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.
Each chapter corresponds to a single week of the semester. The correspondence between exercises and the content of the lectures is somewhat loose, especially at the start of the semester. The course assumes that you are intuitively familiar with common programming concepts like classes, objects, higher-order functions and function closures (which can be stored in variables). However, you do not need a detailed theoretical understanding of the concepts.
NB. The exercise part of this document may be incomplete. Please always refer to the source files that you obtained via `pv248 update` on `aisa` as the authoritative version. We are working on also improving this document, please be patient.

Part A.1: Course Overview

Welcome to PV248 Programming in Python. In a normal semester, the course consists of lectures, seminars and assignments. This is not a normal semester: the course will be entirely online, and your primary source of information will be this collection of code examples and exercises. There are a few lecture recordings from previous years, but only one or so is in English.
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 (15-20 minutes each) and small programs for homework (a few hundred lines).
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 this directory:

- `grading.txt` – what is graded and how; what you need to pass,
- `homework.txt` – general guidelines that govern assignments,
- `reviews.txt` – writing and receiving peer reviews,
- `advisors.txt` – whom to talk to and how when you need help.

Study materials for each week are in directories `01` through `13`. Start by reading `intro.txt`. Assignments are in directories `hw1` through `hw6` and will be made available according to the schedule shown in `grading.txt`.

Part A.2: Grading

To pass the subject, you need to collect a total of 18 points (by any means). The points can be obtained as follows (these are upper limits):

- 12 points for homework (6 assignments, 2 points each)
- 9 points for weekly exercises,
- 6 points for finishing your homework early,
- 3 points for peer review.

You need to pass the 18 point mark by 17th of February, one week after the last deadline of the last homework (this gives you some space to collect the remaining points via peer review).

A.2.1 Homework There will be 6 assignments, one every two weeks. There will be 8 deadlines for each of them, one week apart and each deadline gives you one chance to pass the automated test suite. If you pass on the first or second deadline, you get 1 extra point for the assignment. For the third and fourth deadlines, the bonus is reduced to 0.5 point. Afterwards, you only get the baseline 2 points.
The deadline schedule is as follows:

	given	try 1	try 2	try 3	try 4
		3 points		2.5 points	
hw1	7.10.	14.10.	21.10.	28.10.	4.11.
hw2	21.10.	28.10.	4.11.	11.11.	18.11.
hw3	4.11.	11.11.	18.11.	25.11.	2.12.
hw4	18.11.	25.11.	2.12.	9.12.	16.12.
hw5	9.12.	16.12.	23.12.	30.12.	6.1.
hw6	16.12.	23.12.	30.12.	6.1.	13.1.

	given	try 5	try 6	try 7	try 8
		2 points			
hw1	7.10.	11.11.	18.11.	25.11.	2.12.
hw2	21.10.	25.11.	2.12.	9.12.	16.12.
hw3	4.11.	9.12.	16.12.	23.12.	30.12.
hw4	18.11.	23.12.	30.12.	6.1.	13.1.
hw5	9.12.	13.1.	20.1.	27.1.	3.2.
hw6	16.12.	20.1.	27.1.	3.2.	10.2.

The test suite is strictly binary: you either pass or you fail. More details and guidelines are in `homework.txt`.

A.2.2 Weekly Exercises Besides homework assignments, the main source of points will be weekly exercises. Like with homework, you are not required to do any of these (except to get sufficient points to pass the course). How you split points between homework and the weekly exercises is up to you.
Each week, you will be able to submit a fixed subset of the exercises given to you (i.e. we will select usually 2, sometimes perhaps 3 exercises, which you can submit and get the point). Each week, you will be able to get up to one point (so in theory, 12 points are available, but the maximum you can earn this way is capped at 9). The point will be split between the exercises, i.e. it will be possible to earn fractional points in a given week, too.
If bonuses are present in an exercise, those are not required in submissions (nor they are rewarded with points).
The exercises have test cases enclosed: it is sufficient to pass those test cases to earn the associated points. The deadlines to earn points are as follows (you will have 2 weeks to solve each set):

chapter	given	deadline
01	7.10.	21.10.
02	14.10.	28.10.
03	21.10.	4.11.
04	28.10.	11.11.
05	4.11.	18.11.
06	11.11.	25.11.
07	18.11.	2.12.
08	25.11.	9.12.
09	9.12.	23.12.
10	16.12.	30.12.
11	6.1.	20.1.
12	13.1.	27.1.

A.2.3 Peer Review Reading code is an important skill – sometimes more so than writing it. While the space to practice reading code in this subject is limited, you will still be able to earn a few points doing just that. The rules for peer review are as follows:

- **only homework** is 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 assignment yourself,
- there are no deadlines for requesting or providing peer reviews (other than the deadline on passing the subject),
- writing a review is worth 0.3 points and you can write at most 10.

It is okay to point out correctness problems during peer reviews, with the expectation that this might help the recipient pass the assignment. 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 earn you -6 points per instance. 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 homework and on weekly exercises **which you submit**. If you want to study with your classmates, that is okay – but only cooperate on exercises which are not going to be submitted by either party.

Part A.3: Homework

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 will likely fail the tests.
Do not print anything that you are not specifically directed to. Programs which print garbage (i.e. 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 in the **second or third week**. 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.
If you strongly disagree with a test outcome and you believe you adhered to the specification and resolved any ambiguities in a sensible fashion, please use the online chat or the discussion forum in the IS to discuss the issue (see `advisors.txt` for details).

A.3.1 Submitting Solutions The easiest way to submit a solution is this:

```
$ ssh aisa.fi.muni.cz
$ cd ~/pv248/hw1
<edit files until satisfied>
$ pv248 submit
```

If you prefer to work in some other directory, you may need to specify which homework you wish to submit, like this: `pv248 submit hw1`. The number of times you submit is not limited (but see also below).
NB. **Only** the files listed in the assignment will be submitted and evaluated. Please put your **entire** solution into **existing files**.
You can check the status of your submissions by issuing the following command:

```
$ pv248 status
```

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

```
$ pv248 diff
```

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

A.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 passes mypy.
- The next step is **sanity** and runs every 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 once a week – every Wednesday at midnight, right after the deadline. If you pass the verity suite, the assignment is considered complete and you are awarded the corresponding number of points. The verity suite will **not** run unless the code passes 'sanity'.

If you pass on the first or the second run of the full test suite (7 or 14 days after the assignment is given), you are entitled to a bonus point. If you pass at one of the next 2 attempts, you are entitled to half a bonus point. After that, you have 4 more attempts to get it right. See `grading.txt` for more details.
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).

Part A.4: Advisors

It is hard to anticipate what problems you will run into while programming, and which concepts you will find hard to understand. Normally, those issues would be resolved in the seminar, but this semester, we won't have that luxury.
Instead, we will do our best to give you extended text materials and examples, so that you can resolve as many issues as possible on your own. Of course, that will sometimes fail: for that reason, you will be able to **interactively** ask for **help online**. Unfortunately, as much as we would like to, we cannot provide help 24/7 – there will instead be a few slots in which one of the teachers will be specifically available. You can ask questions at other times, and we will provide a 'best effort' service: if someone is available, they may answer the question, but please do not rely on this. For this, we will use the online chat available at https://lounge.fi.muni.cz – use your faculty login and password to get in, and join the channel (room) ##pv248 (double sharp). The schedule

is as follows:

day	start	end	person
Tue	18:00	20:00	Petr Ročkai
Thu	16:00	18:00	Vladimír Štill

The other option is of course the discussion forum in IS, where you can ask questions, though this is not nearly as interactive, and the delay can be considerable (please be patient).
Please also note that the online chat is meant for **programming discussion**: if you have questions about organisation or technical issues, use the discussion forum instead. Since exercises won't be published until Wednesday, the first session will be held on Thursday, 8th of October.

