = Derived( 2 )
    assert not hasattr( deriv, 'value' )
    assert     hasattr( deriv, 'other' )

    assert deriv.other == 4
```

Let us make another derived class, to show how to call super-class constructors:

```
class MoreDerived( Derived ):
    def __init__( self, value, other ):
```

To call a method in the direct superclass, we can use super, which essentially creates a version of self that skips the current class during (method, attribute) lookup.

```
        super().__init__( other )
```

To reach indirect bases, we need to name them explicitly. Like this (note though that normally, the above call would itself call __init__ of Base, so we wouldn't have to do that here):

```
        Base.__init__( self, value )
```

However, super() does not bind the current instance in a 'normal' way: the object that 'falls out' of super does keep a reference to (our) self, but only uses it to bind the self parameter of methods when we call them through super. Instance attributes are nowhere to be found:

```
        assert not hasattr( super(), 'other' )
        assert     hasattr( super(), 'add' )
```

Last note about super: while it is ostensibly a 'normal' function call, it somehow gains access to self (not by name: it grabs the first argument, whatever it is named) and to its class: it does so by peeking into the call frame of its caller (we will talk in more detail about these next week). Hence, it only works in methods.

```
def test_super(): # demo
    deriv = MoreDerived( 1, 2 )
    assert hasattr( deriv, 'value' )
    assert hasattr( deriv, 'other' )
    assert deriv.value == 1
    assert deriv.other == 4
```

One last remark before we conclude this demo: there is one other crucial difference between C++ and Python. Since Python is dynamically typed through and through, the object bound to self is of the actual, dynamic type of that object and not, like in C++, the sub-object that corresponds to an instance of base itself. Worth keeping in mind.

2.d.3 [generic] In regular Python, generics (parametric types) don't make any sense: we can put any type anywhere, and collections are automatically heterogeneous (they can contain values of different types at the same time). While this is very flexible, it is also somewhat dangerous: you don't necessarily want a Car to find its way into a list of Cat instances.

To stand any chance of typing consistency, mypy needs to know the element types in collections, and for this reason, there are generics in mypy. They behave pretty much the way you would expect them to. Let's first look at generic functions. To express generic types, we need type variables – in Python, these are regular values (like any other type), with special meaning for the type checker. They are created as instances of typing.TypeVar:

```
from typing import TypeVar
```

Unfortunately, when creating a type variable, we need to pass its own name to it as a parameter. This is more than a little ugly.

```
T = TypeVar( 'T' )
```

Now that we have a type variable, we can declare a generic function. Notice that parametric types (list in this case) take their type parameters in square brackets. Remember that annotations are just regular Python expressions? This syntax simply re-uses the standard indexing operator. Hold on to that piece of info – we will look at it more closely when we get to metaclasses.

Within a single prototype, all mentions of T refer to the same type (but it can be any type, of course):

```
def head( records: list[ T ] ) -> T:
    return records[ 0 ]
```

Of course, in a new declaration, T is a fresh type, unrelated to the above T, even though it is technically the same value:

```
def tail( records: list[ T ] ) -> list[ T ]:
    return records[ 1 : ]
```

Unlike dynamic Python, and unlike generics (templates) in C++, there is no 'duck typing' for generic elements (i.e. for values of type T). Which means that there isn't a whole lot that you can do with them.

The things that are available by default are those that every Python object is assumed to provide:

- equality (but not ordering),
- conversion to a string (i.e. values of type T can be printed), and somewhat surprisingly,
- hashing (you can make a set of T, even if set[ T ] wasn't the type you started with, or use T as a key in a dict).

Due to the last point, the following will type-check just fine, but crash with a TypeError at runtime (another of those weak spots):

```
def make_set( value: T ) -> set[ T ]:
    return { value }
```

Let's check – you can run this file through mypy (even mypy --strict) and it will not complain. However, observe:[22]

```
def test_hashable() -> None: # demo
    try:
        make_set( [] )
        assert False

    except TypeError:
        pass
```

To add constraints to a type variable, we can use protocols, that offer additional capabilities for types of that variable[23] – there are a few builtin protocols, or you can make your own. Let's try with SupportsInt:

```
from typing import Sized, Protocol

S = TypeVar( 'S', bound = Sized )

def double( value: S ) -> int:
    return 2 * len( value )

def test_double() -> None: # demo
    assert double( 'foo' ) == 6
```

To make a protocol of your own, simply inherit from Protocol and add whatever methods you want to use, then use the protocol just like the one above.

```
class SupportsThings( Protocol ):
    an_attribute: int
    def a_method( self, value: int ) -> bool: ...

class AThing:
    def __init__( self ) -> None:
        self.an_attribute = 42
    def a_method( self, value: int ) -> bool:
        return True
```

If you only need to accept one value of the given type, you can use the protocol as an annotation directly (but if you mention it twice, unlike type variables, each value can be of a different type):

```
def use_a_thing( a_thing: SupportsThings ) -> None:
    assert a_thing.a_method( 3 )
    assert a_thing.an_attribute == 42
```

Or we can of course bind the protocol to a type variable, like with the one from typing:

```
R = TypeVar( 'R', bound = SupportsThings )

def use_two_things( a_thing: R, b_thing: R ) -> R:
    if a_thing.an_attribute > b_thing.an_attribute:
        return a_thing
    else:
        return b_thing
```

The last thing that we need to know about generics is how to make our classes generic (and hence accept a type parameter). Like with a protocol, simply inherit from Generic. You need to 'index' the Generic with some type variables, which then become bound to a single type in the entire scope of that class:

```
from typing import Generic

class ABox( Generic[ R ] ):
    def __init__( self, a_thing: R ) -> None:
        self.a_thing = a_thing
    def open( self ) -> R:
        return self.a_thing

def test_a_thing() -> None: # demo
    a_thing = AThing()
    b_thing = AThing()
    b_thing.an_attribute = 32

    a_box = ABox( a_thing )
    b_box : ABox[ AThing ] = ABox( b_thing )

    use_a_thing( a_thing )
    assert use_two_things( a_thing, b_thing ) is a_thing
    assert b_box.open() is b_thing
```

**2.d.4** [type] TBD type, isinstance.

**2.d.5** [anno] TBD (__annotations__).

## Part 2.e: Elementary Exercises

**2.e.1** [geometry] In this exercise, you will implement basic types for planar analytic geometry. First define classes Point and Vector (tests expect the coordinate attributes to be named x and y):

```
class Point:
    def __init__( self, x: float, y: float ) -> None:
        self.x = x
        self.y = y
    def __sub__( self, other: Point ) -> Vector: # self - other
        pass # compute a vector
    def translated( self, vec: Vector ) -> Point:
        pass # compute a new point

class Vector:
    def __init__( self, x: float, y: float ) -> None:
        pass
    def length( self ) -> float:
        pass
    def dot( self, other: Vector ) -> float: # dot product
        pass
    def angle( self, other: Vector ) -> float: # in radians
        pass
```

Let us define a line next. The vector returned by get_direction should have a unit length and point from p1 to p2. The point returned by get_point should be p1.

```
class Line:
    def __init__( self, p1: Point, p2: Point ) -> None:
        pass
    def translated( self, vec: Vector ) -> Line:
        pass
    def get_point( self ) -> Point:
        pass
    def get_direction( self ) -> Vector:
        pass
```

The Segment class is a finite version of the same. Also add a get_direction method, like above (or perhaps inherit it, your choice).

```
class Segment:
```

[22] The assert False must trip if make_set works normally. Which it doesn't. But if it threw anything other than a TypeError, we would not catch that and the program would crash, too. So, TypeError it is (if you are wondering if mypy somehow detects that we catch the TypeError and allows the program because of that – no, it doesn't… you can remove the try/except and observe the program crash, though mypy still claims everything is fine.

[23] If that reminds you of Haskell type classes, or C++ concepts, or Java constrained generics, you are spot on. It is the same idea. It is one of these things that keep coming up, just like generics themselves.

```
    def __init__( self, p1: Point, p2: Point ) -> None:
        pass
    def length( self ) -> float:
        pass
    def translated( self, vec: Vector ) -> Segment:
        pass
    def get_endpoints( self ) -> Tuple[ Point, Point ]:
        pass
```

And finally a circle, using a center (a `Point`) and a radius (a `float`).

```
class Circle:
    def __init__( self, c: Point, r: float ) -> None:
        pass
    def center( self ) -> Point:
        pass
    def radius( self ) -> float:
        pass
    def translated( self, vec: Vector ) -> Circle:
        pass
```

Equality comparison.

```
def point_eq( p1: Point, p2: Point ) -> bool:
    return isclose( p1.x, p2.x ) and \
           isclose( p1.y, p2.y )

def dir_eq( u: Vector, v: Vector ) -> bool:
    return isclose( u.angle( v ), 0 ) or \
           isclose( u.angle( v ), pi )

def line_eq( l1: Line, l2: Line ) -> bool:
    return dir_eq( l1.get_direction(), l2.get_direction() ) and \
           ( point_eq( l1.get_point(), l2.get_point() ) or
             dir_eq( l1.get_point() - l2.get_point(),
                 l1.get_direction() ) )
```

Please make sure that your implementation is finished before consulting tests; specifically, try to avoid reverse-engineering the tests to find out how to write your program.

## Part 2.p: Practice Exercises

**2.p.1** [`dsw`] Since a search tree must be ordered, we need to be able to compare (order) the values stored in the tree. For that, we need a type variable that is constrained to support order comparison operators:

```
class SupportsLessThan( Protocol ):
    def __lt__( self: T, other: T ) -> bool: ...

T = TypeVar( 'T', bound = SupportsLessThan )
```

Now that `T` is defined, you can use it to type your code below; values of `T` can be compared using `<` and equality (but not other operators, since we did not explicitly mention them above).

The actual task: implement the DSW (Day, Stout and Warren) algorithm for rebalancing binary search trees. The algorithm is 'in place' – implement it as a procedure that modifies the input tree. You will find suitable pseudocode on Wikipedia, for instance.

The constructor of `Node` should accept a single parameter (the value). Do not forget to type the classes.

```
class Node: pass # add ‹left›, ‹right› and ‹value› attributes
class Tree: pass # add a ‹root› attribute

def dsw( tree ): # add a type signature here
    pass
```

**2.p.2** [`ts3norm`] (continued from `01/p3_ts3esc`) Eventually, we will want to replicate the actual substitution into the templates. This will be done by the `ts3_render` function (next exercise). However, somewhat surprisingly, that function will only take one argument, which is the structured document to be converted into a string. Recall that the template system is recursive: before `ts3_render`, another function, `ts3_combine` combines the document and the templates into a single tree-like structure. One of your less fortunate colleagues is doing that one.

This structure has 5 types of nodes: lists, maps, templates (strings), documents (also strings) and integers. In the original system there are more types (like decimal numbers, booleans and so on) but it has been decided to add those later. Many documents only make use of the above 5.

A somewhat unfortunate quirk of the system is that there are multiple types of nodes represented using strings. The way the original system dealt with this is by prefixing each string by its type; `$document$ ` (with a trailing space!) and `$template$ `. Those prefixes are stored in the database. To make matters worse, there are strings with no prefix: earlier versions looked for `${` and `#{` sequences in the string, and if it found some, treated the string as a template, and as a document otherwise.

The team has rightly decided that this is stupid. You drew the short straw and now you are responsible for function `ts3_normalize`, which takes the above slightly baroque structure and sorts the strings into two distinct types, which are represented using Python classes. Someone else will deal with converting the database 'later'.

```
class Document:
    def __init__( self, text: str ) -> None:
        self.text = text

class Template:
    def __init__( self, text: str ) -> None:
        self.text = text
```

Each of the above classes keeps the actual text in a string attribute called `text`, without the funny prefixes. The lists, maps and integers fortunately arrive as Python `list`, `dict` and `int` into this function. Return the altered tree (without disturbing the original), the strings substituted for their respective types.

The mypy type for the function is simple on the surface, but the machinery that makes it type is ugly. On the logic that it is prepared for you, this exercise still requires passing strict mypy, because given `InputDoc` and `OutputDoc`, the function types are straightforward. You can use either `isinstance` or `type` and equality to guard code that uses specific union members – either is understood by `mypy`.

```
def ts3_normalize( tree: InputDoc ) -> OutputDoc:
    pass
```

**2.p.3** [`ts3render`] At this point, we have a structure made of `dict`, `list`, `Template`, `Document` and `int` instances. The lists and maps can be arbitrarily nested. Within templates, the substitutions give dot-separated paths into this tree-like structure. If the top-level object is a map, the first component of a path is a string which matches a key of that map. The first component is then chopped off, the value corresponding to the matched key is picked as a new root and the process is repeated recursively. If the current root is a list and the path component is a number, the number is used as an index into the list.

If a `dict` meets a number in the path (we will only deal with string keys), or a `list` meets a string, treat this as a precondition violation – fail an `assert` – and let someone else deal with the problem later.

At this point, we prefer to avoid the complexity of rendering all the various data types. Assume that the tree is only made of documents and templates, and that only scalar substitution (using `${path}`) happens. Bail with an `assert` otherwise. We will revisit this next week.

The `${path}` substitution performs scalar rendering, while `#{path}` substitution performs composite rendering. Scalar rendering resolves the path to an object, and depending on its type, performs the following:

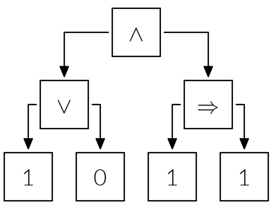
- `Document` → replace the `${…}` with the text of the document; the pasted text is excluded from further processing,

- Template → the ${…} is replaced with the text of the template; occurrences of ${…} and #{…} within the pasted text are further processed.

The top-level entity passed to ts3_render must always be a dict. The starting template is expected to be in the key $template of that dict. Remember that ##{…}, $${…} and so on must be unescaped but not substituted.
If you encounter nested templates while parsing the path, e.g. ${abc${d}}, give up (again via a failed assertion); however, see also exercise r3.

```
def ts3_render( tree: OutputDoc ) -> str:
    pass
```

2.p.4 [bool] In this exercise, we will evaluate boolean trees, where operators are represented as internal nodes of the tree. All of the Node types should have an evaluate method. Implement the following Node types (logical operators): and, or, implication, equality, nand. The operators should short-circuit (skip evaluating the right subtree) where applicable. Leaves of the tree contain boolean constants.
Example of a boolean tree:




In this case the or (∨) node evaluates to True, the implication (⇒) evaluates to True as well, and hence the whole tree (and, ∧) also evaluates to True.
Add methods and attributes to Node and Leaf as/if needed.

```
class Node:
    def __init__( self ) -> None:
        self.left  : Optional[ Node ] = None
        self.right : Optional[ Node ] = None

class Leaf( Node ):
    def __init__( self, value: bool ) -> None:
        self.truth_value = value
```

Complete the following classes as appropriate.

```
class AndNode:         pass
class OrNode:          pass
class ImplicationNode: pass
class EqualityNode:    pass
class NandNode:        pass
```

2.p.5 [intersect] We first import all the classes from e1_geometry, since we will want to use them.
What we will do now is compute intersection points of a few object type combinations. We will start with lines, which are the simplest. You can find closed-form general solutions for all the problems in this exercise on the internet. Use them.
Line-line intersect either returns a points, or a Line, if the two lines are coincident, or None if they are parallel.

```
def intersect_line_line( p: Line, q: Line ) \
        -> Union[ Point, Line, None ]:
    pass
```

A variation. Re-use the line-line case.

```
def intersect_line_segment( p: Line, s: Segment ) \
        -> Union[ Point, Segment, None ]:
    pass
```

Intersecting lines with circles is a little more tricky. Checking e.g. MathWorld sounds like a good idea. It might be helpful to translate both objects so that the circle is centered at the origin. The function returns a either None (the line and circle do not intersect), a single Point (the line is tangent to the circle) or a pair of points.

```
def intersect_line_circle( p: Line, c: Circle ) \
        -> Union[ None, Point, Tuple[ Point, Point ] ]:
    pass
```

It's probably quite obvious that users won't like the above API. Let's make a single intersect() that will work on anything (that we know how to intersect, anyway). You can use type( a ) or isinstance( a, some_type ) to find the type of object a. You can compare types for equality, too: type( a ) == Circle will do what you think it should.

```
def intersect( a: Union[ Line, Segment, Circle ],
               b: Union[ Line, Segment, Circle ] ) \
        -> Union[ None, Point, Line, Segment,
                  Tuple[ Point, Point ] ]:
    pass
```

2.p.6 [list] Implement a linked list with the following operations:

- append – add an item at the end
- join – concatenate 2 lists
- shift – remove an item from the front and return it
- empty – is the list empty?

The class should be called Linked and should have a single type parameter (the type of item stored in the list). The join method should re-use nodes of the second list. The second list thus becomes empty.

```
class Linked: pass
```

## Part 2.r: Regular Exercises

2.r.1 [json] As you might have noticed in prep exercises 2 and 3, the support for recursive data types in mypy is somewhat spotty. When designing new types, though, there is a compromise that types okay in mypy – at the price of creating a monomorphic wrapper for every recursive type in evidence. In other words, you can pick any two of [ recursive types, generic types, sanity ]. The JSON type will look as follows:

```
JSON = Union[ 'JsonArray', 'JsonObject', 'JsonInt', 'JsonStr' ]
JsonKey = Union[ str, int ] # for <get> and <set>
```

Now implement the classes JsonArray and JsonObject, with get and set methods (which take a key/index) and in the case of JsonArray, an append and a pop method. The set methods should also accept 'raw' str and int objects.

```
class JsonArray:  pass
class JsonObject: pass
```

The classes JsonStr and JsonInt are going to be a little special, since they should behave like str and int, but also provide get/set (which fail with an assertion) to make life easier for the user.

```
class JsonInt: pass
class JsonStr: pass
```

2.r.2 [rotate] You might be familiar with the zipper data structure, which is essentially a 'linked list with a finger'. Let us consider traversal of binary trees instead of lists. Implement two methods, rotate_left and rotate_right, on a binary tree object.
These methods shuffle the tree so that the left/right child of the current root becomes the new root. If rotating right, the old root becomes the left child of the new root, and the previous left child of the new root is

attached as the right child of the old root. If rotating left, the opposite. Notably, these rearrangements preserve the in-order of the tree.
Question: can we reach all nodes using just these two rotations? Can you think of an operation that, combined with the two rotations, would make the entire tree reachable? Can you think of a set of operations that make the entire tree reachable and preserve in-order? Learn more in S.1.

```
class Tree:
    def __init__( self, value ) -> None:
        self.left  : Optional[ Tree ] = None
        self.right : Optional[ Tree ] = None
        self.value = value

    def rotate_left( self ): pass
    def rotate_right( self ): pass
```

2.r.3 [ts3bugs] Let's pick up where p3_ts3render left off. It turns out that the original system had a bug, where a template could look like this: ${foo.bar}.baz} – if ${foo.bar} referenced a template and that template ended with ${quux (notice all the oddly unbalanced brackets!), the system would then paste the strings to get ${quux.baz} and proceed to perform that substitution.
The real clincher is that template authors started to use this as a feature, and now we are stuck with it. Replicate this functionality. However, make sure that this does not happen when the first part of the pasted substitution comes from a document!
The original bug would still do the substitution if the second part was a document and not a template. Feel free to replicate that part of the bug too. As far as anyone knows, the variant with template + document is not abused in the wild, so it is also okay to fix it.
Now the other part. If you encounter nested templates while parsing the path, first process the innermost substitutions, resolve the inside path and append the path to the outer one, then continue resolving the outer path.
Example: ${path${inner.tpl}}, first resolve inner.tpl, append the result after path, then continue parsing. If the inner.tpl path leads to a document with text .outside.2, the outer path is path.outside.2.

2.r.4 [treap]

```
class SupportsLessThan( Protocol ):
    def __lt__( self: T, other: T ) -> bool: ...
    def __le__( self: T, other: T ) -> bool: ...

T = TypeVar( 'T', bound = SupportsLessThan )
```

A treap is a combination of a binary search tree and a binary heap. Of course, a single structure cannot be a heap and a search tree on the same value:

- a search tree demands the value in the right child to be greater than the value in the root,
- a max heap demands that the value in both children be smaller than the root (and hence specifically in the right child).

Treap has therefore a pair of values in each node: a key and a priority. The tree is arranged so that it is a binary search tree with respect to keys, and a binary heap with respect to priorities.
The role of the heap part of the structure is to keep the tree approximately balanced. Your task is to implement the insertion algorithm which works as follows:

1. insert a new node into the tree, based on the key alone, as with a standard binary search tree,
2. if this violates the heap property, rotate the newly inserted node toward the root, until the heap property is restored.

The deeper the node is inserted, the more likely it is to violate the heap property and the more likely it is to bubble up, causing the affected portion of the tree to be rebalanced by the rotations. Remember that rotations do not change the in-order of the tree and hence cannot disturb the search tree property.

```
class Treap( Generic[ T ] ):
    def __init__( self, key: T, priority: int ):
        self.left  : Optional[ Node ] = None
        self.right : Optional[ Node ] = None
        self.priority = priority
        self.key = key

    def insert( self, val, prio ): pass
```

2.r.5 [distance] In case there are no intersections, it makes sense to ask about distances of two objects. In this case, it also makes sense to include points, and we will start with those:

```
def distance_point_point( a: Point, b: Point ) -> float:
    pass

def distance_point_line( a: Point, l: Line ) -> float:
    pass
```

If we already have the point-line distance, it's easy to also find the distance of two parallel lines:

```
def distance_line_line( p: Line, q: Line ) -> float:
    pass
```

Circles vs points are rather easy, too:

```
def distance_point_circle( a: Point, c: Circle ) -> float:
    pass
```

A similar idea works for circles and lines. Note that if they intersect, we set the distance to 0.

```
def distance_line_circle( l: Line, c: Circle ) -> float:
    pass
```

And finally, let's do the friendly dispatch function:

```
def distance( a: Union[ Point, Line, Circle ],
              b: Union[ Point, Line, Circle ] ) -> float:
    pass
```

2.r.6 [istree] We define a standard binary tree:

```
class Tree:
    def __init__( self ) -> None:
        self.left  : Optional[ Tree ] = None
        self.right : Optional[ Tree ] = None
```

However, not all structures built from the above data type are necessarily trees, since it's possible to create cycles. Write a predicate, is_tree, which decides if a given instance is actually a tree (i.e. it does not contain an undirected cycle).

```
def is_tree( tree ):
    pass
```

# Part 3: Lexical Closures

Demonstrations:

1. func – a few more features of function definitions
2. closure – the basic intuition and syntax
3. capture – mechanics of capturing variables

Preparatory exercises:

1. `merge` – combine items in a dictionary
2. `dice` – dicing and slicing lists
3. `newton` – finding roots with closures
4. `sort` – sorting and grouping with callbacks
5. `file` – make pure functions work with files
6. `counter` – keeping state

Regular exercises:

1. `fold` – folding lists
2. `trees` – folding trees
3. `bisect` – finding roots of general functions
4. `each` – traversing data structures
5. `objects` – a closure-based object system
6. `inherit` – the same, extended with simple inheritance

Voluntary exercises:

1. (nothing here yet)

## Part 3.1: Functions and Function Calls

The mechanics of function calls in Python are quite standard and very similar to what you would encounter in any mainstream imperative language. That is, the interpreter maintains a stack of activation records (also known as stack frames). Each record keeps essentially two things:

1. an equivalent of a return address – where in the program to continue when the current function returns,
2. an environment (in the technical sense from chapter 1) – i.e. bindings of names to their values; this environment is realized as local bindings and a reference to the lexically enclosing scope, where the interpreter looks for names that aren't bound locally.

The 'return address' is somewhat tricky to visualize in the standard Python syntax with expressions, because we might need to return into the middle of an expression. Consider:

```
x = f(a) + b
```

This is a statement with a function call in it – so far so good. But of course, when `f(a)` returns, we still need to do some work in that statement: namely, we need to take the return value of `f` and add `b` to it, and then assign the result into `x`. So the return address certainly can't be just a line number or some other reference to a statement. If you are tempted to say that we can remember the statement and just go looking for the call to `f`, this is not going to work either:

```
x = f(a) + f(b)
```

What CPython (the standard Python interpreter) does internally is use a bytecode representation of the program, in which the call is a separate 'instruction'. The above program fragment then becomes:

```
x1 = f(a)
x2 = f(b)
x  = x1 + x2
```

Now it is clear what a 'return address' is, because each line has at most a single function call, and if it has a function call, the only thing that additionally happens on that line is binding its result to a name.

There is one other important deviation between 'traditional' languages like C or C++ and Python (shared by many other dynamic languages) – the stack in Python is not continuous, but is rather a linked list of heap-allocated records. Why this is so should become apparent when we deal with lexical closures in the next section, and with coroutines in the next chapter.

There is one last important thing to note before we move on: a function declaration (using `def`) is a statement and hence can appear anywhere in the program where statements are permitted.[24] With that, we can move on to actually talking about closures.

## Part 3.2: Lexical Closures

Since `def` is a statement, this is a legal Python program:

```
def foo():
    def bar():
        pass
```

Of course, in isolation, this is not very interesting: the only difference at the first sight is that `bar` is only defined locally (i.e. it is a local binding, not visible outside of `foo`). That can be useful, but is not a real game changer. The following is:

```
def foo():
    x = 7
    def bar():
        return x
    x = 8
    assert x == bar()
```

In some sense this is nothing new: it is entirely normal that a function can use variables (and functions) defined in the same scope as itself. Say like this:

```
def quux():
    pass

def baz():
    quux()
```

This program is not surprising in any way, yet what is happening is the same thing as above: when a name is not found in the local scope, the lookup continues in the lexically enclosing scope. And when we say lexically, we mean syntactically (but the former term somehow got entrenched, even if it is not technically correct). Which means that, since `x` is not bound locally in `bar` above, when it is mentioned, the lookup initially fails. But the next enclosing lexical scope is that of `foo`, and sure enough, `foo` has a binding for `x`. So that binding is used.

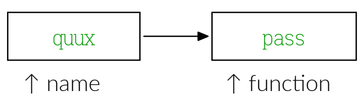
Why is this interesting? We need to go back to chapter 1 to appreciate what is going on here. The identifier `x` is still the same, but if we enter `foo` twice, like this:

```
foo()
foo()
```

we get two different names `x`. And each of those names has a possibly different binding. Which means that there are now two cells (objects) that correspond to the syntactic definition of `bar`. It is in this sense that the localness of that definition becomes important. Consider a simple case of an isolated function:

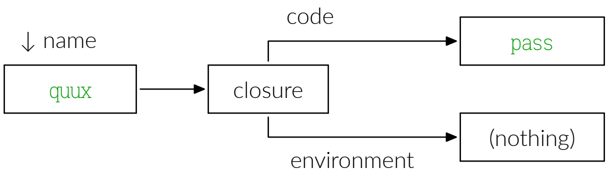
```
def quux():
    pass
```

When we take this apart, executing the two lines results in the following situation:

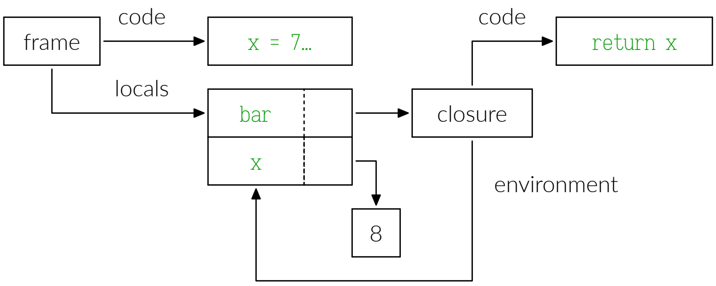



[24] Conversely, if you can put a `def` somewhere, you can put any other statement in that spot too… think about it.

In this sense, the function 'body' is an (immutable) object like any other.[25] Except the situation above is not how things actually look. It is a little more complicated:




For the foo defined earlier (with a nested bar function), we get something like this at the time of the assert (notice that we are now looking at the program in the middle of executing a function – the central object is an activation record, or a frame for short):




Well, that's the logical or semantic picture. The implementation is unfortunately somewhat different – unlike everything else in Python, the captured environment is not represented as a reference to the enclosing environment (a dict) at runtime. Instead, it is flattened into a tuple of reified cells.[26] The local environment then pretends that they are regular bindings, except they are actually accessed through these fake cells (you could call them pointers, or references, because that's basically what they are – a single nameless binding, or 'arrow'). Why? Two reasons:

1. We do not want the captured environment to keep all local variables from the enclosing function alive. If the inner function (the closure) is returned or stored somewhere, that would tie the entire scope's lifetime to that return value. This would be expensive. However, it's not super important right now – we will get back to object lifetime in the week after the next.
2. Remember how = is a (re)binding operator? Now that is a problem.

What's wrong with rebinding? Consider the following:

```
def foo():
    x = 7
    def bar():
        x = 8
        return x
    x = 9
```

If the inner function rebinds a captured binding, the connection breaks – the outer function has a different binding for x than the inner function. Actually, as written, x in bar is completely independent from x in foo, and this is just regular shadowing. But how about this:

```
def foo():
    x = 7
    def bar():
        x = x + 1
        return x
    x = 9
    bar()
```

Now this is an UnboundLocalError. Oh dear. That's because Python notices that you are binding x in bar which can only happen if x is a local variable in bar (i.e. not captured). So, an error. How to get around this? Tell Python explicitly that this is intended to be a capture:

```
def foo():
    x = 7
    def bar():
        nonlocal x
        x = x + 1
        return x
    x = 9
    bar()
    assert x == 10
```

But now we have a new problem. Initially, x is a capture, but if = on a nonlocal behaves like a regular binding, the connection still breaks: x in bar is bound to a new value (cell) and the assert fails. But it doesn't fail. Why doesn't it fail?

Because nonlocal captures are magic, as per the above. When a name is marked nonlocal, its behaviour changes in both the inner and the outer function. In the outer function, the binding is known as a cell variable[27] and all access to it is indirected through the same reified cell mechanism that is used for captures.[28] When you bind a nonlocal name, it is not the name that is being bound: it is the (normally invisible) cell that the name itself is bound to. Yes, it is a terrible hack (I am sorry). But this is the mechanism that makes the 'binding' apparently shared between the inner and the outer function.

You might find some consolation in the fact that the overall effect is equivalent to the inner function updating the outer environment, and the way things are actually implemented is simply an optimisation.

## Part 3.d: Demos

**3.d.1** [func] Before we proceed to look at closures, there are a few items that we should catch up on with regard to 'regular' functions. First, it's been mentioned earlier that def is really just a statement. There is more to that: it is a hidden binding (assignment) – saying def foo(): … is equivalent to foo = …, except there is no anonymous 'function literal' (lambda comes close, but is syntactically too different to be a realistic equivalent).

However, mypy really does not like rebinding names of functions (it treats functions specially, much more so than Python itself), so we won't get away with a demo of that. But you can try the following without mypy and it will work fine:

```
def foo(): return 1
def foo(): return 2
assert foo() == 2
```

Anyway, on to actually useful things. First, let's look at implicit parameters:

```
def power_sum( values: list[ float ], power: float = 1 ) -> float:
    total = 0
    for v in values:
        total += v ** power
    return total

def test_power_sum() -> None: # demo
```

[25] It does remember the name with which it was defined, but this is purely a technicality and can be ignored.

[26] The tuple is accessible as foo.__closure__ and the values stored in that tuple are, for better or worse, of a type actually called cell. Whether that is a happy or an unhappy coincidence is left to figure out as an exercise for the reader.

[27] You can find the names of such variables in a tuple called foo.__code__.co_cellvars.

[28] Access to cell variables and to captures (actually called free variables in the CPython interpreter) is through a separate set of opcodes that deal with the extra indirection through the cell object. When reading, the cell is automatically dereferenced. When binding, it is the cell that is rebound, not the name.

