PV248 Python

Petr Ročkai and Zuzana Baranová

Part A: Introduction

This document is a collection of exercises and commented examples of source code (in Python). All of the source code included here is also available as source files which you can edit and directly execute (we will refer to these files as the source bundle). Additionally, this section contains the rules and general guidelines that apply to the course as a whole.
The latest version of this document along with the source bundle is available both in the study materials in IS and on the student server `aisa`:

- `https://is.muni.cz/auth/el/fi/podzim2021/PV248/um/` – a PDF in `pv248.seminar.pdf` and the source bundle in directories `01` through `12`, `t1` through `t4` and `sol` – use the 'download as ZIP' option in the sidebar to get entire directories in one go,
- log into `aisa` using `ssh` or `putty`, run `pv248 update`, then look under `~/pv248` (this chapter is in subdirectory `00`).

We will update the files as needed, to correct mistakes and to include additional material.[1] On `aisa`, running `pv248 update` at any time will update your working copies, taking care not to overwrite your changes. It will also tell you which files have been updated.
Each of the following chapters corresponds to a single week of the semester. The correspondence between exercises and the content of the lectures is, however, somewhat loose, especially at the start of the semester.
NB. If you are going to attend the lectures (you need to enroll separately, subject code is PV288), all you need at the start is intuitive familiarity with common programming concepts like classes, objects, higher-order functions and function closures (which can be stored in variables), as covered in e.g. PB006. You will get all the details that you may need in the lectures. On the other hand, if you are not going to attend lectures, you either need to already know all the theory, or you need to study it in your free time (this subject is purely practical).

Part A.1: Course Overview

Welcome to PV248 Python Seminar.
Since this is a programming subject, most of the coursework – and grading – will center around actual programming. There will be 2 types of programs that you will write in this seminar: tiny programs for weekly exercises (around 10–30 minutes each) and small programs for homework (a few hundred lines and anything from a few hours to a day or two of work).
As you probably know by now, writing programs is hard and this course won't be entirely easy either. You will need to put in effort to pass the subject. Hopefully, you will have learned something by the end of it. Further details on the organisation of this course are in the remaining files in this directory:

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

Study materials for each week are in directories `01` through `12`. Start by reading `00_intro.txt`. Assignments are in directories `t1` through `t4`, one for each 4-week block and one extra for the exam period. They will be made available according to the schedule shown in `2_grading.txt`. The exercises for any given week will make use of the material covered in the lecture, though some weeks it will be a fairly loose fit. Especially when the lecture material is broad (like in weeks 1, 2 and 5), the seminar will mainly include general programming exercises. Topics will get more specific and focused as the semester progresses. The lectures are divided into 3 blocks, 4 lectures each. They will cover the following topics:

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

Part A.2: Grading

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Part A.3: Tasks

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

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

A.3.2 Evaluation There are three sets of automated tests which are executed on the solutions you submit:

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

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

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

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

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

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

```
$ pv248 status
```

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

```
$ pv248 diff
```

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

A.3.4 Guidelines The general principles outlined here apply to all assignments. The first and most important rule is, use your brain – the specifications are not exhaustive and sometimes leave room for different interpretations. Do your best to apply the most sensible one. Do not try to find loopholes (all you are likely to get is failed tests). Technically correct is not the best kind of correct.
Think about pre- and postconditions. Aim for weakest preconditions that still allow you to guarantee the postconditions required by the assignment. If your preconditions are too strong (i.e. you disallow inputs that are not ruled out by the spec) you may fail the tests.
Do not print anything that you are not specifically directed to. Programs which print anything that wasn't specified will fail tests.
You can use the standard library. Third-party libraries are not allowed, unless specified as part of the assignment. Make sure that your classes and methods use the correct spelling, and that you accept and/or return the correct types. In most cases, either the 'syntax' or the 'sanity' test suite will catch problems of this kind, but we cannot guarantee that it always will – do not rely on it.
If you don't get everything right the first time around, do not despair. The expectation is that most of the time, you will pass on the second or third attempt. In the real world, the first delivered version of your product will rarely be perfect, or even acceptable, despite your best effort to fulfill every customer requirement. Only very small programs can be realistically written completely correctly in one go.

