Python solution manual: 2017-2018 edition

Sam van Elsloo

June-July 2018
Version 1.4

Contents

1 Preface **5**
- 1.1 Changelog . . . 5
  - 1.1.1 Version 1.4 . . . 5
  - 1.1.2 Version 1.3 . . . 5
  - 1.1.3 Version 1.2 . . . 5
  - 1.1.4 Version 1.1 . . . 5
- 1.2 Changes between Python2 and Python3 . . . 6
- 1.3 General exam strategy . . . 6
- 1.4 Next year . . . 7

2 June 2016 exam **11**
- 2.1 Part I: what will this print? . . . 11
  - 2.1.1 Question 1 . . . 11
  - 2.1.2 Question 2 . . . 11
  - 2.1.3 Question 3 . . . 11
  - 2.1.4 Question 4 . . . 11
  - 2.1.5 Question 5 . . . 12
  - 2.1.6 Question 6 . . . 12
- 2.2 Part II: debug the program . . . 12
  - 2.2.1 Question 7 . . . 12
  - 2.2.2 Question 8 . . . 14
  - 2.2.3 Question 9 . . . 14
  - 2.2.4 Question 10 . . . 14
- 2.3 Part III: complete the program . . . 14
  - 2.3.1 Question 11 . . . 14
- 2.4 Part IV . . . 14
  - 2.4.1 Question 12 . . . 15
  - 2.4.2 Question 13 . . . 15
  - 2.4.3 Question 14 . . . 16
  - 2.4.4 Question 15 . . . 17

3 Exam August 2016 **19**
- 3.1 Part I: What will this print? . . . 19
  - 3.1.1 Question 1 . . . 19
  - 3.1.2 Question 2 . . . 20
  - 3.1.3 Question 3 . . . 20
  - 3.1.4 Question 4 . . . 21
  - 3.1.5 Question 5 . . . 21
  - 3.1.6 Question 6 . . . 21
- 3.2 Part II: Debug the program . . . 22
  - 3.2.1 Question 7 . . . 22
  - 3.2.2 Question 8 . . . 22
  - 3.2.3 Question 9 . . . 22
- 3.3 Part III: complete . . . 22
  - 3.3.1 Question 10 . . . 22
- 3.4 Part IV: write . . . 23
  - 3.4.1 Question 11 . . . 23
  - 3.4.2 Question 12 . . . 24
  - 3.4.3 Question 13 . . . 24
  - 3.4.4 Question 14 . . . 25

4 Exam August 2015 **28**
4.1 Part I: Reading Python . . . 28
4.1.1 Question 1 . . . 28
4.1.2 Question 2 . . . 28
4.1.3 Question 3 . . . 28
4.1.4 Question 4 . . . 28
4.1.5 Question 5 . . . 29
4.1.6 Question 6 . . . 29
4.2 Part II: Debugging Python . . . 29
4.2.1 Question 7 . . . 29
4.2.2 Question 8 . . . 29
4.2.3 Question 9 . . . 30
4.3 Part III: Writing Python . . . 30
4.3.1 Question 10 . . . 30
4.3.2 Question 11 . . . 30
4.3.3 Question 12 . . . 31
4.3.4 Question 13 . . . 31
4.3.5 Question 14 . . . 32

5 Exam April 2012 **34**
5.1 Part I: Reading Python . . . 34
5.1.1 Question 1 . . . 34
5.1.2 Question 2 . . . 34
5.1.3 Question 3 . . . 34
5.1.4 Question 4 . . . 34
5.2 Part II: Debugging Python . . . 35
5.2.1 Question 5 . . . 35
5.2.2 Question 6 . . . 35
5.2.3 Question 7 . . . 35
5.2.4 Question 8 . . . 36
5.3 Part III: supplement Python . . . 36
5.3.1 Question 9 . . . 36

6 July 2014 **39**
6.1 Part I: Reading Python . . . 39
6.1.1 Question 1 . . . 39
6.1.2 Question 2 . . . 39
6.1.3 Question 3 . . . 39
6.1.4 Question 4 . . . 39
6.1.5 Question 5 . . . 40
6.1.6 Question 6 . . . 40
6.2 Part II: Debugging Python . . . 40
6.2.1 Question 7 . . . 40
6.2.2 Question 8 . . . 40
6.2.3 Question 9 . . . 41
6.2.4 Question 10 . . . 41

7 January 2012 **43**
7.1 Part I: Reading Python . . . 43
7.1.1 Question 1 . . . 43
7.1.2 Question 2 . . . 43
7.1.3 Question 3 . . . 43
7.1.4 Question 4 . . . 43
7.1.5 Question 5 . . . 43
7.1.6 Question 6 . . . 44
7.2 Part II: Debugging Python . . . 44
7.2.1 Question 7 . . . 44

7.2.2 Question 8 . . . 44
7.2.3 Question 9 . . . 44

8 July 2017 **46**
8.1 Part I: What will this print? . . . 46
8.1.1 Question 1 . . . 46
8.1.2 Question 2 . . . 46
8.1.3 Question 3 . . . 46
8.1.4 Question 4 . . . 47
8.1.5 Question 5 . . . 47
8.2 Part II: Debug the program . . . 47
8.2.1 Question 6 . . . 47
8.2.2 Question 7 . . . 47
8.2.3 Question 8 . . . 47
8.3 Part III: Complete the program . . . 48
8.4 Part IV . . . 48
8.4.1 Question 10 . . . 48
8.4.2 Question 11 . . . 49
8.4.3 Question 12 . . . 50
8.4.4 Question 13 . . . 52

9 August 2017 **56**
9.1 Part I: What will this print? . . . 56
9.1.1 Question 1 . . . 56
9.1.2 Question 2 . . . 56
9.1.3 Question 3 . . . 56
9.1.4 Question 4 . . . 57
9.1.5 Question 5 . . . 57
9.1.6 Question 6 . . . 57
9.2 Part II: Debug the programs . . . 57
9.2.1 Question 7 . . . 57
9.2.2 Question 8 . . . 58
9.2.3 Question 9 . . . 58
9.3 Part III: Complete the program . . . 58
9.3.1 Question 10 . . . 58
9.4 Part IV: Write program . . . 58
9.4.1 Question 11 . . . 58
9.4.2 Question 12 . . . 59
9.4.3 Question 13 . . . 60

10 Exam June 2015 **63**
10.1 Part I: what will this print? . . . 63
10.1.1 Question 1 . . . 63
10.1.2 Question 2 . . . 63
10.1.3 Question 3 . . . 63
10.1.4 Question 4 . . . 63
10.1.5 Question 5 . . . 63
10.1.6 Question 6 . . . 64
10.2 Part II: Debug the program . . . 64
10.2.1 Question 7 . . . 64
10.2.2 Question 8 . . . 64
10.2.3 Question 9 . . . 64
10.2.4 Question 10 . . . 64
10.3 Part III: complete . . . 64
10.3.1 Question 11 . . . 64
10.4 Part IV: write . . . 65
10.4.1 Question 12 . . . 65

10.4.2 Question 13 65
10.4.3 Question 14 65
10.4.4 Question 15 67

1 Preface

In this dropbox folder you'll find a bunch of old exams, together with solution manuals for them. I basically had to rewrite all of the solutions this year (as in, June 2018) because they were rather trash since I was pretty bad at Python back in the day (the only reason I passed was cause I just practised old exams endlessly, so I ended up actually getting a better grade than a lot of people who were programming gods[1]. At the same time, I added the solutions to the 2016 and 2017 exams so that you have access to even more solutions. Of course, I've verified all solutions by actually programming them in Python so they're pretty credible.

Also, if you're like, what's up with those pages in between all of the chapters: it's the last solution manual of the first year and it's almost Summer break[2], we're all longing for holidays and I know how hard it can be to stay motivated, especially if it's Python, so I figured why not have something nice to look at?

1.1 Changelog

1.1.1 Version 1.4

- Finished the June 2015 exam.

The summary is now completely finished, so best of luck with the exam tomorrow and have a great Summer break. Just one small request though: please don't expect replies to your emails during the first few weeks of the holiday break in case you need to resit something. Especially not when you email me on my birthday then I just hate you.

1.1.2 Version 1.3

- Added the August 2017 exam.
- Added the first half of the June 2015 exam; I'll upload the rest of it as early as possible on Wednesday (I hope somewhere in the morning).

1.1.3 Version 1.2

- Added the July 2017 exam. I'll still add the August 2017 exam and the July 2015 exam, which I'll do asap.

Also, of course thank you very much to the people who chimed in for the gift I really appreciated it.

1.1.4 Version 1.1

- Added the April 2012, July 2014 and January 2012 exams. Note: I still have to add the final question of the January 2012 exam; I will do this next Tuesday. Furthermore, I will still add the solution manuals of the July and August 2017 exams; however, due to lack of time, I'll also only add those on Tuesday. They are definitely the hardest of the old exams though, so it's a good final practice for the exam on Thursday (which is also why I didn't include them at the beginning of this solution manual).

[1] So also as a note of hope if you think you're shit at programming: practising old exams is really the way to go for this exam, and it can salvage your grade even if you're really shit. But if you managed to do the bonus assignments (even if it was with a lot of help (and copying) from other people, you'll be fine if you study. Although that holds for pretty much any course in this bachelor.

[2] But to be fair, it's not nearly as bad as having the final exam before the Summer break, on the final Friday afternoon of the year, on your birthday, which is what I personally had to endure last year. Trust me that's horrible.

1.2 Changes between Python2 and Python3

In addition to the old solution manual being utter trash, a reason to completely start over with writing solutions was that since this year, they teach Python3 rather than Python2. There are some minor differences between them; the most important of them are listed below:

- `print()` used to not require brackets, i.e. you could just write `print ‘‘Dynamic TAs are the most amazing guys ever.’’` and it wouldn't give a syntax error. Since the first part of the exam always is questions of “what will this print”, don't be the guy who's like, “lol damn they forget the brackets every time after the `print`, it'll just give a syntax error”. They wouldn't do that for a full exam, let alone would they do it for four years in a row.
- `xrange()` used to exist but sadly it died. In Python2, we used to have `range()` and `xrange()`, where `range()` basically sucked as and `xrange()` was almost always significantly better (especially in terms of program time). Therefore, in Python 3, they removed the Python2 `range()` completely, and renamed `xrange()` to `range()`.
  More specific, the difference between the two was as follows: if you would write `a=range(20,0,-2)` in Python2, it will actually create a list starting at 20, stopping at 0 (so not including 0), with step size of -2, i.e. `[20,18,16,...,4,2]`. This is rather trash if you write stuff like `i in range(1000000)`, as it means it'll automatically create a gigantic list of 1000000 entries, slowing your program down (it's similar to `np.arange()` in this sense). On the other hand, `xrange()` (and thus `range()` in Python3) is lazily evaluated: that is, if you write `xrange(1000000)`, it won't actually do something: it'll remember the initial point, the end point and the step size, but beyond that, it'll only calculate the corresponding value when it actually needs to. Maybe it doesn't make a lot of sense if you don't like Python that much, but it's an important difference that really alleviates the memory usage.
- Python3 saves everything automatically as floats, whereas Python2 would use integers as well. Specifically, the difference occurs in the following:

```
a = 3
b = 2
c = b/a
```

  Python3 behaves like you would expect it to behave: `c` will be equal to 1.5. On the other hand, Python2 saves `a` and `b` as integers, since, well, they are integers. It then performs integer division, which means that the result is rounded *down* to the closest integer, i.e. 1.5 is rounded down to 1 and `c` equals 1. In Python3, this behaviour can be obtained by writing `c=b//a` (note the double backslash); then it'd also save `c` as being equal to 1. In Python2 the problem can be circumvented by saving the numbers as floats, by writing `a=3.` and `b=2.`
  footnoteAs long as at least one of them is a float, it'll take ‘regular’ division rather than integer division. But this is the reason why often in the old exams you'll see things like `25.` (with the dot behind it), just to make sure it's saved as a float rather than an integer and that you'd get integer math rather than ‘regular’ math.. This sometimes will lead to questions having different solutions in Python3 than in Python2, and I'll address them where necessary.

