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 and on the student server `aisa`:

- `https://is.muni.cz/auth/el/fi/podzim2021/PV248/um/` – a PDF in `pv248.seminar.pdf` and the source bundle in directories `01` through `12`, `t1` through `t4` 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.[1] 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

Welcome to PV248 Python Seminar.
Since this is a programming subject, most of the coursework – and grading – will center around actual programming. There will be 2 types of programs that you will write in this seminar: tiny programs for weekly exercises (around 10–30 minutes each) 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 this course won't be entirely easy either. 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:

- `2_grading.txt` – what is graded and how; what you need to pass,
- `3_tasks.txt` – general guidelines that govern assignments,
- `4_reviews.txt` – writing and receiving peer reviews.

Study materials for each week are in directories `01` through `12`. Start by reading `00_intro.txt`. Assignments are in directories `t1` through `t4`, one for each 4-week block and one extra for the exam period. They will be made available according to the schedule shown in `2_grading.txt`.
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, 2 and 5), the seminar will mainly include general programming exercises. Topics will get more specific and focused as the semester progresses. The lectures are divided into 3 blocks, 4 lectures each. They will cover the following topics:

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	29.11.

Part A.2: Grading

To pass the subject, you need to meet the following criteria:

- complete 8 out of the 12 seminars, that is:
  - submit at least 3 preparatory exercises
  - attend the corresponding seminar
- complete 4 tasks selected from 3 or 4 sets
  - the tasks must come from at least 3 different sets
  - which tasks you pick is entirely up to you
- write 6 peer reviews for tasks:
  - you can only review what you passed yourself
  - you must cover 2 different tasks (with at least 2 reviews each)

The deadline for peer reviews is 13.2.2022 (the Sunday after the end of the exam period, or a week and a half after the last deadline for the last task set). Doing more work than required is always OK.

A.2.1 Seminars Each chapter in this exercise collection has 2 types of exercises: 'preparatory' and 'regular'. Completing a seminar, then, has 2 parts: working out 3 of the preparatory exercises and attending the seminar, which will include the following (group) activities:

- analysis of some of the submitted solutions,
- making improvements to the same,
- solving some of the 'regular' exercises live.

In addition to the group activities, the teacher will live-solve one of the regular exercises with comments, input and questions from you as a group.
All weekly exercises have test cases enclosed: it is sufficient to pass those test cases. Any bugs that slip by may be dissected in the following seminar. The submission deadlines for prep exercises are as follows:

[1] The exercises and tasks will be made available, at the latest, on the day before their 'start date' (see tables in the following sections). Of course, we will try to fill everything in sooner than that.

chapter	lecture	deadline	seminar
01	13.9.	18.9. 23:59	20.9. –23.9.
02	20.9.	25.9. 23:59	27.9. –30.9.
03	27.9.	2.10. 23:59	4.10.– 7.10.
04	4.10.	9.10. 23:59	11.10.–14.10.
05	11.10.	16.10. 23:59	18.10.–21.10.
06	18.10.	23.10. 23:59	25.10.–28.10.
07	25.10.	30.10. 23:59	1.11.– 4.11.
08	1.11.	6.11. 23:59	8.11.–11.11.
09	8.11.	13.11. 23:59	15.11.–18.11.
10	15.11.	20.11. 23:59	22.11.–25.11.
11	22.11.	27.11. 23:59	29.11.– 2.12.
12	29.11.	4.12. 23:59	6.12.– 9.12.

If your seminar falls on a holiday (this affects Tuesday 28.9., Thursday 28.10. and Wednesday 17.11.), you can attend with a different group that week.

A.2.2 Tasks There will be 4 sets of 3 tasks each. As mentioned earlier, you are required to complete, at minimum, 4 tasks covering at least 3 of the sets. Submitting more is of course allowed and encouraged. There are 8 deadlines for each set, summarised in the next section.
Please remember that the test suite is strictly binary: you either pass or you fail, and that the deadlines are firm. More details and guidelines are in `3_tasks.txt`.

A.2.3 Peer Review Reading and understanding code is an important skill, and even though it's not easy to practice, we are going to at least try. You will be required to read, understand and provide feedback for 6 task solutions written by your classmates. The rules for peer review are as follows:

- only tasks are eligible for reviews (not the weekly exercises),
- you can submit any code (even completely broken) for peer review,
- to write a review for any given submission, you must have already passed the respective task yourself,
- there are no interim deadlines for requesting or providing peer reviews (only the deadline at the end of the exam period).