Part 1: Python 101

As we have mentioned, each chapter is split into 3 sections: demonstrations, preparatory exercises and regular exercises. The demos are complete programs with comments that should give you a quick introduction to using the constructs that you will need in the actual exercises. The demos for the first week are these:

1. `list` – using lists
2. `dict` – using dictionaries
3. `str` – using strings

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

1. `fibfib` – iterated Fibonacci sequence

The second set of exercises are those that are meant to be solved before the corresponding seminar. The first set should be submitted by 18th of September (you need to solve at least 3 of the exercises). The corresponding seminars are in the week starting on 20th of September. Now for the exercises:

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

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

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

Part 1.d: Demonstrations

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

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

Lists are mutable: the value of a list may change, without the list itself changing identity. Methods like append and operators like `+=` update the list in place.
Lists are internally implemented as arrays. Appending elements is cheap, and so is indexing. Adding and removing items at the front is expensive. Lists are indexed using (again) square brackets and indices start from zero:

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

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

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

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

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

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

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

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

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

1.d.2 [`dict`] Dictionaries (associative arrays) are another basic (and very useful) data structure. Literals are written using curly braces, with colons separating keys from values and commas separating multiple key-value pairs from each other:

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

In Python, dictionaries are implemented as hash tables. This gives constant expected complexity for most single-item operations (insertion, lookup, erase, etc.). One would expect that this also means that dictionaries are unordered, but this is not quite so (details some other day, though).
Like lists, dictionaries are mutable: you can add or remove items, or, if the values stored in the dictionary are themselves mutable, update those. However, keys cannot be changed, since this would break the internal representation. Hence, only immutable values can be used as keys (or, to be more precise, only 'hashable' values).
Most operations on items in the dictionary are written using subscripts,

like with lists. Unlike lists, the keys don't need to be integers, and if they are integers, they don't need to be contiguous. To update a value associated with a key, use the assignment syntax:

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

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

```
a_list = []

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

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

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

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

1.d.3 **[str]** The last data type we will look at for now is `str`, which represents Unicode strings. Unlike lists and dictionaries, but quite like integers, strings in Python are immutable. You can construct new strings from old strings, but once a string exists, it cannot be updated. There are many kinds of string literals in Python, some of them quite complicated. The basic variations use single or double quotes (and there is no difference between them, though some programmers give them different semantics).

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

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

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

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

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

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

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

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

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

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

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

Of course, you can also break out the web browser and point it to https://docs.python.org/3.

Part 1.e: Elementary Exercises

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

```
s[0] = 1 1 2 3 5 8 13 21 …
s[1] = 1 1 1 2 5 21 233 10946 …
s[2] = 1 1 1 1 5 10946 2.2112·10^48 1.6952·10^2287 …
s[3] = 1 1 1 1 5 1.6952·10^2287 …
```

More generally:

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

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

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

Part 1.p: Prep Exercises

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

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

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

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

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

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

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

Find the image of the set represented by `domain` under `f`, as a list of dis-

joint, closed intervals, sorted in ascending order. Produce the shortest list possible.
Values which are not in the image must not appear in the result (e.g. if the image is $1, 2, 4$, the intervals would be $(1, 2), (4, 4)$ – not $(1, 4)$ nor $(1, 1), (2, 2), (4, 4)$).

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

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

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

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

```
def ts3_escape( string ):
    pass

def ts3_unescape( string ):
    pass
```

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

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

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

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

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

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

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

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

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

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

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

Part 1.r: Regular Exercises

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

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

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

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

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

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

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

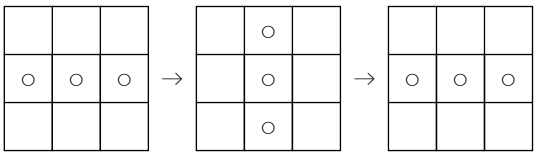
And decipher the text back to the sent message.

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

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

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

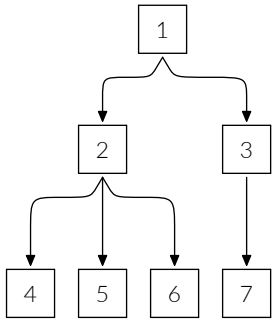
An example of a short periodic game:



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

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

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



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

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

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