```
    assert power_sum( [ 1, 2 ] ) == 3
    assert power_sum( [ 1, 2 ], 2 ) == 5
```

This is basically self-explanatory. When we call the function, we may either supply the parameter (in which case power is bound to the actual value we provided) or not, in which case it is bound to the implicit value (1 in this case).
There is a well-known trap associated with implicit parameters. The problem is that the implicit binding takes place at the time def is evaluated (def is just a statement, remember?). Consider this function with an optional output parameter:

```
def power_list( values: list[ float ], power: float,
                out: list[ float ] = [] ) -> float:
    total = 0
    for v in values:
        out.append( v ** power )
    return sum( out )

def test_power_list() -> None: # demo
    assert power_list( [ 1, 2 ], 2 ) == 5
    assert power_list( [ 1 ], 1 ) == 6
```

Yes, that second assert is correct. When invoked a second time, the implicit binding of out is still the same as it was at the start, to some cell that was created when the def executed. And the second call simply appends more values to that same cell. It all makes sense, if you think about it. But it is not something you would intuitively expect, and it is easy to fall into the trap even if you know about it.
Just to drive the point home, let's consider the following (we need to pull in Callable for typing the outer function):

```
from typing import Callable

def make_foo() -> Callable[ [], list[ int ] ]:
    def foo( l: list[ int ] = [] ) -> list[ int ]:
        l.append( 1 )
        return l
    return foo

def test_foo() -> None: # demo
```

Evaluating the def again creates a new binding for the implicit parameter, as expected:

```
    assert make_foo()() == [ 1 ]
    assert make_foo()() == [ 1 ]
```

If we remember the result of a single def and call it twice, we are back where we started (again, as expected):

```
    foo = make_foo()
    assert foo() == [ 1 ]
    assert foo() == [ 1, 1 ]
```

Another feature worth mentioning are keyword arguments. In Python, with the exception of a couple special cases, all arguments are 'keyword' by default. That is, whether an argument is used as a keyword argument or a positional argument is up to the caller to decide. To wit:

```
def test_keyword() -> None: # demo
    assert power_sum( [ 1, 2 ], power = 4 ) == 17
    assert power_sum( values = [ 3, 4 ] ) == 7
```

There are some limitations: all positional arguments (in the call) must precede all keyword arguments – no backfilling is done, so you cannot skip a positional argument and provide it as a keyword. If you want to pass an argument using a keyword, you must also do that for all the subsequent (formal) arguments. Implicit arguments may be of course left out entirely, but if they are not, they take effect after supplied keyword arguments.
With that out of the way, the main exception from 'all arguments are keyword arguments' are variadic functions[29] that take a tuple of arguments (as opposed to a dict of them). Like this one:

```
def sum_args( *args: int ) -> int:
    total = 0
    for a in args:
        total += a
    return total
```

In the body of the function, args is a tuple with an unspecified number of elements which must all be of the same type, as far as mypy is concerned (Python as such doesn't care, obviously). Of course, that type might be an union, but the body might involve some isinstance gymnastics.
Anyway, the function is used as you would expect (of course, since no names are given to the arguments, they cannot be passed using keywords):

```
def test_sum_args() -> None: # demo
    assert sum_args() == 0
    assert sum_args( 1 ) == 1
    assert sum_args( 1, 2 ) == 3
    assert sum_args( 3, 2 ) == 5
```

There is another type of variadic function, which does permit (and in fact, requires keyword arguments):

```
def make_dict( **kwargs: int ) -> dict[ str, int ]:
    return kwargs

def test_make_dict() -> None: # demo
    assert make_dict() == {}
    assert make_dict( foo = 3 ) == { 'foo': 3 }
```

All of the above can be combined, but the limitations on call syntax remain in place. In particular:

```
def bar( x: int, *args: int, y: int = 0, **kwargs: int ) -> int:
    return x + sum( args ) + y + sum( kwargs.values() )
```

Note that y cannot be passed as a non-keyword argument, because *args is greedy: it will take up any positional arguments after x. On the other hand, if x is passed as a keyword argument, args will be necessarily empty and everything must be passed as keyword args.

```
def test_bar() -> None: # demo
    assert bar( 0 ) == 0
    assert bar( x = 0 ) == 0
    assert bar( 1, 1 ) == 2
    assert bar( 1, 1, y = 3 ) == 5
    assert bar( 1, 1, y = 3, z = 1 ) == 6
    assert bar( x = 1, z = 1 ) == 2
```

---

3.d.2 [closure] As explained earlier, closures happen when a function refers to a local variable of another function. This is really only relevant in languages:

1. with lexical scoping – variables are looked up syntactically, in the surrounding text (i.e. in a surrounding function, or a block, etc.) and not on the execution stack (that would be dynamic scope),
2. where functions are first class entities – that is, we can return them from other functions, bind them to local or global names, or accept them as parameters, etc.,
3. in which functions can be defined in local scopes (i.e. other than global/module and class scopes).

[29] The other exception are certain built-in functions, which have documented parameter names, but those names cannot be used as keywords in calls. E.g. int() takes, according to help(int) an argument called x, but you cannot write int( x = 3 ). You can, however, say int( '33', base = 5 ).

A language with all the above properties (e.g. Python) will naturally gain lexical closures, at least unless they are specifically forbidden. If we look away from implementation details, it is clear why this must be so:

```
from typing import Callable

Writer = Callable[ [ int ], None ]
Reader = Callable[ [], int ]
```

Let us implement a simple 'machine' for keeping a median (upward-biased, for simplicity) of a sequence, without exposing the sequence in any way. We will call it a `median_logger`. The idea is to only use the features from the above list and derive closures.

```
def median_logger() -> tuple[ Writer, Reader ]:
```

When `median_logger` executes, the first thing it does is create a new object – an empty list – and bind it to a local name, `items`.

```
    items: list[ int ] = []
```

Now we define a function that adds a value to the list. We can define a function here as per (3) and we can refer to `items` because it is in the lexical scope, as per (1). Do keep in mind that `items` is bound during execution of `median_logger`: it starts existing after the function is entered, and stops existing when it is left.[30]

```
    def writer( value: int ) -> None:
        items.append( value )
```

And another function to pull out the median (but nothing else). Again, `items` is in the lexical scope, so we can refer to it. It is the same `items` as above, used by `writer`, because we are in the same scope.

```
    def reader() -> int:
        items.sort()
        return items[ len( items ) // 2 ]
```

Since functions are first-class objects, per (2), we can return them. So we do:

```
    return writer, reader
```

For reference, let's add a very simple function which binds an empty list the same way, but returns the bound value directly (please excuse the redundancy):

```
def make_list() -> list[ int ]:
    items: list[ int ] = []
    return items

def test_median_logger() -> None: # demo
```

It should be obvious, but let's triple check that value (cell) construction in functions behaves the way we expect it to:

```
    l_1 = make_list()
    l_2 = make_list()
    assert l_1 is not l_2

    l_1.append( 1 )
    assert l_1 != l_2
```

Now with that sorted, let's get back to `median_logger`. Like above, let us make two 'copies' of whatever the function returns. This is interesting, because normally you would expect a function to be only defined once (and the function 'body' actually is) and the returned value to be the same.

30 Except it doesn't, because it is captured. But normally, that's exactly what would happen.

```
    w_1, r_1 = median_logger()
    w_2, r_2 = median_logger()
```

But alas, it is not: the behaviour aligns with `make_list`, which is just as well. While an implementation detail, we can also check that the 'bodies' actually are the same.

```
    assert w_1 is not w_2
    assert w_1.__code__ is w_2.__code__
```

Let's check the behaviour. We expect that `w_1` and `r_1` internally share the same list, i.e. adding elements using `w_1` will influence the result of `r_1`. And this is so:

```
    w_1( 5 )
    assert r_1() == 5
    w_1( 2 )
    w_1( 3 )
    assert r_1() == 3
```

While `w_1` is entirely independent from `r_2` and vice-versa:

```
    w_2( 10 )
    w_2( 10 )
    assert r_2() == 10 # and not 5
    assert r_1() == 3  # also not 5
```

If the above is clear, we can have a little peek at the internals. First, we check that it is the 'captures' that are different between `w_1` and `w_2`:

```
    assert w_1.__closure__ is not w_2.__closure__
```

And also that they are actually the same between `w_1` and `r_1`, even though those have different bodies:

```
    assert w_1.__code__ is not r_1.__code__
    assert len( w_1.__closure__ ) == 1
    assert len( r_1.__closure__ ) == 1
    assert w_1.__closure__[ 0 ] is r_1.__closure__[ 0 ]
```

## Part 3.e: Elementary Exercises

3.e.1 [`counter`]

```
K = TypeVar( 'K' )
V = TypeVar( 'V' )
R = TypeVar( 'R' )
```

The `make_counter` function should return a pair consisting of a function `fun` and a dictionary `ctr`, where `fun` accepts a single parameter of type `K`, which is also the key type of `ctr`. Calling `fun` on a value `key` then increments the corresponding counter in `ctr`. Don't forget the type annotations.

```
def make_counter(): pass
```

## Part 3.p: Practice Exercises

3.p.1 [`merge`]

```
class SupportsLessThan( Protocol ):
    def __lt__( self: K, other: K ) -> bool: ...

K = TypeVar( 'K', bound = SupportsLessThan )
V = TypeVar( 'V' )
W = TypeVar( 'W' )
```

Write a function `merge_dict` which takes these 3 arguments:

- a `dict` instance, in which some keys are deemed equivalent: the goal of `merge_dict` is to create a new dictionary, where all equivalent

keys have been merged; keys which are not equivalent to anything else are left alone (though the single value is still passed through `combine`),
- a `list` of `set` instances, where each `set` describes one set of equivalent keys (the sets are pairwise disjoint), and finally,
- a function `combine` which takes a `list` of values (not a set, because we may care about duplicates): `merge_dict` will pass, for each set of equivalent keys, all the values corresponding to those keys into `combine`.

In the output dictionary, create a single key for each equivalent set:

- the key is the smallest of the keys from the set which were actually present in the input `dict`,
- the value is the result of calling `combine` on the list of values associated with all the equivalent keys in the input `dict`.

Do not modify the input dictionary.

```
def merge_dict( dict_in: dict[ K, V ],
                equiv: list[ set[ K ] ],
                combine: Callable[ [ list[ V ] ], W ] ) \
        -> dict[ K, W ]:
    pass
```

3.p.2 [`dice`] Typing note: If you decide to use type annotations, be aware that they are quite heavy. What's worse, `zip_n_with` and `chunk_with` cannot be typed using `Callable` without resorting to `Callable[ ..., X ]` which is just a masked way to use `Any`. You have been warned (but it's still an interesting exercise to make it type, with this limitation in mind).
The `zip_with` function takes 2 lists and a callback and constructs a new list from results of applying the callback to pairs of items from the input lists (each item from one of the lists). Stop when the shorter list runs out.

```
def zip_with( func, list_1, list_2 ): pass
```

The `pair_with` function is similar, but only has a single input list and applies the callback to consecutive non-overlapping pairs of items in this list. Any unpaired items at the end of the list are thrown away.

```
def pair_with( func, items ): pass
```

The following two functions are like the above, but work with more than 2 items at a time. The lists in the `zip` case must be all of the same type (to make things typecheck).

```
def zip_n_with( func, *args ): pass
def chunk_with( func, chunk_size, items ): pass
```

3.p.3 [`newton`] Implement Newton's method for finding roots (zeroes) of differentiable, real-valued functions. The function `newton` takes 4 arguments: the function `f` for which we are finding the root, its first derivative `df`, the initial guess `ini` and the precision $p$ = `prec`. Return a number $x$, such that $\exists u \in \langle x-p, x+p\rangle . f(u) = 0$.
How it works: if you have an estimate $x_0$ for $x$, you can get a better estimate by subtracting $f(x_0)/f'(x_0)$ from $x_0$ (where $f'$ is the derivative, `df`). Repeat until satisfied (you can assume quadratic convergence, meaning that the error is bounded by the improvement one step earlier).

```
def newton( f, df, ini, prec ): pass
```

Using `newton`, implement a cube root function. Hint: given $z$ (the number to be cube-rooted), find a function $f(x)$ such that $f(x) = 0$ iff $z = x^3$. Clearly, the zero of $f$ is the cube root of $z$. The meaning of `prec` is the same as in `newton`.

```
def cbrt( z, prec ): pass
```

Note: if all inputs are integers, make sure the functions use integers throughout, so that they can be used with very large numbers. In type annotations, using `float` is OK, because mypy treats float as a superclass of `int` (which is very wrong, but alternatives are… complicated).

3.p.4 [`sort`] Implement the following functions:

- `sort_by` (with an order relation)
- `group_by` (with an equivalence relation)
- `nub_by` (likewise)

The order/equivalence relation are callbacks that take two elements and return a boolean. The order is given as less-or-equal: `order( x, y )` means `x <= y`.
The `sort_by` function should return a new list, sorted according to the order The sort must be stable (i.e. retain the relative order of items which compare equal).
The `group_by` function should return a list of lists, where each sublist contains equivalent items. Joining all the sub-lists together must yield the original list (i.e. the order of input elements is retained). The sub-lists must be as long as possible.
Finally `nub_by` should output a list where each equivalence class has at most one representative – the first one that appears in the input list. The relative order of items must remain unperturbed. In other words, if an item is equivalent (according to the provided equivalence relation) to an earlier item, do not include the new item in the output.

```
def sort_by( data, order ):   pass
def group_by( data, eq_rel ): pass
def nub_by( data, eq_rel ):   pass
```

3.p.5 [`file`] Your task is to write a function which takes:

- a list of input files,
- a function `get_name` which maps input filenames to output filenames.
- a pure function `fun` which maps strings to strings,

For each input file `file`, read the content, apply `fun` to that content and write the result to `get_name( file )`. Make sure things work if `get_name` is an identity function. Process the files left to right. Later files may be overwritten due to processing of earlier files.

```
def with_files( files, get_name, fun ): pass
```

3.p.6 [`ts3comp`] This is the final part of the 'template system 3' series of exercises (previously: `01/p3_ts3esc`, `02/p2_ts3norm`, `02/p3_ts3render` and `02/r3_ts3bugs`).
Our starting point this time is `02/p3_ts3render` – we will add support for missing data types and for rendering of composite data.
For scalar substitution (using `${…}`), add the following data types (on top of existing `Document` and `Template`):

- `int` → it is formatted as a decimal number and the resulting string replaces the `${…}`,
- `list` → the length of the list is formatted as if it was an `int`, and finally,
- `dict` → `.default` is appended to the path and the substitution is retried.

Composite rendering using `#{…}` is similar, but:

- a `dict` is rendered as a comma-separated (with a space) list of its values, after the keys are sorted alphabetically, where each value is rendered as a scalar,
- a `list` is likewise rendered as a comma-separated list of its values as scalars,
- everything else is an error: like before, treat this as a failed precondition, fail an `assert`, and leave it to someone else (or future you) to fix later.

Everything else about ts3_render is unchanged from the last time.

```
def ts3_render( tree: OutputDoc ) -> str:
    pass
```

## Part 3.r: Regular Exercises

**3.r.1** [fold] Implement foldr, a function which takes a binary callback f, a list l and an initial value i. Use the function f to reduce the list to a single value, from right to left. (Note: this is similar, but not the same as functools.reduce, due to different bracketing).

```
def foldr( f, l, i ): pass
```

Now use foldr to implement the following functions:

- fold_len – get the length of a list,
- fold_pairs – create a 'cons list' made of pairs, such that [1, 2, 3] becomes (1, (2, (3, ()))),
- fold_rev – reverse the input list.

You will probably need Any to type fold_pairs (there might be ways around it, but they are going to be ugly).

```
def fold_len( l ): pass
def fold_pairs( l ): pass
def fold_rev( l ): pass
```

**3.r.2** [trees]

```
T = TypeVar( 'T' )
S = TypeVar( 'S' )
```

Given the following representation of trees:

```
class Node( Generic[ T ] ):
    def __init__( self, val: T ) -> None:
        self.left  : Optional[ Node[ T ] ] = None
        self.right : Optional[ Node[ T ] ] = None
        self.val   : T = val

class Tree( Generic[ T ] ):
    def __init__( self ) -> None:
        self.root : Optional[ Node[ T ] ] = None
```

Implement a bottom-up fold on binary trees, with the following arguments:

- a ternary callback f: the first argument will be the value of the current node and the other two the folded values of the left and right child, respectively,
- the binary tree tree,
- an 'initial' value which is used whenever a child is missing (leaf nodes are folded using f( leaf_val, initial, initial)).

```
def fold( f, tree, initial ): pass
```

**3.r.3** [bisect] Write a function bisect, which takes $f$ which is a continuous function, two numbers, $x_1$ and $x_2$ such that $\mathrm{sgn}(f(x_1)) \neq \mathrm{sgn}(f(x_2))$ and precision $p$. Return $x$ such that $\exists z.x - p \leq z \leq x + p \wedge f(z) = 0$.

```
def bisect( f, x_1, x_2, prec ): pass
```

**3.r.4** [each] Write a function each that accepts a unary callback and a traversable data structure (one that is either iterable, or provides an each method). Arrange for f to be called once on each element.

```
def each( f, data ): pass
```

Use each to implement:

- each_len – count the number of elements
- each_sum – count the sum of all the elements
- each_avg – compute the average of all elements
- each_median – likewise, but median instead of average

```
(return the ⌊n/2⌋ element if there is no
 definite median, or None on an empty list)
```

```
def each_len( data ): pass
def each_sum( data ): pass
def each_avg( data ): pass
def each_median( data ): pass
```

**3.r.5** [objects]

```
T = TypeVar( 'T' )

class Obj( Protocol ):
    def __call__( self, __msg: str, *args: Any ) -> Any: ...
```

Build a simple closure-based object system and use it to model a pedestrian crossing with a button-operated traffic light. Design two objects:

- traffic_light – a 2-state light, either 'red' or 'green', toggled by messages set_red, set_green and queried using is_green; the set_green method operates immediately (is_green right after set_green returns True) but set_red has a safety timeout: the light turns red, but is_green will only become False after 5 seconds to clear the crossing,
- button – takes a reference to two traffic lights; when pushed (message push), it requests that the first is turned green, then after a timeout (20s), requests it to go back to red; the second light vice-versa; it must ensure that under no circumstances the lights both return is_green at the same time.

Every second, all objects in the system receive a tick message with no arguments.

```
def traffic_light(): pass
def button( pedestrian_light, vehicle_light ): pass
```

# Part 4: Iterators, Coroutines

Demonstrations:

1. (to be done)

Practice exercises in this chapter:

1. flat – flattening nested data with generators
2. send – understanding full coroutines
3. getline – coroutine-based data streams
4. lexer – more streams
5. parser – coroutine-based lexer + parser combination
6. mbox – event-based (SAX-like) parsing with coroutines

Regular exercises:

1. iscan – iterator-based scanning
2. gscan – similar, but with generators
3. itree – iterating a binary tree
4. gtree – generators vs trees
5. dfs – walking graphs with coroutines
6. guided – A* search with coroutines

Voluntary exercises:

1. (nothing here yet)

## Part 4.1: Iterators

Of the two concepts in this unit, iterators are by far the simpler. An iterator is, conceptually, a 'finger' that points at a particular element of a 'sequence' (you could say 'pointer' instead of 'finger', but that is an already wildly overloaded term).
Further, there are two more concepts with the same root as 'iterator', and that are closely related to them:

1. iterables, which are objects that can be iterated – you could perhaps call them sequences, but iterable is more general (we will get to it),
2. to iterate [an iterable x], which means to create an iterator for x and then use it (usually until it is exhausted, but not necessarily).

The three elementary operations on an iterator are:

1. check whether there are any items left in the sequence,
2. shift to the next item,
3. get the current item.

All three are actually implemented as a single operation in Python, called next. It has these effects:

1. check whether the sequence is empty, and if so, raise StopIteration (yes, really),
2. grab the current item (the sequence is not empty, so there is one) so that it can be returned later,
3. shift the 'finger' to the next element (mutating the iterator),
4. return the value grabbed in step 2.

This essentially tells you everything that you need to know about iterators to use them directly (call next repeatedly to get items and shift the iterator, until it raises StopIteration). However, an overwhelming majority of iterator uses are, in Python, implicit – either in a for loop:

```
for value in iterable:
    pass
```

or passed as an argument to a (library, builtin) function which consumes the iterable (e.g. list, sorted, map, sum and so on). You may notice that the results of many of those are in turn also iterable.
Notice the distinction between iterator and iterable: in Python, every iterator is an iterable, but the converse is not true: a list is iterable but is not an iterator – next([1]) is an error. To get an iterator for an iterable, you need to use the built-in function iter – next(iter([1])) works and evaluates to 1.[31] Notice that the call to iter is implicit in a for loop (i.e. you can really use an iterable that is not an iterator – probably quite obviously, since you can use for to iterate a list).
When using iterators, one additional property needs to be kept in mind – there are two flavours of iterables:

1. 'one shot' iterables, which are consumed by iterating over them, and hence can be iterated at most once (you could call them streams),
2. 'restartable' iterables, which can be iterated multiple times (these are what we normally think of as sequences).

By convention, one-shot iterables are their own iterators (as in, the iterator is literally the same object – not a different instance of the same class), though this is not required. In a complementary convention, iterators are one-shot iterables (recall that each iterator must be also an iterable), even when they are derived from a restartable iterable:

```
i = iter( [1, 2] )      # list is restartable
j = iter( i )           # list_iterator is one-shot
assert next( i ) == 1
assert next( j ) == 2
next( i )               # StopIteration
```

Additional technical details can be found in the demos (including a short 'how to write an iterator' tutorial).

## Part 4.2: Generators

Before confronting coroutines in their full generality, we will make a stop at so-called generators, or semi-coroutines. This puts us on a middle rung of three stages of generalisation:

1. functions can call any number of other functions; they may run forever or they eventually return to their caller once,
2. generators / semi-coroutines differ by the ability to return more than once, but again only to their caller – in this context, the 'returning' is called yielding because they can continue executing if resumed,
3. full coroutines can yield into arbitrary other coroutines – they are not restricted to keep returning into their 'caller'.

Before we go on, it is important to note that coroutines provide additional expressive power – they make certain things much easier to write – but in principle, they can be always simulated with functions and explicit state (or more conveniently using objects).[32]
In some sense, generators represent a 'sweet spot' between expressiveness and intuitiveness: full coroutines can be very hard to grasp (i.e. they can be very unintuitive), even though they provide additional power over generators. On the other hand, generators can provide a huge benefit in both readability and in ease of writing a particular piece of code.[33] We will have some opportunities to contrast explicit iterators with generators and the improvements the latter can yield (excuse the pun).
So how do we represent multiple 'returns' from a function? If these returns are all into the caller (as is the case with generators), we can think of the values that are being returned as a sequence, or a stream. It is not a coincidence that iterators and generators are closely related.
A common pattern for using generators is this:

1. call into the generator to obtain a value,
2. process the value,
3. resume the generator to obtain another value,
4. repeat until the generator is exhausted.

This does look an awful lot like iteration, and that is exactly how generators are commonly used in Python – the result of a generator function is automatically iterable (in fact, an iterator) and as such can be used in a for loop.
Like we did with iterators, we need to clear up some terminology:

- a generator function is what looks like a regular function except that it uses a yield keyword; when called, a generator function returns immediately and the result is
- a generator, which is an object that represents a suspended coroutine – it is this generator object that can be iterated.

Or, using an example:

```
def make_gen():
    print( 'this is gen' )
```

[31] Both iter and next simply delegate to the methods __iter__ and __next__ of the object they are called on. We will see this when we implement our own iterables and iterators.

[32] Of course, any program you can write using coroutines, you can rewrite without them. This is true of essentially all abstractions in all programming languages – all you really need to compute anything that can be computed is 2 unlimited counters and a conditional goto. By comparison, even Turing machines have awfully rich semantics.

[33] Code readability is a struggle between two opposing forces: readable code must be both simple (intuitive) but also succinct – code that is simple but long-winded is not readable, because the reader cannot hold all of it in their short-term 'working' memory. More abstraction usually means shorter code, but also more complicated. And since expressive power is really a measure of abstraction, a language that is 'too powerful' is just as bad as a language that is not powerful enough. Hence the seeking of middle ground.

```
    yield 3

gen = make_gen()
```

In this piece of code, make_gen is a generator function while gen is a generator. As written, the code does not print anything: the body of the generator function does not start executing at the time it is called. Instead, it is captured as a generator object and returned to the caller. This is very closely related to how lexical closures arise. Compare:

```
def make_fun():
    def fun():
        print( 'this is fun' )
        return 3
    return fun

fun = make_fun()
```

The result in this case is a function object (i.e. a lexical closure), while it was a generator object in the generator case above. In both cases, to actually perform the code, the gen / fun object needs to be used. How that is done of course differs: to use fun, we simply call it: fun(). With gen, we instead iterate it – Python has provided a built-in __next__ method for the generator object (just like it provided a __call__ method for the function object) that interacts with the coroutine:

1. calling next resumes the coroutine,
2. if a yield is encountered, the coroutine is suspended and next returns the yielded value,
3. if a return is encountered, the coroutine is destroyed, its return value is wrapped in a StopIteration object and raised by next.

Since for ignores the value inside StopIteration, in most situations, the return value (as opposed to values passed to yield) is worthless. Nonetheless, it can be obtained when a generator is used directly, though this is not common.

## Part 4.3: Coroutines

As explained earlier, coroutines can be understood as a further generalisation of generators. In fact, what Python calls a generator turns out to be, almost by accident, a full coroutine. Like in case of lexical closures, this is the result of a 'conspiracy' of a few seemingly unrelated features:

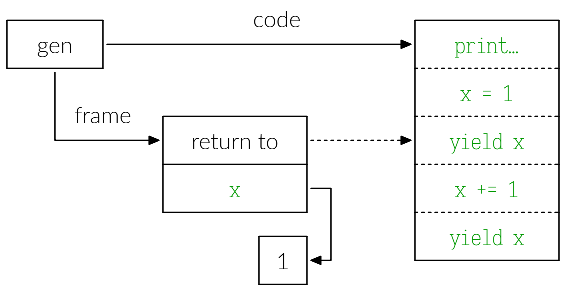
1. generators exist, obviously,
2. suspended generators are first-class objects,
3. generators actually return into the caller of next.

The second point has been made implicitly earlier: a generator function returns a generator object and the latter is the requisite first-class representation of a suspended generator.
Now recall that in the initial definitions, we have demanded that semi-coroutines (generators) can return multiple times into their caller. But in Python, this is coincidental: the generator returns into the function that called next – and usually, that is indeed the caller, because generators are normally used directly in a for loop. But since we can pass the suspended coroutine around, anyone can call next and the next yield in the coroutine's body will transfer control back to the caller of next, not to the original caller of the coroutine. Again, it is purely by convention that these two functions are usually the same.
Then there is an added bonus: suspended generators can be resumed by calling their send built-in method, instead of using next. In fact, next( coro ) is equivalent to coro.send( None ). What does it do is that the yield on which the coroutine was previously suspended returns a value. Specifically, it returns whatever was given to send as an argument.[34] And while this ability is not strictly necessary (we can send values from one coroutine to the next by other means), it makes using Python generators as 'full' coroutines a bit more convenient.
So what does all this mean in practice? We already have some experience with lexical closures, which are functions with some additional captured state. Generators are like that, except they also remember a sort of 'return address' – in this case, an address which tells the interpreter where to continue executing the coroutine when it is resumed. Since all local variables of the generator function are, by construction, used by the generator object, the generator object really keeps the entire frame of its 'parent' function:

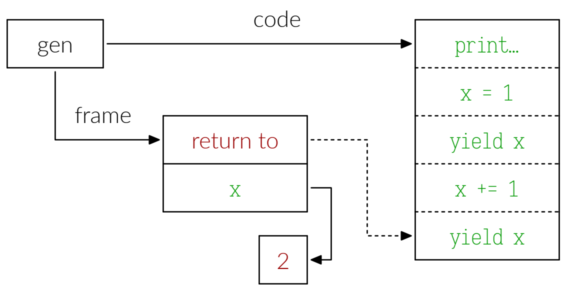
```
def make_gen():
    print( 'hello' )
    x = 1
    yield x
    x += 1
    yield x

gen = make_gen()
next( gen )
```

And the corresponding picture, at the point right after the last statement above:




And after calling next( gen ) a second time (changes are highlighted):




## Part 4.d: Demos

4.d.1 [iter] TBD: How to write an iterator.

4.d.2 [gen] Normally, generators are used in for loops. However, when you simply call a generator, the result is an object of type generator, which represents the suspended computation. (For future reference, native coroutines declared with async def behave the same way, just the object type is different.)
Let's define a generator:

```
def gen1() -> Generator[ int, None, None ]:
    print( "before yield 1" )
    yield 1
    print( "before yield 2" )
    yield 2
```

To actually run the computation, you can call __next__() on the gen-

[34] See the demos for an executable example.

erator object. Alternatively, you can call next with generator object as the argument. Once you do that, the execution of the body of gen1 starts, and continues until it hits a yield. At that point, the yielded value becomes the return value of __next__(), like this:

```
def test_gen1() -> None: # demo
    x = gen1()
    print( "constructed gen1" )
    assert x.__next__() == 1
    print( "no longer interested in gen1...\n" )
```

Since x is just a normal object, we can abandon it at any time. Nothing forces us to keep calling __next__() on it. Let's look at send() now.

```
def gen2() -> Generator[ int, int, None ]:
    v = yield 1
    print( "received", v )
    yield 2
    print( "returning from gen2()" )
    pass # StopIteration is automatically raised here

def test_gen2() -> None: # demo
    y = gen2()
    assert y.__next__() == 1
    assert y.send( 24 ) == 2 # resumes execution of ‹y›
    print( "sent 24, got 2 back" )
    try: y.__next__() # generators do not return
    except StopIteration: print( "generator done" )
```

---

4.d.3 [trampoline] In languages with 'general' or 'full' coroutines, the yield (or resume) operation takes the form:

```
result = yield <value> to <coroutine>
```

This cannot be directly written in Python, yet earlier we have claimed that thanks to first-class nature of coroutines and the fact that they can be resumed from anywhere (though they cannot themselves yield to arbitrary other coroutines).

## Part 4.p: Practice Exercises

4.p.1 [flat] Write a generator that completely flattens iterable structures (i.e. given arbitrarily nested iterables, it will generate a stream of scalars). Note: while strings are iterable, there are no 'scalar' characters, so you do not need to consider strings.
Note: This function is unreasonably hard to type statically with mypy. Feel free to use Any for the items (but try to give a correct 'outer' (top-level) type for both the argument and the return value).

