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	16:00	18:00	(Lukáš Korenčik)
Tue	18:00	20:00	Petr Ročkai
Thu	16:00	18:00	Vladimír Štill
Fri	16:00	18:00	Petr Ročkai

Please note that the Tuesday 16:00 slot is a backup, in case there is too much demand in the three remaining slots. Please only use it if you didn't manage to get a word in at other times. It will be either cancelled or made permanent after a few weeks, depending on traffic in the remaining slots.
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`.

```
def test_main():

    rpn_num = [ 5 ]
    assert rpn_eval( rpn_num ) == [ 5 ]

    rpn_neg = [ 1, "neg" ]
    assert rpn_eval( rpn_neg ) == [ -1 ]

    rpn_rec = [ 2, "recip" ]
    assert rpn_eval( rpn_rec ) == [ 1/2 ]

    rpn_n = [ -1/7, "recip" ]
    assert rpn_eval( rpn_n ) == [ -7 ]

    rpn_simp = [ 1, "recip", "neg" ]
    assert rpn_eval( rpn_simp ) == [ -1 ]

    rpn = [ 4, "neg", "recip", "neg", "neg", "recip", "neg",
            "recip", "recip" ]
    assert rpn_eval( rpn ) == [ 4 ]

    rpn_nums = [ 5, 1/9, "recip", 2, "neg", "recip", -1, "neg" ]
    assert rpn_eval( rpn_nums ) == [ 5, 9, -1/2, 1 ]
```

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`.

```
def test_main():

    rpn = [ 2, -2, '+' ]
    assert rpn_eval( rpn ) == [ 0 ]
```

```
rpn = [ 3, 7, '*' ]
assert rpn_eval( rpn ) == [ 21 ]

rpn = [ 8, 2, "recip", '/' ]
assert rpn_eval( rpn ) == [ 16 ]

rpn = [ -1, 3, '-', 2, '+', 4, "neg", 2, '**' ]
assert rpn_eval( rpn ) == [ -2, 16 ]

rpn = [ 3, -1, 9, '*', 22, 100, "neg", "sum" ]
```

[3, -9, 22, -100, sum]

```
assert rpn_eval( rpn ) == [ -84 ]
```

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`.

```
from functools import reduce
from operator import add

def test_main():
    e = Evaluator()
    e.add_op( '*', 2, lambda x, y: x * y )
    e.add_op( '+', 2, lambda x, y: x + y )
    assert e.evaluate( [ 1, 2, '+', 7, '*' ] ) == [21]

    e.add_op( 'neg', 1, lambda x: -x )
    assert e.evaluate( [ 3, 'neg' ] )  == [ -3 ]

    e.add_op( 'four', 4, lambda a, b, c, d: a - b * c + d )
    e.add_op( 'second', 5, lambda a, b, c, d, e: b )
```

The following test case should evaluate as follows: `[ 2, 4, 7, 'neg', 8, 'four' ]` → `[ 2, 4, -7, 8, 'four' ]` → `2 - 4 * -7 + 8`.

```
    assert e.evaluate( [ 1, 2, 3, 4, 5, 'second', 4, 7, 'neg',
                         8, 'four' ] ) == [ 38 ]
```

```
def test_bonus_1():
    e = Evaluator()
    e.add_op( 'sum', 0, lambda *x: reduce( add, x ) )
    assert e.evaluate( [ 2, -3, 4, 9, 'sum' ] ) == [ 12 ]
```

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.

```
def test_main():
    test_point()
    test_vector()
    test_line()
    test_segment()
    test_circle()

def point_eq( p1, p2 ):
    return p1.x == p2.x and p1.y == p2.y

def test_point():
    p1 = Point( 1, -1 )
    p2 = Point( -7, 2 )

    assert point_eq( p2 - p1, Point( -8, 3 ) )
    assert point_eq( p1 - p2, Point( 8, -3 ) )
```