1.3 General exam strategy

Since I know not all of you are very confident for this exam, let me give some general guidelines on how to study for the exam, and how to do the exam itself:

- Part I: What will this print?: this is, in my opinion, the easiest part of the exam. Just try to think like your compute and go through the code line by line, and write out what is happening. Once you've done a few exams you'll get fluent in it yourself, if you're not already. Don't think too much on the exam for these questions, but just play it safe by writing it out.
- Part II: Debug the program: this can be a relatively difficult part of the exam. However, if you practice a lot of exams, you'll get a lot better in recognising what kind of errors you can expect. Furthermore, I can really recommend to actually type the programs over yourself during studying, and changing some stuff around, just to see what you can do before you break the entire thing and how to cause certain errors. On the exam, this part will then not take too long anymore, as you're so experienced in fucking stuff up that

you can immediately recognise when someone else fucks up.

- Part III: Complete the program: depending on what program to complete, this can be super easy to relatively easy. There are often similar lines of code in the program itself that you can sort of copy; you can also look at example codes in the reader to get inspiration on what you would need to write (e.g. when it's questions about `Pygame` stuff, there's a lot of example code in the reader you can use as reference.
- Part IV: Programming Python: this is undoubtedly the hardest part of the exam as it depends very much on the type of programs they want you to program. However, there are a few things to do to make you better at it: first of all, again, do a lot of exam questions. Programming is something you learn by doing, and just making as many exams as possible is a good idea. You'll also realise that some questions are actually pretty similar; sometimes they are almost exactly the same, other times you'll be able to use the same line of thinking. Next to that, I thoroughly recommend you to look at these questions before you start doing parts I-III of the exam. This will allow you think in the back of your mind how to do these questions whilst you keep working on parts I-III, and it can just give you that bit of extra time to come up with the solution to the question. Furthermore, whilst studying, you should identify what kind of questions you find easiest, and do those at first on the exam: personally, I find numerical simulation the easiest so I'd just always do that one first to make sure I get those first. Euler-like questions (but way easier) I also like, but questions about strings I always found really vague so during the exam I gave them the lowest priority; you should have a similar strategy thought out beforehand[3]. Similarly, if there is a question about root-finding: *always* do that one first. Root-finding is honestly ridiculously easy and still worth like 10 points. Finally, please note that your programs don't have to be 'nice': in principle, you can brute-force everything and it's fine. This for example, this means that if a question asks for something like, for what value of `a` is the maximum value of `h` equal to 50: you can write a very fancy program that calculates this automatically by writing a function that solves for `a`. However, you can also just manually try out several values of `a` and manually tweak it to get close to `h`. Therefore, the solutions in this solution manual are often not the nicest way of doing things[4], but rather the fastest, safest way of doing it on the exam.

1.4 Next year

Finally, with the academic year coming to an end rather soon: in case you're interested, next year there'll be summaries and (if useful) solution manuals for each and every course. More importantly, they get noticeably better as well: honestly, there's a night-and-day difference in terms of quality; in fact, the summaries of second year span a massive 2350 pages, or about 600 pages per quarter[5]. Trust me, next year you'll be able to miss all lectures, not look at any of the lecture slides or the book, but just print the summary, do the examples I included, and you'll be ready to do old exams. They're honestly just on another level, although I may be biased.

Therefore, please take my advice *don't buy any of the books next year; they're simply not worth it at all (and it saves you a couple of hundreds of euros you can spend on beer and what not).* The books are almost all bad and the summaries (sammaries whatever you like) just completely replace them as they all have so many examples in there with worked out solutions that you don't need the actual books at all. It's just a giant waste of money.

Specifically, with regards to the summaries first quarter (assuming it stays the same as for us):

- ADSEE-II: it's really intensive in the first three weeks, with 3 to 4 lectures per week. After that, all the lectures are finished already, and you'll have an exam already in week 1.4 (base on the slides of the 11 lectures). There's a complete summary for those slides (around 100 pages long) covering everything what's in there, with nice boxes indicating what the really important things are, so you won't have to study from slides.

[3] But of course, be flexible; if you see on the exam that the question about strings is super easy whereas the numerical simulation is suddenly way more difficult, then do the strings question of course.

[4] So if you're good at programming and think, man this guy sucks at Python, first why do you even use this manual, and second I am capable with Python but writing the most efficient programs isn't a requirement for passing the exam.

[5] There are days when I'm happy if I can write one page per hour, so you can do the math yourself on how much time I had to spend on it last year, over a period of approximately 10 weeks per quarter. And then I still needed time for myself to actually study myself, and time for projects etc., so it shouldn't be too hard to imagine how much sacrifice I had to make to write the summaries. Since it's basically the first thing you guys ask me if you meet me. There's no real way underestimate the amount of work it was.

For your information, the exam only counts for two-thirds of your grade for ADSEE-II. You'll also have to do a group assignment about something related to aircraft, and a group assignment about something related to spacecraft, both due at the end of week 1.8. They're in groups of 5 and you choose the groups yourself (so you can do it with friends etc., which is really nice imo). The assignments are quite doable, but you shouldn't underestimate how much work it is.

- Probability and statistics: you'll have a midterm exam in week 1.5 and a final exam in week 1.9 (both covering one half of the course). For both of them I made a pretty good summary that explains everything well and contains plenty of example questions (both around 70 pages each), so no need to buy the book (there's also a pdf available online that you can find rather easily I think).
- Aerodynamics I: one of the most fun courses of the bachelor in my opinion. Although I'm pretty sure I'm one of the only ones with that opinion, as you kinda have to be Italian to enjoy aerodynamics for some reason. But honestly, aerodynamics is the coolest thing you'll learn about in your bachelor, and although it's very mathematical, it's really fascinating, and really useful too. The second half of the course is given by Marc Gerritsma, which makes it even more amazing (and the first lecturer is also a cool guy (also Italian, like half the aerodynamics department)). I made a summary for it which is really extensive (150+ pages yayyy) and explains everything really well, well enough to use it instead of the book (again, in my opinion at least).
- Differential equations: just one final exam. There's a very good summary for it, which is much, much, much, much better than the book (some stuff isn't even in the book) or the lectures. Only slight issue is that the passing rate was 15% last year[6]. This was partly because they changed the contents of the exam slightly and there was no practice exam available. The actual exam was genuinely ridiculously difficult. It's kinda similar to how in most courses you have a few moments where you're like, okay I kinda get it but please don't let them ask the most fucked up question of this chapter, and it turns out the entire exam is just those kind of questions. It could have been even worse, as the board of examiners already had asked in advance to make the exam easier than it originally was, so you could have just imagined the massacre it would have been if that wouldn't have happened. Nonetheless, there's a glimmer of hope for you guys: first, you peeps will actually have a practice exam available for reference, which is really helpful. Next to that, there's the following: the lecturer is extremely, extremely boring. If you were in E8 in first semester, or in E9 in fourth quarter, you've had the lecturer for calculus or linear algebra. It's genuinely quite sad how no one shows up to the lectures: half the people leave after the first half of the first lecture[7]. After four lectures, only 10 people go to lectures (and to be clear, in 2nd year you don't have split classes anymore, so although 300+ students are supposed to show up, only *ten* students actually show up. And then it's a lecture in lecture room A of the Aula and it's just super awkward). Now, the lecturer is pretty desperate to get people to go the lectures, and in being so desperate, he tells the handful of people who do go what questions will appear on the exam (pretty specifically which topics, so you can easily practice with those questions, and then on the exam only the numbers are different but other than that it's exactly the same). Some people were nice enough to tell those things to me too, so for me the exam was relatively easy (only time-wise it was crammed, but if you practice very similar questions the evening before, the exam goes a lot better), and some of my friends also got a much nicer grade due to me telling them what would be in the exam.
  Now, I don't suggest that *all* of you go to the lectures, cause then the depression rate would just shoot up which is also undesired obviously. Instead, just send a 'sacrifice' to go to the lectures and take note of what he says will be in the exam[8]. Idk, pay the guy some money for his sacrifices or smth. But have something there willing to find out what the exam will be about, it can be really vital.
  Other than that, like I said, there's an outstanding summary, which says specifically for each kind of question how to solve it, with complete step-by-step guides that you just have to remember by heart. It's a really good summary, and hopefully this year the exam will just be a bit easier (for the resit, the passing rate was already 45% so that's better).

One final thing about second year in general: please bear in mind that 2nd year courses, especially in first semester, are just a lot harder than first year courses. All of the 2nd year first semester courses (except ADSEE-II

[6] Which should have been even lower as many people didn't even show up at the exam as it was widely expected to be so bad; if you'd include that you'd probably obtain a pass rate of like 10%. Also nothing was done about it, they didn't compensate the grades at all.

[7] Personally I was able to withstand 1.5 lecture after that I just couldn't handle anymore. You should have been there to understand how bad it was.

[8] Hell, it's pretty clear when he says it: he literally wrote it on the board last year. Like, question 3: Telegraph equation, with square wave as initial condition; question 5: irregular series solution with roots that differ by integer. Although those terms don't mean much to you now (and as a matter of fact, probably never will), it didn't really leave much to the imagination as to what they'd ask on the exam.

and the projects) have passing rates of 50% or less. Getting behind on courses and having to resit courses is also a really difficult thing to make up for without an extraordinary effect. So not to make you panic, but if you plan on finishing this bachelor in three years, you really have to be on top of your stuff pretty quickly in the semester (maybe the first week after the Summer break can be chill, but after that you'll have to start studying). Also because most of the first semester courses kinda leave you alone (e.g. aerodynamics and differential equations are both just one single exam with no special extras), so it may be difficult initially to start studying. But do try at least; and with the quality of the summaries everyone *should* be able to pass the second year courses in one go.

2 June 2016 exam

2.1 Part I: what will this print?

2.1.1 Question 1

This question doesn't really work well anymore in Python3 due to the way `range` works now; instead, you should treat it as if the first line reads `a = np.arange(20,0,-2)`. In that case, line 1 will create `[20 18 16 14 12 10 8 6 4 2]`, as the beginpoint is 20, the step is -2 and it stops at 0 (so does not include 0). Then, line 3 will print all entries of `a`, but with step 2: it will thus skip the second, fourth, sixth, etc. entry. Thus, it will print `[20 16 12 8 4]`.

2.1.2 Question 2

So, let's first look at `chorus`. It first completely copies `lst`. Then, what happens after that? Look carefully at `lst`: it contains of two entries (two strings), separated by a comma, just after I'm okay,". Thus, `lst` has two entries. Then, `chorus` will add another entry to this lest, equal to the first entry in `lst` (`lst[0]`), where "oh, " is replaced by void, and "I'm" is replaced with "he's". `chorus` then adds a final entry to the list, by adding the second line of `lst` (`lst[1]`), with "I sleep" replaced with "he sleeps", and "I work" with "he works".