It is okay to point out correctness problems during peer reviews, with the expectation that this might help the recipient complete the task. This is the only allowed form of cooperation (more on that below).

A.2.4 Plagiarism Copying someone else's work or letting someone else copy yours will cause that item to be crossed off your achievements, along with one more of the same kind. That is, if you 'borrow' a solution to a preparatory exercise, that week won't count toward the 8 seminars that you need to complete, and you will be required to complete 9 other seminars, instead of 8. Likewise, if you borrow a solution to a task, that task will be crossed off. You will then have to solve 4 tasks in 3 different sets to pass the subject.
You are also responsible for keeping your solutions private. If you only use the `pv248` command on `aisa`, it will make your `~/pv248` directory inaccessible to anyone else (this also applies to school-provided UNIX workstations). Keep it that way. If you work on your solution using other computers, make sure they are secure. Do not publish your solutions anywhere (on the internet or otherwise). All parties in a copying incident will be treated equally.
No cooperation is allowed (not even design-level discussion about how to solve the exercise) on tasks and on weekly exercises which you submit. If you want to study with your classmates, that is okay – but only cooperate on code which is not going to be submitted by either party. If you want to be sure of this, limit teamwork to the 'regular' exercises (in files called `rN_*.py`).

Part A.3: Tasks

A.3.1 Schedule There are 4 sets of tasks and each has a 4-week window when it can be handed in. The dates are as follows (both start and end are at the midnight which ends the given day):

set	start	end
T.1	24.9. 0:00	21.10. 23:59
T.2	22.10. 0:00	18.11. 23:59
T.3	19.11. 0:00	16.12. 23:59
T.4	7.1. 0:00	3.2. 23:59

A.3.2 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.
- The next step is sanity and runs every noon and midnight. 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 twice a week – every Thursday and Monday at midnight, right after the 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 (the task is still considered complete).
The deadlines for verity tests are as follows:

try	day	T.1	T.2	T.3	T.4
start	Fri	24.9.	22.10.	19.11.	7.1.
1	Mon	27.9.	25.10.	22.11.	10.1.
2	Thu	30.9.	28.10.	25.11.	13.1.
3	Mon	4.10.	1.11.	29.11.	17.1.
4	Thu	7.10.	4.11.	2.12.	20.1.
5	Mon	11.10.	8.11.	6.12.	24.1.
6	Thu	14.10.	11.11.	9.12.	27.1.
7	Mon	18.10.	15.11.	13.12.	31.1.
8	Thu	21.10.	18.11.	16.12.	3.2.

A.3.3 Submitting Solutions The easiest way to submit a solution is this:

```
$ ssh aisa.fi.muni.cz
$ cd ~/pv248/t1
<edit files until satisfied>
$ pv248 submit t1_splay
```

The number of times you submit is not limited (but not every submission will be necessarily evaluated, as explained above).
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.3.4 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.
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. The expectation is that most of the time, you will pass on the second or third attempt. 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. Only very small programs can be realistically written completely correctly in one go.

Part A.4: Peer Reviews

You must participate in the peer review system, both as a reviewer (see earlier section on grading) and as a review recipient. In addition to fulfilling the requirements of the course, we hope that the reviews you get will help you better understand how other people see your code, and perhaps also learn something new about writing code.

A.4.1 Requesting Reviews Not all submissions are worth reviewing, and only you can decide which are. Therefore, when you would like to have your code reviewed, use the following command:

```
$ pv248 review --request t1_splay
```

Substitute other tasks for t1_splay as appropriate. You can request a review on a task which you did not pass yet. You may get up to 3 reviews for any given request. The request you create is for your latest submission at the time (so make sure you submit first, ask for reviews later).
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.[2] After you read your review, you should 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 100
```

Instead of 100, you can use a smaller number, indicating a 0–100 rating of the review you received. This does not serve any other purpose except give an idea to the recipient how satisfied you were, except if you indicate 0 (in which case, we will investigate what went wrong). Please rate the reviews honestly!

A.4.2 Writing Reviews To write a review, start with the following command:

```
$ pv248 review --list
```

You will get a list of review requests for which you are an eligible reviewer. In particular, only tasks that you have already successfully solved will show up. If you like one of the entries, note its number (e.g. 7) and type:

```
$ pv248 review --checkout 7
$ cd ~/pv248/reviews/
$ ls
```

There will be a directory for each of the reviews you agreed to write. 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.
```

You can write up to 9 reviews (though you only need to do 6, but you can help out more people if you write more reviews, and get more practice while doing it). The limit of 9 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 9 slots.