check that it did not affect original points

```
    assert point_eq( p1, Point( 1, -1 ) )
    assert point_eq( p2, Point( -7, 2 ) )

    v_0 = Vector( 0, 0 )
    assert point_eq( p1.translated( v_0 ), p1 )

    v_24 = Vector( 2, 4 )
    assert point_eq( p1.translated( v_24 ), Point( 3, 3 ) )
    assert point_eq( p1, Point( 1, -1 ) )  # remains unaffected

def test_vector():
    v1 = Vector( 2, 7 )
    v2 = Vector( -5, 0 )

    assert isclose( v1.length(), 7.28010988928 )
    assert isclose( v2.length(), 5 )

    assert v1.dot( v2 ) == -10
    assert isclose( v1.angle( v2 ), 1.8490959858 )

def test_line():
    p1 = Point( 2, -1 )
    p2 = Point( 3, 4 )
    ln = Line( p1, p2 )

    ln_t = ln.translated( Vector( -2, -2 ) )
    p1_t, p2_t = ln_t.point_point()
    assert point_eq( p1_t, Point( 0, -3 ) )
    assert point_eq( p2_t, Point( 1, 2 ) )

    p1_t, v_t = ln_t.point_vector()
    assert point_eq( p1_t, Point( 0, -3 ) ) or point_eq( p1_t, Point(
1, 2 ) )
    assert isclose( v_t.length(), 5.0990195135927845 )
    assert ( v_t.x == -1 and v_t.y == -5 ) or ( v_t.x == 1 and v_t.y
== 5 )
```

Test line equality.

```
    assert ln == ln
```

Parallel lines.

```
    l1 = Line( Point( 2, 0 ), Point( 3.5, -3 ) )
    l2 = Line( Point( 5, 2 ), Point( 7, -2 ) )
    assert l1 != l2
```

l1 represented by different points

```
    l2 = Line( Point( 1.5, 1 ), Point( -1, 6 ) )
    assert l1 == l2
```

Intersecting lines.

```
    l2 = Line( Point( -3, 2 ), Point( 1, 9 ) )
    assert l1 != l2

def test_segment():
    p1 = Point( 2, -1 )
    p2 = Point( 3, 4 )
    sg = Segment( p1, p2 )

    sg_t = sg.translated( Vector( -1, 3 ) )
    assert sg.length() == sg_t.length()
    p1_t, p2_t = sg_t.point_point()
    assert point_eq( p1_t, Point( 1, 2 ) )
    assert point_eq( p2_t, Point( 2, 7 ) )

def test_circle():
    c = Circle( Point( 1, -1 ), 4 )
    assert point_eq( c.center(), Point( 1, -1 ) )
    assert c.radius() == 4

    c_t = c.translated( Vector( -11, -3 ) )
    assert point_eq( c_t.center(), Point( -10, -4 ) )
    assert c_t.radius() == 4

    assert point_eq( c.center(), Point( 1, -1 ) )
    assert c.radius() == 4
```

Since we will want to import this file into the next two exercises, we use the 'current module is the main program' trick below, which prevents the test code from running on import.

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`.

```
def test_main():
    test_line_line()
    test_line_segment()
    test_line_circle()
    test_intersect()

def test_line_line():

    l1 = Line( Point( 2, 1 ), Point( -3, 7 ) )
    l_i = intersect_line_line( l1, l1 )
    assert type( l_i ) == Line
    assert l_i == l1
```

Same as `l1`, but represented using different points.

```
    l2 = Line( Point( -0.5, 4 ), Point( 7, -5 ) )
    l_i = intersect_line_line( l1, l2 )
    assert type( l_i ) == Line
    assert l_i == l1
    assert l_i == l2

    l3 = Line( Point( 2, 2 ), Point( -1, 4 ) )
    for line in [ l1, l2 ]:
        points = intersect_line_line( line, l3 )
        assert len( points ) == 1
        p = points[ 0 ]
        assert isclose( p.x, 0.125 )
        assert isclose( p.y, 3.25 )
```

Parallel lines.

```
    l1 = Line( Point( 1, 1 ), Point( 3, 5 ) )
    l2 = Line( Point( 6, 4 ), Point( 7, 6 ) )
    assert intersect_line_line( l1, l2 ) == []

def test_line_segment():
```

Segment which lies on a line.

```
    l = Line( Point( -2, -3 ), Point( -1, -2 ) )
    s = Segment( Point( 3, 2 ), Point( 5, 4 ) )
    assert intersect_line_segment( l, s ) == s
```