Part 1: Python Intro

There are two sets of exercises in the first week (exercises within each set are related). The first set is an evaluator of simple expressions in reverse polish notation (files prefixed `rpn_`) and the other is about planar analytic geometry (simple geometric objects, their attributes, transformations on them and interactions between them; these files are prefixed `geom_`). Each of the two blocks is split into three exercises. One thing that you will need but might not be familiar with is **variadic functions**: see `varargs.py` for an introduction.
The order in which the exercises were meant to be solved is this:

1. `rpn_un.py`
2. `rpn_bin.py` (can be submitted)
3. `rpn_gen.py`

4. `geom_types.py`
5. `geom_intersect.py` (can be submitted)
6. `geom_dist.py`

It is okay to flip the two blocks, but the exercises within each block largely build on each other and cannot be as easily skipped or reordered.

Part 1.1: Exercises

1.1.1 [rpn_un] In the first (short) series of exercises, we will implement a simple RPN (Reverse Polish Notation) evaluator. The entry point will be a single function, with the following prototype:

```
def rpn_eval( rpn ):
    pass
```

The `rpn` argument 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 result of `rpn_eval` should be the stack at the end of the computation. Below are a few test cases to check the implementation works as expected. You are free to add your own test cases. When you are done, you can continue with `rpn_bin.py`.

1.1.2 [rpn_bin] The second exercise is rather simple: take the RPN evaluator from the previous exercise, and extend it with the following binary operators: `+`, `-`, `*`, `/`, `**`. On top of that, add two 'greedy' operators, `sum` and `prod`, which reduce the entire content of the stack to a single number.
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.
This exercise is one of the two which you can submit this week, and is worth **0.5 points**.

```
def rpn_eval( rpn ):
    pass
```

Some test cases are included below. Write a few more test cases to convince yourself that your code works correctly. If you didn't see it yet, you should make a short detour to `varargs.py` before you come back to the last round of RPNs, in `rpn_gen.py`.

1.1.3 [rpn_gen] Let's generalize the code. Until now, we had a fixed set of operators hard-coded in the evaluator. Let's instead turn our evaluator into an object which can be extended by the user with additional operators. The class should have an `evaluate` method which takes a list like before.
On top of that, it should also have an `add_op(name, arity, f)` method, where `name` is the string that describes / names the operator, `arity` is the number of operands it expects and `f` is a function which implements it. The function `f` should take as many arguments as `arity` specifies.

```
class Evaluator:
    def __init__( self ):
        pass
    def add_op( self, name, arity, f ):
        pass
    def evaluate( self, rpn ):
        pass

def example():
    e = Evaluator()
    e.add_op( '*', 2, lambda x, y: x * y )
    e.add_op( '+', 2, lambda x, y: x + y )
    print( e.evaluate( [ 1, 2, '+', 7, '*' ] ) ) # expect [21]
```

Bonus 1: Allow `arity = 0` to mean 'greedy'. The function passed to `add_op` in this case must accept any number of arguments.

```
bonus_1 = True   # enable / disable tests for bonus 1
```

Bonus 2: Can you implement `Evaluator` in such a way that it does not require the `arity` argument in `add_op()`? How portable among different Python implementations do you think this is?
As usual, write a few test cases to convince yourself that your code works (in addition to the ones already provided). Be sure to check that operators with arities 1 and 3 work, for instance.
Then, you can continue to `geom_types.py`.

1.1.4 [geom_types] The second set of exercises will deal with planar analytic geometry. First define classes `Point` and `Vector` (tests expect the attributes to be named `x` and `y`):

```
class Point:
    def __init__( self, x, y ):
        pass
    def __sub__( self, other ): # self - other
        pass # compute a vector
    def translated( self, vec ):
        pass # compute a new point

class Vector:
    def __init__( self, x, y ):
        pass
    def length( self ):
        pass
    def dot( self, other ): # dot product
```

```
        pass
    def angle( self, other ): # in radians
        pass
```

Let us define a line next. Whether you use a point and a vector or two points is up to you (the constructor should take two points). Whichever you choose, make both representations available using methods (point_point and point_vector, both returning a 2-tuple). The points returned should be the same as those passed to the constructor, and the vector should be the vector from the first point to the second point.
Apart from the above methods, also implement an equality operator for two lines (__eq__), which will be called when two lines are compared using ==. In Python 2, you were also expected to implement its counterpart, __ne__ (which stands for 'not equal'), but Python 3 defines __ne__ automatically, by negating the result of __eq__.

```
class Line:
    def __eq__( self, other ):
        if not isinstance( other, Line ):
            return False
        pass # continue the implementation
    def translated( self, vec ):
        pass
    def point_point( self ):
        pass
    def point_vector( self ):
        pass
```

The Segment class is a finite version of the same.

```
class Segment:
    def length( self ):
        pass
    def translated( self, vec ):
        pass
    def point_point( self ):
        pass
```

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

```
class Circle:
    def __init__( self, c, r ):
        pass
    def center( self ):
        pass
    def radius( self ):
        pass
    def translated( self, vec ):
        pass
```

As always, write a few test cases to check that your code works. 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.

1.1.5 [geom_intersect] We first import all the classes from the previous exercise, since we will want to use them.

```
from geom_types import *
```

We will want to 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.

This exercise is the second that you can submit. You will need to include geom_types.py as well, but the points are all attached to this exercise (i.e. submitting geom_types.py alone will not earn you any points).
Line-line intersect either returns a list of points, or a Line, if the two lines are coincident.

```
def intersect_line_line( p, q ):
    pass
```

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

```
def intersect_line_segment( p, s ):
    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 list of points.

```
def intersect_line_circle( p, c ):
    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) 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, b ):
    pass
```

Test cases follow. Note that the tests use line equality which you implemented in geom_types. The last exercise for this week can be found in geom_dist.py.

1.1.6 [geom_dist] 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, b ):
    pass

def distance_point_line( a, p ):
    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, q ):
    pass
```

Circles vs points are rather easy, too:

```
def distance_point_circle( a, c ):
    pass
```

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

```
def distance_line_circle( p, c ):
    pass
```

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

```
def distance( a, b ):
    pass
```

Part 2: Data Structures

In the second week, there is one set of 3 related and another set of 3 standalone exercises. The first set is a mock 'reimplement a legacy system' story. If something appears to be stupid, blame the original authors. They were overworked programmers too, probably fresh out

of school.
The first set contains:

1. `ts3_escape.py`
2. `ts3_normalize.py` (this one can be submitted)
3. `ts3_render.py` [tbd]

The second set contains exercises which make use of the basic built-in classes, like `dict`, `set`, `list`, `str` and so on:

- `merge.py` (this one can be submitted)
- `rewrite.py` – fun with rewrite systems
- `magic.py` – identifying file types