[2] 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.

Part 1: Python 101

As we have mentioned, each chapter is split into 3 sections: demonstrations, preparatory exercises and regular 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. dict – using dictionaries
3. str – using strings

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 prep 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 set should be submitted by 18th of September (you need to solve at least 3 of the exercises). The corresponding seminars are in the week starting on 20th 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 set of exercises 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 listed 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

Part 1.d: Demonstrations

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 [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).
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.3 [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 variations use single or double quotes (and there is no difference between them, though some programmers give them different semantics).

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

To access a 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'
```

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. 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 good. 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.

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] = fib(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: Prep 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 (e.g. 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.

```
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 root → relaxing concoction
- 7 ethanol → elixir of life
- 4 water + 2 mandrake root + 2 valerian root + 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)

```
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.

```
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:

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

```
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 the function which, given the number of rails/rows, returns the encrypted text.

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

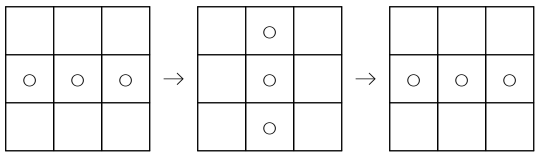
And decipher the text back to the sent message.

```
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

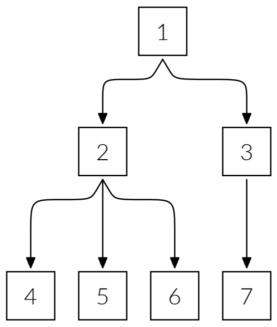
An example of a short periodic game:



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

```
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:

```
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

from math import isclose
```

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

This week, the exercises require static type annotations that can be checked with mypy --strict. In most weeks from now on, prep exercises will require mypy annotations, though they will be optional in most tasks (but you may find them helpful). Please do not use Any in the annotations, or the type: ignore pragma. While not enforced by the submission evaluator (sometimes those are hard to avoid, like the cases already present in the skeletons), over-use will be frowned upon.
Demonstrations:

1. mypy – annotation basics

Elementary exercises:

1. geometry – define basic types for planar geometry

Prep 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

Part 2.d: Demonstrations

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

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:
    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: Prep Exercises

2.p.1 [dsw] 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).

```
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: the strings substituted for their respective types.

```
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.
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,
- 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.

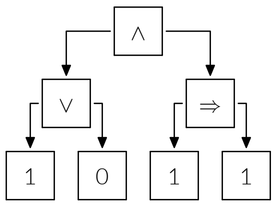
Composite rendering using #{…} is similar, but:

- 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: again, treat this as a failed precondition, fail an assert, and leave it to someone else to fix later.

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 $${…} must remain untouched. 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] 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 T.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`] 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

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

Part 3.p: Prep Exercises

3.p.1 [`merge`] 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] 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 sub-list 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 [counter] 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.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.

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

3.r.2 [trees] 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 $\operatorname{sgn}(f(x_1)) \neq \operatorname{sgn}(f(x_2))$ 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] 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

Preparatory 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: TBD.

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

Part 4.d: Demonstrations

4.d.1 [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 generator 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:

Part 4.p: Prep 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 do 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 un-

derstanding 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 ):
    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.

```
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 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.

```
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] 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.

```
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.

```
def bfs( graph: Graph[ T ], start : T ) -> Gen1[ T ]:

    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 T.1: Introductory Tasks

The programming tasks for this block are as follows:

1. `shelter` – a simple information system,
2. `splay` – a self-balancing binary search tree,
3. `while` – an interpreter for simple 'while programs'.

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' )`).
You may add `mypy` annotations to your solutions if you like, and they will be checked (without `--strict`) and the result will be shown to you as part of the `syntax` test, but any failures will be ignored.

Part T.1.1: [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 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`, where:
    - the date of entry is a `datetime.date` instance,
    - `year_of_birth` is an integer,
    - everything else is a string,

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 present in the shelter at this time (i.e. were not adopted and not in foster care) 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,
    - 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 `adopter_name` and `adopter_address` which are strings,
    - method `start_foster` which takes a `date` (again a `datetime.date` instance) and `parent`, which accepts one of the objects returned by `available_foster_parents` listed below,
    - `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`,
- `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).

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 at the given date,
- attempting to adopt an animal that was in foster care at the time, or attempting to put an animal that has already been adopted into foster care,
- attempting to do a veterinary exam on an animal which is in foster care or already adopted at the time,
- an attempt is made to exceed the capacity of a foster parent.

Part T.1.2: [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/ssyj4d 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.

EDIT 2021-09-27: Clarified that insert should not add duplicate keys.

Part T.1.3: [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