If you're wondering, but why are those [] placed around the stuff between the plus signs: by calling `lst[0]`, you are only asking for the entry in `lst` itself (in this case the entry is a string). If you wouldn't put the `[]` back around it then again, you would be adding strings to lists, which is like comparing apples to oranges.

Finally, the for loop simply prints each line, with the first character of each line being a capital. Thus, the following is printed:

> Oh I'm a lumberjack, and I'm okay,
> I sleep all night and I work all day.
> He's a lumberjack, and he's okay,
> He sleeps all night and he works all day.

2.1.3 Question 3

Honestly, easiest way to do these kind of questions is by simply writing it out yourself. You have scrap paper, so use it. We start with `n` equal to 0. We then do four loops for `i`, and for each loop, we do three loops (`j=1`, `j=2` and `j=3`) over `j`. In each loop, we increment `n` by one. Evidently, this will thus happen $4 \times 3 = 12$ times, so we'll print 12 as well. Not that difficult. Thus, we print 12.

2.1.4 Question 4

Again, best if you just write out the loop yourself. We start with `va=12` and `vb=20`, so then obviously `va==vb` is False so we'll execute the loop. We now have `va<vb` ($12 < 20$), so we must execute the first statement: `va` is incremented to 24. After that, `va` still does not equal `vb`, so we must again execute the loop. We now have `vb<va`, so we must increment `vb` by 20, ending up at 40. We can make a table of the iterations until `va==vb`, and we'd end up at table 2.1. We see that we end up at `va=60` before the while statement is not satisfied anymore, so we print 60.

Of course, you could have thought of this yourself: we're basically looking for the smallest number that's a multiple of both 12 and 20, and this is obviously 60. However, my experience as your Dynamics TA has been

mostly been that whenever you guys start thinking on your own, it goes horribly wrong. So in general I'd recommend to just write it out like this and just be safe.

Table 2.1: Table of iterations.

Iteration	`va`	`vb`
0	12	20
1	24	20
2	24	40
3	36	40
4	48	60
5	60	60

2.1.5 *Question 5*

Let's just go through this line by line. The third line will create a vector

$$t = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{bmatrix}$$

The fourth line will square each entry, and subtract 25 from it, resulting in

$$x = \begin{bmatrix} -25 & -24 & -21 & -16 & -9 & 0 & 11 & 24 & 39 & 56 \end{bmatrix}$$

Now, what will the fifth line do? Well, `(x>0.01)` will create a new vector with the same size as `x`: each entry that is larger than 0.01 will be replaced with 1 (essentially `True`), and every entry that's not will be replaced with 0 (`False`). Thus, it'll create

$$(x > 0.01) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Then, `(x>0.01)*x` will do an elementwise multiplication between these two vectors, so it'll create

$$y = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 11 & 24 & 39 & 56 \end{bmatrix}$$

The sum of this is simply $11 + 24 + 39 + 56 = 130$.

2.1.6 *Question 6*

Again, let's go through this line by line. Line 3 will create an array of $\begin{bmatrix} -10, -9.99, -9.98, ..., 9.98, 9.99 \end{bmatrix}$. Now, `y=sin(x)` will simply compute the sine of each of those entries and put it in an array. The final line will take the maximum value of `y`, and convert it to an integer. Now before you start to actually compute the sine of each entry, realise the following: when we convert it to an integer, we'll automatically round down, i.e. 0.999999 will be rounded down to 0 rather than 1. The only way for it to be rounded to 1 would be if you'd take the sine of $\pi/2$, such that you'd get exactly 1 as a result. However, considering we have a time-step of 0.01, it's extremely unlikely we'll actually obtain the value 1 for any of the entries; we'll probably get like 0.99 at most or something. Thus, it'll automatically be rounded down to 0, and we'll print 0.

2.2 *Part II: debug the program*

2.2.1 *Question 7*

Answer B is correct. It just has two very clear indentation errors after the `if:` and `else:` statements. If those indentations errors would be removed btw, neither answer C nor D would actually be correct; it'd print 25 (positive). Why? Well, it's quintessential to understand how argument passing (i.e. how it works with the arguments you pass to a function) works in Python. You have to realise that Python is actually quite smart, and it makes a distinction between inner scope variables and outer scope variables. Inner scope variables are

variables that only work within the function, and outer scope variables are variables that are called from outside the function.

What do I mean with this? Well, whenever you define a new variable within a function, that variable can only be used within that function. Consider the following:

```
def f(x):
  a = 3
  return a*x

print(a)
```

This will give you `NameError: name 'a' is not defined`. This is because `a` is defined within the function, and is therefore limited to the inner scope of the function: it's only usable within the function itself. On the other hand, you could have also written the following:

```
b = 3
def f(x):
  return b*x

print(b)
print(f(5))
```

This won't give any errors at all! Python is smart enough to realise now that it for the function `f(x)`, it needs to get `b` from the outerscope, i.e. from information *outside* the function itself. Note that you could have even put `b=3` after the function statement; Python is oblivious to where you put functions in a script (as long as you don't define them within an indented block of code). However, what happens in the following?

```
b = 3
def f(x):
  y = b*x
  b = 4
  return y

print(f(5))
```

Now it'll actually give an `UnboundLocalError: local variable 'b' referenced before assignment`. Why does it stop working now? Why is Python so stupid that it now doesn't realise it can take `b` from outside the function? The reason is that you *also* define `b` within the function itself. By doing so, you force Python to treat the variable `b` as an inner scope variable: thus, it won't be able to realise `b=3` is standing above it, and it'll actually say, bitch you didn't define `b` yet within this function pls stop. So, it's fine to use variables you defined outside the function in the function; however, you are then not allowed to change these values! So `b+=1` would also have been resulted in the same error. If you'd want to use the initial value of `b` anyway, you'd either need to put it in the function definition itself, or put it as an argument, i.e. write `def f(x,b)` and `print(f(5,b)`, so that it'll pass on the value of `b` you defined in the outer scope.

This also holds for the arguments you define: if we look back at the question itself, we write `def f(x)`. This means that we define `x` when we pass the arguments to `f`. This essentially makes it an inner-scope variable. This means that the line `x=-1` in the outer scope will be completely ignored! It may as well not have been there. Instead, the function would think: `f(5)`, so `x=5`. Evaluating `if x>0` is true, since `5>0`. So, `y = 5*5=25`, so it will return 25. Note that this does not alter the value of `y` in the outer scope! If you'd write, after `print f(y)`, `print(y)`, it will still print 5.

Inner and outer scope variables

Variables defined within functions will *always* be inner scope variables, i.e. they can only be called upon within the function itself.

Functions can, in principle, use variables that are defined in the outer scope, i.e. outside the function. However, if this variable is redefined anywhere within the function itself, or used as an argument, it will be treated as an inner scope variable, and it will never be taken from the outer scope.

2.2.2 Question 8

Answer A is correct. `range` only works with integers; you'd need to use `np.arange(0.5,2.5,0.5)` for this.

2.2.3 Question 9

Answer E is correct. It is actually completely fine, I don't know what to comment.

2.2.4 Question 10

Yeah so this question depends on which language you'd run it in:

- In Python2, answer C would be correct. Since we write `j=1`, it's saved as an integer. Then, in each loop we divide it by the integer 10, so it becomes 0.1 which is rounded down to 0. So, `i` is never incremented, and it'll always remain 0. Consequentially, you're always stuck in the while loop, and thus answer C is correct: there's no error, just an infinite loop and it will never print anything.
- In Python3, none of the answers would be correct (just run it yourself). Answer E is the one that probably would be correct: every loop, you increment `i` by 0.1. So, in principle you'll end up at `i=10`, and the loop is exited, and it'll print 10 1. However, due to rounding errors, your penultimate value of `i` will be like 9.999999998, so it'll actually increment to `10.0999998`. But this is a rather unpredictable round-off error so you shouldn't really expect something like this on the exam. But this is the reason it's usually a bad idea to use comparisons for floats as rounding errors may have a very large influence on your results: it's common to write stuff like `if 9.99999<x<10.000001` to make sure rounding errors don't affect your results.

2.3 Part III: complete the program

These questions always seem way harder than they actually are, as you can take a lot of "inspiration" from other parts of the code. The first thing to do is always read the text above carefully, and to go through the code carefully before wanting to answer the missing blocks, so that you have a solid understanding of what the code actually should do (this can sometimes mean you need to add a minus sign to something). After that, first think what the part should do, and then think how you can translate this to Python code. Also look at the rest of the code to see whether there's a similar line somewhere else (this is often the case).

2.3.1 Question 11

So the code simulates and plots the falling of a ball. The answers are pretty straightforward:

a) Our simulation will run until the thing is 10 meters through the water. In other words, correct is: (`while`) `h>-10 (:)`.
b) In the text above it is said that when the ball drops in the water, the density is increased to 1000 $\mathrm{kg/m^3}$. In other words, correct is: `if h<0 (:)`.
c) Very simple time integration. We simply obtain (`V =`) `V + ay*dt`, very similar to the line below.
d) We are plotting the time table versus the altitude table. Thus, we simply write `ttab` here.
e) Looking at a few lines above, and looking in the reader, it's pretty clear we should write `122` here: the first 1 corresponds to the number of rows of plots; the second 2 corresponds to the number of columns of plots; the third 2 corresponds to the number of the plot we are currently describing (the 2nd), counting from left-to-right, then top-to-bottom.

2.4 Part IV

2.4.1 Question 12

See the code below.

```
import matplotlib.pyplot as plt
import numpy as np

def f(x):
    return x**4 - 4*x**3 + 3*x**2 + 2*x - 1

x = np.arange(-1,3,0.0001)
y1 = f(x)

plt.plot(x,y1)
plt.show()
```

Just zoom in a lot (until you get the 4 decimal accuracy) on the points where the graph crosses $y = 0$, and then read off the x-value. You obtain $x_1 = -0.6180$, $x_2 = 0.3820$, $x_3 = 1.6180$ and $x_4 = 2.6180$. Sometimes Python will give weird values on the x-axis and say like $+1.618$ on the right-bottom, but yeah you just have to add that then. You can of course write a program that actually computes the roots by solving the equation numerically, but honestly why would you? Trust me, it's more complicated than it sound, and just reading from the graph takes less time than writing a numerical solver (numerical solvers have a few ifs and buts, mostly in this case that getting the initial guess correct such that you end up at the root). This is just one of those things that are like I said: you can just brute force your way through rather than trying to be fancy and program something beautiful, because literally no one cares.

2.4.2 Question 13

This question is arguably a bit difficult at first, as it may seem rather overwhelming. However, in those cases, it's important to think on a larger scale, and think in non-Python terms what your program should do. On a high-level, your program would do the following:

```
while n<1000:
  convert n to binary
  sum binary digits
  if sum is even:
    store n
  n = n+1
sum stored n
```

Now, we see that we probably have two main difficulties: converting n to binary, and summing the binary digits (storing n shouldn't be too hard with lists, and summing afterwards also seems doable). Well, how we do this then? First, note that strictly speaking, you don't really need to write the number as a binary number: consider the example of the 23 they show. You can also just define a counter that starts at 0, and every power of 2 it is divisible by increments the counter by 1, meaning you end up at 4. If this number is even, it means you had an even number of 1s in your binary number, and if it's odd, you'd have an odd number of 1s in your binary too. This simplifies our lives a bit. Only problem is, how are we going to do the division and counting thingy? Well, an if-statement and modulo are pretty helpful for this.