Part 2.1: Exercises

2.1.1 [`ts3_escape`] 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 `${`, throw RuntimeError, since such string could not have been returned from `ts3_escape`. Once you are done, continue to `ts3_normalize.py`.

```
def ts3_escape( string ):
    pass

def ts3_unescape( string ):
    pass

def test_main():

    pairs = [
        ( "", "" ),
        ( "aa", "aa" ),
        ( "$", "$" ),
        ( "{", "{" ),
        ( "${", "$${" ),
        ( "$${", "$$${" ),
        ( "$$ {", "$$ {" ),
        ( "${$", "$${$" ),
        ( "$$${", "$$$${" ),
        ( "ab${ nabc$$${{tsk$$$${asd${${}}}aa$a{",
          "ab$${ nabc$$$${{tsk$$$$${asd$${$${}}}aa$a{" ),

        ( "#", "#" ),
        ( "#{", "##{" ),
        ( "#{{", "##{{" ),
        ( "####{", "#####{" ),
        ( "#}{", "#}{" ),
        ( "$#{", "$##{" ),
        ( "#${", "#$${" ),

        ( "$#{}${##}##{$}%${##${$$${{${",
          "$##{}$${##}###{$}%$${##$${$$$${{$${" )
    ]

    for unescaped, escaped in pairs:
        err = "ts3_escape( '{}' ) did not match '{}'".format(
unescaped, escaped )
        assert ts3_escape( unescaped ) == escaped, err
        err = "ts3_unescape( '{}' ) did not match '{}'".format(
escaped, unescaped )
        assert ts3_unescape( escaped ) == unescaped, err
```

2.1.2 [`ts3_normalize`] Eventually, we will want to replicate the actual substitution into the templates. This will be done by the `ts3_render` function. 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:
    pass

class Template:
    pass
```

Each of the above classes should have an attribute called `text`, which is a string and contains only the actual 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 ):
    pass
```

2.1.3 [ts3_render] 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, raise a RuntimeError 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:

- if it is a Document, replace the ${...} with the text of the document; the pasted text is excluded from further processing,
- if it is a Template, the ${...} is replaced with the text of the template; occurrences of ${...} and #{...} within the pasted text are further processed,
- if it is an int, it is formatted and the resulting string replaces the ${...},
- if it is a list, the length of the list is formatted as if it was an int, and finally,
- if it is a 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: raise a RuntimeError for now, someone else will fix that 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}}, throw an error (but see also bonus 2 below).

```
def ts3_render(tree):
    pass
```

Bonus 1: 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!
PS: 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.
Bonus 2: 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".

```
from ts3_normalize import ts3_normalize
```

2.1.4 [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
- 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, equiv, combine):
    pass
```

2.1.5 [rewrite] Write a function is_generated which checks whether word final can be generated by a rewrite system described by rules starting from word initial.
The rewrite system is given as a dict, where keys are str and values are each a list of str. The key is the left-hand side of a rule (see below) while the list gives all possible right-hand sides to go with it. The initial string and the string to be generated (final) are given as str instances.

```
def is_generated( rules, initial, final ):
    pass
```

A rewrite system is like a grammar, but does not distinguish between terminals and non-terminals. There are only letters, and the rules say that a given substring can be rewritten to some other substring. For instance, consider the rules:

1. x → xx (any x in the string can be doubled)
2. xx → xyz
3. xx → xyx

Starting from xy, a possible derivation would be:

1. use rule 1 to obtain xxy
2. use rule 2 to obtain xyzxy
3. use rule 1 again to obtain xyzxxy

All of the words which appear above are said to be generated by the rewrite system. More formally, a word is generated by the system if it can be obtained by applying a finite sequence of rules. Each rule can be applied in an arbitrary position (i.e. wherever you like). In this exercise, the right side of a rule is always strictly longer than the left side (this reduces the power of the system considerably and makes the exercise much easier to solve).

2.1.6 [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:
  0. offset, an integer,
  a. bits, a bytes object,
  b. mask, another bytes object,
  c. 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
```

Part 3: Text, JSON

The first set of exercises is general text processing, while the second deals with more structured data (JSON and a bit of CSV).

1. `grep.py`
2. `rfc822.py`
3. `multi822.py` – submit for 0.5 points

4. `report.py` (needs report.json)
5. `elements.py` (needs elements.csv) – submit for 0.5 points
6. `flatten.py`

Part 3.1: Exercises

3.1.1 `[grep]` The goal of this exercise is to write a simple program that works like UNIX `grep`. We will start by writing 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:

```
43: This line matched a regex,
```

Hint: check out the `enumerate` built-in.

```
def grep( regex, filename ):
    pass
```

3.1.2 `[rfc822]` 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 ):
    pass
```

When done, go on to `multi822.py`.

3.1.3 `[multi822]` Building on the previous exercise, extend the parser in the following way: if the given field is unique, keep its associated value as a string. However, if a certain field appears multiple times, turn the value into a list. The right-hand-side strings should be listed in the order of appearance.

```
def parse_multirfc822( filename ):
    pass
```

3.1.4 `[report]` The goal here is to load the file `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
```

In the next exercise, we will try to write some JSON instead: `elements.py`.

```
import json # go for `load` (via io) or `loads` (via strings)
```

3.1.5 `[elements]` In this exercise, we will read in a CSV (comma-separated values) file and produce a JSON file. The input is in `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. Name the output file identically to the input file, except for the extension (`.json`).
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(filename):
    pass
```

3.1.6 `[flatten]` In this exercise, your task is to write a function that flattens json data. Flattening works as follows:
The result is a single-level (flat) json with key-value pairs, the keys representing the former structure of data. We could use any (unique) separator to indicate the nested structure, which would allow unflattening without loss of information. We will use the dollar sign '$'. If you do encounter '$' in the original data, replace it with two dollars '$$'. Assume that there are no keys composed entirely of numbers.
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' }
```

The simplest way to go about it is to use recursion.

```
def flatten( data ):
    pass
```

Part 4: SQL

This week, we will look at some basic SQL. First, we will import and export JSON data to/from an SQL database and perform simple searches.

The schema for all three exercises is in books.sql.

1. book_import.py – import data into a database [0.5 pts]
2. book_export.py – the converse
3. book_query.py – use SQL to search for things

The second part will build up a very simple CRUD-type application (CRUD = Create Read Update Delete). The interface will be through Python objects. The subject will be shopping lists. The second exercise in the set can be submitted (though you will need to also include the first, which will not be graded on its own).

4. list_create.py – creating shopping lists
5. list_search.py – searching and reading [0.5 pts]
6. list_update.py – update and delete

Part 4.1: Exercises

4.1.1 [book_import] Load the file 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 books.sql and you can create an empty database with the correct data definitions by running the following command:

```
$ sqlite3 books.dat < 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 abovementioned 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 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
```

```
def test_main():
    import os
    try:
        os.unlink( 'books.dat' )
    except:
        pass
    conn = sqlite3.connect( 'books.dat' )
    import_books( 'books.json', 'books.dat' )
    cur = conn.cursor()

    books_authors_ref = {}
    for item in json.load( open( 'books.json' ) ):
        books_authors_ref[ item[ 'name' ] ] = item[ 'authors' ]

    cur.execute( 'select * from book order by id' )
    books = cur.fetchall()
    book_names = set( [ name for id, name in books ] )
    assert book_names == books_authors_ref.keys()

    cur.execute( 'select * from author order by id' )
    authors = cur.fetchall()
    author_names = set( [ name for id, name in authors ] )
    all_authors = set( sum( books_authors_ref.values(), [] ) )
    assert author_names == all_authors

    cur.execute( 'select * from book_author_list order by book_id' )
    book_author = cur.fetchall()
    assert len( book_author ) == sum( [ len( l ) for l in
books_authors_ref.values() ] )
    for b_id, a_id in book_author:
        _, b_name = books[ b_id - 1 ]
        _, a_name = authors[ a_id - 1 ]
        assert a_name in books_authors_ref[ b_name ]

    conn.close()
```