```
name₀ = operation name₁ name₂
```

Valid operations are:

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

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 = mod 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 5: Memory management, reference counting

Prep 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

Part 5.p: Prep 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 ) -> 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 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.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
```

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
```

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

Prep 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

Part 6.p: Prep 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.
Examples:

```
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

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 [class], such as [5] 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. Use Callable[..., Any] and/or Any as appropriate.

```
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). Ont 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).
Type annotations: You can make Field a Generic and with some fiddling, make the types sort of work (may need a cast in __get__). Use of Any in this exercise won't be held against you.

```
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).
Prep exercises:

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

Regular exercises:

1. xxx
2. xxx
3. xxx
4. xxx
5. xxx
6. xxx

Part 7.p: Prep 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·(b + c) = a·b + a·c
   - bilinearity a·(rb + c) = r(a·b) + (a·c)
   - compatibility with scalar multiplication: (ra)·(rb) = rr(a·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 × b = a × rb = r(a × b)
- Jacobi identity: a × (b × c) + b × (c × a) + c × (a × 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] 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.

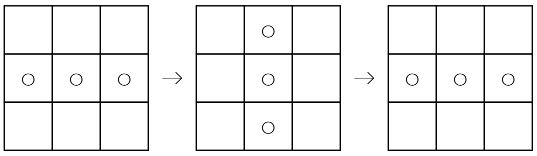
```
def check_sift( sift ): pass
```

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 implementation 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] 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

Prep 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. xxx
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. xxx
5. xxx
6. xxx

Part 8.p: Prep 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 `__await__()`) of 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', 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.5 [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.

```
async def sort( data: Array, suspend: Suspend ) -> None: pass
```

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 T.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 is focused on the use of semi-coroutines (generators).

1. `rec` – recursive programs
2. `ptr` – pointers and garbage collection
3. `rst` – real-time splay trees

Part T.2.1: [rec]

Implement an interpreter for simple recursive programs. The following syntax is taken unchanged from `t1_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 `t1_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. use of an undefined function or bad argument count,
3. runtime errors (division by zero).

Part T.2.2: [ptr]

In this task, you will extend `t1_while` with pointers and garbage collection. The syntax is unchanged, except for addition of 3 new operations:

- `addr_ = set addr val` stores the value in variable `val` 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 `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 may execute an arbitrary sequence of operations, even if it doesn't actually appear in the program).
Only addresses (which are distinct from numbers) can be passed to `get` and `set` as their first argument. Addresses are created by `alloc`. Adding a number and an address results in an address if 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 result of any other operation is a number. Addresses always evaluate as `true` in `while` or `if`.
The numeric values of addresses are unspecified, except that addresses of different objects always compare unequal, and addresses within the same object compare reasonably (higher offsets are greater).
New semantic errors (compared to `t1_while`):

- 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 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 a result.

Part T.2.3: [rst]

This task is based on the splay tree from t1_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[3] 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.)

[3] 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.

Part 9: Text, JSON

Prep 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

Part 9.p: Prep 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.

```
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.

```
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' }

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.

```
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.

```
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' ? ;

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.

```
FS = Tuple[ str, str, str, List[ str ], int, int ]

def read_fstab( path: str ) -> List[ FS ]:
    pass
```

9.r.5 [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.

```
def cpp( filename: str ) -> str:
    pass
```

Part 10: Databases

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
5. xxx
6. xxx

Part 10.p: Prep 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
```

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.

```
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 Shop( list_create.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 Item( list_create.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 ShoppingList( list_create.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] 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 Shop( list_search.Shop ):

    def update( self ):
        pass

    def delete( self ):
        pass

class Item( list_search.Item ):

    def update( self ):
        pass

    def delete( self ):
        pass

class ShoppingList( list_search.ShoppingList ):

    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: ShoppingList, 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).

```python
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.