We can do the following: for each n, we define an inner for-loop with running variable i, which ranges from 10 to 0, in reverse order: `for i in range(10,0,-1)` (why these numbers will become clear very soon). Then we check whether $2^i > n$: if it is, then n is not divisible by 2^i. If $n > 2^i$, then we take the modulo of `n%2**i`, and continue the rest of the operations with this remainder.

In case it's not really clear what I'm doing: consider the number 23 once more. First we check whether $2^{10} = 1024$ fits in it; this is definitely the case, so we check whether $2^9 = 512$ fits in it, etc. etc., until we reach $i = 4$, so that we check whether $2^4 = 16$ fits in 23, which is true. In this case, we will increase our `counter` by 1, and take the modulo with respect to 16 so that we end up at $23\%16 = 7$ (you could also just subtract 16 from 23). We then check whether $2^3 = 8$ fits in 7, which is not true, so we check whether $2^2 = 4$ fits in 7, which is true, so we increase our `counter` by 1, and take the modulo with respect to 4 so that we end up at $7\%4 = 3$, etc. etc. This way our final `counter` will equal 3.

Hopefully it's now also clear why I made the range from 10 to (including) 0, with step -1: I first check the large numbers, and I have to check exponents between 0 and 10, as 1024 is approximately equal to 1000 (you could have also let it range from 0 to 9 but who cares). In Python-language, this would translate to

```
listofnumbers = []
n = 0
while n<1000:
  counter = 0
  for i in range(10,-1,-1):
    if not 2**i>n:
      n = n%2**i
      counter = counter + 1
  if counter%2 == 0:
    listofnummbers.append(temp)
  n = n+1
print(sum(listofnumbers))
```

There is but one thing left that's a problem: we now change the value of n when doing the binary stuff, meaning we reduce it everytime, and it'll indeed be an infinite while-loop. We can solve this by everytime saving the value of n in a temporary variable and doing the computations with this temporary variable, i.e.

```
listofnumbers = []
n = 0
while n<1000:
  counter = 0
  temp = n
  for i in range(10,-1,-1):
    if not 2**i>temp:
      temp = temp%2**i
      counter = counter + 1
  if counter%2 == 0:
    listofnummbers.append(temp)
  n = n+1
print(sum(listofnumbers))
```

And this nicely prints 249750.

Just to be clear: it's completely understandable if you had problems with this questions and if it's something you wouldn't have had come up with yourself. It's something you kinda have to see. However, then again, there are always at least a few other programming questions you can do without requiring insight (see e.g. the previous question). Furthermore, if you're stuck and you have no other questions left to do: first try to think on a high-level what your program should do and then boil those high-level functions down into smaller things. That's the best way to approach such a problem, rather than just randomly trying stuff.

2.4.3 *Question 14*

This is one of those questions where you can take brute-force in the most literal sense of the world. When I took this exam two years ago, I literally just counted each of the letters how many times they occurred, and then did the program by hand basically. This is a perfectly valid strategy, and since it's a rather short text, it's perfectly viable to get you at the correct answer. Just make sure as fuck you can count correctly, cause I remember when I did it I needed like five recounts before I was sure I counted correctly (honestly it's pretty tough you know). But yeah I did not write any Python code at all.

However, if you do want to write it in Python, see the code below:

```
text = "No, no, no, my brain in my head." \
       "It will have to come out." \
       "Out? Of my head?" \
       "Yes! All the bits of it. Nurse! Nurse! Nurse!"

sum = 0
for letter in text:
    letter = letter.lower()
    if letter.islower():
        idx = ord(letter)-ord("a")
        sum += idx**2
```

```
print(sum)
```

Basically:

- Lines 1-6 speak for themselves really.
- Line 7 starts looping through all the symbols in the text.
- Line 8 converts every letter to a lower letter; note that nothing will happen to punctuation signs.
- Line 9 checks if the letter is now a lower letter: this will filter out the punctuation marks.
- Line 10 computes the difference in index with respect to the letter a, much like the decypher assignment.
- Line 11 adds the square of the index to the sum.
- Line 13 prints the sum, and prints 15506.

So, the answer is 15506.

2.4.4 *Question 15*

In my opinion, these questions of numerical simulation are the easiest questions on the exam, and you should really aim to get at least this one correct of the open questions. Honestly, let's just look at the solution as shown below.

```
### Given data
CL = 1.45
CD0 = 0.035
piAe = 23.8761
rho = 1.225
S = 102
g = 9.80665
m = 41467
Vtakeoff = 67
T = 180000
mu = 0.03

### Simulation parameters
t = 0
dt = 0.001

### Initial condition
V = 0
s = 0

### Simulation
while V < Vtakeoff:
    L = CL*0.5*rho*V**2*S
    CD = CD0 +CL**2/piAe
    Daero = CD*0.5*rho*V**2*S
    N = m*g - L
    Df = mu*N
    D = Daero + Df
    ax = (T-D)/m
    V = V + ax*dt
    s = s + V*dt
    t = t + dt

print(t)
print(s)
```

Sure, it's a long code (35 lines), but it's super easy:

- Lines 1-11 are just the data they give in the question.
- Lines 14-15 just sets the initial time and the timestep.
- Lines 18-19 are just the initial condition.
- Line 22 is just realising that the simulation should end when the take-off speed is reached.
- Lines 23-28 are just the six equations they give, but with $W = m \cdot g$ which is completely obvious.
- Line 29 is just the equation of motion in x-direction.
- Lines 30-32 are just the numerical integration.

- Lines 34 and 35 give you the desired output of $t = 17.356\,\text{s}$ and $s = 595\,\text{m}$.

3 Exam August 2016

Please note that although the date on the exam says August 12, 2015, I'm really, really sure this is the August 2016 exam.

3.1 Part I: What will this print?

3.1.1 Question 1

As before, we go through these exercises line by line. Line 1 will make `rx=[12,9,6]`, as it starts at 12, stops at 3 (so not including 3 itself) and has step size -3. Line 2 speaks for itself. Then, in line 4, we define our for-loop, which may be a bit confusing if you aren't fluent in how indexing works in Python.

Indexing in Python

Normally speaking, for indexing works like `start:end:step`, i.e. the first number is the index of the first entry you want to include, the second number is the index of the last entry you want to include (but not include itself), and step is the step size in indices. For example,

```
a = [1,3,6,10,15,21]
b = a[1:4:1]
```

will make `b=[3,6,10]`: index 1 (i.e. the second number) is the first entry, and the step size is then 1, so we must also take index 2 and index 3 (but not 4, as we stop *at* 4). `c=a[0:4:2]` would make `c=[1,6]`, as we take indices 0 and 2 (but not 4). Special cases:

- **Omitting starting entry**: in case the first entry is part of the range of indices you want to include, you may also omit the starting number, i.e. `c=a[:4:2]` is equivalent to `c=a[0:4:2]`.
- **Omitting ending entry**: similarly, if you want to include entries starting from a certain index, but till the very end of the list, you could also omit the `stop`, i.e. `d=a[3::2]` is equivalent to `d=a[3:6:2]`.
- **Omitting both starting and ending entry**: you can also omit both if you want to start at the beginning and stop at the end, but are only interested in skipping entries; i.e. `e=a[::2]` is equivalent to `e=a[0:6:2]`.
- **Omitting the step**: in case you take a step of 1, you may omit the step, meaning that `f=a[1:4]` is equivalent to `f=a[1:4:1]`.
- **Negative start and stop**: sometimes you'll want to exclude only the last entry, without precisely knowing how long the list is. This can be done by using negative indices: the index -1 corresponds to the last entry in the list, -2 to the entry before that one, -3 to the one before that one, etc. In other words, `g=a[1:-2:2]` is equivalent to `g=a[1:4:2]`. One could also use it for the starting entry, i.e. `h=a[-4:-2]` is equivalent to `h=a[2:4]`.
- **Negative step**: if you want to transverse the list in the opposite direction, take a negative step, i.e. `i=a[::-1]` will make `i=[21,15,10,6,3,1]`. Note that in this case, the starting index (if specified) should always be higher than the ending index, otherwise no one understands what you'd be doing (you'd be trying to go from low index to high index, but in negative direction).

Honestly, just play around with this a bit; make a long list yourself and see what happens when you change the indices. It's really the best way to remember it.

Now, back to he question: `i` will transverse `rx` simply in reverse direction, i.e. it will first take the value 6, then 9, and then 12. Let's then just execute the loop:

- First, `i=6`, and `mysum` will equal $6 - 0 = 6$.
- After that, `i=9`, and `mysum=9-6=3`.

- Finally, `i=12`, and `mysum=12-3=9`.

Thus, we'll print 12, 9.

3.1.2 *Question 2*

For this question it's just quintessential you know what the `.join()` thingy does. It's pretty simple: let's consider the first instance of `.join()` first: we put it aft of a string that's simply a space (`' '`). Between the brackets of `.join()`, you put a list of the strings that you want to join together, separated by the string you put it aft of (so in this case separated by a space). Thus, the first part will print `We are the kings who say: "`. Had you written `''.join([v1, v2])`, it'd have printed `We arethe kings who say: "` as you know say they should be separated by nothing (so are and the are written together, not separated by a space).

The second instance of `.join()` works similarly, except you need to remember that if lists contain repeated entries, you may simply write `[v3]*3` to instantly create a list that contains three entries of `v3` (so to be clear, it's not that you now have three lists, each containing one entry of `v3`). So, `.join([v3]*3)` will attempt to join three instances of `"Ni"` together, each separated by `!", "` (an exclamation park, a quotation mark, comma, a space and a quotation mark). Thus, it'll print

- We are the kings who say: "Ni!", "Ni!", "Ni

Note that the final Ni is not ended by an exclamation mark and quotation mark, as there is nothing to be joint together at the end of it.

3.1.3 *Question 3*

Again, these questions you just have to write out rather than trying to do it by head. Initially, we'll have `o` is zero; the while statement is then violated as `j` is also equal to zero and thus not smaller than `2*o`. Thus, `o` is increased to 1. `j` is still equal to 0, so the while statement is satisfied. With `o` equal to 1, the modulo (remainder after division) with respect to 3 will simply be 1 too, so the if-statement is not satisfied. `r` is then incremented by `j`, but as `j` is equal to 0, it says 0. Finally, `j` is incremented by 1.

Then, `j<2*o` is still satisfied, so the while loop is executed again; the if-statement is still not satisfied, so `r` is increased to 1 as `j=1`. `j` will then be incremented to 2. Then, `j<2*o` is no longer satisfied, so the while-loop is exited and `o` is incremented to 2 due to the for-loop. With `j` still equalling 2, `j<2*o`, so we execute the while-loop again. Yet again, `o%3` does not equal zero so we continue on with the rest of the for-loop. `r=1+2=3`, and `j=2+1=3`. Still, `j<2*o`, so we again execute the loop, and once more we do not satisfy the if-statement, and increment `r=3+3=6` and `j=3+1=4`. The while-loop is now exited, and `o` is increased to 3.

With `j=4`, this means that the while-loop must again be executed, but this time, `o%3==0`, so we break this while-loop[1]. So, we immediately increase `o` again to be equal to 4. With `j` equal to 4, this means the while-loop will be executed. In a very similar fashion to what I described before, we get:

- $r = 6 + 4 = 10$
- $j = 4 + 1 = 5$
- $r = 10 + 5 = 15$
- $j = 5 + 1 = 6$
- $r = 15 + 6 = 21$
- $j = 6 + 1 = 7$
- $r = 21 + 7 = 28$
- $j = 7 + 1$

At this point, `j<2*o` is no longer satisfied and the while-loop is exited. As `range(5)` excludes 5 itself, the for-loop is also exited, and thus we print the last value of `r`, i.e. 28.