4.1.2 [book_export] 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

def test_main():
    export_books( 'books.dat', 'books_2.json' )

    with open( 'books.json', 'r' ) as f1:
        js1 = json.load( f1 )
    with open( 'books_2.json', 'r' ) as f2:
        js2 = json.load( f2 )
    assert js1 == js2
```

4.1.3 [book_query] 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
```

```
from math import isclose

def test_main():

    conn = sqlite3.connect( 'books.dat' )
    res = books_by_author( conn, 'Brontë' )
    assert set( res ) == set( ['Poems by Currer, Ellis and Acton Bell',
                               'Jane Eyre', 'The Professor',
'Wuthering Heights'] )
    res = books_by_author( conn, 'son' )
    assert res == ['The Rise and Fall of D.O.D.O.']

    res = authors_by_book( conn, 'Bell' )
    assert set( res ) == set( ['Charlotte Brontë', 'Emily Brontë',
'Anne Brontë',
                               'Ernest Hemingway'] )
    res = authors_by_book( conn, 'н' )
    assert set( res ) == set( ['Аркадий Стругацкий', 'Борис Стругацкий']
)

    res = books_by_author_count( conn, 2 )
    assert set( res ) == set( ['Poems by Currer, Ellis and Acton Bell',
'Улитка на склоне', 'The Rise and Fall of D.O.D.O.'] )

    res = books_by_author_count( conn, 3 )
    assert res == ['Poems by Currer, Ellis and Acton Bell']

    res = books_by_author_count( conn, 4 )
    assert not res

    res = average_author_count( conn )
    assert isclose( res, 1.4444444444444444 )
```

Part 5: Operators, Iterators, Decorators and Exceptions

This week, we will look at some of the more advanced language features of Python.
The first couple of exercises will be about iterators and generators:

1. `iter.py` – simple iterators [submit for 0.5 points]
2. `flat.py` – a simple generator

We will then move on to operator overloading:

3. `poly.py` – polynomials
4. `mod.py` – finite rings of integers mod N (/n)

Finally, the last will cover exceptions and decorators:

5. `with.py` – context managers vs exceptions
6. `trace.py` – decorators part 1 [submit for 0.5 points]
7. `noexcept.py` – decorators and exceptions

Part 5.1: Exercises

5.1.1 [iter] Implement these 4 utilities on iterables:

- prefix and suffix list
- prefix and suffix sum

Examples:

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

def prefixes_iter( list_in ):
    pass

def prefixes_gen( list_in ):
    pass

def suffixes( list_in ):
    pass

def prefix_sum( list_in ):
    pass

def suffix_sum( list_in ):
    pass
```

Implement the prefix list both using an iterator (an object with `__iter__`) and using a generator. Pick one approach for each of the remaining 3, but make sure there's at least one iterator and one generator among them.
Go on to `flat.py`.

```
import types

def test_main():

    res = [ [], [1], [1,2], [1,2,3], [1,2,3,4] ]

    for i in prefixes_iter( [1,2,3,4] ):
        assert i in res
        res.remove( i )

    assert not res # emptied

    res = [ [], [1], [1,2], [1,2,3], [1,2,3,4] ]
    assert isinstance( prefixes_gen( [] ), types.GeneratorType )

    for i in prefixes_gen( [1,2,3,4] ):
        assert i in res
        res.remove( i )

    assert not res

    res = [ [], [7], [8,7], [6,8,7], [5,6,8,7] ]

    for i in suffixes( [5,6,8,7] ):
        assert i in res
        res.remove( i )
    assert not res # emptied

    count = 0
    for item in prefix_sum( [1,2,3,4,5] ):
        count += 1 # is iterable
    assert count == len( [1,2,3,4,5] )

    assert list( prefix_sum( [1,2,3,4,5] ) ) == [1,3,6,10,15]

    count = 0
    for item in suffix_sum( [1,2,3,4,5] ):
        count += 1 # is iterable
    assert count == len( [1,2,3,4,5] )

    assert list( suffix_sum( [1,2,3,4,5] ) ) == [5,9,12,14,15]
```

5.1.2 [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.

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

Let's move on to poly.py.

```
def test_main():

    f = flatten( [ 7, [ 2, 2 ], map( lambda x : x - 8, [ 0, 9 ] ) ]
)
    assert list( f ) == [ 7, 2, 2, -8, 1 ]

    f = flatten( [ map( lambda x: x * x, range( 3 ) ),
                 [ range( i + 1 ) for i in range( 3 ) ] ] )

    assert list( f )  ==  [ 0, 1, 4, 0, 0, 1, 0, 1, 2 ]

    f = flatten( [ 0, ( 1, 7, 3 ), 9, [ [ [ 3 ] ], [ -2, [ 0 ] ] ],
reversed( [ 22, 9 ] ) ] )
    res = [ 0, 1, 7, 3, 9, 3, -2, 0, 9, 22 ]

    f_len = 0

    for item, item_r in zip( f, res ):
        f_len += 1
        assert item == item_r

    assert f_len == len( res )
```

5.1.3 [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: 2x^3 + 7x^2 + 5
print( y )     # prints 2x + 4
print( x + y ) # prints 2x^3 + 7x^2 + 2x + 9

class Poly:
    pass
```

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

```
def test_main():

    x = Poly( 2, 7, 3, 5 )
    y = Poly( 2, 4 )
    z = Poly( 0, 4, 1, -3, 0 )
    a = Poly( 0 )

    assert str( x ) == "2x^3 + 7x^2 + 3x + 5"
    assert str( y ) == "2x + 4"
    assert str( z ) == "4x^3 + x^2 - 3x"
    assert str( a ) == "0"

    assert str( x + a ) == "2x^3 + 7x^2 + 3x + 5"
    assert str( a + a ) == "0"

    assert str( x + y ) == "2x^3 + 7x^2 + 5x + 9"
    assert str( y + x ) == str( x + y )

    assert str( x + z ) == "6x^3 + 8x^2 + 5"
    assert str( z + x ) == str( x + z )

    assert str( y + z ) == "4x^3 + x^2 - x + 4"
    assert str( z + y ) == str( y + z )

    z_inv = Poly( 0, -4, -1, 3, 0 )
    assert str( z_inv + z ) == "0"
```

5.1.4 [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]_n, such as [5]_7 to represent all integers with remainder 5. Implement equality, comparison, printing, and the respective addition and multiplication (all should also work with integers).
The class Mod represents a congruence class x modulo n.

```
class Mod:
    def __init__( self, x, n ):
        pass

def test_main():

    x = Mod( 3, 7 )
    y = Mod( 4, 7 )
    z = Mod( 1, 2 )

    assert str( x + y ) == "[0]_7"
    assert str( x * y ) == "[5]_7"

    try:
        print( x + z ) # TypeError
        assert False
    except TypeError:
        pass

    assert str( x + 1 ) == "[4]_7"
    assert str( y - x ) == "[1]_7"

    assert x == 3
    assert x == x

    try:
        x == z
        assert False
    except TypeError:
        pass

    assert not x == y
    assert not x > y
    assert x < y

    assert Mod( 1, 7 ) == Mod( 8, 7 )
    assert 1 == Mod( 1, 7 )
    assert 2 != Mod( 1, 7 )

    assert str( Mod( 2, 6 ) - 5 ) == "[3]_6"
```

Next thing to look at are exceptions and context managers, in `with.py`.