```python
def sum_depends( db: Connection ) -> List[ Tuple[ str, str, int ] ]:
    pass
```

Part 11: Asynchronous Programming

Prep 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` – TBD another stream pipeline exercise
5. `minilisp` – an asynchronous parser
6. `rot13` – listening on UNIX domain sockets

Part 11.p: Prep 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.)

```python
import asyncio
import time

reached = []
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.

```python
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.

```python
import asyncio
from asyncio.subprocess import PIPE
from typing import Tuple

data = []

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.

```python
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, msg ):
    pass # print( "client sending", ... )
    pass # print( "client received", ... )
```

Start the server and the two clients, gather the data back from the two clients into a list, return this list; starting with the hello client. Use the provided port.

```
async def start( port ):
    pass
```

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
```

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
```

11.r.4 [tokenize] Nothing here. Perhaps 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
```

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

This chapter is work in progress! You have been warned. Submission for this chapter starts on 2021-11-29.
Prep 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

Part 12.p: Prep 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
```

What exactly constitutes an outlier is somewhat domain- and dataset-specific, but using some small integer multiple (3-5) of σ (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 )`).

```
def cmp_outliers( 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 typing import cast, List

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
```

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 T.3: Persistence

This task set is centered around persistent data. There are two database-focused tasks (one of them with objects, another with generators again), while the third is a small parsing project.

1. lisp – a simple context-free parser
2. squelter – storing the shelter objects with SQL
3. merkle – persistent trees

Part T.3.1: [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 T.3.2: [squelter]

In this task, we will add persistence to the Shelter class from the previous installment (t1_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 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).

Beware! This is a departure from the semantics required in t1_shelter, where it was possible to have multiple animals with identical attribute sets. For this assignment, you will need to modify add_animal to return an existing object if all attributes match (again with the exception of date_of_entry: if all other attributes match but not date_of_entry, raise a RuntimeError).

Same with foster parents and max_capacity. A foster parent and an adopter with the same name and address are, however, distinct entities and should be allowed. In this case, the name and address will repeat in the database (once as a foster parent and once as an adopter).

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.

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 T.3.3: [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).

Part T.4: Asynchronous Programming, Math

In the final set, there are 2 tasks focused on programming with asyncio and one that extends t3/lisp.py into a simple numeric language based on numpy.

1. chat – a line-based chat server
2. reversi – an HTTP+JSON game server
3. numeval – a simple expression language for numeric computation

Part T.4.1: [chat]

Write a simple chat server that accepts connections on a UNIX domain socket, which it should create in its working directory with the name chatsock. It should implement a simple line-oriented protocol with the command types described below. In the direction from client to server:

```
nick <nickname>
join <channel>
message <channel> <text>
part <channel>
replay <channel> <unix timestamp>
```

And in the opposite direction:

```
ok <text>
error <text>
message <channel> <unix timestamp> <nickname> <text>
```

Both nicknames and channel names are alphanumeric strings without spaces, the latter must start with a # while the former must not start with a #. Neither channel names nor nicknames can start with an asterisk (*). Unix timestamp is an unsigned integer denoting the number of seconds elapsed since midnight, 1.1.1970. <text> is any utf-8 string with no newlines in it. The server should respond to unknown or malformed commands with an error.

Upon connecting, the client sends the nick command. The server accepts it and replies ok unless the nickname is already in use by another connected client, in which case it replies error. It is also an error to send any other command before nick has been accepted. If the client sends a nick command later, that will change the nick of the connected client.

The join command associates the client which sent the command with the given channel. Joining an already-joined channel is an 'error'. Channels are transparently created when the first client joins them, and are never destroyed. The part command removes the association of the client with the channel. It is an error to part a channel that is not joined. Both confirm the action with an ok if they succeed.

When a client joins an existing channel, the server announces this fact to all other clients already present in the channel. Likewise, when a client changes its nickname while joined to a channel, the server no-