```
def breadth(tree):
    pass

from math import isclose
```

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

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

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

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

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

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

Part 2: Objects, Classes and Types

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

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

Prep exercises:

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

Regular exercises:

1. `json` – recursive data types without gross hacks

2. `treezip` – a zipper on binary trees
3. `ts3bugs` – more fun with template system 3
4. `treap` – randomized search trees
5. `distance` – shortest distance between two 2D objects
6. `cycle` – finding cycles in object graphs

Part 2.d: Demonstrations

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

```
def divisor_count( n: int ) -> int:
    count = 0
    for i in range( 1, n + 1 ):
        if n % i == 0:
            count += 1
    return count

assert divisor_count( 5 )  == 2 # 1 and 5
assert divisor_count( 6 )  == 4 # 1, 2, 3 and 6
assert divisor_count( 12 ) == 6 # 1, 2, 3, 4, 6 and 12
```

For more complex types, we need to import some helper classes from module `typing`. Some of them (e.g. `List`, `Set`, `Dict` or `Tuple`) are no longer needed in Python 3.9, since the built-in types can be used directly. Many of these types are generic, i.e. they have one or more type parameters. You know these from Haskell (they are everywhere) or perhaps C++/Java/C# (templates and generics, respectively). Like in Haskell but unlike C++, generic types have no effect on the code itself – they are just annotations. Type parameters are given in square brackets after the generic type.

```
from typing import List

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

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

```
    res: List[ int ] = []

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

    return res

assert divisors( 5 )  == [ 1, 5 ]
assert divisors( 6 )  == [ 1, 2, 3, 6 ]
assert divisors( 12 ) == [ 1, 2, 3, 4, 6, 12 ]
```

Part 2.e: Elementary Exercises

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

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

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

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

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

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

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

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

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

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

Part 2.p: Prep Exercises

2.p.1 **[dsw]** Implement the DSW (Day, Stout and Warren) algorithm for rebalancing binary search trees. The algorithm is 'in place' – implement it as a procedure that modifies the input tree. You will find suitable pseudocode on Wikipedia, for instance.

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

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

2.p.2 **[ts3norm]** (continued from `01/p3_ts3esc`) Eventually, we will want

to replicate the actual substitution into the templates. This will be done by the ts3_render function (next exercise). However, somewhat surprisingly, that function will only take one argument, which is the structured document to be converted into a string. Recall that the template system is recursive: before ts3_render, another function, ts3_combine combines the document and the templates into a single tree-like structure. One of your less fortunate colleagues is doing that one.

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

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

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

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

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

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

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

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

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

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

- Document → replace the ${…} with the text of the document; the pasted text is excluded from further processing,
- Template → the ${…} is replaced with the text of the template; occurrences of ${…} and #{…} within the pasted text are further processed,
- int → it is formatted as a decimal number and the resulting string replaces the ${…},
- list → the length of the list is formatted as if it was an int, and finally,
- dict → .default is appended to the path and the substitution is retried.

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

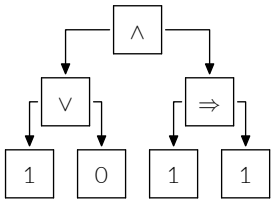
- a dict is rendered as a comma-separated (with a space) list of its values, after the keys are sorted alphabetically, where each value is rendered as a scalar,
- a list is likewise rendered as a comma-separated list of its values as scalars,
- everything else is an error: again, treat this as a failed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

The top-level entity passed to ts3_render must always be a dict. The starting template is expected to be in the key $template of that dict. Remember that ##{…} and $${…} must remain untouched. If you encounter nested templates while parsing the path, e.g. ${abc${d}}, give up (again via a failed assertion); however, see also exercise r3).

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

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

Example of a boolean tree:



In this case the or (∨) node evaluates to True, the implication (⇒) evaluates to True as well, and hence the whole tree (and, ∧) also evaluates to True.

Add methods and attributes to Node and Leaf as/if needed.

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

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

Complete the following classes as appropriate.

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

2.p.5 [intersect] We first import all the classes from e1_geometry, since we will want to use them.

What we will do now is compute intersection points of a few object type combinations. We will start with lines, which are the simplest. You can find closed-form general solutions for all the problems in this exercise on the internet. Use them.

Line-line intersect either returns a points, or a Line, if the two lines are coincident, or None if they are parallel.

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

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

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

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

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

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

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

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

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

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

```
class Linked: pass
```