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	2.12.	9.12.	16.12.	23.12.	30.12.
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	2.12.	6.1.	13.1.	20.1.	27.1.
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	2.12.	16.12.
10	9.12.	23.12.
11	16.12.	30.12.
12	23.12.	6.1.

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)
6. `json_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
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.

4. list_make.py – creating shopping lists
5. list_search.py – searching and reading
6. list_update.py – update and delete

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
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
7. noexcept.py – decorators and exceptions

Part 6: Closures and Coroutines

There will be two sets of exercises, one for closures and another for coroutines. The former is rather general, since closures work more or less the same in all languages. The latter, however,, will mainly look at low-level aspects of how coroutines are implemented in Python.

1. tbd
2. tbd
3. tbd

4. gen.py
5. interleave.py
6. tbd

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.

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.

Part 9: Programming Exercises

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

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

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

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.