tifies each such channel about the change. In both cases, the `nickname` of the `message` will be `*server*`, and its `text` will be `nick has joined the channel` and `oldnick is now known as newnick` respectively.
The `message` command (when sent by a client) causes the given message to be broadcast to all users associated with the given channel, including the originator. There is no other reply from the server. However, if the target channel does not exist or the client is not currently associated with it, the server replies with an `error` and does not broadcast the message. The `nickname` in the `message` command broadcast by the server is that of the originator of the message, and the `timestamp` marks the moment when the server received the command from the sending client.
Finally, the `replay` command causes the server to re-send all `messages` that the given channel received since the given timestamp (i.e. with a `timestamp` that is greater or equal to the one given by the client) to the sender of the command. The server gives `ok` before resending any messages if the request was valid (even in the case that there are no matching messages). The command results in an `error` (and no message replay) in case the client is not associated with the channel, or if the `timestamp` is in the future according to the server clock. The replay includes both server messages (with nick `*server*`) and messages sent by the client requesting the replay.
If the client sends multiple commands without waiting for replies, the replies from the server should come in the order of the commands received. Any `message` commands going from the server to the client are, however, asynchronous to the rest of the command stream, including `message` commands generated in response to a `replay`. That is, a `message` may appear after the client sent a command, but before the server sends the corresponding `ok` or `error`.
The server must remain responsive at all times and to all clients, including during playback of long message histories in response to a `replay` command. It is strongly encouraged that you program the server in an async style, using the `asyncio` library.

Part T.4.2: [`reversi`]

TBD.

Part T.4.3: [`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 S: Exercise Solutions

Part S.1: Python 101

S.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 ) )
```

S.1.r.1 [`permute`]

```
from math import log,floor
from itertools import permutations, dropwhile

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 ) )
```

S.1.r.2 [rfence]

```python
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
```

S.1.r.3 [life]

```python
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

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 )
```

S.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 )
```

S.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
```

S.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 S.2: Objects, Classes and Types

S.2.e.1 [geometry]

```
from __future__ import annotations
from typing import Tuple
from math import sqrt, acos
from math import isclose, pi

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() ) )
```

S.2.r.1 [json]

```
from r1_json import JSON, Union, JsonKey

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
```

S.2.r.2 [rotate]

```
from __future__ import annotations
from typing import TypeVar, Generic, Optional

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 ) -> Tree[ 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 ) -> Tree[ T ]:
        assert self.right is not None
        r = self.right
        detach = r.left
        r.left = self
        self.right = detach
        return r
```

S.2.r.4 [treap]

```
from __future__ import annotations
from typing import TypeVar, Optional, Generic, TYPE_CHECKING

if TYPE_CHECKING:
    S = TypeVar( 'S' )
    class Tree:
        def rotate_left( self: S ) -> S: return self
        def rotate_right( self: S ) -> S: return self

else:
    Tree = __import__( 'sol_import' ).do( 'r2_rotate' ).Tree
    T = TypeVar( 'T' )

class Treap( Tree, Generic[ 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()
```

S.2.r.5 [distance]

```
x = __import__( 'sol_import' ).do( 'e1_geometry' )
from typing import TYPE_CHECKING, Union
from math import isclose
from math import sqrt,acos

if not TYPE_CHECKING:
    Point = x.Point
    Vector = x.Vector
    Line = x.Line
    Segment = x.Segment
    Circle = x.Circle

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
```

S.2.r.6 [istree]

```
from typing import Set, Optional

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 S.3: Lexical Closures

S.3.r.1 [fold]

```
from typing import Callable, TypeVar, Sequence, Any, Tuple, List
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, [] )
```

S.3.r.2 [trees]

```
from r2_trees import *

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 )
```

S.3.r.3 [bisect]

```
from typing import Callable

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 )
```

S.3.r.4 [each]

```
from math import floor
from typing import Generic, Protocol, Callable, TypeVar, Union, \
                   Iterable, cast, Optional, Any

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) ]
```

S.3.r.5 [objects]

```
from r5_objects import *

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 S.4: Iterators, Coroutines

S.4.r.1 [iscan]

```
from __future__ import annotations
import types
from typing import List, Generator

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 )
```

S.4.r.2 [gscan]

```
from __future__ import annotations
import types
from typing import Generator, Iterable

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
```

S.4.r.3 [itree]

```
from __future__ import annotations
from r3_itree import Generic, T, Tree

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
```

S.4.r.4 [gtree]

```
from r4_gtree import Tree, Generator, T, Optional

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
```

S.4.r.5 [dfs]

```
from typing import Dict, List, TypeVar, Set, Generator
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
```

S.4.r.6 [guided]

```
from __future__ import annotations
from r6_guided import Graph, S, T, Gen1, Gen2, Generic, Optional
from queue import PriorityQueue
from typing import Tuple, TYPE_CHECKING