5.1.5 **[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

def foo( x, y ):
    with context( "asserting equality", x, '=', y ):
        assert x == y
```

foo((), ()) foo(7, []) should print something like this (the first call should print nothing):
asserting equality 7 = [] Traceback (most recent call last): File "with.py", line 20, in <module>

```
foo( 7, [] )
```

File "with.py", line 17, in foo

```
assert x == y
```

AssertionError

```
import sys
import traceback
from io import StringIO

def redirect_err( f ):

    stderr = sys.stderr
    out = StringIO()
    sys.stderr = out

    try:
        f()
    except:
        traceback.print_exc()

    sys.stderr = stderr
    return out.getvalue()

def test_main():

    def test():
        foo( (), () )
        foo( 7, [] )

    output = redirect_err( test )

    res = output.split( '\n' )
    assert res[0] == "asserting equality 7 = []"
    assert res[1] == "Traceback (most recent call last):"
    assert res[2].strip().startswith( "File" )
    assert res[2].endswith( "in redirect_err" )
    assert res[3] == "    f()"
    assert res[-2] == "AssertionError"
```

5.1.6 **[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 something like this:
foo [13] bar [13] -> 20 bar [26] -> 33 returned 53

```
def traced( f ):
    pass

@traced
def bar( x ):
    return x + 7

@traced
def foo( x ):
    return bar( x ) + bar( 2 * x )

import sys
from io import StringIO

def redirect_out( test, *args ):

    stdout = sys.stdout
    out = StringIO()
    sys.stdout = out

    test( *args )

    sys.stdout = stdout
    return out.getvalue()

def test_main():

    output = redirect_out( foo, 13 )

    assert output == " foo [13]\n" \
                     "   bar [13] -> 20\n" \
                     "   bar [26] -> 33\n" \
                     " returned 53\n"
```

5.1.7 **[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.

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

def test_main():

    @noexcept( TypeError, AssertionError )
    def foo():
        assert 1 == 2

    @noexcept()
    def bar():
        assert 2 == 3

    assert foo() == None
    assert bar() == None

    @noexcept( TypeError )
    def baz():
        raise AssertionError

    try:
        baz()
        assert False
    except AssertionError:
        pass

    @noexcept( TypeError, AssertionError, RuntimeError )
    def bazz( a1, a2, a3 ):
        assert 1 == 1
        return a1 + a2 + 7 + a3

    assert bazz( 3, -4, a3 = -1 ) == 5
```

Part 6: Closures and Coroutines

There will be two sets of exercises, mostly focused on coroutines (this week, we will use generators – as long as you only use `yield`, but not `send` – those are in fact semi-coroutines; if you also use `send`, generators behave as full coroutines).
We will look at so-called **native coroutines** next week, with `asyncio` (those are declared using `async def` and do not use `yield`). In some sense, native and generator-based coroutines have the same expressive power, but they have different underlying implementations and make slightly different trade-offs.
The first couple of exercises will be quite easy, to get a feeling for yields and coroutines. You can also consult `gen.py` which is really just a short demo.

1. `interleave.py`
2. `stream-getline.py`

The 'real' exercises start with number 3, and all deal with using coroutines for parsing, which is one of the more canonical use cases (except perhaps for using them as generators, which we did last week). In those exercises, you can also practice using lexical closures: they will come in handy when defining the coroutines in `stream-lexer.py` and `parse.py`, at minimum.

3. `stream-lexer.py` [0.5 points]
4. `tbd` one more stream exercise?
5. `parse.py`
6. `mbox.py` [0.5 points]

Part 6.1: Exercises

6.1.1 [`interleave`] Write two generators, one which simply yields numbers 1-5 and another which implements a counter: sending a number to the generator will adjust its value by the amount sent. Then write a driver loop that sends the output of gen1() into gen2(). Add print statements to both to make it clear in which order the code executes. After you are done, implement the same thing with plain objects: Numbers with a get() method and Counter with a `get()` and `put(n)`.

```
def numbers(): # generate numbers 1-5
    pass

def counter():
    pass
```

Write one more generator, the driver loop.

```
def driver():
    pass

class Numbers:
    pass
class Counter:
    pass
```

The driver loop again, now with objects.

```
def driver_obj():
    pass

def test_main():

    for dr in [ driver, driver_obj ]:
        nums = iter( [ 1, 2, 4, 7, 11, 16 ] )
        d = dr()
        for n in d:
            num = next( nums )
            assert n == num, "{} != {} in {}".format( n, num,
d.__name__ )
```

check we exhausted nums

```
        try:
            next( nums )
            assert False
        except StopIteration:
            pass
```

6.1.2 [`stream-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 next week. However, for now, we will do everything by hand, to get a better understanding of the principles.
A **stream processor** will be a (semi)coroutine (i.e. a generator) which takes another (semi)coroutine as an argument. It will extract data from the 'upstream' (the coroutine that it got as an argument) using `next` and it'll send it further 'downstream' using `yield`.
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. A **sink** is another variation: it takes a stream, but does not yield anything. 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.
Let us first 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.

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

def make_test_source():
    counter = Box( 0 )
    def test_source():
        yield "hello "
        counter.value += 1
        yield "world\ni am\n"
        counter.value += 1
        yield " a"
        counter.value += 1
        yield " strea"
        counter.value += 1
        yield "m\nsour"
        counter.value += 1
        yield "ce\n"
        counter.value += 1
        while True:
            yield None
    return ( test_source(), counter )
```

What follows is a very simple sink, which prints the content of the stream to `stdout`:

```
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

def test_main():
    stream, counter = make_test_source()
    assert counter.value == 0
    lines = stream_getline( stream )
    assert counter.value == 0
    assert next( lines ) == "hello world\n"
    assert counter.value == 1
    assert next( lines ) == "i am\n"
    assert counter.value == 1
    assert next( lines ) == " a stream\n"
    assert counter.value == 4
    assert next( lines ) == "source\n"
    assert counter.value == 5
    assert next( lines ) is None
    assert counter.value == 6
```

6.1.3 [stream-lexer] In the second exercise in the 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.

```
IDENT = 1
KW = 2
NUM = 3

def stream_lexer( text_stream ):
    pass
```

Testing code follows.

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

def make_test_source():
    counter = Box( 0 )
    def test_source():
        yield "ident"
        counter.value += 1
        yield "ifier\nm"
        counter.value += 1
        yield "ul 32"
        counter.value += 1
        yield "1 mul"
        counter.value += 1
        yield "tiply add"
        counter.value += 1
        yield "32 1"
        counter.value += 1
        while True:
            yield None
    return ( test_source(), counter )

def bad_stream():
    yield "12"
    yield "x"

def test_main():
    stream, counter = make_test_source()
    assert counter.value == 0
    tokens = stream_lexer( stream )
    assert counter.value == 0
    assert next( tokens ) == ( IDENT, "identifier" )
    assert counter.value == 1
    assert next( tokens ) == ( KW, "mul" )
    assert counter.value == 2
    assert next( tokens ) == ( NUM, "321" )
    assert counter.value == 3
    assert next( tokens ) == ( IDENT, "multiply" )
    assert counter.value == 4
    assert next( tokens ) == ( IDENT, "add32" )
    assert counter.value == 5
    assert next( tokens ) == ( NUM, "1" )
    assert counter.value == 6
    assert next( tokens ) is None
    assert counter.value == 6

    tokens = stream_lexer( bad_stream() )
    try:
        next( tokens )
        assert False
    except RuntimeError:
        pass
```