```
def flatten( g ):
    pass
```

4.p.2 [send] Write two generators, one which simply yields numbers 1-5 and another which implements a counter (which also starts at 1): sending a number to the generator will adjust its value by the amount sent. Then write a driver loop that sends the output of numbers() into counter(). Try adding print statements to both to make it clear in which order the code executes.

```
def numbers(): pass # generate numbers 1-5
def counter(): pass
def driver():  pass # another generator - the driver loop
```

After you are done with the above, implement the same thing with plain objects: Numbers with a get() method and Counter with a get() and with a put( n ) method.

```
class Numbers:    pass
class Counter:    pass
def driver_obj(): pass # a driver loop again, now with objects
```

---

4.p.3 [getline] This is the first in a series of exercises focused on working with streams. A stream is like a sequence, but it is not held in memory all at once: instead, pieces of the stream are extracted from the input (e.g. a file), then processed and discarded, before another piece is extracted from the input. Some of the concepts that we will explore are available in the asyncio library which we will look at later. However, for now, we will do everything by hand, to get a better understanding of the principles.
A stream processor will be a (semi)coroutine (i.e. a generator) which takes another (semi)coroutine as an argument. It will extract data from the 'upstream' (the coroutine that it got as an argument) using next and it'll send it further 'downstream' using yield.
For now, we will use the convention that an empty stream yields None forever (i.e. we will not use StopIteration). A source is like a stream processor, but does not take another stream processor as an argument: instead, it creates a new stream 'from nothing'. A sink is another variation: it takes a stream, but does not yield anything – instead, it consumes the stream. Obviously, stream processors can be chained: the chain starts with a source, followed by some processors and ends with a sink.
To see an example, look near the bottom of the file, where we define a simple source, which yields chunks of text. To use it, do something like: stream, cnt = make_test_source(). The cnt variable will keep track of how many chunks were pulled out of the stream – this is useful for testing.
What follows is a very simple sink, which prints the content of the stream to stdout (might be useful for tinkering and debugging):

```
def dump_stream( stream: Iterator[ Optional[ str ] ] ) -> None:
    while True:
        x = next( stream )
        if x is None: break
        print( end = x )
```

Your first goal is to define a simple stream processor, which takes a stream of chunks (like the test source above) and produces a stream of lines. Each line ends with a newline character. To keep in line with the stated goal of minimizing memory use, the processor should only pull out as many chunks as it needs to, and not more.

```
def stream_getline( stream ):
    pass
```

---

4.p.4 [lexer] In the second exercise in the stream series, we will define a simple stream-based lexer. That is, we will take, as an input, a stream of text chunks and on the output produce a stream of lexemes (tokens). The lexemes will be tuples, where the first item is the classification (a keyword, an identifier or a number) and the second item is the string which holds the token itself.
Let the keywords be set, add and mul. Identifiers start with an alphabetic letter and continue with letters and digits. Numbers are made of digits. You can use StrStream below as a template for writing the type of a 'lexeme stream'.

```
StrStream = Generator[ Optional[ str ], None, None ]

IDENT = 1
KW = 2
NUM = 3

def stream_lexer( text_stream ):
    pass
```

---

4.p.5 [parser] In this exercise, we will write a very simple 2-stage parser (i.e. one with a separate lexer) using coroutines (one for the lexer and one for the parser itself). The protocol is as follows:

- the parser will get the lexer in the form of a generator object as an argument,
- the parser will yield individual statements,
- the parser will use next(lexer) to fetch a token when it needs one,
- the language has 'include' directives: the parser may need to instruct the lexer to switch to a different input file, which it'll do by send-ing it the name of that file.

For simplicity, the lexer will get a dict with file names as keys and file content as values (both strings). It will start by lexing the file named main. When the lexer reaches an end of an included file, it will continue wherever it left off in the stream which was interrupted by the include directive.
There are 4 basic lexeme (token) types: keyword, identifier, number (literal) and a linebreak (which ends statements). The keywords are: set, add, mul, print and include. Identifiers are made of letters (isalpha) and underscores and literals are made of digits (isdecimal). Statements are of these forms:
[set|add|mul] ident [num|ident] print ident include ident
A statement to be yielded is a 2- or 3-tuple, starting with the keyword as a string, followed by the operands (int for literals, strings for identifiers). E.g. mul x 3 shows up as ('mul', 'x', 3). The include statement is never yield-ed.
The following type alias should help you with typing parse. Even though this is not very intuitive, a tuple is also a sequence.

```
Statement = Sequence[ str | int ]

def lexer( program ):
    pass

def parser( lex ):
    pass
```

---

4.p.6 [mbox] Write a coroutine-based parser for mbox files. It should yield elements of the message as soon as it has enough bytes. The input will be an iterable, but not indexable, sequence of characters.
In an mbox file, each message starts with a line like this:

```
From someone@example.com Wed May  1 06:30:00 MDT 2019
```

You do not need to look at the structure of this line, look for the string From (with a trailing space) at the start of a line, and gobble it up to the nearest newline.
After the separator line, an rfc-822 e-mail follows, with any lines that start with From changed to >From (do not forget to un-escape those). The headers are separated from the rest of the body by a single blank line. You can also assume that each header takes exactly one line.
The reported elements are always pairs of strings, with the following content:

- message start: the string 'message' followed by the content of the separator line with the From removed,
- header: yield the name of the field and the content; yield as soon as you read the first character of the next header field, or the body separator,
- body: yield a single string with the entire body in it, as soon as you encounter the end of the file

```
def parse_mbox( chars ):
    pass
```

## Part 4.r: Regular Exercises

4.r.1 [iscan] Implement a prefix sum and a prefix list on arbitrary Iterable instances, using the iterator approach (class with an __iter__ method).
Examples:

```
dump( prefixes( [ 1, 2, 3 ] ) )  # [] [1] [1, 2] [1, 2, 3]
dump( prefix_sum( [ 1, 2, 3 ] ) ) # [ 1, 3, 6 ]

def prefixes( list_in ):
    pass

def prefix_sum( list_in ):
    pass
```

4.r.2 [gscan] Implement suffix list and suffix sum as a generator, with an arbitrary Sequence as an input.
Examples:

```
suffixes( [ 1, 2 ] )     # [] [ 2 ] [ 1, 2 ]
suffix_sum( [ 1, 2, 3 ] ) # [ 3, 5, 6 ]

def suffixes( list_in ):
    pass

def suffix_sum( list_in ):
    pass
```

4.r.3 [itree]

```
T = TypeVar( 'T' )

class Tree( Generic[ T ] ):
    def __init__( self, value: T,
                  left:  Optional[ Tree[ T ] ] = None,
                  right: Optional[ Tree[ T ] ] = None ) -> None:
        self.left  = left
        self.right = right
        self.value = value
        self.parent : Optional[ Tree[ T ] ] = None

        if left is not None:
            left.parent = self
        if right is not None:
            right.parent = self
```

Write an in-order iterator for binary trees. Write it as a class with a __next__ method.

```
class TreeIter: pass
```

4.r.4 [gtree] Write recursive generators which walk a binary tree in pre-/in-/post-order.

```
def preorder( tree ): pass
def inorder( tree ): pass
def postorder( tree ): pass
```

4.r.5 [dfs] Write a semi-coroutine which yields nodes of a graph in the 'leftmost' DFS post-order. That is, visit the successors of a vertex in order, starting from leftmost (different exploration order will result in different post-orders). The graph is encoded using neighbour lists.

```
def dfs( graph, initial ):
    pass
```

---

4.r.6 [guided] Write an A* 'guided search' that finds a shortest path in a graph, implemented using coroutines. The search coroutine should yield the nodes of the graph as they are explored. In response to each yield, the driver (semantically also a coroutine, though not necessarily a coroutine or a generator in the Python sense) will send the corresponding priority which should be assigned to exploring the successors of the given node.

```
T = TypeVar( 'T' )
S = TypeVar( 'S' )

class cor_iter( Generic[ T, S ] ): pass
```

Note: A* is essentially just BFS with a priority queue instead of a regular queue. To simplify matters, here is an implementation of standard BFS.

```
Graph = dict[ T, set[ T ] ]
Gen2 = Generator[ T, S, None ]

def bfs( graph: Graph[ T ], start : T ) -> Gen2[ T, int ]:

    q : Queue[ T ] = Queue()
    q.put( start )
    seen : set[ T ] = set()

    while not q.empty():
        item = q.get()
        for succ in graph[ item ]:
            yield succ
            if succ not in seen:
                q.put( succ )
            seen.add( succ )

def a_star( graph, start ): pass
```

# Part S.1: Introductory Tasks

The programming tasks for this block are as follows:

1. `a_while` – an interpreter for simple 'while programs',
2. `b_splay` – a self-balancing binary search tree,
3. `c_same` – a solver for 'same game',
4. `d_shelter` – a simple information system.

The tasks at hand only require basic programming skills and no special tricks nor advanced Python constructs. Some of the tasks require exceptions to be raised on errors, but again only basic use is needed (you should be fine with `raise RuntimeError( 'foo' )`).
In the `splay` task, type annotations are optional, since they are a little tricky (the tree is generic in the type of values, but these values must be less-than comparable; you can find a 'recipe' for solving this in `02/p1_dsw`).

## Part S.1.a: `while`

Implement an interpreter for simple 'while programs' (so called because their only looping construct is `while`). The syntax is as follows:

- one line = one statement (no exceptions),
- the program is a sequence of statements,
- blocks are delimited by indentation (1–5 spaces),
- there are following statement types:
  - assignment,
  - `if` statement,
  - `while` statement.

All variables are global and do not need to be declared (they come into existence when they are first used, with a default value 0). Variables are always integers. Variable names start with a letter and may contain letters, underscores and digits.
The `if` and `while` statements are followed by a body: a block indented one space beyond the `if` or `while` itself. The body might be empty. The `if` and `while` keywords are followed by a single variable name. Zero means false, anything else means true.
Assignments are of two forms:

- constant assignments of the form `name = number` (where `number` is an integer written in decimal, and might be negative),
- 3-address code operations, of the form

  $\texttt{name}_0$ = operation $\texttt{name}_1$ $\texttt{name}_2$

Valid operations are:

- logic: `and`, `or`, `nand` (the result is always 0 or 1),
- relational (result is again `0` or `1`):
  - `lt`, `gt` (less/greater than),
  - `eq` (equals),
  - `leq` and `geq` (less/greater or equal),
- arithmetic: `add`, `sub`, `mul`, `div`, `mod`.

Example program:

```
x = 0
y = 7
one = 1
if x
 x = add x x
while y
 y = sub y one
 x = add x one
```

Write a function `do_while` which takes a 'while program' (as a string) and returns a dictionary with variable names as keys and their final values as values (of type `int`).
If the program contains an error, create a special variable named `#error` and set its value to the offending line number. Return immediately after encountering the error. In this case, other variables may or may not be included in the resulting dictionary.
Check syntax before you start executing the program (i.e. the following program should return an error on line 3 and should not loop forever):

```
x = 1
while x
x ++
```

Syntax errors may be due to malformed statements (e.g. `while x = 1`, `x ++` above, etc.), or due to undefined operations (e.g. `x = fdiv x y`). Report the first error (nearest to the top of the input). At runtime, detect and report any attempts to divide by zero.

## Part S.1.b: `splay`

Implement the splay tree data structure (an adaptively self-balancing binary search tree). Provide at least the following operations:

- `insert` – add an element to the tree (if not yet present)
- `find` – find a previously added element (return a `bool`)
- `erase` – remove an element
- `to_list` – return the tree as a sorted list
- `filter` – remove all elements failing a given predicate
- `root` – obtain a reference to the root node

Nodes should have (at least) attributes `left`, `right` and `value`. The class which represents the tree should be called `SplayTree`.
You can find the required algorithms online (wikipedia comes to mind, but also check out `https://is.muni.cz/go/uvcjn9` for some intuition how the tree works).
The main operation is 'splaying' the tree, which moves a particular node to the root, while rebalancing the tree. How balanced the tree actually is depends on the order of splay operations. The tree will have an expected logarithmic depth after a random sequence of lookups (splays). If the sequence is not random, the balance may suffer, but the most-frequently looked up items will be near the root. In this sense, the tree is self-optimizing.
Note: it's easier to implement `erase` using splaying than by using the 'normal' BST delete operation:

1. splay the to-be-deleted node to the root, then

2. join its two subtrees L and R:
   - use splay again, this time on the largest item of the left subtree L,
   - the new root of L clearly can't have a right child,
   - attach the subtree R in place of the missing child.

## Part S.1.c: `same`

Your task will be to implement a simple solver for 'same game'. The rules of the game are:

1. the game is played on a rectangular board with $n \times m$ rectangular cells,
2. a cell can be empty, or occupied by a 'stone' of a particular type (we will use numbers to represent these types, and `None` to represent an empty cell),
3. the player can remove an area made up of 3 or more identical adjacent stones (each stone has 4 neighbours); all matching stones are removed at once,
4. the game ends when no more stones can be removed.

Unlike most versions of the game, to keep things simple, we will not implement gravity or replenishment of the stones (at least not in this iteration). The scoring rules are as follows:

1. the base score of removal is the value of the stone $v$ times the number of stones $n$ removed from the board times the base-2 logarithm of the same: $v \cdot n \cdot \log_2(n)$, the entire number rounded to the closest integer,
2. this score is doubled if at least one of the removed stones is directly adjacent to a cell that was occupied before the last round (i.e. it belonged to a stone that was removed in the last round),
3. it is also doubled if the last removal was of the same type of stone (note that conditions 2 and 3 are mutually exclusive).

When the game ends, the scores for each round are summed: this is the game score.
Write a function `samegame` which takes 2 arguments: the initial board in the form of a single list of numbers (with `None` used to represent empty spaces) and the width of the playing board. You can assume that the number of items in the list will be an integer multiple of the width. The result of the function should be the maximum achievable game score.

```python
def samegame( board: list[ int ], width: int ) -> int:
    pass
```

## Part S.1.d: `shelter`

You volunteer for a local animal shelter, and they really need to get more organized. Since you are a programmer, you decide to step up to the job and write a small information system for them. Here is what it needs to do:

- track all the resident animals and their basic stats: name, year of birth, gender, date of entry, species and breed,
- store veterinary records: animals undergo exams, each of which has a date, the name of the attending vet and a text report,
- record periods of foster care: animals can be moved out of the shelter, into the care of individuals for a period of time – record the start and end date of each instance, along with the foster parent,
- for each prospective foster parent, keep the name, address, phone number and the number of animals they can keep at once,
- record adoptions: when was which animal adopted and by whom,
- keep the name and address of each adopter.

In the remainder of the spec, we will make full use of duck typing: for each entity, we will only specify the interface, the exact classes and their relationships are up to you, as long as they provide the required methods and attributes. The only class given by name is `Shelter`, which is the entry point of the whole system.
The `Shelter` class needs to provide the following methods:

- `add_animal` which accepts keyword arguments for each of the basic stats listed above: `name`, `year_of_birth`, `gender`, `date_of_entry`, `species` and `breed`, plus `adoption_date`, where:
  - the date of entry is a `datetime.date` instance,
  - `year_of_birth` is an integer,
  - everything else is a string,
  - `name`, `species` and `date_of_entry` are required, the rest is optional,
  - `adoption_date` can be set in cases where an animal is being added retroactively and is equivalent to calling `adopt` (see below) atomically,

and returns the object representing the animal (see `list_animals` below for details about its interface),

- `list_animals` which accepts:
  - optional keyword arguments for each of the basic stats: only animals that match all the criteria (their corresponding attribute is equal to the value supplied to `list_animals`, if it was supplied) should be listed,
  - a `date` keyword argument: only animals which were possibly present in the shelter at this time (i.e. were not adopted on an earlier date, and were not in foster care that entire day) should be listed;

  The elements of the list returned by `list_animals` should have:
  - each of the basic stats as an attribute of the corresponding type (see `add_animal`),
  - method `add_exam` which accepts keyword arguments `vet` and `date` and `report`, where `vet` and `report` are strings and `date` is a `datetime.date` instance, and returns an object representing the exam, with attributes corresponding to the keyword arguments,
  - method `list_exams` which takes keyword arguments `start` and `end`, both `datetime.date` instances, or `None` (the range is inclusive; in the latter case, the range is not limited in that direction),
  - method `adopt` which takes keyword arguments `date` (a `datetime.date` instance) and optionally `adopter_name` and `adopter_address` which are strings,
  - method `start_foster` which takes a `date` (again a `datetime.date` instance), `parent` (which accepts one of the objects returned by `available_foster_parents` listed below) and an optional `end_date` (for cases when the fostering is recorded retroactively),
  - `end_foster` which takes a `date`,
- `add_foster_parent` which accepts keyword arguments `name`, `address` and `phone_number` (all strings) and `max_animals` which is an `int` and returns the object representing the foster parent,
- `available_foster_parents` which takes a keyword argument `date` and lists foster parents with free capacity at this date (i.e. those who can keep more animals than they are or were keeping at the given date – if an animal is taken or returned on a given date, it still counts into the limit).

Raise a `RuntimeError` in (at least) these cases:

- `start_foster` was called on an animal that was already in foster care at the given date, or `end_foster` on an animal that was not in foster care on the given date, or `start_foster` is called without an `end_date` on a date that predates an existing fostering record, or `start_foster` is called with an `end_date` that overlaps an existing fostering record,
- attempting to adopt an animal that was in foster care on that day, or attempting to put an animal that has been adopted on that or earlier day into foster care, or was not at the shelter that day at all for some other reason,

- attempting to do a veterinary exam on an animal which is in foster care or already adopted at the time (however, exams can be performed on the same day as fostering is started or ended, or on the day of adoption),
- an attempt is made to exceed the capacity of a foster parent,
- adoption of an animal that has already been adopted is attempted (regardless of dates),
- adoption of an animal at a date that predates a recorded veterinary exam (i.e. the exam record would be rendered invalid by the adoption),
- to avoid confusion, an action is prevented if it would cause two animals with the same name and species to be housed by the shelter at the same time (it is still an error even if they would never meet due to fostering – an animal of the same name & species can only be accepted into the shelter after the first was adopted; of course, having 'Jesenius' and 'Jesenius II' at the same time is perfectly acceptable).

# Part 5: Memory management, reference counting

Please note that the content of this chapter might change considerably before it comes up.
Demonstrations:

1. (to be added)

Practice exercises:

1. refcnt – a reference counting manager
2. final – deterministic object finalization
3. reach – reachability from a set of roots
4. sweep – a mark and sweep collector
5. malloc – low-level memory management
6. trace – tracing composite objects

Regular exercises:

1. refcnt – reference counting with data
2. reach – reachability again
3. sweep – mark and sweep v2
4. semi – a copying 'semi-space' collector
5. cheney – improved version of the same
6. python – reference counting + mark & sweep

Voluntary exercises:

1. (nothing here yet)

## Part 5.p: Practice Exercises

5.p.1 [refcnt] Implement a simple reference-counting garbage collector. The interface is described in the class Heap below. The root objects are immortal (those are established by add_root). The count method returns the number of reachable live objects. The alive method checks whether a given object is alive. All objects start out dead.
References are added/removed using add_ref and del_ref. You can assume that the number of del_ref calls on given arguments is always at most the same as the number of corresponding add_ref calls. Assume that no reference cycles are created. You need to keep track of the references yourself.

```
class Heap:
    def add_root( self, obj: int ) -> None: pass
    def add_ref( self, obj_from: int, obj_to: int ) -> None: pass
    def del_ref( self, obj_from: int, obj_to: int ) -> None: pass
    def count( self ) -> int: pass
    def alive( self, obj: int ) -> bool: pass
```

5.p.2 [final] Same as previous exercise, but with the additional requirement that whenever an object becomes garbage (unreachable), a finalizer is immediately called on it. The finalizer may perform arbitrary heap manipulation (as long as it is otherwise legal; in particular, it may 're-animate' the object it is finalizing, by storing a reference to this object). A finalizer must not be called on an object if a reference exists to this object (even if that reference is from another dead object).

```
class Heap:
    def add_root( self, obj ): pass
    def add_ref( self, obj_from, obj_to ): pass
    def del_ref( self, obj_from, obj_to ): pass
    def set_finalizer( self, callback ): pass
```

5.p.3 [reach] Implement the 'mark' phase of a mark & sweep collector. That is, find all objects which are reachable from the root set.
Like before, roots are marked using add_root and references are added/removed using add_ref and del_ref. You can assume that the number of del_ref calls on given arguments is always at most the same as the number of corresponding add_ref calls.

```
class Heap:
    def add_root( self, obj: int ) -> None: pass
    def add_ref( self, obj_from: int, obj_to: int ) -> None: pass
    def del_ref( self, obj_from: int, obj_to: int ) -> None: pass
    def reachable( self ) -> Set[ int ]: pass
```

5.p.4 [sweep] Add the 'sweep' phase to the mark & sweep collector from previous exercise. That is, find all objects which are reachable from the root set, then 'free' all objects which were previously alive but are not anymore. Freeing objects is simulated using a callback, which is passed to the constructor of Heap. The callback must be passive (unlike the finalizer from p2_final).
Again, roots are marked using add_root and references are added/removed using add_ref and del_ref. You can assume that the number of del_ref calls on given arguments is always at most the same as the number of corresponding add_ref calls.

```
class Heap:
    def __init__( self, free: Callable[ [ int ], None ] ) -> None:
        pass
    def add_root( self, obj: int ) -> None: pass
    def add_ref( self, obj_from: int, obj_to: int ) -> None: pass
    def del_ref( self, obj_from: int, obj_to: int ) -> None: pass
    def collect( self ) -> None: pass
```

5.p.5 [malloc] In this exercise, we will move one level down and one step closer to reality. Your task is to implement simplified versions of the malloc and free functions, in a fixed-size memory represented as a Python list of integers.
For simplicity, the memory will be 'word-addressed', that is, we will not deal with individual bytes – instead, each addressable memory cell will be an int. To further simplify matters, free will get the size of the object as a second parameter (you can assume that this is correct).
Use a first-fit strategy: allocate objects at the start of the first free chunk of memory. It is okay to scan for free memory in linear time. The malloc method should return None if there isn't enough (continuous) memory left.

```
class Heap:
    def __init__( self, size: int ) -> None: pass
    def read( self, addr: int ) -> int: pass
    def write( self, addr: int, value: int ) -> None: pass
    def malloc( self, size: int ) -> Optional[ int ]: pass
    def free( self, addr: int, size: int ) -> None: pass
```

# Part 5.r: Regular Exercises

5.r.1 [refcnt] Implement a simple reference-counting garbage collector. The interface is described in the class Heap below. Objects are represented using lists of integers, and the heap as a whole is a list of such objects. Negative numbers are data, non-negative numbers are references (indices into the main list of objects). The root object (with index 0) is immortal.
The interface:

- the count method returns the number of live objects,
- the write method returns True iff the write was successful (the object was alive and the index was within its bounds)
- likewise, the read method returns None if the object is dead or invalid or the index is out of bounds.
- the make method returns an unused object identifier (and grows the heap as required).

The first call to make creates the root object. A freshly-made objects starts out with zero references. A reference to this object must be written somewhere into the heap.

```
class Heap:
    def __init__( self ):
        self.data : List[ List[ int ] ] = []
        pass # …

    def read( self, obj_id: int, index: int ) -> Optional[int]: pass
    def write( self, obj_id: int, index: int,
               value: int ) -> bool: pass
    def make( self, size: int ) -> int: pass
    def count( self ) -> int: pass
```

5.r.2 [reach] Implement the mark part of a mark & sweep collector. The interface of Heap stays the same as it was in r1.

```
class Heap:
    def __init__( self ):
        self.data : List[ List[ int ] ] = []

    def read( self, obj_id: int, index: int ) -> Optional[int]: pass
    def write( self, obj_id: int, index: int,
               value: int ) -> bool: pass
    def make( self, size: int ) -> int: pass
    def count( self ) -> int: pass
```

5.r.3 [sweep] Add the sweep procedure to the Heap implementation from previous exercise.

```
class GcHeap( Heap ):
    def collect( self ) -> None: pass
```

5.r.4 [semi] Write a semi-space collector, using the same interface as before. The requirement is that after a collection, the objects all occupy contiguous indices. For simplicity, we index the semispaces independently, so the objects always start from 0. Make sure that the root always retains index 0.

```
class Heap:
    def __init__( self ):
        self.data : List[ List[ int ] ] = []

    def read( self, obj_id: int, index: int ) -> Optional[int]: pass
    def write( self, obj_id: int, index: int,
               value: int ) -> bool: pass
    def make( self, size: int ) -> int: pass
    def count( self ) -> int: pass
    def collect( self ) -> None: pass
```

5.r.5 [cheney] Write a Cheney-style semi-space collector, using the same interface and requirements as before. The main difference is in the overhead of the algorithm (only 2 pointers outside of to/from spaces are available in the implementation of collect in this exercise).

```
class Heap:
    def __init__( self ):
        self.data : List[ List[ int ] ] = []

    def read( self, obj_id: int, index: int ) -> Optional[int]: pass
    def write( self, obj_id: int, index: int,
               value: int ) -> bool: pass
    def make( self, size: int ) -> int: pass
    def count( self ) -> int: pass
    def collect( self ) -> None: pass
```

5.r.6 [python] Implement the 'Python' collector: reference counting, with mark & sweep to deal with cycles. Objects that are not on loops, or reachable from loops, are destroyed immediately when last reference to them is lost. Unreachable loops are destroyed on collect.

```
class Heap:
    def __init__( self ):
        self.data : List[ List[ int ] ] = []

    def read( self, obj_id: int, index: int ) -> Optional[int]: pass
    def write( self, obj_id: int, index: int,
               value: int ) -> bool: pass
    def make( self, size: int ) -> int: pass
    def collect( self ) -> int: pass
```

# Part 6: Objects 2

Demonstrations:

1. (to be done)

Practice exercises:

1. poly – polynomials with operator overloading
2. mod – finite rings (integers mod N)
3. noexcept – turn exceptions into None returns
4. with – a simple context manager
5. numeric – a simple meta-class exercise
6. record – 'data classes' using data descriptors

Regular exercises:

1. trace – advanced print debugging
2. profile – a very simple profiler
3. record – more data classes
4. array – array with automatic resizing
5. bitset – a compact set of small integers
6. undo – a data descriptor with a history

Voluntary exercises:

1. (nothing here yet)

## Part 6.p: Practice Exercises

6.p.1 [poly] Implement polynomials which can be added and printed. Do not print terms with coefficient 0, unless it is in place of ones and the only term. For example:

```
x = Poly( 2, 7, 0, 5 )
y = Poly( 2, 4 )

print( x )    # prints 2x³ + 7x² + 5
```

```
print( y )     # prints 2x + 4
print( x + y ) # prints 2x³ + 7x² + 2x + 9
```

The implementation goes here:

```
class Poly:
    pass
```

We will do one more exercise with operators, `mod.py`, before moving on to exceptions.

6.p.2 [`mod`] In this exercise, you will implement the ring $\mathbb{Z}/n\mathbb{Z}$ of integers modulo $n$. Welcome to abstract algebra: a ring is a set with two operations defined on it: addition and multiplication. The operations must have some nice properties. Specifically, the set we consider in this exercise is the set of all possible remainders in the division by $n$; you can read up on the necessary axioms on e.g. Wikipedia (under `Ring (mathematics)`).
Interaction of elements in different modulo classes results in a `TypeError`. When printing, use the notation $[x]_n$, such as $[5]_7$ to represent all integers with remainder 5. Implement equality, comparison, printing, and the respective addition and multiplication (all should also work with plain integer operands on either side).
An instance of `Mod` represents a congruence class $x$ modulo $n$.

```
class Mod:
    def __init__( self, x: int, n: int ) -> None:
        pass
```

6.p.3 [`noexcept`] Write a decorator `@noexcept()`, which turns a function which might throw an exception into one that will instead return `None`. If used with arguments, those arguments indicate which exception types should be suppressed.
Note: typing this correctly with mypy is probably impossible. You can try using `Callable[ ..., Any ]` and/or `Any` if you want to add annotations.

```
def noexcept( *ignore ):
    def decorate( f ):
        return f
    return decorate
```

6.p.4 [`with`] Write a simple context manager to be used in a `with` block. The goal is to enrich stack traces with additional context, like this:

```
def context( *args ):
    pass
```

For example:

```
def foo( x: int, y: int ) -> None:
    with context( "asserting equality", x, '=', y ):
        assert x == y
```

Calling `foo( 1, 1 )` should print nothing (the assertion does not fail and no exceptions are thrown). On the other hand, `foo( 7, 8 )` should print something like this:

```
asserting equality 7 = 8
Traceback (most recent call last):
  File "with.py", line 20, in <module>
    foo( 7, 8 )
  File "with.py", line 17, in foo
    assert x == y
AssertionError
```

---

6.p.5 [`numeric`] Implement a meta-class `Numeric` such that numbers (floats, integers, …) may appear to be instances of `Numeric`-based classes (the normal, non-meta class itself should be able to decide which, if any; you may find a class attribute useful here).
Don't forget to derive your custom metaclass from the builtin (default) metaclass, `type`. When dealing with `mypy`, you can get away with annotating the type of the (non-meta!) class attribute you want to use in the `isinstance` override directly in the metaclass.

```
class Numeric: pass
```

Now implement classes `Complex` which represents standard complex numbers (based on `float`) and `Gaussian`, which represents Gaussian integers (complex numbers with integer real and imaginary part). The following should hold:

- integer values (including literals) are instances of `Gaussian`,
- float values are not instances of `Gaussian`,
- both integer and float values (including literals) are instances of `Complex`.

Other than that, implement addition and equality so that all reasonable combinations of parameters work (integers can be added to Gaussian integers and all of floats, normal integers and Gaussian integers can be added to Complex numbers).