[1] But not the for-loop itself! That one still continues on as usual.

3.1.4 Question 4

The first part is knowing how `.split()` works. Basically, you put a string before it, and it will split the string into a list of separate entries. If you don't supply an argument, it'll split the string on the basis of spaces between the entries (and remove those spaces). Thus, it'll create a list with entries `"Wink"`, `"Wink"`, `"Nudge"` etc. (note that the spaces themselves are destroyed). Had you written `.split("n")`, it'd have looked for the letter n, and split the string whenever it saw an n (and removed the n in the process), i.e. it'd have created `'Wi', 'k wi', 'k ', 'udge '...`. Note that spaces now are preserved, as we are no longer splitting by them.

Then we set `t=1`, and then loop through the previously created list in reverse order. Note that we are not using `range()` or anything, but rather `a` itself: this means that the first value of `x` will not be 0, but it will literally be the last (since reverse order) entry of `a`, i.e. `x="more"`. We then simply multiply `t` by -2, meaning `t=-2`. Afterwards, `x` is changed to `"no"`, and `t=-2*-2=4`. This is done for each entry in `a`, meaning we get

- $t = -2 \cdot 4 = -8$
- $t = -2 \cdot -8 = 16$
- $t = -2 \cdot 16 = -32$
- $t = -2 \cdot -32 = 64$
- $t = -2 \cdot 64 = -128$

as we have seven entries in total in `a`. Note that this is simply equal to $(-2)^7$, and note that it does not matter than `x` will take the same values multiple times (it'll equal `"nudge"` twice for example). In conclusion, it'll print `-128 Wink`.

3.1.5 Question 5

Line 2 will create a row vector

$$t = \begin{bmatrix} -1 & -0.5 & 0 & 0.5 & 1 \end{bmatrix}$$

as `np.linspace` creates 5 numbers between -1 and 1, *in*cluding the endpoint. Line 3 then does an elementwise operation with it, which can be represented as follows:

$$y = \begin{bmatrix} -1 \\ -0.5 \\ 0 \\ 0.5 \\ 1 \end{bmatrix} \odot \begin{bmatrix} -1 \\ -0.5 \\ 0 \\ 0.5 \\ 1 \end{bmatrix} + \begin{bmatrix} -1 \\ -0.5 \\ 0 \\ 0.5 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ -0.25 \\ 0 \\ 0.75 \\ 2 \end{bmatrix}$$

Then, line 4 computes the final values of `y`. It does so by first creating a vector of equal to size of `t`, and with all entries larger than 0 equal to 1, and with all entries not larger than 0 equal to 0. Thus, it will temporarily create $\begin{bmatrix} 00011 \end{bmatrix}$. Thus, `y` can be straightforwardly computed:

$$y = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} \odot \begin{bmatrix} 0 \\ -0.25 \\ 0 \\ 0.75 \\ 2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0.75 \\ 2 \end{bmatrix}$$

Thus, the following will be printed: `[0. 0. 0. 0.75 2.]`.

3.1.6 Question 6

So in line 2 it'll create a vector for `t`, with values 10, 9, 8, ..., 2, 1 (not including 0). `1.0/exp(t)` then represents doing e^{-t} for each of these entries and putting them in a vector of same size. Evidently, the minimum value will then be e^{-10} and the maximum value will be e^{-1}.

3.2 Part II: Debug the program

3.2.1 Question 7

Answer C is correct, although I think matplotlib must have changed the way it calls these errors. The problem is that `myfun(x)` is a perfectly fine working function, except it doesn't return anything ever. So, if you call `y=myfun(t)`, it won't make `y` actually equal to anything; it'll be `None`. Thus, if you try to plot it, `matplotlib` is like what you trying to do bitch, please give me something I can actually plot. Indeed, it gives the error `ValueError: x and y must not be None`, although I have no idea why the exam says different. But if you would include `return y` in the function, answer D would be correct.

3.2.2 Question 8

Answer C is correct. `np.linspace` creates 5 entries between -2 and 2, *in*cluding the stop value (`np.linspace` is basically the exception to the rule of never including the stop value in Python). Also the commas are perfectly allowed in print statements.

3.2.3 Question 9

So this question is one of those questions that doesn't work in Python3 anymore. In Python3, none of the answers is correct: Python3 automatically saves all input as a string, so you'd need to write `float(input("Temperature (C) :")) + 273.15` for example, otherwise it thinks you're adding a number to a word which doesn't make sense. In Python2, the way it was written on the exam was still allowed, so let's consider how it would have been there (so let's assume that it says `float()` around the inputs). Then in principle, the entire program works almost perfectly. It computes the new pressure and temperature and prints them correctly. It then asks you to continue or not: if this answer is empty (e.g. by hitting a spacebar) or the first letter is neither a n or a y (from No or Yes), it'll keep asking until you give it a satisfying answer (it'll make the answer lowercase first, so it doesn't matter whether you write Yes or yes).

Then, `more` is set equal to the boolean `ans[0] == 'y'`: if the first letter of the answer is a y, it'll return True (and thus `more = True`") and thus continue the while-loop, else it'll make it return False (and thus `more = False`") and then exit the while-loop.

From this discussion, it is evident that answer A is false. Answer B is also false, the calculated pressure is corrected, but the temperature obviously isn't correct. Answer C is correct, but D is false as pressing enter will mean that `ans` will be empty, and thus `len(ans)==0` is true and it'll ask again for an answer. Answers E and F are correct, and G is bullshit.

Thus, answers C, E and F are correct.

3.3 Part III: complete

3.3.1 Question 10

Once again, these questions always seem way harder than they actually are, as you can take a lot of "inspiration" from other parts of the code. The first thing to do is always read the text above carefully, and to go through the code carefully before wanting to answer the missing blocks, so that you have a solid understanding of what the code actually should do (this can sometimes mean you need to add a minus sign to something). After that, first think what the part should do, and then think how you can translate this to Python code. Also look at the rest of the code to see whether there's a similar line somewhere else (this is often the case).

We have the following correct answers:

(a) We need to create an empty list of balls, and from the remainder of the program, it is clear that this needs to be called `balls`. Thus, we must write `balls=[]`.

(b) We return all of the variables `x`, `y`, `vx` and `vy`. However, we still need to update the values of `x` and `y`, so we must do so in the same return statement. Thus, we obtain `(return ) (x+vx*dt,y+vy*dt,vx,vy)`, where I use brackets as it is required to be put in a tuple.
(c) We'll take the time between the previous timestep and the current time, i.e. simply `tick-tick0`.
(d) Seemingly easy as we just need to tell something like `y>0`. However, how do we obtain `y`? Well, remember that we are currently looking for the current ball, `balls[i]`. In the answer of (b), we state that the `y` position is returned as second output, in other words, we need to use index 1 to refer to it. Thus, we get `(if )balls[i][1] > 0`.
(e) We need a random value between -2.5 and 2.5. `uniform` creates a random value between 0 and 1. Thus, looking at what is written after it, we should write `5*uniform()-2.5`.

3.4 Part IV: write

I'll admit that across the board the following questions were all pretty bad. But then again, doing as many exams as you can will help you get better.

3.4.1 Question 11

This question is scary if you get frightened by complex numbers and logarithmically spaced numbers. However, they tell you rather specifically what to do. Therefore, let's look at the final code and see what we should do:

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  def f(x):
5      return (1+0.2*x)*(1+0.1*x)/((1+x)*(1+0.02*x)**2*(1+0.01*x)**2)
6
7  x = np.logspace(-4,2,10000)*1j
8  y = f(x)
9
10 plt.plot(np.real(y),np.imag(y))
11 plt.show()
```

We see:

- Lines 4-5 are just the function they wrote.
- Lines 7 does what may have sounded very difficult: creating a logarithmically spaced array. This is where the Numpy documentation helps a lot, as it's probably the first thing that shows up when you search for logarithmic spacing, and it'll tell you how the function exactly works. Another way of doing it would be to realise that a logarithmically spaced array of points is just an array where the *exponents* are spaced logarithmically, i.e. the array they show is simply $\begin{bmatrix}10^{-1} & 10^{-0.5} & 10^{0} & 10^{0.5} & 10^{1}\end{bmatrix}$ which is clearly linearly spaced in the exponents. Thus, we could also create a linear spacing between -4 (as $0.0001 = 10^{-4}$) and 2 (as $100 = 10^{2}$), with 10000 steps in between (just a number I used to create a smooth plot), which will represent the exponents. We would then create an array of 10 to the power of these exponents. Finally, we would multiply with $1j$ as they instructed us to. In other words, `x=10**np.linspace(-4,2,10000)*1j`.
- Line 8 just computes the array of function values. Nothing more, nothing less.
- Line 10 plots, as they instruct, the real parts versus the imaginary parts. Line 11 shows the plot.

That's all there is to it, really. We can then zoom in on the points where the imaginary parts equal -0.2. Doing so you'll find three real parts where the imaginary part equals -0.2: -0.133, 0.308, 0.377 and 0.943.

Please realise how easy this question actually was, if you just did what the question told you to do. Only creating a logarithmically spaced array was admittedly a bit difficult if you didn't know how to.

3.4.2 Question 12

Again, try to first think on a high-level. We get that the program should do the following steps:

- Loop through all of the numbers below 10000.
- For each number below 10000, extract the digits, and sum their factorials.
- If this sum is divisible by the original number, append it to some list.
- Take the last two numbers in this list to be our answer.

Evidently, the second bullet seems to be the hardest task. How do we extract digits from a number in an elegant way? The easiest way by far, and it's a way you should remember, is to simply convert the number to a string. After all, for a string, we are able to indicate indices for which character we want. Thus, by writing `stringofnumber=str(number)`, we have a variable `stringofnumber` for which we can write e.g. `stringofnumber[2]` to extract the third digit from it. Thus, we could loop through this variable, and write `for digit in stringofnumber`, and this way we can loop through each of the digits of the number.

From `math` we can import `m.factorial` to automatically compute the factorial of a digit. However, when we write `for digit in stringofnumber`, it'll think that `digit` is a string too, so we need to convert it to an integer first, by writing `int(digit)`. We can then just sum the factorials of the digits together in a variable called `sumoffactorials` and when we're done looping through the digits, we'll take the modulo with respect to the original `number` and see whether this is 0; if it is, then it's a divisible number.

Thus, so far, we could write it as

```
import math as m

number = 19
stringofnumber = str(number)
sumoffactorials = 0
for digit in stringofnumber:
    sumoffactorials += m.factorial(int(digit))
if sumoffactorials%number == 0:
    print("Divisible number")
```

You can check that this functions properly. We then merely need to extend our program a bit, by making it a for-loop that loops through all the numbers between 1 and 10000 (we don't include 0 as then we'd have to divide in the end by 0 and get an error), and appending divisible numbers to a list. We then print the resulting last two entries of this list. This is shown in the code below:

```
import math as m

listofdivisiblenmumbers = []
for number in range(1,10000):
    stringofnumber = str(number)
    sumoffactorials = 0
    for digit in stringofnumber:
        sumoffactorials += m.factorial(int(digit))
    if sumoffactorials%number == 0:
        listofdivisiblenmumbers.append(number)

print(listofdivisiblenmumbers[-2],listofdivisiblenmumbers[-1])
```

We get as output 7684 9696.

3.4.3 Question 13

Pretty similar in a sense to the previous question, since you again have to somehow extract digits from a number. However, once again, we can do that by simply converting it to a string, although this time it'll be more work. On a top-level, we can distinguish the following things we need to do:

- Convert number to a string
- Save last digit
- Delete last digit from string
- Subtract the double value of the last digit we saved

- Check if number is smaller than 100 already

Deleting the last digit from the string is essentially is the hardest thing, and even that isn't that difficult. Personally, the way I like to do it is by writing `stringofnumber = stringofnumber[:-1]` which basically tells Python to select all but the last entry of the string. We then rather straightforwardly get the code shown below:

```
1  import random as rnd
2  L = 29
3  rnd.seed(0)
4  bignumber = int(''.join([rnd.choice('0123456789') for i in range(L)]))
5
6  steps = 0
7  while bignumber > 99:
8      stringofbignumber = str(bignumber)
9      lastdigit = stringofbignumber[-1]
10     stringofbignumber = stringofbignumber[:-1]
11     bignumber = int(stringofbignumber) - int(lastdigit)*2
12     steps = steps + 1
13
14 print(steps, bignumber)
```

- Lines 1-4 are just copied from the question.
- Line 6 initiates the step counter (as it's also asked to print that).
- Line 7 checks whether the big number is smaller than 100.
- Line 8 converts the big number to a string.
- Line 9 takes the last digit from this big number.
- Line 10 deletes the last digit.
- Line 11 subtracts twice the last digit from the original large number; note that I first must convert the strings to integers as otherwise it thinks I'm multiplying strings and it doesn't know what to do.
- Line 12 then increments the stepcounter.

Honestly, I just want to show how easy the questions actually are if you first just think in non-Python language what you want to do. Don't immediately start programming as this is barely ever a good idea.

3.4.4 *Question 14*

This question may seem incredibly overwhelming to you at first, and it actually is if you want to do it properly, i.e. actually doing what the question wants you to. However, there is a giant loophole in this question that you can easily exploit. Rather than programming an 'AI' to be the second player, programming the output as they show it in the question *and* something that checks whether someone already won or not, why not do these things yourself? Let me just show what I would have coded had I done this exam:

```
1  import random as rnd
2  rnd.seed(0)
3
4  row = rnd.randint(0,2)
5  col = rnd.randint(0,2)
6  print(row, col)
7
8  row = rnd.randint(0,2)
9  col = rnd.randint(0,2)
10 print(row, col)
11
12 row = rnd.randint(0,2)
13 col = rnd.randint(0,2)
14 print(row, col)
15
16 row = rnd.randint(0,2)
17 col = rnd.randint(0,2)
18 print(row, col)
19
20 row = rnd.randint(0,2)
21 col = rnd.randint(0,2)
22 print(row, col)
```

This looks rather simplistic, doesn't it? Basically, I just copied their code, print the row and column each time,

and just copied and pasted a few times. So how would it work? Well, you can just write along with the tic-tac-toe on a scrap paper. If you run my amazing program, the first combination that is printed is 1 1. Since it's the first move, you can put a cross at the middle box of the tic-tac-toe square. Then, player two would have to place a circle at the first free square, starting from top left. There's nothing there yet, so you can easily put a circle there. The second output generated by the Python program is 0 1, i.e. the first row, second column. You can just put a cross there. Player 2 now has to put a circle in the top right box, as middle box of the top row has obviously been filled now. The third output generated by Python is 2 1, i.e. third row, second column. You can also put a cross there. If you put the crosses and circles where you should have put them, it'll be clear now that the game has been won by the RNG. So, the answer to the question will simply be

○ × ○
. × .
. × .

which looks remarkably similar to a dick if you squint your eyes.

Is this the most proper way to do this question? No. Does it show you understand Python well? No. However, imagine you were sitting this exam: it's almost Summer break, it's nice weather outside and beer is awaiting you. What would you do? If your priority is writing a nice program over escaping to freedom, you should reconsider your priorities.

4 Exam August 2015

4.1 Part I: Reading Python

4.1.1 Question 1

Once again, just get your scrap paper out and do it by hand. We start with `i=0`, and then `j=0`. In that case, `(i+j)%3==0` is true, so `total` is incremented to 1. `j=0` is increased to 1 (but then nothing will happens as then `(i+j)%3==0` is evidently not true); then to 2 (again nothing), and then to 3 and then `(i+j)%3==0` is actually true. So, `total` is increased to 2.

We then increment `i` to 1; in this case, `j` will run between 1 and 3 (including 3), and thus `(i+j)%3==0` is true only when `j=2`, so we only increment `total` once, meaning `total` now equals 3. Then `i` is set to 2; in this case, `j` will run between 2 and 3, and thus `(i+j)%3==0` is never true. Finally, `i` is set to 3, and `j` runs between 3 and 3 (so it's only set equal to 3); in that case, `(i+j)%3==0`, so `total` is finally increased to 4. So, we print 4.

4.1.2 Question 2

Again, just one of those questions you just have to manually go through to get it right. For the while-loop, the first time we check whether `a[-1]==5`; note that a negative index means you count from the end (with the first entry from the right having index -1, and the one to the left of that has -2, etc.). However, `a[-1]` equals 1 (it's the last entry of `a`), so we go into the while-loop and decrement `idx` by 1, so we get `idx=-2` now. However, `a[-2]` equals 0, so again we decrement `idx`, and we basically keep doing this until we reach the value 5 in `a`. If you can count, you'll see that it's the seventh entry from the right, i.e. once we obtain `idx=-7`, the while-loop will be exited.

Finally, `len(a)` is simply the number of entries in `a`, so it's equal to 15. This means we print 8.

4.1.3 Question 3

Once more, just go through it as if you were a computer. Only problem is that once more the question is difference in Python2 than in Python3 (due to integer division), so I'll answer as if instead of `/` they wrote `//` (which triggers integer division in Python3).

So, first we start at `i=1`, `rabbit=10` and `fox=3`. We then update the value of `rabbit`: we simply obtain `rabbit = 10 + 10//10 - 3*10//10=10+1-3=8`. However, now for `fox`, we must use this *new* value for `rabbit`, in other words, we get `fox=3+8*3//20=3+24//20=3+1=4`, as `24//20` gets rounded down to 1.

Then, `i=2`, and we update `rabbit` again; we now get `rabbit = 8 + 8//10 - 4*8//10 = 8 + 0 - 32//10 = 8 + 0 - 3 = 5`. We then get for `fox = 4 + 5*4//20 = 4 + 1 = 5`.

Then `i=3` and the while loop is exited. Consequently, we print `5 5`.

4.1.4 Question 4

Looks intimidating at first, but again, let's just go through the code line-by-line. We start outside the function. We simply have a list containing two entries; `lst[0]="Not ok."` and `lst[1]="Ok."`. What the hell happens after that then? Well, apparently, we are taking the entry corresponding to the index equal to `int(check("renault","def"))`, so apparently we are converting the output of the function `check` to an integer. Let's now focus on what `check` outputs then.

First, it creates a list `cars` of several sublists, each of those sublists having the first entry be a string containing the car brand, and the second entry being a letter. But nothing happens with the list yet. Then we set some variable `sw` equal to `False`. Then we loop through the list `cars` (so `c` is one of those eight sublists each loop). It is checked for each sublist whether, if the first letter of the first entry (i.e. the brand name) is converted to a lower letter, is the same letter as the first letter of the `car` that is being inputted, if this letter is also converted to a lower letter[1]. This is indeed the case for the first sublist when we input `renault"`, as it obviously starts with the same letter as `Renault`. It then converts the `txt` you inputted (in our case `def`) to capitals, and counts whether the letter corresponding to the brand (for the Renault it's F) appears in the `txt` (each time the letter appears the counter is incremented by 1). `def` contains an F, so `sw` is set to `True`, and the entire for-loop is broken. `sw` is returned, so we return `True`.

Now, originally, we were converting the output of `check` to an integer: as you hopefully know, a `True` gets converted to 1 (`False` gets converted to 0). So, we take `lst[1]`, which is `Ok.` so that's what we'll print.

4.1.5 *Question 5*

Super easy if you remember how `numpy` indexing works. `a[:,2]` simply means all rows, third column (since it's a 2). So, that's $1+0+5=6$, so we'll print a 6.

4.1.6 *Question 6*

Slightly trickier, but `a[a%2==0]` simply means the values of `a` where the entry is even (since `%2==0` checks whether a number is divisible by 2, i.e. whether it is even or not). The entries of `a` that are even are 2, 4, 0 and 2, so the sum is $2+4+0+2=8$. So we print 8.

4.2 *Part II: Debugging Python*

4.2.1 *Question 7*

Answer A is bullshit; answer B is also wrong (if you use the `math` module, it is `atan`, but for `numpy` they do call it `arctan` for some reason). Answer C is definitely correct though, no doubts about it.

4.2.2 *Question 8*

The number of choices may seem scary but just go through them one-by-one:

A. The program does not have an evident identation error.
B. Bullcrap.
C. Nahhh, this would only happen if you'd make `i` a float somehow (e.g. by adding `0.5` to it for whatever reason).
D. No, this absolutely never happens.
E. Yeah this is actually true. `color[i]=255` and `color[i]=0` will both give errors: you are changing values of a tuple (since `color` is defined as `color = (0,0,0)` and the round brackets indicate it's a tuple), which is something you are simply not allowed to do.
F. No, this is just plain wrong.
G. No, wrong.
H. No, wrong.
I. No, depends on what the programmer wants to do.
J. No, wrong.
K. No, wrong.
L. No, wrong.

[1] So it basically just checks if the first letters of the input and the brand name are the same, and does so by ensuring that both letters are lower case.

So clearly answer E is the only correct one.

4.2.3 Question 9

Long code, but let's first try to get a global overview of the code. The function `getdata` is simply a function that reads a given filename, and returns the data in it in a `data` list. `csvline` will first take the very first entry in a list `lst`, and convert it to a string. Then, it will start looping through the list, skipping the very first entry as we already took that entry. It'll append every string to the already existing string, so that we get a mighty long string; each of the original strings are separated by ; now.

Then, we retrieve the exam and bonus data from some files. Finally, we go through all the exams, and check if a student did the exam, and if it did, we'll save some values in some variables. That's roughly the idea of what's supposed to happen. Let's then go through the answers:

A. False: `getdata` returns a list `data`; the indexing `[1:]` will regard this output list, so it's perfectly fine to write it like this.
B. False: again, `getdata` simply returns a list, and the indexing will just select part of this list. Lists can be added with the plus-sign (it'll simply concatenate the two lists) so this one is false.
C. False: a single backslash is sufficient at the end of the lines near where `bonusdata` is defined.
D. False: just completely wrong.
E. False: no it doesn't.
F. False: it doesn't.
G. True: at the very end, it saves the student number, name, etc. correctly, but it doesn't actually store those values anywhere. So he still needs to add something that it automatically gets stored in a list or something. But there are currently no errors present yet.

4.3 Part III: Writing Python

4.3.1 Question 10

Very simple question. See the code below.

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  def f(x):
5      return np.sin(x)
6
7  def g(x):
8      return 0.7+np.cos(2*x)
9
10 x = np.arange(0,10,0.01)
11 f = f(x)
12 g = g(x)
13 plt.plot(x,f,x,g)
14 plt.show()
```

- Lines 4-8 simply define the functions they ask for.
- Line 10 creates the x-values.
- Line 11 and 12 create the corresponding function values.
- Line 13-14 plots them. Note that we need to write `x,f,x,g`, as `matplotlib` doesn't understand if you put `x,f,g`.