6.1.4 [parse] 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

def test_main():
    program = {
        "main": "set x 3 \n" \
                "set a 2 \n" \
                " add a 9\n" \
                "include sub_a\n" \
                "print x \n",
        "sub_a": "mul x a\n" \
                 "print x\n" \
                 " print a  \n" \
                 "   include file\n",
        "file": "add x 1\n" \
                " set y 7\n" \
                "print y\n"
}

lex = lexer( program )
par = parser( lex )

res = [ ( 'set', 'x', 3 ),
        ( 'set', 'a', 2 ),
        ( 'add', 'a', 9 ),
        ( 'mul', 'x', 'a' ),
        ( 'print', 'x' ),
        ( 'print', 'a' ),
        ( 'add', 'x', 1 ),
        ( 'set', 'y', 7 ),
        ( 'print', 'y' ),
        ( 'print', 'x' ) ]

res_it = iter( res )

for item in par:
    expected = next( res_it )
    assert item == expected, "{} != {}".format( item, expected )

try:
    next( res_it )
    assert False
except StopIteration:
    pass

multi_include = {
    "main": "set Axt 3 \n" \
            "set a 2 \n" \
            " add a 9\n" \
            "include file\n" \
            "print Axt\n" \
            "include file\n" \
            "print  b\n",
    "file": "mul a 7 \n" \
            " set b a\n" \
            "add   a b  \n"
}

lex = lexer( multi_include )
par = parser( lex )

res = [ ( 'set', 'Axt', 3 ),
        ( 'set', 'a', 2 ),
        ( 'add', 'a', 9 ),
        ( 'mul', 'a', 7 ),
        ( 'set', 'b', 'a' ),
        ( 'add', 'a', 'b' ),
        ( 'print', 'Axt' ),
        ( 'mul', 'a', 7 ),
        ( 'set', 'b', 'a' ),
        ( 'add', 'a', 'b' ),
        ( 'print', 'b' ) ]

res_it = iter( res )

for item in par:
    expected = next( res_it )
    assert item == expected, "{} != {}".format( item, expected )

try:
    next( res_it )
    assert False
except StopIteration:
    pass
```

6.1.5 [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. The reported elements are as follows:

- message: Yield a tuple of two integers: the index of the mail in the input file (i.e. 1 for the first mail, 2 for the second and so on) and the offset from the start of the file to the 'F' in 'From '. Yield as soon as you read the 'From ' sequence that delimits the message.
- header: Yield a tuple with two strings, 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.

In an mbox file, each message starts with a line like this:
From someone@example.com Wed May 1 06:30:00 MDT 2019
After which, 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. Headers may continue on the next line with an indent.

```
def parse_mbox( chars ):
    pass

class FileIter:

    def __init__( self, filename ):
        self.file = open( filename, "r" )
        self.chars_given = 0

    def __iter__( self ):
        return self

    def __next__( self ):
        c = self.file.read( 1 )
        if not c:
            self.file.close()
            raise StopIteration
        self.chars_given += 1
        return c

def test_main():
    f = FileIter( "mbox.txt" )
    g = parse_mbox( f )

    item = next( g )
    assert item == ( 1, 0 )
    assert f.chars_given == 5

    item = next( g )
    assert item == ( 'From', 'Author <author@example.com>' )
    assert f.chars_given == 79 # 44 + 34 + 1  ( line1 + line2 + check that

    item = next( g )
    assert item == ( 'To', 'Recipient <recipient@example.com>' )
    assert f.chars_given == 117 # 79 + 38

    item = next( g )
    assert item == ( 'Subject', 'Sample message 1' )
    assert f.chars_given == 143

    item = next( g )
    assert item == 'This is the body.\n' \
                   'From (should be escaped).\n' \
                   'Fromage?\n' \
                   'There are 4 lines.\n' \
                   '\n'
    assert f.chars_given == 222

    item = next( g )
    assert item == ( 2, 217 )
    assert f.chars_given == 222

    item = next( g )
    assert item == ( 'From', 'Author <author2@example.com>' )
    assert f.chars_given == 297

    item = next( g )
    assert item == ( 'To', 'Rec <rec@example.com>, Rec2 <rec2.2@example.co
)
```

```
    assert f.chars_given == 351

    item = next( g )
    assert item == ( 'Subject', 'Sample message 2' )
    assert f.chars_given == 377

    item = next( g )
    assert item == "This is the second body.\n"
    assert f.chars_given == 407

    item = next( g )
    assert item == ( 3, 402 )
    assert f.chars_given == 407

    item = next( g )
    assert item == ( 'From', 'Author <author3@example.com>' )
    assert f.chars_given == 482

    item = next( g )
    assert item == ( 'To', 'Recipient <recipient36@example.com>' )
    assert f.chars_given == 522

    item = next( g )
    assert item == ( 'Subject', 'Msg 3' )
    assert f.chars_given == 537

    item = next( g )
    assert item == "Msg body 3.\n"
    assert f.chars_given == 549

    try:
        next( g )
        assert False
    except StopIteration:
        pass
```

Part 7: asyncio Basics

This week, we will get acquainted with asyncio, the framework for writing asynchronous programs in Python, based on native coroutines and non-blocking IO. The exercises are as follows:

1. cor-sleep.py
2. cor-semaphore.py [submit for 0.5 points]
3. aio-process.py
4. aio-multi.py (tbd)
5. aio-counter.py [submit for 0.5 points]
6. http-client.py

Part 7.1: Exercises

7.1.1 [cor-sleep] Demonstrate the use of native coroutines and basic asyncio constructs. Define 2 coroutines, say cor1() and cor2(), along with an asynchronous main(). 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 main(), 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).

```
import asyncio

import sys
from io import StringIO
import time

def test_main():

    old = sys.stdout
    out = StringIO()
    sys.stdout = out

    start = time.time()
    asyncio.run( main() )
    end = time.time()

    sys.stdout = old
    result = out.getvalue().strip().split( '\n' )

    assert result[0] == "cor1"
    assert result[1] == "cor2"
    assert result[2] == "cor1"
    assert result[-1] == "cor2"
    assert result[-2] == "cor2"
    assert result[-3] == "cor1"

    assert end - start >= 5
    assert end - start < 5.2
```

7.1.2 [cor-semaphore] Use gather() to spawn 10 tasks, each running an infinite loop. Create a global semaphore that is shared by all those tasks and set its initial value to 3. In each iteration, each task should queue on the semaphore and when it is allowed to proceed, sleep 2 seconds before calling notify, and relinquishing the semaphore again. notify adds a tuple – containing the task id (1 - 10) and the time when the task reached the semaphore – to the global list reached.
Observe the behaviour of the program. Add a short sleep **outside** of the critical section of the task. Compare the difference in behaviour.
After your program works as expected, i.e. only 3 tasks are active at any given moment and the tasks alternate fairly, switch the infinite loop for a bounded loop: each task running twice, to be consistent with the tests.
NOTE: Most asyncio objects, semaphores included, are tied to an event loop. You need to create the semaphore from within the same event loop in which your tasks will run. (Alternatively, you can create the loop explicitly and pass it to the semaphore.)

```
import asyncio
import time

reached = []
begin = time.time()

def notify( i ):
    t = time.time() - begin
    print( "task {} reached semaphore at {}".format( i, t ) )
    reached.append( ( i, t ) )

async def main():
    pass

asyncio.run( main() )

def test_main():
    asyncio.run( main() )
    for i in [ 1, 2, 3 ]:
        assert i in [ t[0] for t in reached[0:3] ]
    for i in [ 4, 5, 6 ]:
        assert i in [ t[0] for t in reached[3:6] ]

    for i in range( 1, 11 ):
        assert i in [ t[0] for t in reached ]
    assert len( reached ) == 20
```

time difference at least 2 seconds from last 3-batch

```
    for i in range( 3, len( reached ) ):
        assert reached[ i ][1] - reached[ i - 3 ][1] > 2
```

7.1.3 [aio-process] 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. Keep the type: if the output is an integer, add an integer to the result; otherwise, add a string.
NOTE: The data that goes into the process and that comes out is bytes, not strings. Make sure to encode and decode the bytes as needed.