```
class Gaussian: pass
class Complex: pass
```

---

6.p.6 [`record`] Implement `Field`, a data descriptor which can be used to create classes that simply keep attributes (records, data classes), without having to type out the `__init__` method. The use case is similar to the `dataclass` decorator, though our approach will be much simpler (and also much more limited). When initializing an instance, make sure that the default value is copied, so that default lists and other mutable values are not accidentally shared between instances (see also standard module `copy`).
Hint: The data descriptor can keep the value in the regular instance `__dict__`. Remember the diagram used by the default `__getattribute__` for lookup? You can even use the same name, so the value is not directly exposed.
Bonus: If you like a challenge, extend `Field` so that it monkey-patches an `__init__` method into the 'data' class (i.e. the one with `Field`-typed attributes). This synthetic `__init__` should accept arguments in the declaration order of the fields and initialize them to non-default values, if provided (see tests below).
PS: You can make `Field` a `Generic` and with some fiddling, make the types sort of work (may need a `cast` in `__get__`)

```
class Field: pass

class Data: # helper to silence ‹mypy› in the bonus part
    def __init__( self, *args: Any ) -> None: pass
```

## Part 6.r: Regular Exercises

6.r.1 [`trace`] Write a decorator that prints a message every time a function is called or it returns. The output should be indented when calls are nested, and should include arguments and the return value. Aim for output like this:

```
foo [13]
  bar [13] -> 20
  bar [26] -> 33
returned 53
```

```
def traced( f ):
    pass
```

6.r.2 [`profile`] Implement a decorator which will keep track of how many times which function was called. The decorator should be available as `@profile` and calling `profile.get()` should return a dictionary with function names as keys and call counts as values.

```
def profile( f ): pass
```

**6.r.3** [record] Re-do p6_record, including the bonus, but using a class decorator. That is, implement a decorator record which takes a class which only contains (class) variables and turn it into a proper class with instance attributes of the same names, and with appropriate default values.

```
def record( cls ): pass

class Data: # helper to silence <mypy>
    def __init__( self, *args: Any ) -> None: pass
```

**6.r.4** [array] Implement a class Array which acts like a list, with the following differences:

- no push, pop, remove and similar 'list-like' methods – only item access via indexing,
- the constructor takes a default value, which is used as the initial value for cells that have not been explicitly set,
- all indices are always valid: both reading and writing an index automatically resizes the underlying list (using the default given above to fill in missing cells).

The default value should be copied into new cells, so that arrays with mutable work reasonably. Use shallow copies.

```
class Array: pass
```

# Part 7: Pitfalls, testing, profiling

This week will cover hypothesis, a rather useful tool for testing Python code. Hypothesis is a property-based testing system: unlike traditional unit testing, we do not specify exact inputs. Instead, we provide a description of an entire class of inputs; hypothesis then randomly samples the space of all inputs in that class, invoking our test cases for each such sample.
The main interface to hypothesis is the hypothesis.given decorator. It is used like this:

```
import hypothesis
import hypothesis.strategies as s

@hypothesis.given( s.lists( s.integers() ) )
def test_sorted( x ):
    assert sorted( x ) == x # should fail

@hypothesis.given( x = s.integers(), y = s.integers() )
def test_cancel( x, y ):
    assert ( x + y ) - y == x # looks okay
```

Calling the decorated function will perform a number of randomized tests. The strategies dictate what values will be attempted for each argument (arguments and strategies are matched by name).
Demonstrations:

1. (to be done)

In practice exercises this week, you will write tests for different pieces of (better or worse) code. The 'tests for the tests' that are enclosed try to make sure your tests can tell bad code from good code, though of course there are limitations.

1. inner – dot product on 3D vectors with integer components
2. cross – same, but cross product
3. part – partitioning lists based on a predicate
4. search – binary search, an off-by-one bonanza
5. sort – sorting lists
6. heap – tests for heap-organized arrays

Unlike other weeks, we will not be writing new programs in the seminar either. Instead, you get the following programs that are already written and your task is to write tests for them, to make sure they are correct (or find and fix bugs if they aren't).
The rules for activity points will be relaxed, so that you can split into subgroups and compete with each other to decide the correctness of as many programs as you can. Your tutor will arrange the details with you.

1. life – game of life
2. dfs – depth first search, the perennial favourite
3. record – a decorator for making record types
4. bipartite – checking whether a graph is bipartite
5. treap – testing data structures
6. itree – an in-order tree iterator

Voluntary exercises:

1. (nothing here yet)

## Part 7.p: Practice Exercises

**7.p.1** [inner]

1. Implement the standard dot product on 3D integer vectors.
2. Use hypothesis to check its properties:
   - commutativity
   - distributivity over addition $a \cdot (b + c) = a \cdot b + a \cdot c$
   - bilinearity $a \cdot (rb + c) = r(a \cdot b) + (a \cdot c)$
   - compatibility with scalar multiplication: $(ra) \cdot (rb) = rr(a \cdot b)$

Bonus: Try the same with floats. Cry quietly. Disallow inf. And nan. Then cry some more.

```
Vector = Tuple[ int, int, int ]
Inner = Callable[ [ Vector, Vector ], int ]

def add( a: Vector, b: Vector ) -> Vector:
    ax, ay, az = a
    bx, by, bz = b
    return ( ax + bx, ay + by, az + bz )

def mul( r: int, a: Vector ) -> Vector:
    ax, ay, az = a
    return ( r * ax, r * ay, r * az )

def dot( a, b ): pass

def check_commutativity( dot: Inner ) -> None: pass
def check_distributivity( dot: Inner ) -> None: pass
def check_bilinearity( dot: Inner ) -> None: pass
def check_compatibility( dot: Inner ) -> None: pass
```

**7.p.2** [cross] Implement the cross product and check the following properties:

- anti-commutativity
- distributivity over addition
- compatibility with scalar multiplication: $ra \times b = a \times rb = r(a \times b)$
- Jacobi identity: $a \times (b \times c) + b \times (c \times a) + c \times (a \times b) = 0$

Check all of them on integer inputs.

```
Vector = Tuple[ int, int, int ]
BinOp  = Callable[ [ Vector, Vector ], Vector ]

def add( a: Vector, b: Vector ) -> Vector:
    ax, ay, az = a
    bx, by, bz = b
    return ( ax + bx, ay + by, az + bz )
```

```
def mul( r: int, a: Vector ) -> Vector:
    ax, ay, az = a
    return ( r * ax, r * ay, r * az )

def cross( a, b ): pass

def check_anticommutativity( cross: BinOp ) -> None: pass
def check_distributivity( cross: BinOp ) -> None: pass
def check_jacobi( cross: BinOp ) -> None: pass
def check_compatibility( cross: BinOp ) -> None: pass
```

7.p.3 [part]

```
T = TypeVar( 'T' )
```

Write a function, `partition`, which takes a predicate and a list and returns a pair of lists: one with items that pass the predicate (like filter) and the other with items which don't.

```
def partition( predicate, items ): pass
```

Then write tests using `hypothesis` that show a given implementation of `partition` works as expected.

```
def check_partition( part ): pass
```

7.p.4 [search] Write a function, `search`, which takes an item and a `sorted` list of integers and returns a `bool` indicating whether the item was present in the list. Implement it using binary search.

```
def search( needle, haystack ): pass
```

As before, make sure the `search` predicate is correct. Write some tests by hand and then write a hypothesis check. Which do you reckon is easier and which harder?

```
def check_search_manual( part ): pass
def check_search_auto( part ): pass
```

7.p.5 [sort] Write a procedure which sorts the input list and removes any duplicated entries (in place).

```
def sort_uniq( items ): pass
```

Write a hypothesis-based test function which ensures a given sort-uniq procedure is correct.

```
def check_sort( sort ): pass
```

7.p.6 [heap] Write `sift_down`, a procedure which takes 2 parameters: a list, and an index `idx`. The list is a max-heap, with the possible exception of the node stored at index `idx`, which may be out of place.
The children of the node stored at an arbitrary index $i$ are stored at indices $2i + 1$ and $2i + 2$.

```
def sift_down( heap: List[ int ], idx: int ) -> None:
    pass
```

Write a hypothesis-based test function which ensures that `sift_down` is correct.

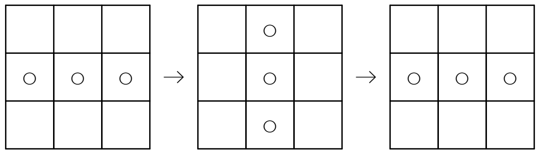
```
def check_sift( sift ): pass
```

## Part 7.r: Regular Exercises

7.r.1 [life] Remember the game of life from week 1? A quick reminder: it is a 2D cellular automaton where each cell is either alive or dead. In each generation (step of the simulation), the new value of a given cell is computed from its value and the values of its 8 neighbours in the previous generation. The rules are as follows:

| state | alive neigh. | result |
|---|---|---|
| alive | 0–1 | dead |
| alive | 2–3 | alive |
| alive | 4–8 | dead |
| dead | 0–2 | dead |
| dead | 3 | alive |
| dead | 4–8 | dead |

An example of a short periodic game:



Enclosed is an implementation of the game that is maybe correct, but maybe not. Write tests and find out which it is. Fix the bugs if you find any.

```
def updated( x, y, cells ):
    count = 0
    alive = ( x, y ) in cells

    for dx in [ -1, 0, 1 ]:
        for dy in [ -1, 0, 1 ]:
            if dx and dy:
                count += ( x + dx, y + dy ) in cells

    return count in { 2, 3 } if alive else count == 3

def life( cells, n ):
    if n == 0:
        return cells

    todo = set()

    for x, y in cells:
        for dx in [ -1, 0, 1 ]:
            for dy in [ -1, 0, 1 ]:
                todo.add( ( x + dx, y + dy ) )

    ngen = { ( x, y ) for x, y in todo if updated( x, y, cells ) }
    return life( ngen , n - 1 )
```

7.r.2 [dfs] You are given a semi-coroutine which is supposed to yield nodes of a graph in the 'leftmost' DFS post-order. That is, it visits the successors of a vertex in order, starting from leftmost. The input graph is encoded using a dictionary of neighbour lists.
Make sure it is correct (and if not, fix it).

```
T = TypeVar( 'T' )

def dfs( graph: Dict[ T, List[ T ] ], initial: T ) \
        -> Iterable[ T ]:
    seen : Set[ T ] = set()
    yield from dfs_rec( graph, initial, seen )

def dfs_rec( graph: Dict[ T, List[ T ] ], initial: T,
             seen: Set[ T ] ) -> Iterable[ T ]:

    seen.add( initial )

    for n in graph[ initial ]:
        yield from dfs_rec( graph, n, seen )

    yield initial
```

7.r.3 [record] Below is an implementation of a `@record` decorator which can be used to create classes that simply keep attributes (records, data classes), without having to type out the `__init__` method.
The use case is similar to the `dataclass` decorator, but the below imple-

mentation is much simpler. Default values must always be set, and they are shallow-copied into each instance. Additionally, the synthetic `__init__` method takes an optional argument for each attribute, in which case the given attribute is initialized to that value, instead of the default.
Make sure the decorator works as advertised. If not, fix it.

```
def record( cls: type ) -> type:
    class rec:
        def __init__( self, *args: Any ) -> None:
            from copy import copy
            counter = 0
            for k, v in cls.__dict__.items():
                if not k.startswith( '__' ):
                    if len( args ) > counter:
                        self.__dict__[ k ] = args[ counter ]
                    else:
                        self.__dict__[ k ] = copy( v )
                    counter += 1
    return rec
```

---

7.r.4 [bipartite] An undirected graph is given as a set of edges $E$. For any $(u, v) \in E$, it must also be true that $(v, u) \in E$. The set of vertices is implicit (i.e. it contains exactly the vertices which appear in $E$).
Below is a predicate which should decide whether a given graph is bipartite (can be coloured with at most 2 colours, such that no edge goes between vertices of the same colour). Make sure it is correct, or fix it.

```
def is_bipartite( graph ):
    colours = {}
    queue = []

    vertices = list( set( [ x for x,_ in graph ] ) )
    for v in vertices: # can be disconnected
        if v in colours:
            continue
        queue.append( v )
        colours[ v ] = 1
        colour = 1

        while queue:
            v = queue.pop( 0 )
            colour = 2 if colours[ v ] == 1 else 1
            for neighb in [ y for x, y in graph if x == v ]:
                if neighb in colours and \
                   colours[ neighb ] != colour:
                    return False
                if neighb not in colours:
                    colours[ neighb ] = colour
                    queue.append( neighb )
    return True
```

---

7.r.5 [treap]

```
class SupportsLessThan( Protocol ):
    def __lt__( self, other: Any ) -> bool: ...
    def __le__( self, other: Any ) -> bool: ...

T = TypeVar( 'T', bound = SupportsLessThan )
```

Remember treaps from week 2? A treap is a combination of a binary search tree and a binary heap: each node has a key (these form a search tree) and a randomized priority (these form a heap).
The role of the heap part of the structure is to keep the tree approximately balanced. Your task is to decide whether the below treap implementation works correctly. Keep in mind that treaps are only approximately balanced: your tests need to take this into account.

```
class Treap( Generic[ T ] ):
    def __init__( self, key: T, priority: int ):
        self.left  : Optional[ Treap[ T ] ] = None
        self.right : Optional[ Treap[ T ] ] = None
        self.key = key
        self.priority = priority

    def rotate_left( self: Treap[ T ] ) -> Treap[ T ]:
        assert self.left is not None
        r = self.left
        detach = r.right
        r.right = self
        self.left = detach
        return r

    def rotate_right( self: Treap[ T ] ) -> Treap[ T ]:
        assert self.right is not None
        r = self.right
        detach = r.left
        r.left = self
        self.right = detach
        return r

    def _insert( node: Optional[ Treap[ T ] ], key: T, prio: int )
-> Treap[ T ]:
        if node is None:
            return Treap( key, prio )
        else:
            return node.insert( key, prio )

    def _fix_right( self ) -> Treap[ T ]:
        assert self.right is not None

        if self.priority > self.right.priority:
            return self
        else:
            return self.rotate_right()

    def _fix_left( self ) -> Treap[ T ]:
        assert self.left is not None

        if self.priority > self.left.priority:
            return self
        else:
            return self.rotate_left()

    def insert( self, key: T, prio: int ) -> Treap[ T ]:
        if key > self.key:
            self.right = Treap._insert( self.right, key, prio )
            return self._fix_right()
        else:
            self.left = Treap._insert( self.left, key, prio )
            return self._fix_left()
```

---

7.r.6 [itree] Below, you will find an implementation of an in-order iterator for binary trees. Make sure it is correct and fix it if it isn't.

```
T = TypeVar( 'T' )

class Tree( Generic[ T ] ):
    def __init__( self, value: T,
                  left:  Optional[ Tree[ T ] ] = None,
                  right: Optional[ Tree[ T ] ] = None ) -> None:
        self.left  = left
        self.right = right
        self.value = value
        self.parent : Optional[ Tree[ T ] ] = None

        if left is not None:
            left.parent = self
        if right is not None:
            right.parent = self

class TreeIter( Generic[ T ] ):

    def __init__( self, tree: Tree[ T ] ) -> None:
```

```
        self.n : Optional[ Tree[ T ] ] = tree

    def descend( self ) -> None:
        assert self.n is not None

        while self.n.left is not None:
            self.n = self.n.left

    def ascend( self ) -> None:
        assert self.n is not None

        while ( self.n.parent is not None and
                self.n == self.n.parent.right ):
            self.n = self.n.parent

        self.n = self.n.parent # coming from left
```

```
    def __iter__( self ) -> TreeIter[ T ]:
        assert self.n is not None
        i = TreeIter( self.n )
        i.descend()
        return i

    def __next__( self ) -> T:
        v = self.n.value

        if self.n.right is not None:
            self.n = self.n.right
            self.descend()
        else:
            self.ascend()

        return v
```

# Part 8: Coroutines 2, `async def`

This chapter extends what we know about coroutines and generators to include `async` coroutines, how they are used, how they are related to generators and how they 'tick' in general.
Demonstrations:

1. `request` – communication with the scheduler
2. `fibres` – how to schedule fibres (aka green threads)
3. `spawn` – creating new fibres on demand
4. `yield` – asynchronous generators and `async for`
5. `context` – context managers and `async`

Practice exercises:

1. `rrsched` – a round-robin coroutine scheduler
2. `priority` – a simple priority-driven scheduler
3. `exchange` – coordinate `async` producers and consumers
4. `box` – a simplified version of the above
5. (exercise missing)
6. `sort` – sorting with real-time latency constraints

Regular exercises:

1. `sleep` – planning execution of sleepy coroutines
2. `ioplex` – multiplex incoming IO to multiple coroutines
3. `search` – low-latency binary search trees
4. (3 more exercises missing)

Voluntary exercises:

1. (nothing here yet either)

## Part 8.1: `async` Coroutines

We have already dealt with generators (aka 'semi-coroutines') and how to extend them to full coroutines using a trampoline. Python has another system of coroutines that is related, but in some sense more restricted. The main use case for `async` coroutines is asynchronous IO (we will look at that more specifically in a few weeks) and the syntax is tailored to this use case.
When using generators[35], we are mainly interested in `yield` and extracting the values that were passed to `yield` (mainly through iteration, sometimes through direct calls to `next` or `send`). The return value of a generator is usually ignored (after all, the only way to get this return value is to catch `StopIteration`).
In some sense, `async def` and particularly `await` is the polar opposite. In `x = await y`, the `x` is the return value of the coroutine object `y`. Or to be more specific:

```
async def foo():
    return 3

async def bar():
    x = await foo() # x is set to 3 here
```

On the other hand, using the `async def` syntax, there is no way to yield anything, even though internally, coroutine objects created by `async def` are very similar to generators. The entire interaction with `yield` and the mechanics of `next` and `StopIteration` are hidden in the `await` expression (and in the scheduler – more on that in the next section).
Before we go on, let us recall the distinction between generator objects and generator functions (and their relationship to lexical closures). Given:

```
def foo():
    yield 3
```

`foo` itself is a generator function, the result of calling `foo()` is a generator object and calling `__next__` on this object actually runs the code written in `foo` (until it yields).
Unsurprisingly, `async def` works the same way, though the result is not called a generator object but a coroutine object, and there is one more twist: you cannot directly call `__next__` on a coroutine object (i.e. it is not an `Iterator`). Instead, it is `Awaitable`, which means you first need to call `__await__` on it, and that gives you an iterator. Like this:

```
async def foo():
    return 3

coro_awaitable = foo()
coro_iterator = coro_awaitable.__await__()
next( coro_iterator ) # raises StopIteration( 3 )
```

Knowing this, we can unpack what the common construct `x = await foo()` actually means:

```
foo_awaitable = foo()
foo_iterator  = foo_awaitable.__await__()

try:
    while True:
        yield next(foo_iterator)
except StopIteration as e:
    x = e.value
```

Besides the awaitable/iterator dance (which is just a technicality), what happens is that `await` transparently passes through every `yield` from the callee to the caller. That is, given:

[35] And 'normal' coroutines – to avoid confusion, we will call them generators in this unit, though everything said about generators applies equally to normal coroutines.

```
async def async_1():
    return await magic_sleep(0)

async def async_2():
    return await async_1()

async def async_3():
    return await async_2()
```

If a return happens, the callee grabs that value and uses it as the result of the await expression.

However, if magic_sleep yields (which its real-world equivalents normally do), the await in async_1 takes the value yielded by the callee (magic_sleep) and passes it to its caller (async_2). Same process repeats in async_2, which takes the value that async_1 secretly yielded and passes it onto its own caller, async_3.

Basically, async def coroutines form a stack, which is sandwiched between two magic pieces:

1. at the top, a magic (library-provided) function which can yield[1],
2. at the bottom, a scheduler, which is the piece that actually calls next (or rather send) and is what we are going to look at next.

## Part 8.2: An async Scheduler

Syntactic restrictions on async def mean that it isn't possible to use them normally (via await) from the toplevel, nor from standard functions. Usually, the missing piece is provided by a library (asyncio in most cases): to transition from the world of functions to the world of coroutines, you need a standard function to which you can pass a coroutine. One such function is asyncio.run, but it's entirely possible to write such function with what we already know. Of course, the other way (calling normal functions from async coroutines) works fine (with some caveats related to latency).

However, asyncio.run is not simply glue that lets you call an async coroutine from a normal function – that wouldn't be very useful. Nonetheless, let's have a look at this minimal glue, for future reference:

```
try:
    while True:
        next( coro )
except StopIteration as e:
    return e.value
```

We will have to refine that, because nothing interesting is happening above: all values that were yielded are ignored and the suspended coroutine is immediately woken up again. We need to add two things to make it actually useful:

1. we need to be able to switch between coroutines (that's the entire point of the exercise, after all),
2. we need to react to the values that the other magic half (which typically comes from the same library, so in this case from us) yields (all the intermediate async def coroutines just forward it, until it reaches the schedule).

Let's start with the second part. Schematically:

```
result = None
while True:
    try:
        request = coro.send( result )
        result = process( request )
    except StopIteration as e:
        return e.value
```

The heavy lifting is done by process, but we are not really interested in the details of that. In asyncio, the requests are IO requests and process dispatches those IO requests to the operating system. We will discuss that in more detail another week. For a more complete sketch, see d1_request.

The other half of scheduler's job is implementing fibres, or green threads. Notable features of fibres are:

1. The most important feature of fibres is that they are cheap, in the sense you can make lots of them, and switching from one to the next is also cheap. This is universally true across many languages that provide them.
2. Python brings another feature with its implementation of fibres: the only place where a fibre can be interrupted (suspended) is during an await. This makes concurrency much easier to deal with, because it is immediately obvious where a thread might be suspended and another might be resumed. There is no parallelism: at any given time, at most a single fibre is executing. A data race is only possible if you split a complex update of a shared data structure across an await – something that is much harder to do by accident than, say, forgetting to lock a mutex.
3. Combined with asyncio[36], fibres can provide IO parallelism where multiple IO requests from multiple fibres are processed in parallel by the low-level IO loop. The actual Python code still runs sequentially, but since IO causes a lot of latency, using the delays while IO is executing in the OS to run other fibres can considerably improve overall throughput, and/or per-client latency in applications with multiple client connections.

To get fibres, we need to be able to do two things, essentially:

1. suspend an entire coroutine stack, which is easily done: await already propagates a yield from special methods all the way to the scheduler,
2. put suspended coroutines on a queue (or into a system of queues) – again easily done, since suspended coroutines are just regular, inert objects and can be put into a list or a deque like any other object,
3. pick and resume a particular fibre from the queue: this is done by calling next or send on the coroutine object that we picked from the queue.

The system of queues is usually arranged the same way an OS scheduler is: there is a run queue for fibres that are ready to execute (i.e. they are not waiting for any IO operation), and then additional queues are created for resources that can block the execution of a fibre (whether it is a synchronisation device, a communication queue or an IO operation). Whenever the resource becomes available, the fibre is moved to the run queue and eventually resumed.

The only thing that remains is that we need to be able to actually create new fibres. But since fibres are nothing but stacks of suspended coroutines, we can create a new one by creating a coroutine object (by calling a coroutine function, aka an async function) and sending the result to the scheduler using a 'please put this on your queue' request. Along the lines of:

```
async def fibre():
    pass # do stuff here

async def main():
    coro = fibre()              # create a suspended coroutine
    await async_spawn( coro ) # send it to the scheduler
```

The implementation of async_spawn is then straightforward. For a worked example, see d3_spawn.

[36] Notably, asyncio is more or less modelled after node.js, which is itself modelled after the IO loop used in traditional, single-threaded UNIX daemons. This approach to concurrency has a long tradition, but the introduction of node.js and asyncio made it considerably easier to use.

# Part 8.d: Demos

8.d.1 [request] In this demo, we will look at first part of the scheduler's job: handling requests from 'special' functions. First, however, let's define a helper class to represent the requests that we are going to pass from the async functions to the scheduler. To make things simpler, the scheduler will pass back the result by updating the request (in particular its result attribute, which we set up in __init__).

```
class Request:
    pass

class ReadRequest( Request ):
    def __init__( self, file: str ):
        self.file = file
        self.result : str

class WriteRequest( Request ):
    def __init__( self, file: str, data: str ):
        self.file = file
        self.data = data
        self.result = None
```

To simplify working with type annotations, we will define a pair of generic aliases. The first, AwaitGen is going to be the type of __await__ methods that cooperate with our scheduler (and hence they yield instances of Request). The latter, Coro, is the type of coroutines that we want to use. Somewhat unfortunately, mypy does not actually care about the yield (or send) type of the async def – we are sufficiently deep into plumbing that we are simply expected to get this right without static types.

```
ResultT  = TypeVar( 'ResultT' )
AwaitGen = Generator[ Request, None, ResultT ]
Coro     = Coroutine[ Request, None, ResultT ]
```

With that out of the way, we can define some 'special' functions that can be awaited, but are not defined using async def, which means that they will be able to yield into the scheduler. Recall that await expects an awaitable object and calls __await__ on it. The result of __await__ should be an iterator.
Of course, we can simply provide __await__ as a method, and to make things particularly easy, we can make it a generator. That way, calling __await__ automatically gets us a generator object, which happens to also be an iterator. We have already prepared a type alias for this occasion above: AwaitGen.
As always, calling async_read( 'foo' ) will use __init__ to initialize the object, at which point we create the request so that __await__ can forward it into the scheduler using yield. When control returns to __await__, we extract the result and pass it onto our caller.

```
class async_read:
    def __init__( self, file: str ):
        self.req = ReadRequest( file )
    def __await__( self ) -> AwaitGen[ str ]:
        yield self.req
        return self.req.result
```

Basically the same as above. Notice the different annotation on __await__, and how that matches the type of self.result in the above request types.

```
class async_write:
    def __init__( self, file: str, data: str ):
        self.req = WriteRequest( file, data )
    def __await__( self ) -> AwaitGen[ None ]:
        yield self.req
        return self.req.result
```

A helper function to actually process requests in the scheduler. We fake everything, for the sake of a demonstration.

```
def process( request: Request ) -> None:
    if isinstance( request, ReadRequest ):
        request.result = f'content of {request.file}'
    if isinstance( request, WriteRequest ):
        print( 'async_run: writing',
               request.data, 'into', request.file )
```

Finally the scheduler itself (not the most accurate name in this case, since it 'schedules' a single coroutine). We will look at the other aspect (actually scheduling green threads) in the next demo. Notice that we always send None as the response – we could actually send a response, but that would make the types uglier, and updating the request is also quite reasonable.

```
def async_run( coro: Coro[ ResultT ] ) -> ResultT:
    while True:
        try:
            request = coro.send( None )
            process( request )
        except StopIteration as e:
            return cast( ResultT, e.value )
```

Finally, we write a couple of 'user' functions using async def. To call into other async coroutines, We use the standard await construct now (we are no longer doing plumbing).

```
async def read_foo() -> str:
    foo = await async_read( 'foo' )
    return f'read_foo: {foo}'

async def main() -> int:
    x = await read_foo()
    print( f'main: result of read_foo was "{x}"' )
    await async_write( 'bar', 'stuff' )
    return 13
```

8.d.2 [fibres] In this demonstration, we will leave out requests (except a very simple one, that will allow us to actually switch fibres) and focus on fibre switching. For this purpose, our scheduler will take two coroutines at the start and switch between them whenever one of them yields[37] the CPU. First the trivial request:

```
class sched_yield:
    def __await__( self ) -> Generator[ None, None, None ]:
        yield None
```

A couple of type aliases for later convenience.

```
T = TypeVar( 'T' )
Coro = Coroutine[ None, None, T ]
```

That done, we can focus on the scheduler. As mentioned, we will pass two coroutines (each of them becoming a 'main' function of a single fibre) to the scheduler. We will collect their results and return them as a 2-tuple. For simplicity, we require both coroutines to have the same return type.

```
def run_scheduler( coro_a: Coro[ T ],
                   coro_b: Coro[ T ] ) -> tuple[ T, T ]:

    result: dict[ Coro[ T ], T ] = {}
```

Since we have exactly two fibres, we can simply bind them to a pair

[37] Please note that if you use the pseudo-code from the Sleator & Tarjan paper, you need to be careful about parallel assignment – in Python, it does not have the semantics intended by the authors and you will need to write it out in multiple steps.

of variables to indicate their status. The active fibre will be the one to execute in the next 'timeslot'.

```
    active:  None | Coro[ T ] = coro_a
    waiting: None | Coro[ T ] = coro_b
```

And the main loop: while we have a fibre to run, run it. If it yields (using sched_yield), swap it with the waiting fibre (if we have one, i.e. it did not terminate yet). If a fibre terminates, stash its result in a dictionary.

```
    while active:
        try:
            active.send( None )
        except StopIteration as e:
            result[ active ] = e.value
            active = None

        if waiting:
            active, waiting = waiting, active
```

Both fibres have terminated, give back their results to the caller.

```
    return result[ coro_a ], result[ coro_b ]
```

That's all there is for the scheduler. We can now write a simple (async) function which will serve as the main function of both our test fibres. It will simply print some progress messages and yield the processor in between. What message order do we expect?

```
async def fibre( n: int ) -> int:
    print( f'fibre {n} runs' )
    await sched_yield()

    for i in range( 2 * n ):
        print( f'fibre {n} continues' )
        await sched_yield()

    print( f'fibre {n} done' )
    return n
```

---

8.d.3 [spawn] The final piece of using async coroutines to implement fibres is creation of fibres on demand. In some sense, this is just a straightforward extension of the previous example: we simply need to realize that coroutine objects (and thus fibres) can be created by existing fibres and that they can be passed to the scheduler using the same request mechanism we have been using earlier (but with a new twist, combining the special function and the 'request' into a single entity – notice the yield self):

```
class async_spawn:
    def __init__( self, coro: Coro ):
        self.coro = coro
    def __await__( self ) -> AGen:
        yield self

Coro = Coroutine[ async_spawn, None, None ]
AGen = Generator[ async_spawn, None, None ]
```

We also need to extend the scheduler from the previous example to support an arbitrary number of fibres (instead of just two). We will put them on a queue (implemented using a deque), running a fibre until we can, then popping it off when it returns.

```
def run_scheduler( main: Coro ) -> None:

    queue  : deque[ Coro ] = deque()
    active : None | Coro   = main
    reqs   = 0
```

Request processing: there is only one type of request, so this is really simple. When spawning a new fibre is requested, put the 'main' of that fibre at the end of the queue. Eventually, it will get to run as fibres that spawned earlier terminate.

```
    def process( req ):
        if isinstance( req, async_spawn ):
            queue.append( req.coro )
        else:
            assert False # no other type of request exists
```

And the main loop: while we have a fibre to run, run it. If it terminates, pull out the next one from the queue. If the queue is empty, we are done. We also keep and return the count of requests that we served, as a simple sanity check.

```
    while active:
        try:
            process( active.send( None ) )
            reqs += 1
        except StopIteration as e:
            active = queue.popleft() if queue else None

    return reqs
```

That's our last demo scheduler. You can make a guess how the execution goes (i.e. what fibres will run and in what order).