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

if TYPE_CHECKING:
    def bfs( graph: Graph[ T ], start : T ) -> Gen1[ T ]:
        pass
```

Part S.5: Memory management, reference counting

S.5.r.1 [refcnt]

```
from typing import Optional, List

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
```

S.5.r.2 [reach]

```
from typing import Optional, List

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
```

S.5.r.3 [sweep]

```
from typing import Optional, List

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 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
```

S.5.r.4 [semi]

```
from typing import Optional, List, Dict, Set

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
```

S.5.r.5 [cheney]

```
from typing import Optional, List, Dict, Set

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
```

S.5.r.6 [python]

```
from typing import Optional, List, Set

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 S.6: Objects 2

S.6.r.1 [trace]

```
from typing import Any, Callable, TypeVar, Generic

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
```

S.6.r.2 [profile]

```
from typing import Dict, Any, Callable

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 )
```

S.6.r.3 [record]

```
from typing import Callable, Any, Protocol

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
```

S.6.r.4 [array]

```
from typing import TypeVar, Generic, List
from copy import copy
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 S.8: Coroutines 2

S.8.r.1 [sleep]

```
from typing import Coroutine, Generator, Iterator, List, Tuple
from queue import PriorityQueue
from time import time, 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 )
```

S.8.r.2 [ioplex]

```
from typing import Coroutine, Generator, Iterator, List, Tuple
from queue import PriorityQueue
from time import time, sleep

Task = Coroutine[ object, object, object ]

class IOPlex:
    def __init__( self, factory ) -> None:
        self.tasks = {}
        self.factory = factory
        self.reply = None
        self.counter = 0

    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 ):
        self.reply = data

    def connect( self ):
        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 ):
        del self.tasks[ ident ]

    def send( self, ident, data ):
        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
```

S.8.r.3 [search]

```
from typing import TypeVar, Generic

T = TypeVar( 'T' )

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 S.9: Text, JSON

S.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 (for your convenience, the above example can be also found in the file `rfc822.txt`). 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.

```
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
```

S.9.r.2 [toml]

```
from typing import Dict, Optional

Section = Dict[ str, str ]
TOML = Dict[ str, Section ]

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()
```

S.9.r.3 [resolv]

```
from typing import Dict, Optional

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':
            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()
```

S.9.r.4 [fstab]

```
import re

FS = tuple[ str, str, str, list[ str ], int, int ]

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
```

S.9.r.6 [cpp]

```
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 S.10: Databases

S.10.r.1 [schema]

```
import json
from sqlite3 import Connection, OperationalError, connect

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} )' )
```

S.10.r.2 [upgrade]

```
import json
from sqlite3 import Connection, OperationalError, connect

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
```

S.10.r.3 [pkgs]

```
from sqlite3 import Connection, connect, Cursor

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 S.11: Asynchronous Programming

S.11.r.1 [sleep]

```
import asyncio

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() )
```

S.11.r.2 [counter]

```
from asyncio.subprocess import PIPE
import asyncio
from typing import List

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)) ] )
```

S.11.r.3 [pipeline]

```
import asyncio
from typing import Any, List, Optional

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
```

S.11.r.5 [minilisp]

```
import asyncio
from typing import Any, List

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
```

S.11.r.6 [rot13]

```
import asyncio
import os
from asyncio import StreamReader, StreamWriter

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 S.12: Math and Statistics

S.12.r.1 [hist]

```
from collections import Counter
from typing import List

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
```

S.12.r.2 [dft]

```
import numpy as np
from typing import List

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 ) ]
```

S.12.r.3 [null]

```
import numpy as np
from typing import cast, List

def null( A: np.ndarray ) -> np.ndarray:
    A = np.atleast_2d(A)
    u, s, vh = np.linalg.svd(A)
    tol = max(1e-13, 0)
    nnz = (s >= tol).sum()
    return cast( np.ndarray, vh[nnz:].conj() )
```

S.12.r.4 [frames]

```
import pandas as pd
from typing import Any, cast, Union, Dict

def max_at( data: pd.DataFrame, col: str ) -> Any:
    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 ) -> pd.Series:
    x = data['weekly'].apply( lambda x: min( x, 9 ) ) + data['assignments']
+ data['reviews']
    assert isinstance( x, pd.Series )
    return x

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 ]:
    return dict( add_total( data ).mean() )
```

S.12.r.5 [regress]

```
import numpy as np
from typing import List, Tuple

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] )
```