```
import asyncio
from asyncio.subprocess import PIPE

async def pipe_cmd( command, inputs ):
    pass

async def run():

    p = "while read x; do echo x is $x; done"
    inputs = [ 'a', 7, "input", 3.6, "`echo 2`" ]
    outputs = [ 'x is a', 'x is 7', 'x is input', 'x is 3.6',
                'x is `echo 2`' ]

    res = await pipe_cmd( p, inputs )
    assert res == outputs

    p = "bc"
    inputs = [ "231-19", "sqrt(2+7)*3", "8^7-5432^2" ]
    outputs = [ 212, 9, -27409472 ]

    res = await pipe_cmd( p, inputs )
    assert res == outputs

    p = "while read var; do echo \"${#var}\"; done"
    inputs = [ 229, 10239, 1343, 1, "06" ]
    outputs = [ 3, 5, 4, 1, 2 ]

    res = await pipe_cmd( p, inputs )
    assert res == outputs

def test_main():
    asyncio.run( run() )
```

7.1.4 [aio-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 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 cor-semaphore.py) – as a tuple.
Don't forget to clean up at the end.
Continue with `aio-counter.py`.

```
import asyncio
from asyncio.subprocess import PIPE

data = []

async def main():
    pass

def test_main():
    asyncio.run( main() )
    error = 0

    def check( result, p, char, t ):
        nonlocal error
        p_c, tt = result
        assert tt >= t and abs(tt - t) - error < 0.05
        error = abs(tt - t)
        p_, c = p_c.split(': ')
        assert p_ == p
        assert c == char

    check( data[0], 'p1', 'a', 0.0 )
    check( data[1], 'p2', 'a', 0.5 )
    check( data[2], 'p1', 'b', 0.7 )
    check( data[3], 'p2', 'b', 1.2 )
    check( data[4], 'p1', 'c', 1.4 )
    check( data[5], 'p2', 'c', 1.9 )
    check( data[6], 'p1', 'd', 2.1 )
    check( data[7], 'p2', 'd', 2.6 )
    check( data[8], 'p1', 'e', 2.8 )
    check( data[9], 'p2', 'e', 3.3 )
    check( data[10], 'p1', 'f', 3.5 )
    check( data[11], 'p2', 'f', 4.0 )
    check( data[12], 'p1', 'g', 4.2 )
    check( data[13], 'p2', 'g', 4.7 )
    check( data[14], 'p1', 'h', 4.9 )
    check( data[15], 'p2', 'h', 5.4 )
```

7.1.5 [aio-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.

```
import asyncio

async def counters( queue, sleeps, iterations ):
    pass

def fuzzy( a, b ):
    for i, j in zip( a, b ):
        if abs( i - j ) > 1:
            return False
    return True

async def check( q ):
    assert fuzzy( await q.get(), [ 1, 2, 10 ] )
    assert fuzzy( await q.get(), [ 2, 4, 20 ] )
    assert fuzzy( await q.get(), [ 3, 6, 30 ] )

async def main():
    q = asyncio.Queue()
    await asyncio.gather( check( q ), counters( q, [ 1, 0.5, 0.1 ],
3 ) )

def test_main():
    asyncio.run( main() )
```

7.1.6 [http-client] Use aiohttp to fetch a given URL and stream the HTML into tidy. Specifically, use `tidy 2>&1` as the command that you start with `asyncio.create_subprocess_shell`. Capture the stdout and return the second line that tidy prints (should say there were no errors).

```
import aiohttp
import asyncio
from asyncio.subprocess import PIPE

async def tidy( url ):
    pass

async def main():
    out = await tidy( 'http://example.com' )
    assert out == b'No warnings or errors were found.\n'
```

out = await tidy('http://www.fi.muni.cz') assert out == b'line 125 column 81 - Warning: replacing unexpected button with </button>\n';

```
def test_main():
    asyncio.run( main() )
```

Part 8: More asyncio

While the lecture this weeks goes into low-level details of asyncio, we will stick with the high-level interface. We will write some simple servers and clients using sockets.
Simple async servers:

1. `tcp-echo.py`
2. `unix-rot13.py` [submit for 0.5 points]
3. `unix-merge.py` [bonus! submit for 0.5 points]

Passing data between native (`async def`) coroutines:

4. `pipeline.py`
5. `tokenize.py` (tbd)
6. `minilisp.py` [submit for 0.5 points]

Part 8.1: Exercises

8.1.1 [tcp-echo] Start a server, on localhost, on the given port (using asyncio.start_server) and have two clients connect to it. The server takes care of the underlying sockets, so we will not be creating them manually. Data is, again, transferred as `bytes` object.
The server should return whatever data was sent to it. Clients should send 'hello' and 'world', respectively, then wait for the answer from the server and return this answer. Add print statements to make sure your server and clients behave as expected; print data received by the server, sent to the clients and sent and received by the clients on the client side. Make sure to close the writing side of sockets once data is exhausted.

```
import asyncio
```

Server-side, handler for connecting clients. Read the message from the client and echo it back to the client.

```
async def handle_client( reader, writer ):
```

print("server received & sending", ...)

```
    pass
```

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 before sending 'world', to ensure message order.

```
async def client( port, msg ):
```

print("client sending", ...) print("client received", ...)

```
    pass
```

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 main( port ):
    pass

import sys
import random
from io import StringIO

def test_main():

    stdout = sys.stdout
    out = StringIO()
    sys.stdout = out

    data = asyncio.run( main( random.randint( 9000, 13000 ) ) )
    print( data )
    assert data == ['hello', 'world']

    sys.stdout = stdout
    output = out.getvalue()
    print( output )
    output = output.split('\n')

    assert 'client sending hello' in output[ 0 : 3 ]
    assert 'client sending world' in output[ 0 : 3 ]
    assert 'server received & sending hello' in output[ 1 : 3 ]
    assert 'server received & sending world' in output[ 2 : ]
```

8.1.2 [unix-rot13] We will do something similar to tcp-echo, but this time we will use a UNIX socket. UNIX sockets exist on 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 on the filesystem otherwise. Note that this should only require very small changes from your previous solution to tcp-echo.

```
import asyncio

async def handle_client( reader, writer ):
```

print("server received", ...) print("server sending", ...)

```
    pass

async def client( msg, path ):
```

print("client sending", ...) print("client received", ...)

```
    pass

async def main( path ):
    pass

import sys
from io import StringIO
```

```
import os

def test_main():

    stdout = sys.stdout
    out = StringIO()
    sys.stdout = out

    path = "./tmp.socket"
    data = asyncio.run( main( path ) )
    print( data )
    assert data == ['uryyb', 'jbeyq']

    sys.stdout = stdout
    output = out.getvalue()
    print( output )
    output = output.split('\n')

    assert 'client sending hello' in output[ 0 : 3 ]
    assert 'client sending world' in output[ 0 : 3 ]
    assert 'server received hello' in output[ 1 : 3 ]
    assert 'server sending uryyb' in output
    assert 'server received world' in output
    assert 'server sending jbeyq' in output

    assert not os.path.exists( path )
```

8.1.3 [unix-merge] This exercise is quite hard, but also rather interesting, which is why it is marked as a bonus.
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

def test_main():
    lines_read = 0
    sem_start, sem_end = None, None

    async def check_line( reader, expect ):
        nonlocal lines_read
        expect += b'\n'
        got = await reader.readline()
        assert got == expect, f"{got} == {expect}"
        lines_read += 1

    async def check_complex( reader, writer ):
        await sem_start.get()