```
async def fibre( n: int ) -> int:
    for i in range( n % 10 ):
        print( f'fibre {n} spawns {10 * n + i}' )
        await async_spawn( fibre( 10 * n + i ) )

    print( f'fibre {n} done' )
    return n
```

---

8.d.4 [yield] TBD

8.d.5 [context] TBD

## Part 8.p: Practice Exercises

8.p.1 [rrsched] Write an async (coroutine) scheduler which executes a given list of coroutines (the async def type) in a round-robin fashion. That is:

- provide suspend, an async method, which, when awaited, suspends the currently executing coroutine and allows the others to be scheduled (that is, given sched, a reference to the scheduler, a coroutine should be able to perform await sched.suspend()),
- tasks are added using add, which takes an unstarted (never awaited) coroutine as an argument and appends it to the end of the round-robin execution order (i.e. the coroutine that is added first is executed first, until it suspends, then the second executes until it suspends, and so on; when the last coroutine on the list suspends, wake up the first to continue, until it suspends, wake up the second, …),
- after at least one coroutine is added, calling run on the scheduler will start executing the tasks; run returns normally after all the tasks finish (note, however, that some tasks may terminate earlier than others).

See test_basic for a simple usage example. A few hints follow (you can skip them if you know what you are doing):

1. To implement suspend, you will want to create a low-level awaitable object (i.e. one which is not the result of async def). This is done by providing a special method __await__, which is a generator (i.e. it uses yield).
2. This yield suspends the entire stack of awaitables (most of which will be typically async coroutines), returning control to whoever called next on the iterator (the result of calling __await__ on the top-level awaitable).
3. Regarding mypy:

- the task passed to add should be a Coroutine (since the scheduler won't touch any of its outputs, these can be all set to object, instead of the much more problematic Any),
- the Awaitable protocol needs __await__ to be a Generator (you will need this for implementing suspend),
- when you call __await__() on an awaitable, the result is, among others, an Iterator.

```
class RoundRobin: pass
```

---

8.p.2 [priority] Write an async scheduler which executes a given list of coroutines in a priority-driven fashion. The add method takes, in addition to the coroutine itself, a static priority. Higher priorities get executed more often. Here is how it works:

1. In addition to the static priority (a fixed number), each task is assigned a dynamic priority. The dynamic priority starts out equal to the static one, but is decremented each time a coroutine is awakened.
2. The next coroutine to be awakened is always the one with the highest dynamic priority.
3. Whenever the highest dynamic priority in the system drops to zero, all tasks get their dynamic priority reset to their static priority.

Except as noted above, the interface and semantics of the scheduler carry over from p1.

```
class PrioritySched: pass
```

8.p.3 [exchange] Implement a class which coordinates a multi-producer, multi-consumer system built out of async coroutines. Each coroutine can either produce items (by calling put) or consume them (by calling get). Constraints:

- a given coroutine cannot call both put and get,
- a producer is blocked until the item can be consumed,
- a consumer is blocked until an item is produced.

These constraints mean that there can be at most one unconsumed item per producer in the system. If multiple producers have a value ready, the system picks up the one that has been waiting the longest. If multiple consumers are waiting for an item, again, the longest-waiting one is given the next item.
When run is called, all coroutines are started up, until each blocks on either put or get. The system terminates when no further items can be produced (there are no producers left).

```
T = TypeVar( 'T' )

class Exchange( Generic[ T ] ): pass
```

8.p.4 [box] Implement a class which coordinates a single producer and a single consumer (the producer puts the value in the 'box', where the consumer fetches it). The roles (producer vs consumer) are known upfront. The coroutines are passed to the constructor unevaluated (i.e. not as coroutine objects, but as functions which take the box as a parameter and return coroutine objects; see also below).

```
T = TypeVar( 'T' )
class Box( Generic[ T ] ): pass
```

---

8.p.6 [sort] You are given sched_yield, an awaitable that allows the scheduler to switch to a different coroutine, if needed. Given that, write a 'low-latency' sort function – one that does only O(1) work between two consecutive calls to sched_yield. Requirements:

- the sort should be in-place,
- the total runtime should be O(n·logn),
- use data.compare( a, b ) to compare items:
  - -1 means data[ a ] < data[ b ],
  - 0 means data[ a ] == data[ b ]
  - finally 1 means data[ a ] > data[ b ],
- use data.swap( a, b ) to swap values with indices a, b,
- len( data ) gives you the number of items.

```
class Array( Sized ):
    def compare( self, a: int, b: int ) -> int: ...
    def swap( self, a: int, b: int ) -> None: ...

async def sort( data: Array, suspend: Suspend ) -> None: pass

def check_run( data: Sequence[ int ] ) -> List[ int ]:
    counter = 0
    work_done = []

    def tick() -> None:
        nonlocal counter
        counter += 1
    def lap() -> None:
        nonlocal counter
        work_done.append( counter )
        counter = 0

    class array( Array ):
        def __init__( self, data: List[ T ] ) -> None:
            self.__data = data
        def compare( self, idx_a: int, idx_b: int ) -> int:
            tick()
            a = self.__data[ idx_a ]
            b = self.__data[ idx_b ]
            return 0 if a == b else 1 if a > b else -1
        def swap( self, idx_a: int, idx_b: int ) -> None:
            tick()
            val_a = self.__data[ idx_a ]
            val_b = self.__data[ idx_b ]
            self.__data[ idx_b ] = val_a
            self.__data[ idx_a ] = val_b
        def __len__( self ) -> int:
            return len( self.__data )

    Pause = Generator[ Tuple[ () ], None, None ]

    class pause( Awaitable[ None ] ):
        def __await__( self ) -> Pause: yield ()

    to_sort = array( list( data ) )
    the_sort = sort( to_sort, pause ).__await__()

    try:
        while True:
            assert next( the_sort ) == ()
            lap()
    except StopIteration:
        lap()

    for i in range( len( data ) - 1 ):
        assert to_sort.compare( i, i + 1 ) <= 0

    return work_done
```

## Part 8.r: Regular Exercises

8.r.1 [sleep] Write an async (coroutine) scheduler which executes a given list of coroutines (the async def type). When a coroutine suspends (using sched.suspend) it specifies how long it wants to sleep, in milliseconds. The scheduler wakes up a particular coroutine when its sleep timer expires (it should try to do it exactly on time, but sometimes this will be impossible because another coroutine blocks for too long). Like before, implement add to attach coroutines to the scheduler and run to start executing them. The latter returns when no coroutines remain.

```
class Sched: pass
```

8.r.2 [ioplex] Write an IO multiplexer for async coroutines. The constructor is given a 'coroutine function' (i.e. an async def, that is a function which returns a coroutine object) which serves as a factory. There are 3 methods:

- connect, which creates a new connection (i.e. it spawns a new server coroutine to handle requests) and returns a connection identifier,
- close which, given a valid identifier, terminates the corresponding connection,
- send which, given a connection identifier and a piece of data, sends the latter on to the corresponding server coroutine and returns the reply of that coroutine.

When creating server coroutines, the multiplexer passes read and write as arguments to the factory, where read is an async function (i.e. its result is await-ed) and returns the data that was passed to send; write, on the other hand, is a regular function and is called when the server coroutine wants to send data to the client. In other words, reading may block, but not writing.

```
class IOPlex: pass
```

8.r.3 [search] The class Tree represents a binary search tree. Implement search that performs a search for a given key, in logarithmic time and constant latency (between two calls to suspend). In each step, pass the value through which the search has passed to suspend, so that the caller can monitor the progress of the search.

```
T = TypeVar( 'T' )

class Tree( Generic[ T ] ):
    def __init__( self, value ) -> None:
        self.left  : Optional[ Tree ] = None
        self.right : Optional[ Tree ] = None
        self.value = value

    async def search( self, key, suspend ):
        pass
```

# Part S.2: Interpreters, Coroutines

In this set, there are 2 interpreters of simple languages – one with recursion and closures, another with explicit pointers and garbage collection. The third task explores an extension of the game solver from the first set (did you know that generators can be used to nicely encode backtracking?), while the fourth task is focused on the use of semi-coroutines (generators) in a latency-sensitive context.

1. a_rec – recursive programs
2. b_ptr – pointers and garbage collection
3. c_gravity – same game, iteration 2
4. d_rst – real-time splay trees

## Part S.2.a: rec

Implement an interpreter for simple recursive programs. The following syntax is taken unchanged from s1/a_while:

- one line = one statement (no exceptions),
- blocks are delimited by indentation (1–5 spaces),
- there are following statement types:
  - assignment,
  - if statement.

There are also two important changes:

1. The right-hand side of an assignment can be a function call, in addition to a built-in operation, written as:

   ```
   name₀ = func name₁ name₂ … nameₙ
   ```

2. There is a new statement type, function definition, which can only appear in the top-level scope (and is the only statement than can appear there), of the form:

   ```
   def funcname name₁ name₂ … nameₙ
   ```

   All functions can call all other functions, regardless of the order in which they are defined in the source. Function names follow the same rules as variable names.

Semantics change in the following way:

- all variables are local to the function in which they are used (declarations are still not needed),
- the result of a function call is the value of a variable with the same name, i.e. in function foo, the statement foo = 7 sets the return value to 7 (but does not terminate the function),
- the namespaces for variables and for functions are separate; operation names (add, and, …) must not be used for functions (but they can be used for variables).

Like if, a def statement is followed by a body, indented by a single space. Other restrictions on blocks remain the same as in s1/a_while. Example program:

```
def fib n
 one = 1
 two = 2
 fib = 1
 rec = gt n two
 if rec
  n_1 = sub n one
  n_2 = sub n two
  fib_1 = fib n_1
  fib_2 = fib n_2
  fib = add fib_1 fib_2
```

Write a function do_rec which takes a recursive program (as a string), a function name, and an arbitrary number of integers. The result is the return value of the function invoked, or a tuple of (line number, error string) in case the program fails. Return the first error in this order (within a group, return the number of the first line with an error):

1. syntax errors (including attempts to redefine a function),
2. errors in function calls:
   - use of an undefined function or
   - mismatch in the number of arguments,
3. runtime errors (division by zero).

Errors of type 2 should be reported even if they are in unused code (i.e. the test must be static). If the function passed to do_rec does not exist or the number of arguments does not match, return an error on (virtual) line 0.

## Part S.2.b: ptr

In this task, you will extend s1/a_while with pointers and garbage collection. The syntax is unchanged, except for addition of 3 new operations:

- `addr_ = set addr val` takes the value from variable `val` and stores it at the address `addr`; the result is `addr` shifted one cell to the right,
- `val = get addr off` loads the value from address `addr + off` and stores it in `val`,
- `addr = alloc count init` allocates a new object with `count` cells; all the cells are set to the value of variable `init`.

The memory available to the program is a fixed-size array of cells (its size is given to the interpreter at the start). It is an error if the program attempts to allocate more memory than it has available.
However, if the total size of reachable objects never exceeds that of the fixed-size memory, the program must not die with an out-of-memory error. A reachable object is one that the program can, at least in principle, read using a `get` operation ('in principle' means, in this case, that the program might need to execute an arbitrary sequence of operations to read the memory – even if the sequence doesn't actually appear in the program).
Addresses are treated as a distinct data type from numbers:

- the first argument of `get` and `set` must be a number,
- new addresses are created by `alloc`,
- adding a number and an address results in an address iff the result is within the bounds of the same object as the original address (same limitation applies to the result of `set`),
- an address may be stored in memory using `set`, and will still be an address if it is later retrieved by `get`,
- the numeric values of addresses are unspecified, except that:
  - addresses of different objects always compare unequal,
  - addresses within the same object compare reasonably (higher offsets are greater),
  - addresses always evaluate as `true` in `while` or `if`, or when used as an operand in a logical operator,
- the result of any other operation is a number (if any addresses appear as operands, the result will depend on their unspecified numeric values).

New semantic errors (compared to `s1/a_while`) – these are all reported at runtime i.e. when the offending operation executes:

- passing a number (i.e. not an address) as a first argument of `get` or `set`, or an address as the first argument to `alloc` or as a second argument to `get`,
- adding the address and the offset passed to `get` is out of bounds of the object into which the address originally pointed,
- memory allocation which would exceed the permitted memory size.

The error reporting mechanism is otherwise unchanged. An example program:

```
one = 1
two = 2
off = 2
x = alloc off two
while off
 off = sub off one
 y = get x off
 z = add z y
```

The interpreter shall be available via `do_ptr` with the program and the memory size in cells as arguments, and a dictionary of variables as the result.

## Part S.2.c: `gravity`

In this task, we will continue with the implementation of 'same game' from the first set. All the rules remain the same (ha-ha) except that gravity causes the board to reshuffle when more than $nm/10$ stones are removed all at once (where the board has $n \times m$ cells). A stone will fall if it is are either:

- unsupported - there is an empty cell right below it, or
- unstable – a stone is unstable on the left if it is missing both its direct left neighbour and the bottom-left diagonal neighbour (instability on the right is symmetrical); however edges of the board are considered stable (they do not count as 'missing a neighbour').

At most one stone is falling at any given time. The first stone to fall is the one nearest to the bottom (if there are multiple such stones, the leftmost one falls first).
The mechanics of the fall are as follows:

1. an unsupported stone will fall in a straight line toward the bottom until it hits another stone,
2. an unstable stone will roll off its position, by moving either to the empty cell below and to its left (or right, if it cannot roll to the left),
3. a stone that started to fall will continue to fall until it is both supported and stable (on both sides),
4. if a stone becomes unsupported due to another stone falling, it will be the next to fall (this does not apply to stones that become unstable – those are processed in the usual bottom-up, left-to-right order).

The reshuffle is considered part of the round that caused it. The scoring rule about adjacent removals remains otherwise unchanged (i.e. it might be triggered by a cell whose stone went missing due to gravity, and vice-versa, the bonus is not awarded when removing stones that got buried by an earthquake).
The entry point, `samegame`, is also unchanged.

## Part S.2.d: `rst`

This task is based on the splay tree from `s1/b_splay`. The changes are aimed at making the tree useful in low-latency applications: all operations become coroutines which must perform at most a constant amount of work between yields. This way, if the application needs to attend to other tasks while a lengthy splay is ongoing, it can simply keep the coroutine suspended. At any point of execution, the time until the next suspend is bounded by a constant, giving us a worst-case latency guarantee (i.e. the data structure is, in principle, suitable for hard-realtime systems).
To achieve the required properties, the tree needs to use top-down splaying, where the lookup is performed as part of the splay. Resources describing the top-down splay operation can be found here: https://www.link.cs.cmu.edu/splay/ (including pseudocode[1] and a C implementation of the operation). Here is my own description of the top-down splay operation:

- set up 2 subtrees, initially both empty, called `l` and `r`,
- there are 3 or 4 helper functions:
  - `link_left`, which takes a subtree and hangs it onto `l` using the rightmost link (i.e. as the right child of the bottom-right node) – you must maintain a pointer to that bottom-right node, to ensure `link_left` runs in O(1),
  - `link_right`, which is the mirror image of `link_left`,
  - the usual `rotate` (with two nodes as arguments, or possibly split into `rotate_left` and `rotate_right`);
- repeat until not interrupted:
  - if the value belongs into the left-left subtree of the current root, `rotate` the root with its left child (if this child exists) [first step of the zig-zig case],
  - if the new root lacks its left child, break the loop,
  - if the value belongs to the left subtree, perform `link_right` on the root and shift the root pointer to its right child [completes the zig-zig, or performs a simplified zig-zag],
  - the right-right and right cases are mirror images of the same.
- reassemble the tree:

- perform `link_left` on the left child of the current root,
- `link_right` on its right child,
- attach `l` and `r` to the root, `l` as the left and `r` as the right child (replacing the now invalid links).

Remaining operations (`find`, `insert` and `erase`) must perform all operations that are not O(1) by splaying the tree (and yield to the caller whenever the splay operation yields). The 'splay maximum to the top' operation (needed for erase) can be implemented by repeatedly 'splaying to the right' (in the sense of `splay( root.right.value )`, though of course taking 2 steps at a time will leave the tree in a significantly better shape),

The splay itself proceeds in a standard manner, except that after each step (zig, zig-zag, or zig-zig as appropriate), it yields the key of the new root. If the result of that yield is `None` (as happens when simply iterating the coroutine), the splay continues as usual. If it is anything else (delivered via `send`), the tree is reassembled into a consistent state (this must still happen in constant time!) and the operation is aborted.

The code to perform tree operations looks like this:

```
for _ in tree.insert( 7 ): pass
for _ in tree.erase( 3 ): pass
for _ in tree.filter( pred ): pass
for x in tree.find( 5 ):
    if x == 5:
        found()
```

Finally, the `to_list` operation is replaced by an iterator. This iterator is the only exception to the O(1) latency bound – it should not use `splay`. Instead, it should implement standard in-order traversal of the tree (i.e. yielding the keys stored in the tree in sorted order). It must be possible to have multiple simultaneously-active iterators over a single tree. All iterators however become invalid upon invocation of any of the remaining 4 operations. (Note: it is possible to implement an O(1)-latency iterator with a standard interface, but not one that also iterates the tree in sorted order.)

# Part 9: Text, JSON

This chapter will focus on working with text and structured textual data (JSON and related formats, such as YAML and TOML). We will look at both writing parsers 'from scratch' (using both regular expressions and recursive descent), but also using parsing libraries (the `json` and `csv` modules) and working with binary data.

Demonstrations:

1. (to be done)

Practice exercises:

1. `grep` – match regular expressions against text files
2. `magic` – identify file type by content
3. `report` – parse JSON and print human-readable output
4. `elements` – convert CSV to JSON
5. `mueval` – evaluate LISPy (prefix) expressions
6. `flatten` – convert JSON to TOML(-ish)

Regular exercises:

1. `email` – parsing e-mails the simple way
2. `toml` – recursive descent and INI files
3. `resolv` – parse a simplified `resolv.conf`
4. `fstab` – read and parse `/etc/fstab`
5. `yaml` – convert JSON to (readable) YAML
6. `cpp` – a simplified C preprocessor

Voluntary exercises:

1. (nothing here yet)

## Part 9.p: Practice Exercises

**9.p.1** [`grep`] The goal of this exercise is to write a simple program that works like UNIX `grep`.

Part 1: Write a procedure which takes 2 arguments, a string representation of a regex and a filename. It will print the lines of the file that match the regular expression (in the same order as they appear in the file). Prefix the line with its line number like so (hint: check out the `enumerate` built-in):

```
43: This line matched a regex,
```

Part 2: Change the code in the `if __name__ …` block below to only run `test_main` if an argument `--test` is given. Otherwise, expect 2 command-line arguments: a regular expression and a file name, and pass those to the `grep` procedure.

```
def grep( regex, filename ):
    pass
```

**9.p.2** [`magic`] Write function `identify` which takes `rules`, a list of rules, and `data`, a `bytes` object to be identified. It then tries to apply each rule and return the identifier associated with the first matching rule, or `None` if no rules match. Each rule is a tuple with 2 components:

- name, a string to be returned if the rule matches,
- a list of patterns, where each pattern is a tuple with:
  a. offset, an integer,
  b. bits, a `bytes` object,
  c. mask, another `bytes` object,
  d. positivity, a `bool`.

The mask and the pattern must have the same length. A rule matches the `data` if all of its patterns match.

A pattern match is decided by comparing the slice of `data` at the given offset to the 'bits' field of the pattern, after both the slice and the bits have been bitwise-anded with the mask. The pattern matches iff:

- the bits and slice compare equal and positivity is `True`, or
- they compare inequal and positivity is `False`.

```
def identify( rules, data ):
    pass
```

**9.p.3** [`report`] The goal here is to load the file `zz.report.json` which contains a report about a bug in a C program, and print out a simple stack trace. You will be interested in the key `active stack` (near the end of the file) and its format. The output will be plain text: for each stack frame, print a single line in this format:

```
function_name at source.c:32
```

```
import json # go for ‹load› (via io) or ‹loads› (via strings)

def report():
    pass
```

**9.p.4** [`elements`] In this exercise, we will read in a CSV (comma-separated values) file and produce a JSON file. The input is in `zz.elements.csv` and each row describes a single chemical element. The columns are, in order, the atomic number, the symbol (shorthand) and the full name of the element. Generate a JSON file which will consist of a list of objects, where each object will have attributes `atomic number`, `symbol` and `name`. The first of these will be a number and the latter two will be strings. The names of the input and output files are given to `csv_to_json` as strings.

Note that the first line of the CSV file is a header.

```python
import csv  # we want csv.reader
import json # and json.dumps

def csv_to_json( source, target ):
    pass
```

9.p.5 [mueval] Write an evaluator for a very small lisp-like language. Let there only be compound expressions (delimited by parentheses) which always have an integer arithmetic operator in the first position (+, -, *, /) and the remainder of the compound are either non-negative integer constants or other compounds. Assume the input is well-formed.

```python
def mueval( expr: str ) -> int:
    pass
```

9.p.6 [flatten] In this exercise, your task is to write a function that flattens JSON data to a form suitable for storing as TOML.
The result is a single-level (flat) dictionary, where the keys represent the previous structure of the data. We will use the period . for subobjects and # for subarrays. To make unambiguous un-flattening possible, if you encounter . or # in the original data, prefix it with a dollar sign, $ (i.e. write out $. or $#), if you encounter $. or $#, escape it with another dollar sign, to $$. or $$#, etc.
Example:

```
{ 'student': { 'Joe': { 'full name': 'Joe Peppy',
                        'address': 'Clinical Street 7',
                        'aliases': ['Joey', 'MataMata'] } } }
```

Flattened:

```
{ 'student.Joe.full name': 'Joe Peppy',
  'student.Joe.address': 'Clinical Street 7',
  'student.Joe.aliases#0': 'Joey',
  'student.Joe.aliases#1': 'MataMata' }
```

```python
def flatten( data: str ) -> str:
    pass
```

## Part 9.r: Regular Exercises

9.r.1 [email] In this exercise, we will parse a format that is based on RFC 822 headers, though our implementation will only handle the simplest cases. The format looks like this:

```
From: Petr Ročkai <xrockai@fi.muni.cz>
To: Random X. Student <xstudent@fi.muni.cz>
Subject: PV248
```

and so on and so forth. In real e-mail (and in HTTP), each header entry may span multiple lines, but we will not deal with that.
Our goal is to create a dict where the keys are the individual header fields and the corresponding values are the strings coming after the colon. In this iteration, assume that each header is unique.

```python
def parse_rfc822( filename ):
    pass
```

9.r.2 [toml] Write a recursive descent parser for simplified TOML (essentially an old-style INI file with restricted right-hand sides), with the following grammar:

```
top    = { line } ;
line   = ( header | kvpair ), '\n' ;
header = '[' word ']' ;
kvpair = word, '=', word ;
word   = alpha, { alnum } ;
alpha  = ? any letter on which isalpha() is true ? ;
alnum  = ? any letter on which isalnum() is true ? ;
```

If the input does not conform to the grammar exactly, reject it and return None. Otherwise return a dictionary of sections (see the type below). If the initial section does not have a header, it is stored under '' (empty string) in the section dictionary.

```python
Section = Dict[ str, str ]
TOML = Dict[ str, Section ]

def parse_toml( toml: str ) -> Optional[ TOML ]:
    pass
```

9.r.3 [resolv] Write a parser (of any kind) that validates a resolv.conf file (which contains DNS configuration). The simplified grammar is as follows:

```
top     := { stmt | comment } ;
stmt    := server, ( comment | [ spaces ], '\n' ) ;
server  := 'nameserver', spaces, addr ;
addr    := num, '.', num, '.', num, '.', num ;
num     := '0' | nonzero, { digit } ;
nonzero := '1' | '2' | … | '9' ;
digit   := '0' | nonzero ;
spaces  := ws_char, { ws_char } ;
ws_char := ? isspace() is True, except newline ? ;
comment := [ ws ], '#', { nonnl }, '\n' ;
nonnl   := ? any char except '\n' ? ;
```

```python
def resolv_valid( rc: str ) -> bool: pass
```

9.r.4 [fstab] Write a non-validating parser for the fstab file, which in traditional UNIXes contains information about filesystems. The format is as follows:
Comments start with # and extend until the end of line. Comments, additional whitespace, and blank lines are ignored. After comments and blanks are stripped, each line of the file describes a single filesystem. Each such description has 6 columns:

1. the device (path to a block device or an UUID),
2. the mount point,
3. the file system type,
4. a comma-separated list of mount options,
5. dump frequency in days (a non-negative integer, optional),
6. file system check pass number (same).

The type below describes the form in which to return the parsed data. If items 5 or 6 are missing, set them to 0.

```python
FS = Tuple[ str, str, str, List[ str ], int, int ]

def read_fstab( path: str ) -> List[ FS ]:
    pass
```

9.r.6 [cpp] Implement a C preprocessor which supports #include "foo" (without a search path, working directory only), #define without a value, #undef, #ifdef and #endif. The input is provided in a file, but the output should be returned as a string. Do not include line and filename information that cpp normally adds to files.

```python
def cpp( filename: str ) -> str:
    pass
```

# Part 10: Databases

This chapter is all about SQL and using relational databases to store and query data using Python. We will only look at the 'bare bones' low level interfaces (things like SQL Alchemy and ORMs are topics a little too big to tackle in this small course). We will be using SQLite3 in

place of a 'real' database, but replacing it with PostgreSQL or MariaDB or some big commercial database is a question of changing a few lines. And then learning a lot of SQL to make good use of them.
Demonstrations:

1. (to be done)

Prep exercises:

1. `bimport` – import books into a database
2. `bexport` – export books from a database
3. `bquery` – query the book database
4. `lcreate` – shopping with Python and SQL
5. `lsearch` – retrieve shopping lists from the database
6. `lupdate` – update lists in the database

Regular exercises:

1. `schema` – create tables given as JSON
2. `upgrade` – same, but with schema upgrade
3. `pkgs` – simple queries on a package database
4. `depends` – fetching transitive dependencies

Voluntary exercises:

1. (nothing here yet)

## Part 10.p: Practice Exercises

10.p.1 [`bimport`] Load the file `zz.books.json` and store the data in a database with 3 tables: `author`, `book` and `book_author_list`. Each author is uniquely identified by their name (which is a substantial simplification, but let's roll with it). The complete schema is defined in `zz.books.sql` and you can create an empty database with the correct data definitions by running the following command:

```
$ sqlite3 books.dat < zz.books.sql
```

```
import sqlite3
import json
```

NB. You want to execute `pragma foreign_keys = on` before inserting anything into sqlite. Otherwise, your foreign key constraints are just documentation and are not actually enforced. Let's write an `opendb` function which takes a filename and returns an open connection. Execute the above-mentioned `pragma` before returning.

```
def opendb( filename ):
    pass
```

Of course, you can also create the schema using Python after opening an empty database. See `executescript`. Define a function `initdb` which takes an open `sqlite3` connection, and creates the tables described in `sql_file` (in our case `zz.books.sql`). You can (and perhaps should) open and read the file and feed it into sqlite using `executescript`.

```
def initdb( conn, sql_file ):
    pass
```

Now for the business logic. Write a function `store_book` which takes a `dict` that describes a single book (using the schema used by `books.json`) and stores it in an open database. Use the `execute` method of the connection. Make use of query parameters, like this (`cur` is a cursor, i.e. what you get by calling `conn.cursor()`):

```
cur.execute( "insert into ... values ( ? )", ( name, ) )
```

The second argument is a tuple (one-tuples are written using a trailing comma). To fetch results of a query, use `cur.fetchone()` or `cur.fetchall()`. The result is a tuple (even if you only selected a single column). Or rather, it is a sufficiently tuple-like object (quacks like a tuple and all that).

```
def store_book( conn, book ):
    pass
```

With the core logic done, we need a procedure which will set up the database, parse the input JSON and iterate over individual books, storing each:

```
def import_books( file_in, file_out ):
    pass
```

---

10.p.2 [`bexport`] In the second exercise, we will take the database created in the previous exercise (`books.dat`) and generate the original JSON. You may want to use a join or two.
First write a function which will produce a `list` of `dict`'s that represent the books, starting from an open `sqlite` connection.

```
import sqlite3
import json
```

```
def read_books( conn ):
    pass
```

Now write a driver that takes two filenames. It should open the database (do you need the foreign keys pragma this time? why yes or why not? what are the cons of leaving it out?), read the books, convert the list to JSON and store it in the output file.

```
def export_books( file_in, file_out ):
    pass
```

---

10.p.3 [`bquery`] In the final exercise of this set, you will write a few functions which search the book data. Like you did for export, get a cursor from the connection and use `execute` and `fetchone` or `fetchall` to process the results. Use SQL to limit the result set.
Fetching everything (`select * from table` without a `where` clause) and processing the data using Python constructs is bad and will make your program unusable for realistic data sets.
The first function will fetch all books by a given author. Use the `like` operator to allow substring matches on the name. E.g. calling `books_by_author( conn, "Brontë" )` should return books authored by any of the Brontë sisters.

```
def books_by_author( conn, name ):
    pass
```

The second will fetch the set of people (i.e. each person appears at most once) who authored a book with a name that contains a given string. For instance, `authors_by_book( conn, "Bell" )` should return the 3 Brontë sisters and Ernest Hemingway. Try to avoid fetching the same person multiple times (i.e. use SQL to get a set, instead of a list).

```
def authors_by_book( conn, name ):
    pass
```

Another function will return names of books which have at least `count` authors. For instance, there are 3 books in the data set with 2 or more authors.

```
def books_by_author_count( conn, count ):
    pass
```

Finally, write a function which returns the average author count for a book. The function should return a single `float`, and ideally it would not fetch anything from the database other than the result: try to do the computation only using SQL.

```
def average_author_count( conn ):
    pass
```

---

10.p.4 [`lcreate`] The file `zz.lists.sql` contains a database schema for

keeping shopping lists. Besides shopping lists themselves, we will keep a table of item descriptions, a table of shops (vendors) and a table of supplies currently in your pantry. This last table also keeps track of a 'minimal' and 'preferred' amount for each item. Those will come in handy when we will want to create shopping lists automatically.
Each item may be available from multiple vendors, and of course each vendor stocks multiple items. Therefore, items and shops are in an M:N relationship, and we will keep this relationship in an auxiliary table. Finally, each vendor has, for each item, an individual unit price that is valid starting on a given date. A null price indicates that the item is not available in the given timespan. New start date overrides the price.
A shopping list, then, is a list of items to obtain. Each item on the list comes with:

- the quantity to obtain,
- the shop where to buy it and
- the quantity actually obtained.

Besides the list of items, the shopping list has a date attached to it. In this exercise, we will start by providing an interface for creating new lists.

```
from datetime import date
from sqlite3 import Connection
from typing import Optional, Callable, Type, Union
```

The classes in this exercise (and its follow-ups) will be associated with records in the database. Each class will hold onto an optional `id`: if the `id` is `None`, the record is not stored in the database (yet). So far, we will only set the `id` in the `create` method.
The only method which is allowed to change the database is `create` (in a later exercise, we will add `update`). All `set_*` and `add_*` methods (and later `remove_*`) methods should simply remember the changes and additions, until the user calls `create`, which then stores everything at once. Other methods may, however, query the database for data, if it is convenient to do so.
Finally, feel free to add a suitable base class, from which the other classes can be derived.

```
SQLT = Union[ str, int, float, date,
              Optional[ str ], Optional[ int ] ]
SQLP = tuple[ SQLT, ... ] # the 2nd parameter of Connection.execute

class Shop:
```

Creates an empty item, not yet associated with anything in the database. Set the internal `id` to `None`.