Then, simply use your eyes, and see that there will be four intersections. So the answer is 4.

4.3.2 Question 11

Again, super easy.

```
sum = 0
for i in range(1,2000):
    sum += 1/i**3

print(sum)
```

We set up a sum, and then simply loop through $1 \leq i \leq 1000$, and perform the summation. At the end we print the sum, which prints 1.2020564. Verifying that this is the value it converges to can be done by also looping through $1 \leq i \leq 2000$, it then prints 1.2020568. So, with five digits accuracy, the answer is 1.20206 (as it's incredibly unlikely it'll ever go beyond 1.2020649 seeing how slowly it increases).

4.3.3 *Question 12*

A bit of a tougher question. First thinking on a higher-level, what we want to do is that we loop through all integer values of `x`, and then compute the corresponding solution for `y`, and then check whether this is actually an integer and that is larger than 0. So, how do we do that? Well, it's shown in the code below.

```
import math as m
import numpy as np
lst = []
for x in range(1000):
    y = np.sqrt((1-x**2)/-2)
    if abs(y-round(y))<0.00001 and y>0 and y<1000:
        lst.append([x,y])

print(np.sum(lst))
```

Lines 1-5 are pretty basic and don't need much explanation. I use `np.sqrt` rather than `m.sqrt` as `np.sqrt` can deal with complex numbers but `m.sqrt` can't. Then, how do we check whether `y` is an integer? Well, we simply round `y` to the nearest integer (which is what `round(y)` does), then check whether the actual value of `y` differs less than a very small number from it (note that we need to use `abs()` as well). If this is the case, it's probably an integer. If `y` is an integer, and it is larger than 0 and smaller than 1000, we'll append the `x` and `y` values to the list, and the end we'll sum up all of the values in the list and we're done. The result is 1188.0.

4.3.4 *Question 13*

Very easy question again. See the code below.

```
W = 10
g = 9.81
m = W/g
V = 45
h = 0
t = 0
dt = 0.001
while h ≥ 0:
    if V>0:
        D = -0.05*V**2
    else:
        D = 0.05*V**2
    Fy = D - W
    a = Fy/m
    V = V + a*dt
    h = h + V*dt
    t = t + dt

print(t)
```

- Lines 1-7 are very basic and just initialise all parameters. Note that we'll need to find the mass of the ball, which is simply 10/9.81. Furthermore, you need to define a coordinate system; personally I find it easiest to just define up to be positive. Thus, `V` is initially positive.
- Line 8 just initiates the while-loop; as long as the height doesn't become negative, it should run.

- Line 9-12 determine the drag force, where I define a positive drag force to be upwards. Note that if the velocity is positive (so upward), the drag will point downward, so if `V>0`, we need to add a minus sign. If `V<0`, then the velocity is downward, so drag is upward, so then it should be a plus.
- Line 13 determines the sum of forces in y-direction; as I said before, I assumed drag to be positive upwards, so it needs to be `Fy = D - W`.
- Line 14 determines the acceleration; once more positive upwards.
- Lines 15-17 perform the rest of the numerical integration.

Now personally it's been my experience that even two years from now during DSE some of you'll find this to be already mind-blowingly difficult, but honestly, it's really, really easy. As long as you've seen a few sample questions it should be the easiest question on the exam.

4.3.5 *Question 14*

This question is a bit tougher than the previous ones, but fortunately it borrows some principles from question 12. See the code below:

```
W = 10
g = 9.81
m = W/g
V = 45
h = 0
t = 0
dt = 0.001
while h ≥ 0:
    if V>0:
        D = -0.05*V**2
    else:
        D = 0.05*V**2
    Fy = D - W
    a = Fy/m
    V = V + a*dt
    h = h + V*dt
    t = t + dt

print(t)
```

The basic idea is as follows: for a given `N`, check for $1 \leq a \leq N$ all possible $a \leq b \leq N$, and check whether for those combinations $c = \sqrt{a^2 + b^2}$ is an integer; if so, increase `counter` by 1 to count how many times this happens in total for a given `N`. Do this for all integer `N` starting from 20, till whenever you reach the point where `counter>N`. Once `counter` for a given `N` is larger than `N` itself, the program should be quit, else `N` should be incremented by 1 and the loop should be run again, initialising `counter` back to 0. That's basically what the entire code does.

Note that `a` is in `range(1,N+1)`: it is said that $0 < a \leq N$, so we need to exclude 0 (by starting at 1) and include N itself, so we need to range to `N+1`. Same goes for `b`.

Furthermore: normally speaking I'd have recommend to just try to find `N` manually: just finding by trying different values of `N` when the counter becomes larger than `N`. However, the issue with this is that you may find two numbers where you think, okay so for this number, the counter is still smaller than `N`, but for the next number, it's larger than `N`; however, bear in mind that there may be multiple numbers where the counter crosses the boundary so to say! It may be that for some smaller `N`, the counter was also already higher than `N` itself, but that there were some unfortunate numbers after that at which there were no new Pythagorean triplets.

5 Exam April 2012

5.1 Part I: Reading Python

5.1.1 Question 1

The program first says s is 0. It will then enter a for-loop, for i in range(20). So, it'll start at i=0, and when the loop is complete, it'll go to i=1, then to i=2, etc. all the way to i=19. Let's see what the program does in the loop itself.

First, it checks whether i is divisible by 2[1]. If it is, it will print the current value of i. Now, elif will only go into effect if the previous if statement was not true. It's basically a combination of 'else' + 'if': if none of the previous if (or elif) statements was true, then it will act as if statement. So, if the current value of i is indivisible by 2, it will check whether it is divisible by 3. If it is divisible by 3, it will increase the value of s. After this, the loop is restarted, until and including i=19.

So, let's see what happens. 0 is divisible by 2, so it'll print i, and it won't check for divisibility by 3. i will then increase to 1, which is not divisibly by 2 nor 3. i will then increase to 2, which is divisible by 2. It is then increased to 3, which is not divisible by 3, so it'll check for divisibility by 3. It is, so it will increase s to 1. For i is 4 and 5, again nothing happens to s. However, at i is 6, you might be inclined to think that s will be increased as well, because 6 is divisible by 3. However, 6 was also divisible by 2, so it won't be checking whether it is divisible by 3 as well. So, 6 is skipped. Applying this logic tells us that it will only increase further at the values i = 9 and i = 15, meaning that the final value of s will be 3. The correct answer is thus 3.

5.1.2 Question 2

The program starts at total=0. Then, for i=0 and j=0, it starts at k=0. The total is increased (well not really) to 0+0=0. k is then increased to 1, and the total becomes 0+1=1. k then becomes 2, so total=1+2=3. k becomes 3, so total=3+3=6. k becomes 4, so total=6+4=10. k then becomes out of range (because 5 doesn't count anymore), and j is increased to 1 (k is reset to 0). The entire process is repeated, and again, 10 is added. j increases to 2 (k is reset to 0), and another 10 is added. Then, i becomes one and j and k are reset to 0. The same thing happens all over again, so another 30 is added, leading to a final value of 60.

5.1.3 Question 3

Let's first ignore the function until we actually use it. i starts at one. As long as i stays smaller than f(i), it will do the statements in the whileloop. Now, what is f(i)? That's where the earlier function was defined for. If we plug in f(1), it will 'recognize' that we have stated n=1 in the function. It will thus return the value 22-1=21. Thus, for i=1, it'll obviously be correct, as 1<21, so it will execute the loop. i is increased to 2, and one stripe is added. f(2)=22-2=20, so for i=2, it'll also work and i becomes 3 and another stripe is added. You can continue on, and it'll only stop once i=11, because 11 is not smaller than f(11)=22-11=11. So, it'll print a stripe for i=1,2,...,10, and in total, 10 stripes will appear.

5.1.4 Question 4

np.arange is similar to how list(start,stop,step) works. You indicate a starting value, a stop value (which will *not* be included itself), and a stepvalue. So, np.arange(1.,15.,0.5) will give an array which contains the values 1., 1.5, 2. etc. all the way to 14.5. Now, let's take a closer look at what the for loop does.

[1] % indicates the modulo. For example, when you divide 11 by 3, the modulo is 2 (because you can get 9 'out' of 11, but then you're left with 2). In feite berekent het het restgetal.

xrange is the same as the regular range, it only differs 'behind the scenes'[2]. So, it'll check the numbers 0, 1, 2,...,28. It'll check when they are divided by 3, whether they are larger than 8. Now, please note that these are *integers* you are dividing by and equating with, not floats. So, when you divide by 3, the answer will be rounded down (e.g. 25/3=8.3333=8). So, we need to have i at least be 27 before i/3>8. The program will print vm[27], so the $27 + 1 = 28$th entry, which is 14.5. If i was increased to 28, it will fall out of range and thus our program is finished[3].

5.2 Part II: Debugging Python

5.2.1 Question 5

Everything seems perfectly OK. Note the following:

- Lists and arrays can be plotted at the same time, as long as their length is the same (so both the list and array must have the same number of entries, which makes sense).
- The small step is needed to prevent the graph from being inaccurate (as Python simply connects the dots, if they're very far apart, it starts to be inaccurater, etc.).

However, the only problem is in what is actually plotted. They only tell the plt.plot to plot the x and y-value, instead of the array and list. So, you only plot one point and you don't end up with a nice line or graph. So, answer E is correct.

Also note, the program could have been more nicely written as:

```
import numpy as np
import matplotlib.pyplot as plt
xtab = np.arange(0.0, 2.0*pi, 0.01)
ytab = np.sin(xtab)

plt.plot(xtab,ytab)
plt.show()
```

Listing 5.1: Alternative program for question 5

By using np.sin (note the required use of np.sin rather than sin), we execute the sine command on every entry in xtab to create ytab. This means ytab is an array as well.

5.2.2 Question 6

Initially b=0, so the while statement is correct, and b will be increased to 1. While statement is still correct, and b is increased to 2. The while statement stays true until and including when b reaches 10. When b reaches 10, it is increased one final time, so the program will print 11.

5.2.3 Question 7

The program will look for j in the range (0,4,2), so it'll only check the values 0 and 2 (because 4 is excluded). It'll first print 0/3, which is 0. Then, 2/3 will be printed, which is, since we divide by an integer here, will also be rounded down to 0[4]. This means the program will print '0' twice.

Please note that the following things would have resulted in TypeErrors:

- Replacing any of the entries in the range by floats. Floats are not accepted in a range.

[2] If you need to program for the exam, range is preferable, because when you use range, you apparently can do more. The only problem is that xrange uses far less memory and is hence preferable in large programs, but for this exam, memory isn't an issue so I guess range is safer.

[3] Had there been xrange(29), then the program would have given an error because vm[28] does not exist in the array. Had there been a 29th entry in the array, then that entry would have been printed as well, etc.

[4] Had we divided an integer by a float, then it'd give a float as result, however. The 'final' thing is dominant, so to say. Python works from left to right, so it'll first convert something to an integer, and then it'll convert it to a float if it encounters a float, etc.

5.2.4 Question 8

The most important thing here is the fact that it does not say print hola(8), but simply hola(8). hola(a) will do the following: it'll check if it is divisible by 3; if it is, then it prints "True". If it's indivisible by 3, then it prints "False". However, it returns nothing to hola(a). This means that if we have hola(8), it'll simply print "False", because 8 is indivisible by 3. If we had said print hola(8), we would have gotten "False", followed by "None": hola(8) itself has no returnvalue, and thus its print is "None".