Line which crosses a segment.

```
    s = Segment( Point( -1, -5 ), Point( -4, -2 ) )
    points = intersect_line_segment( l, s )
    assert len( points ) == 1
    p = points[ 0 ]
    assert isclose( p.x, -2.5 )
    assert isclose( p.y, -3.5 )
```

Line crosses the line in which a segment lies, but not the segment itself.

```
    s = Segment( Point( -5, -1 ), Point( -4, -2 ) )
    assert intersect_line_segment( l, s ) == []
```

A line parallel to a segment.

```
    s = Segment( Point( 1, -2 ), Point( 2, -1 ) )
    assert intersect_line_segment( l, s ) == []

def test_line_circle():
```

A tangent line.

```
    l = Line( Point( 0, 5 ), Point( 3, 5 ) )
    c = Circle( Point( 3, 3 ), 2 )
    res = intersect_line_circle( l, c )
    assert len( res ) == 1
    assert isclose( res[0].x, 3 )
    assert isclose( res[0].y, 5 )
```

Line which crosses a circle.

```
    l = Line( Point( 0, 3 ), Point( 7, 3 ) )
    res = intersect_line_circle( l, c )
    assert len( res ) == 2
    p1, p2 = res[0], res[1]
    assert ( isclose( p1.x, 1 ) and isclose( p2.x, 5 ) ) or \
           ( isclose( p2.x, 1 ) and isclose( p1.x, 5 ) )
    assert isclose( p1.y, 3 )
    assert isclose( p2.y, 3 )
```

No intersection.

```
    l = Line( Point( 6, -1 ), Point( 8, 3 ) )
    assert intersect_line_circle( l, c ) == []

def test_intersect():
```

Circle with a line, swapped order.

```
    l = Line( Point( 1, 3 ), Point( -1, -3 ) )
    c = Circle( Point( 2, 0 ), 3 )
    res = sorted( intersect( c, l ), key=lambda point : point.x )
    p1_exp = Point( -0.5348469228349533, -1.6045407685048603 )
    p2_exp = Point( 0.9348469228349539, 2.80454076850486 )

    assert isclose( res[0].x, p1_exp.x ) and isclose( res[0].y,
p1_exp.y )
    assert isclose( res[1].x, p2_exp.x ) and isclose( res[1].y,
p2_exp.y )
```

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
```

Probably time for some testcases. That wraps up the seminar for today.

```
from math import isclose

def test_main():
    test_point_point()
    test_point_line()
    test_line_line()
    test_point_circle()
    test_line_circle()
```

```
    test_distance()

def test_point_point():
    p1 = Point( 9, 7 )
    p2 = Point( 3, 2 )
    assert isclose( distance_point_point( p1, p2 ), 7.81024967590665 )

def test_point_line():
    p = Point( 2 , -1 )
    l = Line( Point( 3, 6 ), Point( -4, -2 ) )
    assert isclose( distance_point_line( p, l ), 3.85695556037274 )

def test_line_line():
    l1 = Line( Point( -3, -6 ), Point( 3 , 1 ) )
    l2 = Line( Point( 3, 6 ), Point( -3 , -1 ) )
    assert isclose( distance_line_line( l1, l2 ), 3.25395686727984 )

def test_point_circle():
```

point outside circle

```
    p = Point( 0, -2 )
    c = Circle( Point( 2, 9 ), 2 )
    assert isclose( distance_point_circle( p, c ), 9.18033988749894 )
```

point within circle

```
    p = Point( 3, 2 )
    c = Circle( Point( 2, 5 ), 4 )
    assert isclose( distance_point_circle( p, c ), 0.83772233983162 )
```

point on circle

```
    p = Point( 0, 1 )
    c = Circle( Point( 0, 5 ), 4 )
    assert isclose( distance_point_circle( p, c ), 0 )

def test_line_circle():
    l = Line( Point( 1, -3 ), Point( 2, -1 ) )
    c = Circle( Point( 2, 7 ), 2 )
    assert isclose( distance_line_circle( l, c ), 1.57770876399966 )

def test_distance():
    p1 = Point( 9, 7 )
    p2 = Point( 3, 2 )
    assert isclose( distance( p1, p2 ), 7.81024967590665 )

    p = Point( 3, 2 )
    c = Circle( Point( 2, 5 ), 4 )
    assert isclose( distance( c, p ), 0.83772233983162 )
```