```
    def __init__( self, db: Connection ):
        pass

    def set_name( self, name: str ):
        pass
```

Create a record in the database. If the instance is already associated with a record, raise a `RuntimeError`. If the shop does not have a name, raise a `RuntimeError`.

```
    def create( self ):
        pass
```

All the remaining classes are analogous to `Shop`.

```
class Item:

    def __init__( self, db: Connection ):
        pass

    def set_name( self, name: str ):
        pass
```

Prices are associated not with just an item, but also a time period and a specific shop.

```
    def set_price( self, vendor: Shop, price: Optional[int],
                   start: date ):
        pass
```

If the item does not have a name, raise a `RuntimeError`.

```
    def create( self ):
        pass

class ShoppingList:

    def __init__( self, db: Connection ):
        pass

    def set_date( self, when: date ):
        pass

    def add_item( self, item: Item, qty: int ):
        pass
```

A shopping list might be empty, but it must have a date set. If it does not, refuse to create it (raise a `RuntimeError`).

```
    def create( self ):
        pass
```

---

10.p.5 [`lsearch`] In this exercise, we will extend the classes from `list_create` by adding various ways to fetch them from the database.

```
class FetchableShop( Shop ):
```

Find the shop in the database by its name. If no such shop is in the database, raise a `RuntimeError`. If found, set the internal `id` of the instance. Only allow fetching if the calling Shop instance's `id` is not set yet. If there are several shops with the same name, raise a `RuntimeError`.

```
    def fetch_by_name( self, name: str ):
        pass

    def fetch_by_id( self, ID: int ):
        pass
```

The top-level function `find_shops` will do a substring search on all the shops in the database, and return a `Shop` instance for each match.

```
def find_shops( db: Connection, pattern: str ):
    pass

class FetchableItem( Item ):

    def fetch_by_name( self, name: str ):
        pass

    def fetch_by_id( self, ID: int ):
        pass
```

Find a price at the given time in the given shop. Return `None` if the item is not available from the vendor at the time.

```
    def get_price( self, vendor: Shop, when: date ):
        pass
```

Find the best price available on a given date. Return a tuple of `int` (the price) and a `Shop` (the vendor which has this price), or `None` if the item is not available at all. Tie breaks alphabetically (prefer vendors with names that come first in a dictionary).

```
    def get_best_price( self, when: date ):
        pass

class FetchableShoppingList( ShoppingList ):

    def fetch_by_id( self, ID: int ):
        pass
```

Find all shopping lists that have a given item on it, in quantity at least qty. Returns a list of ShoppingList instances.

```
def find_lists_by_item( db: Connection, item: Item, qty: int ):
    pass
```

10.p.6 [lupdate]

```
T = TypeVar( 'T' )
```

In this exercise, we will extend the classes from list_search by adding an update method to each. If the entity does not exist in the database, update should raise a RuntimeError. After update, the database should reflect any changes and additions that have been done on the instance since it was either created, fetched or last updated.
Also add a delete method, which removes the entry and all the records it owns, from all relevant tables in the database. If you are deleting an entry that has associated records in other tables but does not own these records, raise a RuntimeError instead (an example would be removing a shop, while a pricing entry for that shop exists). After delete, the instance can no longer be used for anything (but you do not need to enforce this).

```
class UpdatableShop( FetchableShop ):

    def update( self ):
        pass

    def delete( self ):
        pass

class UpdatableItem( FetchableItem ):

    def update( self ):
        pass

    def delete( self ):
        pass

class UpdatableShoppingList( FetchableShoppingList ):

    def remove_item( self, item: Item ):
        pass

    def update( self ):
        pass

    def delete( self ):
        pass
```

The following function will check the current supplies and update the given shopping list so that afterwards, fetching everything on the list results in all supplies being at least at their 'minimum' level (if preferred is False) or at their 'preferred' level (if preferred is True). Do not remove anything from the list.
Note that some of the required items might be already on the list (but possibly in an insufficient quantity). Do not add more of an item than required for the restock, unless it already was on the list (specifically, calling add_missing a second time should have no effect, unless the current supply levels changed in the meantime).

```
def add_missing( shop_list: UpdatableShoppingList, preferred: bool
):
    pass
```

## Part 10.r: Regular Exercises

10.r.1 [schema] You are given a JSON file which describes a (very rudimentary) database schema. The top-level value is an object (dictionary) with table names as keys and objects which describe the columns as values.

The keys in the table description are column names and values (strings) are SQL types of those columns. Given a database connection and a path to the JSON file, create the tables. If one of them already exists, raise an error.

```
from sqlite3 import Connection, OperationalError, connect

def create_tables( schema: str, db: Connection ):
    pass
```

10.r.2 [upgrade] This exercise is the same as the previous one, with one important difference: if some of the tables already exist, this is not an error. However, the columns of the existing table and those specified by the schema might be different. In this case, create any missing columns, but do not touch columns that already exist.
Optional extension: print names of any extra columns, as a warning to the user that they no longer appear in the current schema and should be removed.
Note: the alter table command in sqlite is very limited. In a 'real' database, it is possible to alter column types, add and remove constraints and so on, all transactionally protected.

```
from sqlite3 import Connection, OperationalError, connect

def upgrade_tables( schema: str, db: Connection ) -> None:
    pass
```

10.r.3 [pkgs] You are given a database which stores information about packages, with the following tables:

```
package: id (primary key), name (string)
version: id (primary key), package_id (foreign key),
         number (string)
depends: version_id (foreign key), depends_on (foreign key)
```

Where depends_on also refers to version.id. Write the following functions.

```
from sqlite3 import Connection, connect
from typing import List, Tuple
```

Return a list of packages, along with the number of distinct versions of each package.

```
def list_packages( db: Connection ) -> List[ Tuple[ str, int ] ]:
    pass
```

Return the package versions (as a tuple of the package name and version 'number') that are not required by any other package (i.e. they form leaf nodes in the dependency tree).

```
def list_leaves( db: Connection ) -> List[ Tuple[ str, str ] ]:
    pass
```

For each package version, give the number of packages (package versions) which directly depend on it.

```
def sum_depends( db: Connection ) -> List[ Tuple[ str, str, int ] ]:
    pass
```

-- >% -- >% --

```
def mkdb() -> Connection:
    conn = connect( ':memory:' )
    c = conn.cursor()
    c.execute( 'create table package ( id integer primary key, ' + \
               'name varchar )' )
    c.execute( 'create table version ( id integer primary key, ' + \
               'package_id integer, number varchar )' )
    c.execute( 'create table depends ( version_id integer,' + \
               'depends_on integer )' )

    def add_pkg( name: str, *vers: str ) -> None:
```

```
        c.execute( 'insert into package ( name ) values ( ? )',
                   ( name, ) )
        pid = c.lastrowid
        for v in vers:
            c.execute( 'insert into version ( package_id, ' + \
                       'number ) values ( ?, ? )', ( pid, v ) )

    def add_dep( p1: str, v1: str, p2: str, v2: str ) -> None:
        get = '( select version.id from version join ' + \
              'package on package.id = package_id ' + \
              'where name = ? and number = ? )'
        c.execute( 'insert into depends ( version_id, depends_on ) ' + \
                   f'values ( {get}, {get} )', ( p1, v1, p2, v2 ) )

    add_pkg( 'libc', '2.0', '2.1', '2.2' )
    add_pkg( 'ksh', '1.0', '1.1' )
    add_pkg( 'dummy' )

    add_dep( 'ksh', '1.0', 'libc', '2.0' )
    add_dep( 'ksh', '1.1', 'libc', '2.1' )

    return conn
```

# Part 11: Asynchronous Programming

Coroutines again, this time in a very practical context: asyncio (finally). We will be writing both clients and servers using a modern approach based on an IO dispatch loop (bundled with asyncio) and suspending coroutines. I am sure you will be happy to learn that asyncio is essentially what node.js is for JavaScript (and we all know how popular that is). Low latency, high-throughput applications, here we come (yes, I know it's Python).

Demonstrations:

1. (to be done)

Practice exercises:

1. sem – semaphore synchronisation in asyncio
2. proc – asyncio processes
3. multi – more processes
4. tcp – a simple TCP echo server
5. http – an HTTP client with a subprocess
6. merge – process data from multiple sockets

Regular exercises:

1. sleep – sleep, running tasks in parallel
2. counter – two-way communication with a process
3. pipeline – multi-stage asynchronous processing
4. tokenize – another stream pipeline exercise
5. minilisp – an asynchronous parser
6. rot13 – listening on UNIX domain sockets

Voluntary exercises:

1. (nothing here yet)

## Part 11.p: Practice Exercises

**11.p.1** [sem] Use gather() to spawn 10 tasks, each running an infinite loop. Create a global semaphore that is shared by all those tasks and set its initial value to 3. In each iteration, each task should queue on the semaphore and when it is allowed to proceed, sleep 2 seconds before calling notify, and relinquishing the semaphore again.

notify adds a tuple – containing the task id ( 1 - 10 ) and the time when the task reached the semaphore – to the global list reached.

Observe the behaviour of the program. Add a short sleep outside of the critical section of the task. Compare the difference in behaviour.

After your program works as expected, i.e. only 3 tasks are active at any given moment and the tasks alternate fairly, switch the infinite loop for a bounded loop: each task running twice, to be consistent with the tests.

Note: Most asyncio objects, semaphores included, are tied to an event loop. You need to create the semaphore from within the same event loop in which your tasks will run. (Alternatively, you can create the loop explicitly and pass it to the semaphore.)

```
import asyncio
import time

reached: list[ tuple[ int, float ] ] = []
begin = time.time()

def notify( i: int ) -> None:
    t = time.time() - begin
    print( "task {} reached semaphore at {}".format( i, t ) )
    reached.append( ( i, t ) )

async def semaphores() -> None:
    pass
```

**11.p.2** [proc] In this exercise, we will look at talking to external programs using asyncio. There are two coroutines in the asyncio module for spawning new processes: for simplicity, we will use create_subprocess_shell.

However, before you start working, try the following shell command:

```
$ while read x; do echo x is $x; done
```

and type a few lines. Use ctrl+d to terminate the loop.

This is one of the programs we will interact with. Use stdout and stdin streaming to talk to this simple shell program from python: send a line and read back the reply from the program. Copy it to the standard output of the python program. Apart from printing, return a list of all outputs from the shell program. There are two arguments, the command to run and a list of inputs to serve this program one-by-one.

NOTE: The data that goes into the process and that comes out is bytes, not strings. Make sure to encode and decode the bytes as needed.

```
import asyncio
from asyncio.subprocess import PIPE
from typing import List

async def pipe_cmd( command: str,
                    inputs: List[ str ] ) -> List[ str ]:
    pass
```

**11.p.3** [multi] Spawn 2 slightly different instances of the shell program from previous exercise, then use gather to run 3 tasks in parallel:

- two that print the output from each of the processes
- one that alternates feeding data into both of the subprocesses

First shell program reads its input and outputs p1: [input value]. Second shell program reads its input and outputs p2: [input value]. Process 3 sends characters a through h to the two printing processes; it first sends the character to process 1, then waits 0.5 seconds, then it sends the same character to process 2 and waits 0.2 seconds. The outputs of the two main processes are printed to stdout, so that you can follow what is going on, and added to the global data list, along with a timestamp (see p1) – as a tuple.

Don't forget to clean up at the end.

```
import asyncio
from asyncio.subprocess import PIPE
```

```
from typing import Tuple

data : list[ tuple[ str, float ] ] = []

async def multi() -> None:
    pass
```

11.p.4 [tcp] Start a server, on localhost, on the given port (using asyncio.start_server) and have two clients connect to it. The server takes care of the underlying sockets, so we will not be creating them manually. Data is, again, transferred as bytes object.
The server should return whatever data was sent to it. Clients should send hello and world, respectively, then wait for the answer from the server and return this answer. Add print statements to make sure your server and clients behave as expected; print data received by the server, sent to the clients and sent and received by the clients on the client side. Make sure to close the writing side of sockets once data is exhausted.

```
import asyncio
```

Server-side handler for connecting clients. Read the message from the client and echo it back to the client.

```
async def handle_client( reader, writer ):
    pass # print( "server received & sending", ... )
```

Client: connect to the server, send a message, wait for the answer and return this answer. Assert that the answer matches the message sent. Sleep for 1 second after sending world, to ensure message order.

```
async def client( port: int, msg: str ):
    pass # print( "client sending", ... )
    pass # print( "client received", ... )
```

The start function should start the server on the provided port and return it. The stop function should stop the server returned by start.

```
async def start( port: int ):
    pass

async def stop( server ) -> None:
    pass

async def test_main() -> None:

    import sys
    import random
    from io import StringIO

    stdout = sys.stdout
    out = StringIO()
    sys.stdout = out

    port = random.randint( 9000, 13000 )
    server = await start( port )
    data = await asyncio.gather( client( port, 'hello' ), client(
port, 'world' ) )
    await stop( server )

    assert data == [ 'hello', 'world' ], data

    sys.stdout = stdout
    output_ = out.getvalue()
    output = output_.split('\n')

    assert 'client sending hello' in output[ 0 : 4 ], output
    assert 'client sending world' in output[ 0 : 4 ], output
    assert 'server received & sending hello' in \
            output[ 1 : 3 ], output
    assert 'server received & sending world' in \
            output[ 1 : ], output
```

11.p.5 [http] Use aiohttp (python -mpip install aiohttp) to fetch a given URL and stream the HTML into tidy (html-tidy.org). Specifically, use tidy 2>&1 as the command that you start with asyncio.create_subprocess_shell. Capture the stdout and return the output until the first blank line, as a list of bytes objects.

```
import aiohttp
import asyncio
from asyncio.subprocess import PIPE

async def tidy( url ):
    pass
```

11.p.6 [merge] Write a 'merge server', which will take 2 string arguments, both paths to unix sockets. The first socket is the 'input' socket: listen on this socket for client connections, until there are exactly 2 clients. The clients will send lines, sorted lexicographically.
Connect to the 'output' socket (second argument) as a client. Read lines as needed from each of the clients and write them out to the output socket, again in sorted order. Do not buffer more than 1 line of input from each of the clients.
Use readline on the input sockets' streams to fetch data, and relational operators (<, >, ==) to compare the bytes objects.

```
import asyncio
```

The merge_server coroutine will simply start the unix server and return the server object, just like asyncio.start_unix_server does.

```
async def merge_server( path_in, path_out ):
    pass
```

## Part 11.r: Regular Exercises

11.r.1 [sleep] Demonstrate the use of native coroutines and basic asyncio constructs. Define 2 coroutines, say cor1() and cor2(), along with an asynchronous driver, sleepy(). Make the coroutines suspend for a different amount of time (say 0.7 seconds and 1 second) and then print the name of the function, in an infinite loop.
Use asyncio.gather to run them in parallel (from your sleepy(), which you should invoke by using asyncio.run() at the toplevel) and observe the result. What happens if you instead await cor1() and then await cor2()? Try making the loops in corN finite (tests are meant for 5 iterations, but feel free to play around with them).

```
async def sleepy():
    pass
```

11.r.2 [counter] Spawn a given number of instances of the following shell program:

```
while true; do echo .; sleep {n}; done
```

Where the values for {n} are given in the argument sleeps. Run all these programs in parallel and monitor their output (asserting that each line they print is exactly a single dot).
Once a second, use queue.put to send a list of numbers, each of which gives the number of dots received from the $i$-th subprocess. For instance, the first list should be approximately [ 1, 2, 10 ] if sleeps were given as [ 1, 0.5, 0.1 ]. The last parameter, iterations tells you how many one-second intervals to run for (and hence, how many items to put into the queue). After the given number of iterations, kill all the subprocesses.

```
async def counters( queue, sleeps, iterations ):
    assert False
```

–– >% –– >% ––

```
def fuzzy( a: List[ int ], b: List[ int ] ) -> bool:
    from math import ceil
    for i, j in zip( a, b ):
        if abs( i - j ) > ceil( j / 10.0 ):
            return False
    return True

async def check( q: asyncio.Queue[ List[ int ] ] ) -> None:
    assert fuzzy( await q.get(), [ 1, 2, 10 ] )
    assert fuzzy( await q.get(), [ 2, 4, 20 ] )
    assert fuzzy( await q.get(), [ 3, 6, 30 ] )

async def main() -> None:
    q : asyncio.Queue[ List[ int ] ] = asyncio.Queue()
    await asyncio.gather( check( q ), counters( q, [ 1, 0.5, 0.1 ],
3 ) )
```

11.r.3 [pipeline] In this (and the next) exercise, we will write coroutines which can be connected into a sort of pipeline, like what we did with generator-based streams in week 4. Again, there will be sources, sinks and processors and the coroutines will pass data to each other as it becomes available.
Native coroutines have an arguably a more intuitive and more powerful construct to send data to each other than what is available with generators: asyncio.Queue. The queues are of two basic types: bounded and unbounded. The former limits the amount of memory taken up by 'backlogs' and enforce some level of synchronicity in the system.
In the special case where the size bound is set to 1, the queue behaves a lot like send/yield. Trying to get an item from a queue that is empty naturally blocks the coroutine (making it possible for the writer coroutine to run) – this is quite obvious. However, if the queue is bounded, the opposite is also true: writing into a full queue blocks the writer until space becomes available. This lets the reader make progress at the expense of the writer. Recall also the schedulers from week 8.
We will use such queues to build up our stream pipelines: sinks and sources will accept a single queue as a parameter each (sink as its input, source as its output), while a processor will accept two (one input and one output). Like before, we will use None to indicate an empty stream, however, we will not repeat it forever (i.e. only send it once).
In this exercise, we will write two simple processors for our stream pipelines:

- a chunker which accepts str chunks of arbitrary sizes and produces chunks of a fixed size,
- getline which accepts chunks of arbitrary size and produces chunks that correspond to individual lines [TBD pre-made tests missing].

Note: if you use Python 3.8, asyncio.Queue is not a generic type. You will need to adjust the type annotations accordingly.

```
def chunker( size ):

    async def process( q_in, q_out ):
        await q_out.put( None )

    return process

async def test_main() -> None:
    sink_done = False
    Queue = asyncio.Queue[ Optional[ str ] ]

    async def source( q_out: Queue ) -> None:
        await q_out.put( 'hello ' )
        await q_out.put( 'world' )
        await q_out.put( None )

    async def check( pipe: Queue, expect: Optional[ str ] ) -> None:
        x = await pipe.get()
        assert x == expect, f"{x} == {expect}"

    async def sink_4( q_in: Queue ) -> None:
        nonlocal sink_done
        await check( q_in, 'hell' )
        await check( q_in, 'o wo' )
        await check( q_in, 'rld' )
        await check( q_in, None )
        sink_done = True

    async def sink_2( q_in: Queue ) -> None:
        nonlocal sink_done
        await check( q_in, 'he' )
        await check( q_in, 'll' )
        await check( q_in, 'o ' )
        await check( q_in, 'wo' )
        await check( q_in, 'rl' )
        await check( q_in, 'd' )
        await check( q_in, None )
        sink_done = True

    def pipeline( *elements: Any ) -> List[ Any ]: # coroutines
        q_out : Queue = asyncio.Queue( 1 )
        line = [ elements[ 0 ]( q_out ) ]
        for e in elements[ 1 : -1 ]:
            q_in = q_out
            q_out = asyncio.Queue( 1 )
            line.append( e( q_in, q_out ) )
        line.append( elements[ -1 ]( q_out ) )
        return line

    async def run( *pipe: Any ) -> None:
        nonlocal sink_done
        sink_done = False
        await asyncio.gather( *pipeline( *pipe ) )
        assert sink_done

    await run( source, chunker( 4 ), sink_4 )
    await run( source, chunker( 2 ), chunker( 4 ), sink_4 )
    await run( source, chunker( 7 ), chunker( 4 ), sink_4 )
    await run( source, chunker( 7 ), chunker( 2 ), sink_2 )
    await run( source, chunker( 4 ), chunker( 2 ), sink_2 )
    await run( source, chunker( 3 ), chunker( 2 ), sink_2 )
```

11.r.4 [tokenize] Nothing here yet. Please try again later.

11.r.5 [minilisp] Write an asynchronous parser for a very limited subset of the LISP grammar from t3/lisp.py. Specifically, only consider compound expressions and atoms. Represent atoms using str and compound expressions using lists (note: it might be hard to find a reasonable mypy type – it is quite okay to skip mypy in this exercise). The argument to the parser is an asyncio.StreamReader instance. Your best bet is reading the data using readexactly( 1 ). The parser should immediately return after reading the closing bracket of the initial expression.

```
async def minilisp( reader ):
    pass

async def test_main() -> None:
    import os
    loop = asyncio.get_running_loop()
    r_fd, w_fd = os.pipe()
    w_file = os.fdopen( w_fd, 'w' )
    r_stream = asyncio.StreamReader()
    await loop.connect_read_pipe( lambda:
            asyncio.StreamReaderProtocol( r_stream ),
            os.fdopen( r_fd ) )

    def send( data: str ) -> None:
        w_file.write( data )
        w_file.flush()

    async def check( *expect: Any ) -> None:
```

```
        got = await minilisp( r_stream )
        assert got == list( expect ), f"{got} == {expect}"

    send( '(hello)' )
    await check( 'hello' )
    send( '(hello world)' )
    await check( 'hello', 'world' )
    send( '(hello (world))' )
    await check( 'hello', [ 'world' ] )
    send( '((hello) (cruel (or not) world))' )
    await check( [ 'hello' ],
                 [ 'cruel', [ 'or', 'not' ], 'world' ] )
```

11.r.6 [rot13] We will do something similar to p4_tcp, but this time we will use a UNIX socket. UNIX sockets exist in the filesystem and need to be given a (file)name. Additionally, instead of simply echoing the text back, we will use Caesar cipher (rotate the characters) with right shift (the intuitive one) of 13. We will have to explicitly remove the socket once we are done with it, as it will stay in the filesystem otherwise.

```
async def handle_client( reader, writer ):
    pass # print( "server received", ... )
    pass # print( "server sending", ... )

async def client( msg, path ):
    pass # print( "client sending", ... )
    pass # print( "client received", ... )

async def unix_rot( path ):
    pass
```

# Part 12: Math and Statistics

The last chapter (and possibly the least popular part of the course) is about using Python for math. It is here mainly because Python is very popular in scientific computation, aka number crunching (with numpy and scipy doing all the heavy lifting there), and in data science (mainly with pandas). It is also a gateway to more advanced statistics that is also very often driven by Python scripts (think machine learning). All in all, getting a feel for using the big toys cannot hurt, even if the math perhaps does, a little.
NB. For exercises in this chapter, you need numpy ≥ 1.22, pandas and pandas-stubs ≥ 2022.2.
Demonstrations:

1. (to be done)

Practice exercises:

1. linear – matrices warmup
2. volume – polyhedron volume
3. signal – generating sine waves
4. stats – simple stats with pandas
5. outliers – dealing with irregularities in data
6. student – the t-test

Regular exercises:

1. hist – drawing histograms with ASCII art
2. dft – discrete Fourier transform
3. null – the null space of a matrix
4. frames – slicing and dicing pandas dataframes
5. regress – linear regression, with outliers
6. anova – TBD analysis of variance

Voluntary exercises:

1. (nothing here yet)

## Part 12.p: Practice Exercises

12.p.1 [linear] The goal of this exercise is to learn about numpy arrays. Write a function which takes a list of numbers, interprets it as a square matrix and computes the inverse, second power, the determinant. The function should return those values as a 3-tuple, with matrices represented the same way as input: as a flat list of numbers. Return None in place of inverse if the matrix is singular, i.e. has no inverse.

```
import numpy as np

def linalg( matrix ):
    pass
```

12.p.2 [volume] Compute the volume of an $n$-dimensional simplex, given as a list of $n + 1$ points. A 2D simplex is a triangle, given by 3 points, a 3D simplex is a 3-sided pyramid given by 4 points and so on.

```
def volume( pts ):
    pass
```

12.p.3 [signal] Write a function that generates 1 second of signal as a sequence of amplitude samples, built from a given mix of sinus frequencies. The result should contain count samples, including the initial state at t = 0. 1 second is the time of 1 full cycle of a sine wave with frequency 1. Return it as an ndarray.
Then write a function logscale, which takes a histogram represented as a list of floats and converts its x axis to logscale. That is: the first item is discarded, the second item becomes first, the average of 3rd and 4th item comes second, the average of 5th through 8th items comes third, and so on. Compare np.ceil( np.log2( range( 1, 32 ) ) ).

```
import numpy as np

def freq( count, freqs ):
    pass

def logscale( data ):
    pass
```

12.p.4 [stats] Grab the data from the given filename and compute the average, median, first and last quartile and variance of each numeric column. Put the data into a dictionary with sub-dictionaries as values, e.g.

```
{
    'average': { 'age': 39.207, 'bmi': 30.663, … },
    'variance': …,
    'first quartile': { 'age': 27, … },
    'last quartile': { 'age': 51, … },
    …
}

def stats( filename ):
    pass
```

12.p.5 [outliers] Write a function that removes outliers from an otherwise normally distributed data set, given as a list of 2-tuples (x, y). You can create random inputs for testing with numpy.random.normal( mean, stddev, count ) and then add a few outliers manually.

```
import numpy as np
from numpy.typing import NDArray
from typing import List, Tuple
```

What exactly constitutes an outlier is somewhat domain- and dataset-

specific, but using some small integer multiple (3-5) of $\sigma$ (the standard deviation) as the cutoff is quite common.
You can use pandas data frames in the implementation if you like, or even construct them outside and pass them to the function directly. Remove all outliers strictly outside the range given by the `nsigmas` argument. Return the filtered list.

```
def drop_outliers( data, nsigmas ):
    pass
```

Now that we have a function to remove outliers, let's look at what effect it has. The following function should call `f` on both the original data, and the outlier-culled variant. Return a 2-tuple of (original data, outliers removed) where each is itself a 2-tuple (x, y). Apply `f` on each axis separately (i.e. for a dataset with $x$ values `xs` and $y$ values `ys`, return `f( xs ), f( ys )`).

```
Data = List[ Tuple[ float, float ] ]

def cmp_outliers( data: Data, nsigmas, f ):
    pass
```

Try computing mean, median, quartiles and standard deviation of a few data sets with a more or less severe outlier problem.

**12.p.6** [`student`] The t-test is used, among other things, to assess whether two population means of some attribute are the same, based on a sample of each of the two populations. The test makes a few assumptions, the most important being:

1. the attribute is normally distributed,
2. the variances of the two samples are similar,
3. the sample sizes are equal.

The assumptions are not exact: small deviations only lead to small inaccuracy in the result. Hence, we can set up some tolerances. Implement a predicate `t_validate` that takes 2 sets of numbers, and tolerance arguments as follows:

- `normality` is the maximum p-value that we are willing to accept for a normality test on the input data (use a Shapiro-Wilk test to obtain the p-value),
- `variance` is the difference of variances that we are willing to tolerate, and finally
- `relsize` is the relative size difference that we are willing to accept (i.e. we accept the samples if their size difference divided by their size average is less than `relsize`).

```
def t_validate( s_1, s_2, normality, variance, relsize ):
    pass
```

Then implement a function `split` that takes:

- `data`, a pandas data frame,
- `col`, the column to test,
- `split_col`, the column by which the data is split into two disjoint sets,
- `split_val` if `None`, `split_col` must have exactly 2 values, which are taken to be the sample sets to compare, otherwise `split_val` is a number and `split_col` is numeric: then the two sets are given by `data[split_col] < split_val` and `data[split_col] >= split_val`.

The result of `split` is two sets of numbers (in the form of single-column data frames).

```
def split( data, col, split_col, split_val = None ):
    pass
```

Finally implement `pvalue` which takes 2 samples (sets of numbers) and produces a p-value indicating the likelihood that the means of the corresponding populations are equal.

```
def pvalue( s_1, s_2 ):
    pass
```

Note on typing: if you decide to use `scipy.stats`, you will need to import it with `# type: ignore`, since `scipy` does not have `mypy` stubs.

# Part 12.r: Regular Exercises

**12.r.1** [`hist`] Write a function that takes a list of numbers and draws an ASCII histogram (into a string). Normalize the height to 25 characters. You can compare your output with example output which uses the `*` character to represent value frequency.

```
def histogram( bins ):
    pass
```

**12.r.2** [`dft`] Write a function which reconstructs the frequencies which were given to `freq` in `p3_signal.py`, as an ascending list of integers.
Note that the FFT algorithm used in numpy will give you non-zero amplitudes for every frequency – use `isclose` to check if the amplitude is almost zero.
You can assume that the input only contains integer frequencies. When testing, be careful to avoid aliasing (i.e. make sure the highest frequency passed to `freq` from `p3_signal.py` is less than half the number of samples).

```
def dft( amp ):
    pass
```

**12.r.3** [`null`] Given a square matrix, find its 'null space' in the form of a list of unit-length basis vectors for that space. The null space (or a kernel) of a matrix is the space of all vectors which, multiplied by the matrix, come out as zero. For instance:

$$\begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{matrix} \times \begin{matrix} x \\ y \\ z \end{matrix} = \begin{matrix} x \\ y \\ 0 \end{matrix}$$

This comes out zero if $x = y = 0$, regardless of $z$. Hence, the null space is spanned by the single vector (0, 0, 1). Indeed:

$$\begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{matrix} \times \begin{matrix} 0 \\ 0 \\ 1 \end{matrix} = \begin{matrix} 0 \\ 0 \\ 0 \end{matrix}$$

If we consider another matrix, we see:

$$\begin{matrix} 1 & 1 & 0 \\ 2 & 2 & 0 \\ 0 & 0 & 0 \end{matrix} \times \begin{matrix} x \\ y \\ z \end{matrix} = \begin{matrix} x + y \\ 2x + 2y \\ 0 \end{matrix}$$

The vector is zero whenever $x = -y$ (and irrespective of $z$). Hence, the null space is two-dimensional, spanned by (for instance) the vectors (1, -1, 0) and (0, 0, 1).

$$\begin{matrix} 1 & 1 & 0 \\ 2 & 2 & 0 \\ 0 & 0 & 0 \end{matrix} \times \begin{matrix} 1 \\ -1 \\ 0 \end{matrix} = \begin{matrix} 0 \\ 0 \\ 0 \end{matrix}$$

$$\begin{matrix} 1 & 1 & 0 \\ 2 & 2 & 0 \\ 0 & 0 & 0 \end{matrix} \times \begin{matrix} 0 \\ 0 \\ 1 \end{matrix} = \begin{matrix} 0 \\ 0 \\ 0 \end{matrix}$$

Notice that we have chosen the basis so that it is orthogonal:

$$\begin{matrix} 1 & -1 & 0 \end{matrix} \times \begin{matrix} 0 \\ 0 \\ 1 \end{matrix} = 0$$

It's easy to make it orthonormal, just divide the first vector by a square root of 2. In the exercise, however, orthogonality is not required (it just makes it easy to see that the vectors are linearly independent).