        _, s111 = await asyncio.open_unix_connection( "sock_11" )
        _, s112 = await asyncio.open_unix_connection( "sock_11" )
        _, s121 = await asyncio.open_unix_connection( "sock_12" )
        _, s122 = await asyncio.open_unix_connection( "sock_12" )
        _, s21 = await asyncio.open_unix_connection( "sock_2" )
        _, s22 = await asyncio.open_unix_connection( "sock_2" )

        s111.write( b'b\n' )
        s112.write( b'c\n' )

        s121.write( b'd\n' )
        s122.write( b'e\n' )

        s21.write( b'a\n' )
        s22.close()

        await check_line( reader, b'a' )

        s121.write( b'f\n' )
        s111.close()
        s112.close()
        s121.close()
        s122.close()
        s21.close()

        await check_line( reader, b'b' )
        await check_line( reader, b'c' )
        await check_line( reader, b'd' )
        await check_line( reader, b'e' )
        await check_line( reader, b'f' )

        await sem_end.put( 0 )

    async def main_complex():
        nonlocal sem_start, sem_end
        sem_start, sem_end = asyncio.Queue( 1 ), asyncio.Queue( 1
)
        chk = await asyncio.start_unix_server( check_complex,
"sock_o" )
        m   = await merge_server( "socket",  "sock_o" )
        m1  = await merge_server( "sock_1",  "socket" )
        m11 = await merge_server( "sock_11", "sock_1" )
        m12 = await merge_server( "sock_12", "sock_1" )
        m2  = await merge_server( "sock_2",  "socket" )
        await sem_start.put( 0 )
        await sem_end.get()
        assert lines_read == 6, f"{lines_read} == 6"

    async def check_simple( reader, writer ):
        await sem_start.get()
        _, s0 = await asyncio.open_unix_connection( "socket" )
        _, s1 = await asyncio.open_unix_connection( "socket" )

        s0.write( b'b\n' )
        s1.write( b'c\n' )
        await check_line( reader, b'b' )

        s0.write( b'f\n' )
        s1.write( b'd\n' )
        await check_line( reader, b'c' )

        s1.close()
        await check_line( reader, b'd' )

        s0.close()
        await check_line( reader, b'f' )

        await sem_end.put( 0 )

    async def main_simple():
        nonlocal sem_start, sem_end
        sem_start, sem_end = asyncio.Queue( 1 ), asyncio.Queue( 1
)
        chk = await asyncio.start_unix_server( check_simple,
"sock_o" )
        m   = await merge_server( "socket", "sock_o" )
        await sem_start.put( 0 )
        await sem_end.get()
        assert lines_read == 4, f"{lines_read} == 4"

    asyncio.run( main_simple() )
    lines_read = 0

    asyncio.run( main_complex() )
```

8.1.4 [`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 6. 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 into 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.
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.

```
import asyncio

def chunker( size ):

    async def process( q_in, q_out ):
        pass

    return process

async def main():
    sink_done = False

    async def source( q_out ):
        await q_out.put( 'hello ' )
        await q_out.put( 'world' )
        await q_out.put( None )

    async def check( pipe, expect ):
        x = await pipe.get()
        assert x == expect, f"{x} == {expect}"

    async def sink_4( q_in ):
        nonlocal sink_done
        await check( q_in, 'hell' )
        await check( q_in, 'o wo' )
        await check( q_in, 'rld' )
        await check( q_in, None )
        sink_done = True

    async def sink_2( q_in ):
        nonlocal sink_done
        await check( q_in, 'he' )
        await check( q_in, 'll' )
        await check( q_in, 'o ' )
        await check( q_in, 'wo' )
        await check( q_in, 'rl' )
        await check( q_in, 'd' )
        await check( q_in, None )
        sink_done = True

    def pipeline( *elements ):
        q_out = asyncio.Queue( 1 )
        line = [ elements[ 0 ]( q_out ) ]
        for e in elements[ 1 : -1 ]:
            q_in = q_out
            q_out = asyncio.Queue( 1 )
            line.append( e( q_in, q_out ) )
        line.append( elements[ -1 ]( q_out ) )
        return line

    async def run( *pipe ):
        nonlocal sink_done
        sink_done = False
        await asyncio.gather( *pipeline( *pipe ) )
        assert sink_done

    await run( source, chunker( 4 ), sink_4 )
    await run( source, chunker( 2 ), chunker( 4 ), sink_4 )
    await run( source, chunker( 7 ), chunker( 4 ), sink_4 )
    await run( source, chunker( 7 ), chunker( 2 ), sink_2 )
    await run( source, chunker( 4 ), chunker( 2 ), sink_2 )
    await run( source, chunker( 3 ), chunker( 2 ), sink_2 )

def test_main():
    asyncio.run( main() )
```

8.1.5 [`minilisp`] Write an asynchronous parser for a very limited subset of the hw3 lisp grammar. 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, so if you normally use annotations, it's quite okay to skip them 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.
Note: This exercise is designed to help you in adapting your hw3 parser for use with hw4 (i.e. in making it asynchronous). Writing a predictive (LL) parser for the hw3 grammar is easy and worth a shot even if you originally wrote something else.

```
import asyncio

async def minilisp( reader ):
    pass

import os

async def main():
    loop = asyncio.get_running_loop()
    r_fd, w_fd = os.pipe()
    w_file = os.fdopen( w_fd, 'w' )
    r_stream = asyncio.StreamReader()
    await loop.connect_read_pipe( lambda:
            asyncio.StreamReaderProtocol( r_stream ),
            os.fdopen( r_fd ) )

    def send( data ):
        w_file.write( data )
        w_file.flush()

    async def check( *expect ):
        expect = list( expect )
        got = await minilisp( r_stream )
        assert got == expect, f"{got} == {expect}"

    send( '(hello)' )
    await check( 'hello' )
    send( '(hello world)' )
    await check( 'hello', 'world' )
    send( '(hello (world))' )
    await check( 'hello', [ 'world' ] )
    send( '((hello) (cruel (or not) world))' )
    await check( [ 'hello' ],
            [ 'cruel', [ 'or', 'not' ], 'world' ] )
```

```
def test_main():
    asyncio.run( main() )
```

Part 9: Programming Exercises

This week will consist of a few generic exercises, since the lecture does not cover a specific topic.

1. `alchemy.py` – transmute and mix inputs to reach a goal
2. `minieval.py` – evaluate simple lisp-y expressions
3. `word-chain.py` – solve a simple puzzle
4. `reduce.py` – reduce composite values with a bit of Python magic
5. `cycles.py` – a simple graph algorithm

Submit `minieval`, `alchemy` and/or `cycles` for 0.5 points each.

Part 10: Testing

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.
Unlike other weeks, you won't get pre-made tests, since writing tests is the point of the exercises. To get started, try e.g. `https://hypothesis.readthedocs.io/en/latest/quickstart.html`.

We will look at two types of programs to use hypothesis with, first some integer and floating-point linear math:

1. `inner.py` – properties of the inner vector product
2. `cross.py` – same but cross product

And some classic computer science problems:

4. `sort.py` – sorting algorithms
5. `bsearch.py` – binary search
6. `heap.py` – binary heaps

Part 11: Numeric Math

Since Python is often used as a driver for numeric algorithms, with `numpy` as the backend, we will try to explore some of its basic functions. The first part will focus on linear algebra.

1. `linear.py` – warm up
2. `volume.py` – volume of n-dimensional polyhedra
3. `tbd`

While in the second we will look at signal processing.

4. `histogram.py` – drawing histograms
5. `sig.py` – generating signals
6. `dft.py` – analysing signals

Part 12: Statistics

TBD.