```
import numpy as np
from numpy.typing import NDArray
from typing import cast, List

FloatArray = NDArray[ np.float64 ]

def null(m):
    pass
```

**12.r.4** [frames] The data for this exercise is in `zz.frames.csv`. The data represents grading of a programming subject (with made-up names and numbers, of course). The columns are names, number of points from weekly exercises, from assignments and from reviews. Implement the following functions:
Return a `DataFrame` which only contains rows of students, which achieved the best result among their peers in one of the categories (weekly, assignments, reviews). If there are multiple such students for a given category, include all of them.

```
def best( data ):
    pass
```

Return a `DataFrame` which contains the name and the total score (as the only 2 columns). Don't forget that the weekly exercises contribute at most 9 points to the total.

```
def compute_total( data ):
    pass
```

Return a dictionary with 4 keys (`weekly`, `assignments`, `reviews` and `total`) where each value is the average number of points in the given category. Consider factoring out a helper function from `compute_total` to get a `DataFrame` with 5 columns.

```
def compute_averages( data ):
    pass
```

-- >% -- >% --

```
def eq( data: pd.DataFrame, student: str, col: str, val: float ) ->
bool:
    matches = data[ data[ 'student' ] == student ][ col ]
    return cast( bool, ( matches == val ).all() )
```

**12.r.5** [regress] In this case, the input data will again be (x, y) tuples, but distributed around a straight line and we will compute linear regression on the data. This time, we will remove outliers iteratively: find the term with the greatest squared residual and if the squared residual is larger than `cutoff`-times the sum of all squared residuals, drop the data point and restart the regression. Stop when there are no more outliers.
Feel free to use `pandas` and/or `numpy`.

```
def drop_outliers( data, cutoff ): # add arguments if you like
    pass # return filtered data

def regress( data, cutoff ):
    pass # remove outliers iteratively
         # return the slope and the intercept of the regression line
```

NOTE: In both `p5` and in this exercise, we have taken a rather cavalier approach to outlier removal. For real statistics on real data, you often need to be much more careful and take the origin of the data set into account. Always disclose any outliers you have removed from further consideration.

# Part S.3: Persistence

This task set is centered around persistent data. There are two database-focused tasks and two parsing-focused tasks.

1. `a_lisp` – a simple context-free parser
2. `b_squelter` – storing the shelter objects with SQL
3. `c_merkle` – persistent trees
4. `d_numeval` – syntax + linear algebra

## Part S.3.a: `lisp`

Write a simple LISP (expression) parser, following this EBNF grammar:

```
expression  = atom | compound ;
compound    = '(', expression, { whitespace, expression }, ')' |
              '[', expression, { whitespace, expression }, ']' ;
whitespace  = ( ' ' | ? newline ? ), { ' ' | ? newline ? } ;
atom        = literal | identifier ;
literal     = number | string | bool ;

nonzero    = '1' | '2' | '3' | '4' |
             '5' | '6' | '7' | '8' | '9' ;
digit      = '0' | nonzero ;
sign       = '+' | '-' ;
digits     = '0' | ( nonzero, { digit } ) ;
number     = [ sign ], digits, [ '.', { digit } ] ;

bool       = '#f' | '#t' ;

string     = '"', { str_char }, '"' ;
str_lit    = ? any character except '"' and '\' ? ;
str_esc    = '\"' | '\\' ;
str_char   = str_lit | str_esc ;

identifier = id_init, { id_subseq } | sign ;
id_init    = id_alpha | id_symbol ;
id_symbol  = '!' | '$' | '%' | '&' | '*' | '/' | ':' | '<' |
             '=' | '>' | '?' | '_' | '~' ;
id_alpha   = ? alphabetic character ? 
id_subseq  = id_init | digit | id_special ;
id_special = '+' | '-' | '.' | '@' | '#' ;
```

Alphabetic characters are those for which `isalpha()` returns `True`. It is okay to accept additional whitespace where it makes sense. For the semantics of (ISO) EBNF, see e.g. wikipedia.
The parser should be implemented as a toplevel function called `parse` that takes a single `str` argument. If the string does not conform to the above grammar, return `None`. Assuming `expr` is a string with a valid expression, the following should hold about `x = parse(expr)`:

- an `x.is_foo()` method is provided for each of the major non-terminals: `compound`, `atom`, `literal`, `bool`, `number`, `string` and `identifier` (e.g. there should be an `is_atom()` method), returning a boolean,
- if `x.is_compound()` is true, `len(x)` should be a valid expression and `x` should be iterable, yielding sub-expressions as objects which also implement this same interface,
- if `x.is_bool()` is true, `bool(x)` should work,
- if `x.is_number()` is true, basic arithmetic (`+`, `-`, `*`, `/`) and relational (`<`, `>`, `==`, `!=`) operators should work (e.g. `x < 7`, or `x * x`) as well as `int(x)` and `float(x)`,
- `x == parse(expr)` should be true (i.e. equality should be extensional),
- `x == parse(str(x))` should also hold.

If a numeric literal `x` with a decimal dot is directly converted to an `int`, this should behave the same as int( float( x ) ). A few examples of valid inputs (one per line):

```
(+ 1 2 3)
(eq? [quote a b c] (quote a c b))
12.7
(concat "abc" "efg" "ugly \"string\"")
(set! var ((stuff) #t #f))
(< #t #t)
```

Note that str(parse(expr)) == expr does not need to hold. Instead, str should always give a canonical representation, e.g. this must hold:

```
str( parse( '+7' ) ) == str( parse( '7' ) )
```

## Part S.3.b: squelter

In this task, we will add persistence to the Shelter class from the previous installment (s1_d_shelter). You should provide 2 new functions, load and store. The basic requirement is that doing a store → load → store sequence will produce two identical copies of the same data.

- Both load and store expect a db keyword argument, which takes an open sqlite3 connection.
- The load function accepts a single positional argument, an id of the Shelter snapshot to load and returns a Shelter instance.
- The store function takes a Shelter instance as its only positional argument, and returns an id (which can be then passed to load).

Please note that storing multiple Shelter instances in a single database must be possible. Moreover, each animal and human should appear in the entire database only once, even if they appear in multiple Shelter snapshots (copies) stored in that database. We consider two people or two animals the same if all their attributes match, with two exceptions:

- the max_capacity of a foster parent: the same foster parent may appear in multiple Shelter instances with a different capacity,
- the date_of_entry of an animal, which works the same way (the same animal still cannot re-enter the same shelter though).

A foster parent and an adopter with the same name and address are the same person, and should only appear in the database once. Since addresses for veterinarians are not stored, they are distinct from foster parents and adopters, even if they have the same name.
Finally, if store is called on a Shelter with the keyword argument deduplicate set to True, and a snapshot with the exact same information (i.e. the list of animals, adopters, foster parents, fostering records and vet exams) is already present, do not add anything to the database and return the id of the existing snapshot. It is okay for this check to be, in the worst case, linear in the number of snapshots already stored (but it should be still reasonably efficient, allowing a database to hold several years worth of weekly snapshots).
The database schema is up to you, subject to the constraints above. If store is called on an empty database, it should create the necessary tables.

## Part S.3.c: merkle

Implement class Merkle which provides the following methods:

- __init__( conn ) sets up the object, using conn as the database connection (you can assume that this is an sqlite3 connection),
- store( path ) stores the tree corresponding to the directory path from the filesystem into the database (see below about format) and returns its hash,
- diff_path( hash_old, path_new ) computes a recursive diff between the directory given by the hash_old stored in the database and the directory given by path_new (in the filesystem),
- diff( hash_old, hash_new ) computes a recursive diff between two directories stored in the database,
- fetch( hash, path ) creates directory path in the filesystem (it is an error if it already exists, or if anything else is in the way) and makes a copy of the tree with root directory given by hash (from the database into the filesystem), returning True on success and False on error,
- find( root_hash, node_path ) returns the hash of a node that is reached by following node_path starting from the directory given by root_hash, or None if there is no such node.

The format of the trees is as follows:

- a regular file corresponds to a leaf node, and its hash is simply the hash of its content,
- a directory node is a list of (item hash, item name) tuples; to compute its hash, sort the tuples lexicographically by name, separate the item hash from the name by a single space, and put each tuple on a separate line (each line ended by a newline character).

These are the only node types. The same node (two nodes are the same if they have the same hash) must never be stored in the database twice. The find operation must be fast even for very large directories (i.e. do not scan directories sequentially). Paths are given as strings, components separated by a single / (forward slash) character.
The recursive diff should be returned as a dict instance with file paths as its keys, where:

- a path appears in the dictionary if it appears in either of the trees being compared (except if it is in both, and the content of the associated files is the same),
- the values are Diff objects, with the following methods:
  - predicates is_new, is_removed and is_changed,
  - old_content, new_content which return a bytes object with the content of the respective file (if applicable),
  - unified which returns a str instance with a difflib-formatted unified diff (it is the responsibility of the caller to make sure the files are utf8-encoded text files).

For instance, doing diff( foo, foo ) should return an empty dict. You are encouraged to fetch the file content lazily. Diffing trees with a few hundred files each, where most files are 100MiB, should be very fast and use very little memory if we only actually read the content diff for a single small file.
The hashes are SHA-2 256 and in the API, they are always passed around as a bytes object (which contains the raw hash, 32 bytes long). When hashing directories, the hashes are written out in hex (base 16).

## Part S.3.d: numeval

Write an evaluator based on the grammar from t3/lisp.py. The basic semantic rules are as follows: the first item in a compound expression is always an identifier, and the compound expression itself is interpreted as a function application. Evaluation is eager, i.e. innermost applications are evaluated first. Literals evaluate to themselves, i.e. 3.14 becomes a real with the value 3.14. Only numeric literals are relevant in this homework, and all numeric literals represent reals (floats). Besides literals, implement the following objects (<foo>+ means 1 or more objects of type foo):

- (vector <real>+) – creates a vector with given entries
- (matrix <vector>+) – 1 vector = 1 row, starting from the top

And these operations on them:

- (+ <vector> <vector>) vector addition, returns a vector
- (dot <vector> <vector>) dot product, returns a real
- (cross <vector> <vector>) cross product, returns a vector
- (+ <matrix> <matrix>) matrix addition, returns a matrix
- (* <matrix> <matrix>) matrix multiplication, returns a matrix
- (det <matrix>) determinant, returns a real

- (solve <matrix>) solve a system of linear equations, returns a vector

For solve, the argument is a matrix of coefficients and the result is an assignment of variables – if there are multiple solutions, return a non-zero one.

| system | matrix | written as |
|---|---|---|
| x + y = 0 | 1 1 | (matrix (vector 1 1) |
| -y = 0 | 0 -1 | (vector 0 -1)) |

Expressions with argument type mismatches (both in object constructors and in operations), attempts to construct a matrix where the individual vectors (rows) are not of the same length, addition of differently-shaped matrices, multiplication of incompatible matrices, addition or dot product of different-sized vectors, and so on, should evaluate to an error object. Attempt to get a cross product of vectors with dimension other than 3 is an error. Any expression with an error as an argument is also an error.

The evaluator should be available as evaluate() and take a string for an argument. The result should be an object with methods is_real(), is_vector(), is_matrix() and is_error(). Iterating vectors gives reals and iterating matrices gives vectors. Both should also support indexing. float(x) for x.is_real() should do the right thing.

You can use numpy in this task (in addition to standard modules).

# Part K: Solution Key

## Part K.1: Week 1

### K.1.e.1 [fibfib]

```
def fibfib( n, k ):
    if n == 0:
        a = 1
        b = 1
        for i in range( k - 2 ):
            c = a + b
            a = b
            b = c
        return b
    else:
        return fibfib( 0, fibfib( n - 1, k ) )
```

### K.1.r.1 [permute]

```
def int_to_list( number, base ):
    r = []
    while number:
        r.append( number % base )
        number //= base
    return r

def unique( lists ):
    return list( set( lists ) )

def list_to_int( list_, base ):
    res = 0
    for i in range(len(list_)):
        res += list_[i] * ( base ** (len(list_)-i-1))
    return res

def permute_digits( n, b ):
    perms = list( permutations( int_to_list( n, b ) ) )
    return unique( map( lambda x : list_to_int( x, b ), perms ) )
```

### K.1.r.2 [rfence]

```
def encrypt(text, rails):
    res = ""

    for i in range( 1, rails + 1 ):
        j = 0
        res += text[ i - 1 ]
        next_i = False

        while not next_i:
            lines_until_up = None
            lines_until_down = None

            if i % rails != 0: # (==0) last row, no down
                lines_until_down = rails - i
            if i % rails != 1: # (==1) first row, no up
                lines_until_up = i - 1
            for shift in [ lines_until_down, lines_until_up ]:
                if shift is not None:
                    j += shift * 2
                    if i + j - 1 >= len( text ):
                        next_i = True
                        break
                    res += text[ i + j - 1 ]
    return res

def decrypt(text, rails):
    switches, rest = divmod( len( text ) - 1, rails - 1 )
    first_row_len = switches // 2 + 1

    rows = [ text[ 0 : first_row_len ] ]

    i = first_row_len
    while i < len( text ):
        mid_row = ""
        if len( text ) - i < switches: # last row
            rows.append( text[ i : ] )
            break
        for j in range( switches ):
            mid_row += text[ i ]
            i += 1
        if rest > 0:
            mid_row += text[ i ]
            i += 1
            rest -= 1
        rows.append( mid_row )

    res = ""
    while any( rows ):
        for i in list( range( 0, len( rows ) ) ) + \
                list( range( len( rows ) - 2, 0, -1 ) ):
            if len( rows[ i ] ) == 0:
                break
            res += rows[ i ][ 0 ]
            rows[ i ] = rows[ i ][ 1 : ]
    return res
```

### K.1.r.3 [life]

```
def updated( x, y, cells ):
    count = 0
    alive = ( x, y ) in cells

    for dx in [ -1, 0, 1 ]:
        for dy in [ -1, 0, 1 ]:
            if dx or dy:
                count += ( x + dx, y + dy ) in cells

    return count in { 2, 3 } if alive else count == 3
```

```python
def life( cells, n ):
    if n == 0:
        return cells

    todo = set()

    for x, y in cells:
        for dx in [ -1, 0, 1 ]:
            for dy in [ -1, 0, 1 ]:
                todo.add( ( x + dx, y + dy ) )

    ngen = { ( x, y ) for x, y in todo if updated( x, y, cells ) }
    return life( ngen , n - 1 )
```

### K.1.r.4 [breadth] XXX

```python
from statistics import median, mean

def breadth(tree):
    last_level = [1]
    widths = []

    while last_level:
        next_level = []
        for i in last_level:
            next_level += tree[i]

        widths.append( len( last_level ) )
        last_level = next_level

    return mean( widths ), median( widths ), max( widths )
```

### K.1.r.5 [radix]

```python
def radix_sort( strings, idx = 0 ):
    buckets = {}
    result = []
    for s in strings:
        if len( s ) > idx:
            buckets.setdefault( s[ idx ], [] ).append( s )
        else:
            result.append( s )
    for _, b in sorted( buckets.items(), key = lambda x: x[ 0 ] ):
        result.extend( radix_sort( b, idx + 1 ) )
    return result
```

### K.1.r.6 [bipartite]

```python
def is_bipartite( graph ):
    colours = {}
    queue = []

    vertices = list( set( [ x for x,_ in graph ] ) )
    for v in vertices: # can be disconnected
        if v in colours:
            continue
        queue.append( v )
        colours[ v ] = 1
        colour = 1

        while queue:
            v = queue.pop( 0 )
            colour = 2 if colours[ v ] == 1 else 1
            for neighb in [ y for x,y in graph if x == v ]:
                if neighb in colours and \
                   colours[ neighb ] != colour:
                    return False
                if neighb not in colours:
                    colours[ neighb ] = colour
                    queue.append( neighb )
    return True
```

# Part K.2: Week 2

### K.2.e.1 [geometry]

```python
class Point:
    def __init__( self, x: float, y: float ) -> None:
        self.x = x
        self.y = y

    def __sub__( self, other: 'Point' ) -> Vector:
        return Vector( self.x - other.x, self.y - other.y )

    def translated( self, vec: Vector ) -> 'Point':
        return Point( self.x + vec.x, self.y + vec.y )

class Vector:

    def __init__( self, x: float, y: float ) -> None:
        self.x = x
        self.y = y

    def __mul__( self, s: float ) -> Vector:
        return Vector( self.x * s, self.y * s )

    def length( self ) -> float:
        return sqrt( self.x * self.x + self.y * self.y )

    def dot( self, other: Vector ) -> float:
        return self.x * other.x + self.y * other.y

    def angle( self, other: Vector ) -> float:
        cos = self.dot( other ) / ( self.length() * other.length() )
        if isclose( cos, 1 ): cos = 1
        if isclose( cos, -1 ): cos = -1
        return acos( cos )

class Line:
    def __init__( self, p1: Point, p2: Point ):
        self.p1 = p1
        self.p2 = p2

    def translated( self, vec: Vector ) -> Line:
        return Line( self.p1.translated( vec ),
                     self.p2.translated( vec ) )

    def get_point( self ) -> Point:
        return self.p1

    def get_direction( self ) -> Vector:
        v_dir = self.p2 - self.p1
        return v_dir * ( 1 / v_dir.length() )

class Segment( Line ):

    def __init__( self, p1: Point, p2: Point ) -> None:
        super( Segment, self ).__init__( p1, p2 )

    def length( self ) -> float:
        return ( self.p2 - self.p1 ).length()

    def translated( self, vec: Vector ) -> Segment:
        return Segment( self.p1.translated( vec ),
                        self.p2.translated( vec ) )

    def get_endpoints( self ) -> Tuple[ Point, Point ]:
        return ( self.p1, self.p2 )

class Circle:
    def __init__( self, c: Point, r: float ) -> None:
        self.c = c
        self.r = r

    def translated( self, vec: Vector ) -> Circle:
        return Circle( self.c.translated( vec ), self.r )
```

```
    def center( self ) -> Point:
        return self.c

    def radius( self ) -> float:
        return self.r

def point_eq( p1: Point, p2: Point ) -> bool:
    return isclose( p1.x, p2.x ) and \
           isclose( p1.y, p2.y )

def dir_eq( u: Vector, v: Vector ) -> bool:
    return isclose( u.angle( v ), 0 ) or \
           isclose( u.angle( v ), pi )

def line_eq( l1: Line, l2: Line ) -> bool:
    return dir_eq( l1.get_direction(), l2.get_direction() ) and \
           ( point_eq( l1.get_point(), l2.get_point() ) or
             dir_eq( l1.get_point() - l2.get_point(),
                     l1.get_direction() ) )
```

K.2.r.1 [json]

```
def toJSON( val: Union[ JSON, int, str ] ) -> JSON:
    if isinstance( val, str ):
        return JsonStr( val )
    if isinstance( val, int ):
        return JsonInt( val )

    return val

class JsonArray:
    def __init__( self ) -> None:
        self.arr : list[ JSON ] = []
    def get( self, key: JsonKey ) -> JSON:
        assert isinstance( key, int )
        return self.arr[ key ]
    def set( self, key: int, val: Union[ JSON, int, str ] ) -> None:
        assert isinstance( key, int )
        self.arr[ key ] = toJSON( val )
    def append( self, val: JSON ) -> None:
        self.arr.append( val )

class JsonObject:
    def __init__( self ) -> None:
        self.assoc : dict[ str, JSON ] = {}

    def get( self, key: JsonKey ) -> JSON:
        return self.assoc[ str( key ) ]

    def set( self, key: JsonKey, val: Union[ JSON, int, str ] ) ->
None:
        self.assoc[ str( key ) ] = toJSON( val )

class JsonWrapper:
    def get( self, key: Union[ str, int ] ) -> JSON:
        assert False
    def set( self, key: Union[ str, int ], val: JSON ) -> None:
        assert False

class JsonInt( int, JsonWrapper ): pass
class JsonStr( str, JsonWrapper ): pass
```

K.2.r.2 [rotate]

```
T = TypeVar( 'T' )

class Tree( Generic[ T ] ):

    def __init__( self, value: T ) -> None:
        self.left  : Optional[ Tree[ T ] ] = None
        self.right : Optional[ Tree[ T ] ] = None
        self.value = value

    def rotate_left( self ) -> Any:
        assert self.left is not None
        r = self.left
        detach = r.right
        r.right = self
        self.left = detach
        return r

    def rotate_right( self ) -> Any:
        assert self.right is not None
        r = self.right
        detach = r.left
        r.left = self
        self.right = detach
        return r
```

K.2.r.4 [treap]

```
class Treap( Tree[ T ] ):
    def __init__( self, key: T, priority: int ):
        self.left  : Optional[ Treap[ T ] ] = None
        self.right : Optional[ Treap[ T ] ] = None
        self.priority = priority
        self.key = key

    def _insert( node: Optional[ Treap[ T ] ], key: T, prio: int )
-> Treap[ T ]:
        if node is None:
            return Treap( key, prio )
        else:
            return node.insert( key, prio )

    def _fix_right( self ) -> Treap[ T ]:
        assert self.right is not None

        if self.priority > self.right.priority:
            return self
        else:
            return self.rotate_right()

    def _fix_left( self ) -> Treap[ T ]:
        assert self.left is not None

        if self.priority > self.left.priority:
            return self
        else:
            return self.rotate_left()

    def insert( self, key: T, prio: int ) -> Treap[ T ]:
        if key > self.key:
            self.right = Treap._insert( self.right, key, prio )
            return self._fix_right()
        else:
            self.left = Treap._insert( self.left, key, prio )
            return self._fix_left()
```

K.2.r.5 [distance]

```
def distance_point_point( a: Point, b: Point ) -> float:
    p = a - b
    return Vector( p.x, p.y ).length()

def distance_point_line( a: Point, l: Line ) -> float:
    p1 = l.get_point()
    p2 = p1.translated( l.get_direction() )
    x1, y1, x2, y2 = p1.x, p1.y, p2.x, p2.y

    return ( abs( ( y2 - y1 ) * a.x - ( x2 - x1 ) * a.y +
                  ( x2 * y1 ) - ( y2 * x1 ) ) /
             sqrt( ( y2 - y1 ) * ( y2 - y1 ) +
                   ( x2 - x1 ) * ( x2 - x1 ) ) )

def distance_line_line( p: Line, q: Line ) -> float:
    p1 = p.get_point()
```

```
    return distance_point_line( p1, q )

def distance_point_circle( a: Point, c: Circle ) -> float:
    return abs( distance_point_point( a, c.center() ) - c.radius() )

def distance_line_circle( l: Line, c: Circle ) -> float:
    dist = distance_point_line( c.center(), l ) - c.radius()
    return 0 if dist <= 0 else dist

def distance( a: Union[ Point, Line, Circle ],
              b: Union[ Point, Line, Circle ] ) -> float:

    if type( a ) == Point and type( b ) == Point:
        return distance_point_point( a, b )
    if type( a ) == Point and type( b ) == Line:
        return distance_point_line( a, b )
    if type( a ) == Line and type( b ) == Point:
        return distance_point_line( b, a )
    if type( a ) == Line and type( b ) == Line:
        return distance_line_line( a, b )
    if type( a ) == Point and type( b ) == Circle:
        return distance_point_circle( a, b )
    if type( a ) == Circle and type( b ) == Point:
        return distance_point_circle( b, a )
    if type( a ) == Line and type( b ) == Circle:
        return distance_line_circle( a, b )
    if type( a ) == Circle and type( b ) == Line:
        return distance_line_circle( b, a )

    assert False
```

---

K.2.r.6 [istree]

```
class Tree:
    def __init__( self ) -> None:
        self.left  : Optional[ Tree ] = None
        self.right : Optional[ Tree ] = None

def is_tree_rec( root: Tree, visited: Set[ Tree ] ) -> bool:
    if root in visited:
        return False

    visited.add( root )
    result = True

    if root.left is not None:
        result = result and is_tree_rec( root.left, visited )
    if root.right is not None:
        result = result and is_tree_rec( root.right, visited )

    return result

def is_tree( root: Tree ) -> bool:
    return is_tree_rec( root, set() )
```

# Part K.3: Week 3

K.3.e.1 [counter]

```
def make_counter() -> Tuple[ Callable[ [ K ], None ], Dict[ K, int ]
]:
    ctr : Dict[ K, int ] = {}

    def fun( key: K ) -> None:
        ctr.setdefault( key, 0 )
        ctr[ key ] += 1

    return ( fun, ctr )
```

K.3.r.1 [fold]

```
T = TypeVar( 'T' )
S = TypeVar( 'S' )

def foldr( f: Callable[ [ S, T ], T ], l: Sequence[ S ], i: T ) ->
T:
    res = i
    for x in reversed( l ):
        res = f( x, res )
    return res

def fold_len( l: Sequence[ T ] ) -> int:
    return foldr( lambda x, y: y + 1, l, 0 )

def fold_pairs( l: Sequence[ T ] ) -> Sequence[ Any ]:
    return foldr( lambda x, y: (x, y), l, () )

def fold_rev( l: Sequence[ T ] ) -> List[ T ]:
    def app( x: T, y: List[ T ] ) -> List[ T ]:
        y.append( x )
        return y

    return foldr( app, l, [] )
```

K.3.r.2 [trees]

```
def fold_node( f: Callable[ [ S, T, T ], T ],
               node: Optional[ Node[ S ] ], init: T ) -> T:
    if node is None:
        return init
    return f( node.val,
              fold_node( f, node.left, init ),
              fold_node( f, node.right, init ) )

def fold( f: Callable[ [ S, T, T ], T ], tree: Tree[ S ],
          initial: T ) -> T:
    return fold_node( f, tree.root, initial )
```

K.3.r.3 [bisect]

```
def bisect( f: Callable[ [ float ], float ],
            x_1: float, x_2: float, prec: float ) -> float:

    mid = ( x_1 + x_2 ) / 2

    if abs( x_1 - x_2 ) < 2 * prec:
        return mid

    if f( mid ) * f( x_1 ) <= 0:
        return bisect( f, x_1, mid, prec )
    else:
        return bisect( f, mid, x_2, prec )
```

---

K.3.r.4 [each]

```
T = TypeVar( 'T' )
S = TypeVar( 'S', covariant = True )

class EachProto( Protocol, Generic[ S ] ):
    def each( self, __f: Callable[ [ S ], object ] ) -> None: ...

Each = Union[ Iterable[ T ], EachProto[ T ] ]

def each( f: Callable[ [ T ], object ], data: Each[ T ] ) -> None:
    if hasattr( data, "each" ):
        cast( EachProto[ T ], data ).each( f )
    else:
        for x in cast( Iterable[ T ], data ):
            f( x )

def each_len( data: Each[ T ] ) -> int:
    counter = 0
    def inc( _: T ) -> None:
        nonlocal counter
        counter += 1
    each( inc, data )
    return counter

def each_sum( data: Each[ int ] ) -> int:
```

```
    sum_ = 0
    def add( x: int ) -> None:
        nonlocal sum_
        sum_ += x
    each( add, data )
    return sum_

def each_avg( data: Each[ int ] ) -> float:
    items = 0
    sum_ = 0
    def add( x: int ) -> None:
        nonlocal items, sum_
        items += 1
        sum_ += x
    each( add, data )
    return sum_ / items

def each_median( data: Each[ int ] ) -> Optional[ int ]:
    items = []
    def add( x: int ) -> None:
        items.append( x )
    each( add, data )

    if not items:
        return None
    len_ = len( items )
    return sorted( items )[ len_ // 2 - ((len_ + 1) % 2) ]
```

## K.3.r.5 [objects]

```
def traffic_light() -> Obj:

    is_green = False
    timeout = 0

    def dispatch( what: str, *args: Any ) -> Any:
        nonlocal is_green, timeout

        if what == 'is_green':
            return is_green
        if what == 'set_green':
            is_green = True
        if what == 'set_red':
            timeout = 5
        if what == 'tick':
            if timeout > 0:
                timeout -= 1
                if timeout == 0:
                    is_green = False

    return dispatch

def button( pedestrian_light: Obj, vehicle_light: Obj ) -> Obj:

    timeout = 0
    to_green = True

    def dispatch( what: str, *args: Any ) -> Any:
        nonlocal to_green, timeout

        if what == 'push':
            vehicle_light( 'set_red' )

        if what == 'tick':
            if not vehicle_light( 'is_green' ) and \
               not pedestrian_light( 'is_green' ):

                if to_green:
                    pedestrian_light( 'set_green' )
                    timeout = 20
                else:
                    vehicle_light( 'set_green' )
                    to_green = True

            if timeout > 0:
                timeout -= 1
                if timeout == 0:
                    pedestrian_light( 'set_red' )
                    to_green = False

    return dispatch
```

# Part K.4: Week 4

## K.4.r.1 [iscan]

```
class Prefix:
```

FIXME list_in should be iterable, not list

```
    def __init__( self, list_in: List[ int ] ) -> None:
        self.slice = 0
        self.list = list_in
        self.lenlist = len( list_in )

    def __iter__( self ) -> Prefix:
        return self

    def __next__( self ) -> List[ int ]:
        slice_ = self.slice
        self.slice += 1
        if slice_ > self.lenlist:
            raise StopIteration
        return self.list[ 0 : slice_ ]

class Sum:
    def __init__( self, list_in: List[ int ] ) -> None:
        self.prefix = Prefix( list_in )
        next( self.prefix )

    def __iter__( self ) -> Sum: return self
    def __next__( self ) -> int:
        return sum( next( self.prefix ) )

def prefixes( list_in: List[ int ] ) -> Prefix:
    return Prefix( list_in )

def prefix_sum( list_in: List[ int ] ) -> Sum:
    return Sum( list_in )
```

## K.4.r.2 [gscan]

```
def suffixes( iter_in: Iterable[ int ] ) \
        -> Generator[ Iterable[ int ], None, None ]:
    list_in = list( iter_in )
    for i in range( len( list_in ), -1, -1 ):
        yield list_in[ i : ]

def suffix_sum( iter_in: Iterable[ int ] ) \
        -> Generator[ int, None, None ]:
    count = 0
    for item in reversed( list( iter_in ) ):
        count += item
        yield count
```

## K.4.r.3 [itree]

```
class TreeIter( Generic[ T ] ):

    def __init__( self, tree: Tree[ T ] ) -> None:
        self.n : Optional[ Tree[ T ] ] = tree

    def descend( self ) -> None:
        assert self.n is not None

        while self.n.left is not None:
```

```
            self.n = self.n.left

    def ascend( self ) -> None:
        assert self.n is not None

        while ( self.n.parent is not None and
                self.n == self.n.parent.right ):
            self.n = self.n.parent

        self.n = self.n.parent # coming from left

    def __iter__( self ) -> TreeIter[ T ]:
        assert self.n is not None
        i = TreeIter( self.n )
        i.descend()
        return i

    def __next__( self ) -> T:
        if self.n is None:
            raise StopIteration()

        assert self.n is not None # srsly
        v = self.n.value

        if self.n.right is not None:
            self.n = self.n.right
            self.descend()
        else:
            self.ascend()

        return v
```

---

K.4.r.4 [gtree]

```
def preorder( tree: Optional[ Tree[ T ] ] ) \
        -> Generator[ T, None, None ]:
    if tree is not None:
        yield tree.value
        yield from preorder( tree.left )
        yield from preorder( tree.right )

def inorder( tree: Optional[ Tree[ T ] ] ) \
        -> Generator[ T, None, None ]:
    if tree is not None:
        yield from inorder( tree.left )
        yield tree.value
        yield from inorder( tree.right )

def postorder( tree: Optional[ Tree[ T ] ] ) \
        -> Generator[ T, None, None ]:
    if tree is not None:
        yield from postorder( tree.left )
        yield from postorder( tree.right )
        yield tree.value
```

K.4.r.5 [dfs]

```
T = TypeVar( 'T' )

def dfs( graph: Dict[ T, List[ T ] ], initial: T ) \
        -> Generator[ T, None, None ]:
    seen : Set[ T ] = set()
    yield from dfs_rec( graph, initial, seen )

def dfs_rec( graph: Dict[ T, List[ T ] ], initial: T,
             seen: Set[ T ] ) -> Generator[ T, None, None ]:

    if initial in seen:
        return

    seen.add( initial )

    for n in graph[ initial ]:
        yield from dfs_rec( graph, n, seen )
```

```
        yield initial
```

K.4.r.6 [guided]

```
def a_star( graph: Graph[ T ], start: T ) -> Gen2[ T, int ]:
    q : PriorityQueue[ tuple[ int, T ] ] = PriorityQueue()
    q.put( ( 0, start ) )
    while not q.empty():
        prio, item = q.get()
        for succ in graph[ item ]:
            nprio = yield succ
            q.put( ( nprio, succ ) )

class cor_iter( Generic[ T, S ] ):
    def __init__( self, cor: Gen2[ T, S ] ) -> None:
        self.to_send : Optional[ S ] = None
        self.cor = cor

    def __iter__( self ) -> cor_iter[ T, S ]:
        return self

    def __next__( self ) -> T:
        if self.to_send is not None:
            return self.cor.send( self.to_send )
        else:
            return next( self.cor )

    def reply( self, v: S ) -> None:
        self.to_send = v
```

# Part K.5: Week 5

K.5.r.1 [refcnt]

```
class Heap:
    def __init__( self ) -> None:
        self.data : List[ List[ int ] ] = []
        self.refs : List[ int ] = []

    def boundcheck( self, obj_id: int, index: int ) -> bool:
        return obj_id >= 0 and \
               obj_id < len( self.data ) and \
               index < len( self.data[ obj_id ] )

    def put( self, obj_id: int ) -> None:
        if obj_id <= 0 or not self.boundcheck( obj_id, 0 ):
            return
        self.refs[ obj_id ] -= 1
        if self.refs[ obj_id ] == 0:
            for val in self.data[ obj_id ]:
                self.put( val )
            self.data[ obj_id ] = []

    def get( self, obj_id: int ) -> None:
        if self.boundcheck( obj_id, 0 ):
            self.refs[ obj_id ] += 1

    def read( self, obj_id: int, index: int ) -> Optional[int]:
        if not self.boundcheck( obj_id, index ):
            return None
        return self.data[ obj_id ][ index ]

    def write( self, obj_id: int, index: int, value: int ) -> bool:
        if not self.boundcheck( obj_id, index ):
            return False
        self.get( value )
        self.put( self.data[ obj_id ][ index ] )
        self.data[ obj_id ][ index ] = value
        return True

    def count( self ) -> int:
        return 1 + sum( 1 if x else 0 for x in self.refs )
```

```
    def make( self, size: int ) -> int:
        self.data.append( [ 0 for _ in range( size ) ] )
        self.refs.append( 0 )
        return len( self.data ) - 1
```

---

K.5.r.2 [reach]

```
class Heap:
    def __init__( self ) -> None:
        self.data : List[ List[ int ] ] = []
        self.marks : List[ bool ] = []

    def boundcheck( self, obj_id: int, index: int ) -> bool:
        return obj_id >= 0 and \
               obj_id < len( self.data ) and \
               index < len( self.data[ obj_id ] )

    def read( self, obj_id: int, index: int ) -> Optional[int]:
        if not self.boundcheck( obj_id, index ):
            return None
        return self.data[ obj_id ][ index ]

    def write( self, obj_id: int, index: int, value: int ) -> bool:
        if not self.boundcheck( obj_id, index ):
            return False
        self.data[ obj_id ][ index ] = value
        return True

    def mark( self, now: int ) -> None:
        if not self.boundcheck( now, 0 ) or self.marks[ now ]:
            return
        self.marks[ now ] = True
        for x in self.data[ now ]:
            self.mark( x )

    def count( self ) -> int:
        self.marks = [ False for _ in self.data ]
        self.mark( 0 )
        return sum( self.marks )

    def make( self, size: int ) -> int:
        self.data.append( [ 0 for _ in range( size ) ] )
        return len( self.data ) - 1
```

---

K.5.r.3 [sweep]

```
class GcHeap:
    def __init__( self ) -> None:
        self.data : List[ List[ int ] ] = []
        self.marks : List[ bool ] = []

    def boundcheck( self, obj_id: int, index: int ) -> bool:
        return obj_id >= 0 and \
               obj_id < len( self.data ) and \
               index < len( self.data[ obj_id ] )

    def read( self, obj_id: int, index: int ) -> Optional[int]:
        if not self.boundcheck( obj_id, index ):
            return None
        return self.data[ obj_id ][ index ]

    def write( self, obj_id: int, index: int, value: int ) -> bool:
        if not self.boundcheck( obj_id, index ):
            return False
        self.data[ obj_id ][ index ] = value
        return True

    def mark( self, now: int ) -> None:
        if not self.boundcheck( now, 0 ) or self.marks[ now ]:
            return
        self.marks[ now ] = True
        for x in self.data[ now ]:
            self.mark( x )

    def count( self ) -> int:
        self.marks = [ False for _ in self.data ]
        self.mark( 0 )
        return sum( self.marks )

    def collect( self ) -> None:
        self.marks = [ False for _ in self.data ]
        self.mark( 0 )
        for obj_id, live in enumerate( self.marks ):
            if not live:
                self.data[ obj_id ] = []

    def make( self, size: int ) -> int:
        self.data.append( [ 0 for _ in range( size ) ] )
        return len( self.data ) - 1
```

---

K.5.r.4 [semi]

```
class Heap:
    def __init__( self ) -> None:
        self.fro : List[ List[ int ] ] = []
        self.to  : List[ List[ int ] ] = []

    def boundcheck( self, obj_id: int, index: int ) -> bool:
        return obj_id >= 0 and \
               obj_id < len( self.to ) and \
               index < len( self.to[ obj_id ] )

    def read( self, obj_id: int, index: int ) -> Optional[int]:
        if not self.boundcheck( obj_id, index ):
            return None
        return self.to[ obj_id ][ index ]

    def write( self, obj_id: int, index: int, value: int ) -> bool:
        if not self.boundcheck( obj_id, index ):
            return False
        self.to[ obj_id ][ index ] = value
        return True

    def collect( self ) -> None:
        refmap : Dict[ int, int ] = {}
        self.fro = self.to
        self.to = []
        self.copy( 0, refmap )

    def copy( self, now: int, refmap: Dict[ int, int ] ) -> int:
        if now < 0:
            return now
        if now in refmap:
            return refmap[ now ]

        refmap[ now ] = len( self.to )
        copy : List[ int ] = []
        self.to.append( copy )

        for val in self.fro[ now ]:
            copy.append( self.copy( val, refmap ) )
        return refmap[ now ]

    def make( self, size: int ) -> int:
        self.to.append( [ 0 for _ in range( size ) ] )
        return len( self.to ) - 1
```

---

K.5.r.5 [cheney]

```
class Heap:
    def __init__( self ) -> None:
        self.fro : List[ List[ int ] ] = []
        self.to  : List[ List[ int ] ] = []
        self.scan = 0

    def boundcheck( self, obj_id: int, index: int ) -> bool:
        return obj_id >= 0 and \
```

```
            obj_id < len( self.to ) and \
            index < len( self.to[ obj_id ] )

    def read( self, obj_id: int, index: int ) -> Optional[int]:
        if not self.boundcheck( obj_id, index ):
            return None
        return self.to[ obj_id ][ index ]

    def write( self, obj_id: int, index: int, value: int ) -> bool:
        if not self.boundcheck( obj_id, index ):
            return False
        self.to[ obj_id ][ index ] = value
        return True

    def collect( self ) -> None:
        self.fro = self.to
        self.to = []

        self.scan = 0
        assert self.copy( 0 ) == 0

        while self.scan < len( self.to ):
            o = self.to[ self.scan ]
            for i in range( len( o ) ):
                o[ i ] = self.copy( o[ i ] )
            self.scan += 1

        print( self.to )

    def copy( self, ref: int ) -> int:
        if ref < 0:
            return ref

        nref = self.fro[ ref ][ 0 ] - len( self.fro )

        if nref < 0:
            nref = len( self.to )
            self.to.append( self.fro[ ref ].copy() )
            self.fro[ ref ][ 0 ] = nref + len( self.fro )

        return nref

    def make( self, size: int ) -> int:
        self.to.append( [ 0 for _ in range( size ) ] )
        return len( self.to ) - 1
```

**K.5.r.6** [python]

```
class Heap:
    def __init__( self ) -> None:
        self.data : List[ List[ int ] ] = []
        self.refs : List[ int ] = []
        self.marks : List[ bool ] = []

    def boundcheck( self, obj_id: int, index: int ) -> bool:
        return obj_id >= 0 and \
            obj_id < len( self.data ) and \
            index < len( self.data[ obj_id ] )

    def read( self, obj_id: int, index: int ) -> Optional[int]:
        if not self.boundcheck( obj_id, index ):
            return None
        return self.data[ obj_id ][ index ]

    def write( self, obj_id: int, index: int, value: int ) -> bool:
        if not self.boundcheck( obj_id, index ):
            return False
        self.get( value )
        self.put( self.data[ obj_id ][ index ] )
        self.data[ obj_id ][ index ] = value
        return True

    def put( self, obj_id: int ) -> None:
        if obj_id <= 0 or not self.boundcheck( obj_id, 0 ):
            return
        self.refs[ obj_id ] -= 1
        if self.refs[ obj_id ] == 0:
            for val in self.data[ obj_id ]:
                self.put( val )
            self.data[ obj_id ] = []

    def get( self, obj_id: int ) -> None:
        if self.boundcheck( obj_id, 0 ):
            self.refs[ obj_id ] += 1

    def mark( self, now: int ) -> None:
        if not self.boundcheck( now, 0 ) or self.marks[ now ]:
            return
        self.marks[ now ] = True
        for x in self.data[ now ]:
            self.mark( x )

    def collect( self ) -> None:
        self.marks = [ False for _ in self.data ]
        self.mark( 0 )
        for obj_id, live in enumerate( self.marks ):
            if not live:
                self.data[ obj_id ] = []

    def make( self, size: int ) -> int:
        self.data.append( [ 0 for _ in range( size ) ] )
        self.refs.append( 0 )
        return len( self.data ) - 1
```

# Part K.6: Week 6

**K.6.r.1** [trace]

```
T = TypeVar( 'T' )

class traced( Generic[ T ] ):
    indent = 0
    counter = 0

    def __init__( self, f: Callable[ ..., T ] ) -> None:
        self.f = f

    def __call__( self, *args: Any, **kwargs: Any ) -> T:
        traced.counter += 1
        cnt = traced.counter
        if cnt > 1:
            print()
        print( ' ' * traced.indent, self.f.__name__, list( args ),
end = '' )
        print( kwargs if len(kwargs) else '', end = '' )
        traced.indent += 2
        r = self.f( *args, **kwargs )
        traced.indent -= 2
        if cnt != traced.counter:
            print( '\n' + ' ' * traced.indent, "returned", r )
        else:
            print( ' ->', r, end = '' )
        return r
```

**K.6.r.2** [profile]

```
class profile:
    stats : Dict[ str, int ] = {}

    @staticmethod
    def get() -> Dict[ str, int ]:
        return dict( profile.stats )

    def __init__( self, fun: Callable[ ..., Any ] ) -> None:
        self.fun = fun

    def __call__( self, *args: Any, **kwargs: Any ) -> Any:
        profile.stats.setdefault( self.fun.__name__, 0 )
```

```
        profile.stats[ self.fun.__name__ ] += 1
        return self.fun( *args, **kwargs )
```

K.6.r.3 [record]

```
class Data: # helper to silence <mypy>
    def __init__( self, *args: Any ) -> None: pass

def record( cls: type ) -> type:
    class rec:
        def __init__( self, *args: Any ) -> None:
            from copy import copy
            counter = 0
            for k, v in cls.__dict__.items():
                if not k.startswith( '__' ):
                    if len( args ) > counter:
                        self.__dict__[ k ] = args[ counter ]
                    else:
                        self.__dict__[ k ] = copy( v )
                    counter += 1
    return rec
```

K.6.r.4 [array]

```
T = TypeVar( 'T' )

class Array( Generic[ T ] ):

    def __init__( self, defval: T ) -> None:
        self.defval = copy( defval )
        self.data : List[ T ] = []

    def extend( self, idx: int ) -> None:
        while len( self.data ) <= idx:
            self.data.append( copy( self.defval ) )

    def __setitem__( self, idx: int, val: T ) -> None:
        self.extend( idx )
        self.data[ idx ] = val

    def __getitem__( self, idx: int ) -> T:
        self.extend( idx )
        return self.data[ idx ]
```

# Part K.8: Week 8

K.8.r.1 [sleep]

```
Task = Coroutine[ object, object, object ]

class Sched:
    def add( self, task: Task ) -> None:
        self.tasks.append( task.__await__() )

    def __init__( self ) -> None:
        self.tasks : List[ Iterator[ int ] ] = []
        self.queue : PriorityQueue[ Tuple[ float, int ] ]
        self.queue = PriorityQueue()

    class suspend:
        def __init__( self, n: int ) -> None:
            self.msec = n
        def __await__( self ) -> Generator[ int, None, None ]:
            yield self.msec

    def schedule( self, i: int ) -> None:
        try:
            task = self.tasks[ i ]
            delay = next( task )
            item = ( time() + delay / 1000, i )
            self.queue.put( item )
        except StopIteration:
            pass

    def run( self ) -> None:
        for i in range( len( self.tasks ) ):
            self.schedule( i )

        while not self.queue.empty():
            when, what = self.queue.get()
            pause = when - time()
            if pause > 0:
                sleep( pause )
            self.schedule( what )
```

K.8.r.2 [ioplex]

```
Task = Coroutine[ object, object, object ]

class IOPlex:
    def __init__( self, factory: Any ) -> None:
        self.tasks: dict[ int, Any ] = {}
        self.factory = factory
        self.reply: Optional[ str ] = None
        self.counter = 0
        self.queue: Any = PriorityQueue()

    class read:
        def __await__( self ):
            x = yield (); return x

    def schedule( self, i: int ) -> None:
        try:
            task = self.tasks[ i ]
            delay = next( task )
            item = ( time() + delay / 1000, i )
            self.queue.put( item )
        except StopIteration:
            pass

    def write( self, data: str ) -> None:
        self.reply = data

    def connect( self ) -> int:
        ident = self.counter
        self.counter += 1
        self.tasks[ ident ] = \
            self.factory( self.read, self.write ).__await__()
        next( self.tasks[ ident ] )
        return ident

    def close( self, ident: int ) -> None:
        del self.tasks[ ident ]

    def send( self, ident: int, data: str ) -> Optional[ str ]:
        if ident not in self.tasks:
            return None
        try:
            self.tasks[ ident ].send( data )
        except StopIteration:
            return None
        r, self.reply = self.reply, None
        return r
```

K.8.r.3 [search]

```
class Tree( Generic[ T ] ):
    def __init__( self, key ) -> None:
        self.left  : Optional[ Tree ] = None
        self.right : Optional[ Tree ] = None
        self.key = key

    async def search( self, key, suspend ) -> bool:
        await suspend( self.key )
        r = False

        if key == self.key:
```

```
            r = True
        if key < self.key and self.left is not None:
            r = await self.left.search( key, suspend )
        if key > self.key and self.right is not None:
            r = await self.right.search( key, suspend )

        return r
```

# Part K.9: Week 9

## K.9.r.1 [email]

```
def parse_rfc822( filename: str ) -> dict[ str, str ]:

    d = {}

    with open( filename, "r" ) as f:
        for line in f:
            parts = line.split( ": ", 1 ) # incl. the space
```

drop line endings

```
            if parts[1][-1] == '\n':
                parts[1] = parts[1][:-1]

            d[parts[0]] = parts[1]
    return d
```

## K.9.r.2 [toml]

```
class ParseTOML:
    def __init__( self, toml: str ) -> None:
        self.text = toml
        self.idx = 0
        self.sec : Section = {}
        self.key = ''
        self.parsed : TOML = {}
        self.error = False
        self.top()

    def eof( self ) -> bool:
        return self.error or self.idx >= len( self.text )

    def peek( self ) -> str:
        if self.eof():
            return ''
        else:
            return self.text[ self.idx ]

    def shift( self ) -> str:
        x = self.peek()
        self.idx += 1
        return x

    def require( self, x: str ) -> None:
        if self.shift() != x:
            self.error = True

    def top( self ) -> None:
        while not self.eof():
            self.line()

    def line( self ) -> None:
        if self.peek() == '[':
            self.header()
        else:
            self.kvpair()

        self.require( '\n' )

    def header( self ) -> None:
        self.parsed[ self.key ] = self.sec
        self.sec = {}
        self.require( '[' )
        self.key = self.word()
        self.require( ']' )

    def kvpair( self ) -> None:
        k = self.word()
        self.require( '=' )
        v = self.word()
        self.sec[ k ] = v

    def word( self ) -> str:
        x = self.shift()

        if not x.isalpha():
            self.error = True

        while self.peek().isalnum():
            x += self.shift()

        return x

    def get( self ) -> Optional[ TOML ]:
        self.parsed[ self.key ] = self.sec
        return None if self.error else self.parsed

def parse_toml( toml: str ) -> Optional[ TOML ]:
    return ParseTOML( toml ).get()
```

## K.9.r.3 [resolv]

```
class Validate:
    def __init__( self, text: str ) -> None:
        self.text = text
        self.idx = 0
        self.error = False
        self.top()

    def eof( self ) -> bool:
        return self.error or self.idx >= len( self.text )

    def peek( self ) -> str:
        if self.eof():
            return ''
        else:
            return self.text[ self.idx ]

    def shift( self ) -> str:
        x = self.peek()
        self.idx += 1
        return x

    def require( self, x: str ) -> None:
        check = self.text[ self.idx : self.idx + len( x ) ]
        if check != x:
            self.error = True
        self.idx += len( x )

    def top( self ) -> None:
        while not self.eof():
            self.stmt()

    def stmt( self ) -> None:
        if self.peek() == 'n':
            self.server()
            self.spaces()
        if self.peek() == '\n':
            self.shift()
        else:
            self.comment()

    def comment( self ) -> None:
        self.spaces()
        self.require( '#' )
        while not self.eof() and self.peek() != '\n':
```

```python
            self.shift()
        self.require( '\n' )

    def spaces( self, req: bool = False ) -> bool:
        if not self.peek().isspace():
            if req:
                self.error = True
            return False

        while self.peek().isspace() and self.peek() != '\n':
            self.shift()
        return True

    def server( self ) -> None:
        self.require( 'nameserver' )
        self.spaces( True )
        self.address()

    def address( self ) -> None:
        self.num()
        for i in range( 3 ):
            self.require( '.' )
            self.num()

    def num( self ) -> None:
        x = self.shift()

        if x == '0':
            return

        if not x.isdecimal():
            self.error = True
        while self.peek().isdecimal():
            self.shift()

    def ok( self ) -> bool:
        return not self.error

def resolv_valid( text: str ) -> bool:
    return Validate( text ).ok()
```

K.9.r.4 [fstab]

```python
def read_fs( line: str ) -> FS:
    items = line.split()
    dev    = items[ 0 ]
    path   = items[ 1 ]
    fstype = items[ 2 ]
    opts   = items[ 3 ].split( ',' )
    freq   = int( items[ 4 ] ) if len( items ) > 4 else 0
    fsck   = int( items[ 5 ] ) if len( items ) > 5 else 0

    return dev, path, fstype, opts, freq, fsck

def read_fstab( path: str ) -> list[ FS ]:
    comment = re.compile( '#.*' )
    ws = re.compile( '\s*' )
    res : list[ FS ] = []

    with open( path, 'r' ) as f:
        for line in f:
            line = comment.sub( '', line )
            if ws.fullmatch( line ):
                continue
            res.append( read_fs( line ) )

    return res
```

K.9.r.6 [cpp]

```python
def cpp( path: str ) -> str:
    defined = set()
    out = ''
    emit : list[ bool ] = []

    def process( line: str ) -> None:
        if line.startswith( '#ifdef' ):
            cmd, macro = line.split()
            emit.append( macro in defined )
        if line.startswith( '#endif' ):
            emit.pop()

        if not emit or emit[ -1 ]:
            if line.startswith( '#define' ):
                cmd, macro = line.split()
                defined.add( macro )
            if line.startswith( '#undef' ):
                cmd, macro = line.split()
                defined.remove( macro )
            if line.startswith( '#include' ):
                cmd, path = line.split()
                read( path[ 1: -1 ] )

    def read( path: str ) -> None:
        nonlocal out
        with open( path, 'r' ) as f:
            for line in f:
                if line[ 0 ] == '#':
                    process( line )
                else:
                    if not emit or emit[ -1 ]:
                        out += line

    read( path )
    return out
```

# Part K.10: Week 10

K.10.r.1 [schema]

```python
def create_tables( schema: str, db: Connection ) -> None:
    tabs = json.load( open( schema ) )
    for name, cols in tabs.items():
        cdesc = ', '.join( f'{c} {t}' for c, t in cols.items() )
        db.execute( f'create table {name} ( {cdesc} )' )
```

K.10.r.2 [upgrade]

```python
def upgrade_tables( schema: str, db: Connection ) -> None:
    tabs = json.load( open( schema ) )
    for tab, cols in tabs.items():
        cdesc = ', '.join( f'{c} {t}' for c, t in cols.items() )
        cmd = f'create table if not exists {tab} ( {cdesc} )'
        db.execute( cmd )

        for c, t in cols.items():
            cmd = f'alter table {tab} add column {c} {t}'
            try:
                db.execute( cmd )
            except OperationalError as e:
                if not str( e ).startswith( 'duplicate column' ):
                    raise
```

K.10.r.3 [pkgs]

```python
def list_packages( db: Connection ) -> Cursor:
    return db.execute( 'select name, count( number ) from ' + \
                       'package left join version ' + \
                       'on package_id = package.id group by name' )

def list_leaves( db: Connection ) -> Cursor:
    return db.execute( 'select name, number from ' + \
                       'package join version ' + \
                       'on package_id = package.id ' + \
                       'where version.id not in ' + \
                       '( select depends_on from depends )' )
```

```
def sum_depends( db: Connection ) -> Cursor:
    return db.execute( 'select name, number, ' + \
                       '( select count(*) from depends where ' + \
                       '  depends_on = version.id ) from ' + \
                       'package join version ' + \
                       'on package_id = package.id' )
```

# Part K.11: Week 11

### K.11.r.1 [sleep]

```
async def cor1() -> None:
    for i in range( 5 ):
        await asyncio.sleep( 0.7 )
        print( "cor1" )

async def cor2() -> None:
    for i in range( 5 ):
        await asyncio.sleep( 1 )
        print( "cor2" )

async def sleepy() -> None:
    await asyncio.gather( cor1(),
                          cor2() )
```

### K.11.r.2 [counter]

```
async def counters( queue: asyncio.Queue[ list[ int ] ],
                    sleeps: list[ float ], iterations: int ) ->
None:
    ctr = [ 0 for _ in sleeps ]
    proc = [ await asyncio.create_subprocess_shell( f"while true; do echo .; sleep {i}; done",
                                                    stdin=PIPE,
                                                    stdout=PIPE )
                                                    for i in sleeps
    ]

    async def monitor( idx : int ) -> None:
        out = proc[ idx ].stdout
        assert out is not None
        async for l in out:
            assert l == b".\n"
            ctr[ idx ] += 1

    async def printer() -> None:
        await asyncio.sleep( 1 )
        for i in range( iterations ):
            await queue.put( ctr )
            await asyncio.sleep( 1 )
        for p in proc:
            p.kill()
            await p.wait()

    await asyncio.gather( printer(), *[ monitor( i ) for i in
        range(len(sleeps)) ] )
```

### K.11.r.3 [pipeline]

```
def chunker( limit: int ) -> Any:

    async def process( q_in: asyncio.Queue[ Any ],
                       q_out: asyncio.Queue[ Any ] ) -> None:

        s = ''
        while True:
            item = await q_in.get()
            print( "{}, retrieved {}".format( limit, item ) )

            if item is None:
                while s:
                    if len( s ) <= limit:
                        await q_out.put( str( s ) )
                        s = ''
                    else:
                        await q_out.put( str( s[:limit] ) )
                        s = s[ limit : ]
                break

            s += item
            if len( s ) < limit:
                continue
            if len( s ) <= limit:
                await q_out.put( str( s ) )
                s = ''
                continue
            else:
                await q_out.put( str( s[ :limit ] ) )
                s = s[ limit : ]
                continue

            await q_out.put( s )

        await q_out.put( None )

    return process
```

### K.11.r.5 [minilisp]

```
async def minilisp( reader: asyncio.StreamReader ) -> Any:
    stack : list[ Any ] = []
    token = b''

    def shift() -> None:
        nonlocal token
        if token:
            stack[ -1 ].append( token.decode() )
        token = b''

    while True:
        byte = await reader.readexactly( 1 )
        if byte == b'(':
            shift()
            stack.append( [] )
        elif byte == b')':
            shift()
            x = stack.pop()
            if stack:
                stack[ -1 ].append( x )
            else:
                return x
        elif byte.isspace():
            shift()
        else:
            token += byte
```

### K.11.r.6 [rot13]

```
def rotate_13( s: str ) -> str:
    ss = ''
    def f( c: str ) -> str:
        return chr( ( ord( c ) + 13 - 97 ) % 26 + 97 )
    for c in s:
        ss += f( c )
    return ss

async def handle_client( reader: StreamReader,
                         writer: StreamWriter ) -> None:
    while True:
        data = await reader.read( 10 )
        if not data:
            break
        response = data.decode( 'utf8' )
        print( 'server received', response )
        msg = rotate_13( response )
```

```
        print( 'server sending', msg )
        writer.write( msg.encode( 'utf8' ) )
        await writer.drain()
    print( 'closing connection to server' )
    writer.close()

async def client( msg: str, path: str ) -> str:
    reader, writer = await asyncio.open_unix_connection( path )
    print( "client sending", msg )
    writer.write( msg.encode() )
    if msg == 'world':
        await asyncio.sleep( 1 )

    data_ = await reader.read( 10 )
    data = data_.decode()

    print( "client received", data )
    print( "closing" )
    writer.close()
    return data

async def unix_rot( path: str ) -> list[ str ]:
    server = await asyncio.start_unix_server( handle_client, path )
    data = await asyncio.gather( client( 'hello', path ), client(
'world', path ) )
    server.close()
    await server.wait_closed()
    os.unlink( path )
    return list( data )
```

# Part K.12: Week 12

**K.12.r.1** [hist]

```
def normalize( n: int, max_ : int ) -> int:
    return round( ( n / max_ ) * 25 )

def histogram( bins: List[ int ] ) -> str:
    count = Counter( bins )
    m = max( count.values() )
    for b in count:
        count[ b ] = normalize( count[ b ], m )

    i = 1
    height = 25
    s = ""
    while height > 0:
        i = 0
        for j in sorted( count.keys() ):
            while i < j:
                i += 1
                s += ' '
            if i == j and count[j] >= height:
                s += '*'
            else:
                s += ' '
            i += 1
        s += '\n'
        height -= 1
    return s
```

**K.12.r.2** [dft]

```
def dft( a: List[ float ] ) -> List[ float ]:
    return [ i for i, v in enumerate( np.abs( np.fft.rfft( a ) ) )
            if not np.isclose( v, 0 ) ]
```

**K.12.r.3** [null]

```
def null( A: NDArray[ np.float64 ] ) -> NDArray[ np.float64 ]:
    A = np.atleast_2d(A)
    u, s, vh = np.linalg.svd(A)
    tol = max(1e-13, 0)
    nnz = (s >= tol).sum()
    return vh[nnz:].conj()
```

**K.12.r.4** [frames]

```
def max_at( data: pd.DataFrame, col: str ) -> pd.DataFrame:
    return data[data[col] == data[col].max()]

def best( data: pd.DataFrame ) -> pd.DataFrame:
    d = max_at( data, 'weekly' )
    e = max_at( data, 'assignments' )
    f = max_at( data, 'reviews' )
    return cast( pd.DataFrame, d.combine_first( e ).combine_first( f
) )

def get_total( data: pd.DataFrame ):
    weekly_min = data['weekly'].apply( lambda x: min( x, 9 ) )
    return weekly_min + data['assignments'] + data['reviews']

def add_total( data: pd.DataFrame ) -> pd.DataFrame:
    return data.assign( total = get_total )

def compute_total( data: pd.DataFrame ) -> pd.DataFrame:
    tot = add_total( data )
    return tot[ [ 'student', 'total' ] ]

def compute_averages( data: pd.DataFrame ) -> Dict[ str, float ]:
    data = add_total( data )
    cols = data[ [ 'weekly', 'assignments', 'reviews' ] ]
    return dict( cols.mean() )
```

**K.12.r.5** [regress]

```
Data = List[ Tuple[ float, float ] ]

def drop_outliers( data: Data, cutoff: float ) -> Data:
    x_ = [ x for x,_ in data ]
    y_ = [ y for _,y in data ]
    p = np.polyfit( x_, y_, 1 )

    idx_max = 0
    max_dist = 0
    sum_dist = 0
    for i in range( len( y_ ) ):
        dist = ( y_[ i ] - ( p[ 0 ] * x_[ i ] + p[ 1 ] ) ) ** 2
        if dist > max_dist:
            idx_max = i
            max_dist = dist
        sum_dist += dist

    if max_dist > ( sum_dist * cutoff ):
        x_.pop( idx_max )
        y_.pop( idx_max )
        return drop_outliers( list( zip( x_, y_ ) ), cutoff )

    return list( zip( x_, y_ ) )

def regress( data: Data, cutoff: float ) -> Tuple[ float, float ]:
    data = drop_outliers( data, cutoff )
    x_ = [ x for x,_ in data ]
    y_ = [ y for _,y in data ]
    p = np.polyfit( x_, y_, 1 )
    return ( p[0], p[1] )
```