5.3 Part III: supplement Python

5.3.1 Question 9

See the following program.

```
import matplotlib.pyplot as plt

g = 9.81 # m/s2

m = 20. # kg
c = 10. # N/m
k = 10. # Ns/m

y = 0. # m
vy = 0. # m

t = 0. # s
dt = 0.01 # s

ytab = []
vytab = []
ttab = []

while t<25:
    Fspring = -c*y # Spring force
    Fdamp = -k*vy # Damper force
    Fgrav = m*g # Gravity force

    Ftot = Fspring+Fdamp-Fgrav

    ay = Ftot/m
    vy = vy + ay*dt
    y = y + vy*dt

    t = t + dt

    ytab.append(y)
    vytab.append(vy)
    ttab.append(t)

# End of loop: plotting results

plt.subplot(211)
plt.plot(ttab,ytab)

plt.subplot(212)
plt.plot(ttab,vytab)
plt.show()
```

Listing 5.2: Question 9

Let me explain each line you have to add:

(a) t should be smaller than 25, as we want the plot to run for the first 25 seconds.
(b) For the resultant force, it is important that you are consistent in what forces are pointing up etc. I find it the easiest to define my coordinate system to be positive upwards, just because. This means that for each force:

  - The spring force: the spring will be compressed, so y will become increasingly negative, but it'll give a force upward (so positive in my coordinate system). With the way the exam defines Fspring, this checks out, so we can just take the positive value of it.
  - Similarly, we have for the damping force, if we define the velocity upward to be positive, that if there's a very large velocity upward, then there's a damping force pointing downward (so negative). The formula for the damping force already takes care of this, so again, we can just take the value as it is.
  - Fgrav always points downward. However, the formula currently gives it as positive. Thus, we must multiply it with -1 for the total force to make sure it points downwards in our calculations.

(c) The acceleration is simply the force divided by the mass.

(d) We only miss the appending of the ttab, so we'll do that now.

(e) Just look at the line just after this line. It's very similar to that line. plt.subplot(212) means that we want 2 columns, 1 row, and the 2nd graph. We now want the 1st graph, and thus we write plt.subplot(211)[5].

Note that there are of course multiple solutions to this questions (for example if you established your coordinate system differently).

[5] If we have multiple grpahs, the order is determined by first going from left to right, then going down one row, etc. If you have a 3×3 plot, the third plot is the plot in the topright.

6 July 2014

6.1 Part I: Reading Python

6.1.1 Question 1

We start with n=1. Then, it will check the values 0, 3, 6 and 9. If i is divisible by 2, it'll double the value of 0. So, for i=0, which is divisible by 2, n will be increased to n=1+1=2. For i=3, nothing happens. For i=6, n is increased to n=2+2=4. For i=9, nothing happens.

So, the final answer is 4.

6.1.2 Question 2

We start at count=0 and n=4. It will then enter the for loop, where i will be in range(4). Let's start with i=0: range(0) does not contain any numbers (not even zero), so we immediately go to i=1. Then, j=0 is checked, and count is increased by 1. j=1 falls outside the range, so i is increased to 2. j=0 and j=1 increase count each by 1, so count now equals 3. For i=3, j=0, j=1 and j=2 each increase the count by one, so the count now equals 6. i will not increase to 4, so the final count printed is 6.

6.1.3 Question 3

We start at n=0. The whilestatement is true, so n is increased to n=0+1=1. k=1 as well. For i in range(1), n is increased by i. Only 0 is in range(1), so n=1+0=1. n is then increased by 5 to n=1+5=6, and the loop is started again. n is increased to n=6+1=7. k=7, and the values i=0, i=1,...,i=6 are checked, each increasing n by the value of i, so n becomes 7+0+1+2+3+4+5+6=28. Finally, n is increased by 5, so n=28+5=33, meaning it'll print 33.

6.1.4 Question 4

First, ignore the function until we actually need to use it. encrypt('Hello World'.lower()) means that it needs to 'encrpypt' the string 'Hello world' after it has made all characters lower case. Now, note that when performing actions on a string, the '' are basically ignored (they are merely used to indicate to the computer that these are strings). Thus, when we plug in 'hello world' in the function, we get the following process:

The length of "hello world" is 11 (5 characters, one space, 5 characters). We start at i=0. As long as i is smaller than 11-6=5, the whileloop will be performed. In the whileloop, for i=0, the 1st entry in the string (thus h) will be chosen, and it will be converted to an ordinary number, following ASCII. Then, 1 is added to this ASCII-number, and it is converted back to a character (thus basically, an a becomes b, b becomes c, etc.). This value is printed, and i is increased by 1. Thus, this process repeats for i=1,i=2,i=3 and i=4 (because i=5 would mean the whileloop condition is no longer met, as 5 is not smaller than 5), and the second to fifth character are also increased by one. So, it'll print the letters i, f, m, m and p. Now, how exactly it is printed depends on how the print statement is stated. The comma at the end indicates that when it starts printing, the next print may be printed right after it, with a space inbetween. So, it'll print: i f m m p

If there hadn't been a comma there, the next print statement would have been created on a new line, and you'd have gotten:

i

f

m

m

p

6.1.5 Question 5

Note the two ways to create an array:

- np.linspace makes an array by looking the start and stop value, followed by the *number of steps* in between. Contrary to what you're used to, however, it will *include* the stop value. So, it'll consist of 0., 2.5, 5., 7.5, 10.
- np.arange makes an array based on the start and stop value, followed by the *step size*. It does not include the stop value. So, it'll create an array containing the values -10.,-9.9,...,9.9 (200 values in total).

Now, a[-1] indicates the final entry in a[1] and b[1] the second entry in b. So, the value 10.+-9.9=0.1 will be printed.

6.1.6 Question 6

a=array(range(1,21)) means that it creates an array of the list [1,2,3,...,20]. The wonderful thing about arrays is that it is a lot easier to just select values that meet certain conditions, instead of first having to scroll through a list, checking which entries meet the requirements (requiring a for-loop) and then capturing these in a list. sum(a[(a%4==0)+(a%5==0)]) means that it will take the sum of the values in the array that are divisible by either 4 or 5 (the two statements are a%4==0 and a%5==0. These are combined with a 'OR' statement via +. The reason for the brackets around them is that otherwise you'd get a%4==0+a%5==0, which is very bad math of course. So, it'll take the values 4, 5, 8, 10, 12, 15, 16, 20, which has a total of 90. So, it'll print 90.

6.2 Part II: Debugging Python

6.2.1 Question 7

Considering the error only occurs in line 4, we know that A-C are wrong, as if there was an error with np.ones(), it would have shown earlier.

The problem arises with y=cos(x) because it uses the math.cos, but that's incompatible with an array. You need to use np.cos() to perform the action on each value in the array. So, E is correct.

6.2.2 Question 8

Once again a question that's different based on Python2/Python3. Assume we use Python2. The program contains no error whatsoever. However, the program won't be doing what you want it to be doing. Let's see what happens:

1. For i=0, x=20+0/20=20
2. For i=1, x=20+1/20=20
3. For i=2, x=20+2/20=20
4. For i=3, x=20+3/20=20
5. For i=4, x=20+4/20=20

Why? Because x is an integer, if we divide for examnple 2 by 20, which equals 0.1, it is converted back to an integer (because it takes the type of the last value (so an integer in this case). So, it is rounded down to 0, and nothing happens to x. This is not what we wanted, so D is correct. Had x=20. in the beginning, we'd been fine.

[1] The - indicates that you scroll through it from back to front. Only difference is that now you do start at 1 instead of 0.

6.2.3 *Question 9*

The problem already arises in the fourth line: range does not accept floats, only integers. arange needs to be used here.

6.2.4 *Question 10*

I'm not entirely sure what options would be considered correct. Two possibilities are:

- In line 6): h0=25.0*np.ones(len(t))
- In line 11): plt.plot(t,h0*np.ones(len(t)),'b.')

What is done is both times is that an array is created containing only 1 as entries, with the same length as the length of the array t. Then, everything is multiplied by 25.0, which means we now have an array which contains of only 25.0, but which contains the same number of entries as t (so for each t, there's a 25.0). This allows us to graph the two arrays.

I'd be honest that I don't know how correct the combination of these two possibilities are, because basically they're exactly the same, but the second one just procrastinates making an array of h0. However, it should be allowed to write down both of these as it answers the question, don't they?

7 January 2012

7.1 Part I: Reading Python

7.1.1 Question 1

Let's just go through this step by stap. i and flag are initially 0 and stop is false. Then, i is increased by 1. If i is divisible by 7, several stuff happens, but otherwise, the while loop is restarted. So, once i is increased to 7, flag is increased to 1. Once i hits 14, flag is increased to 2, and stop becomes true. So 14 will be printed.

7.1.2 Question 2

Again, let's just see what happens. N equals 4, and Arr is an empty list initially.

i is thus scrolled between 0 and 3. Then, for each i, j is scrolled between 0 and 3. If either i or j are zero, or equal to 4-1=3, then the list Arr is appended by the tuple of i and j. So, for i=0 and j=0, Arr is appended by (0,0). For i=0 and j=1, it is appended by (0,1), same goes for (0,2) and (0,3). (1,0) is appended, but (1,1) and (1,2) are not, but (1,3) is appended. Same goes for (2,0) and (2,3), and finally (3,0), (3,1), (3,2) and (3,3), leading to: [(0,0),(0,1),(0,2),(0,3),(1,0),(1,3),(2,0),(2,3),(3,0),(3,1),(3,2),(3,3)].

7.1.3 Question 3

Note how a is divided by 10 rather than 10. So, i needs to be 20 before a becomes 2, and thus 20 is printed.

7.1.4 Question 4

two_words.find(' ') will yield 1 (because the space occurs one). Thus, two_words[:1] yields the first element, thus hallo. Similarly, second will be 1+1::, which are all elements except the first two (so, the first word and the space), so it'll yield world. Thus, world hallo is printed.

7.1.5 Question 5

So, it'll print those with index maximum 20, entries that are smaller than 22, but whose square root is larger than 3, i.e.:

10

11

12

13

14

15

16

17

18

19

20

7.1.6 Question 6

Let's go through this step by step. f(10,120) will be checked, so in the function, n=10 and a=120. As 10/120=0 (mind the integers), and thus n=10%120=10. Thus, a,n=120,10 is returned. n,a=120,10, so a,n is 10,120 and 10 120 will be printed.

7.2 Part II: Debugging Python

7.2.1 Question 7

Answer C is correct. Note that `condition` is `True`. However, in the for-loop, `a=a+1` is only executed when `condition` is not `True`. This is never the case, as `condition` is never updated. Thus, `a=a+1` is never executed and `a` remains zero, whereas `condition` remains True. Thus, answer C is correct.

7.2.2 Question 8

Answer B is correct. The `if` statement shouldn't be indented, obviously.

7.2.3 Question 9

Answer B is correct. The error is admittedly pretty hard to spot in this program, as almost everything seems fine. However, not that the value of `t` is never reset after the while-loop! This means that after the first mass, the value of `t` is still 3 s, and it's not reset back to 0 s, such that the second, third, fourth and fifth mass all will have their while-loops immediately exited. Thus, answer B is correct.

8 July 2017

This exam was arguably rather hard and last year they bumped up the grades a tiny bit iirc (in the sense that you only needed 45 points to pass rather than 48).

8.1 Part I: What will this print?

8.1.1 Question 1

Okay so once again this question is fucked up in Python3, but in Python2, the first line would create two lists, `[0,3,6,9,12,15,18]` and `[0,4,8,12,16]`, and then puts them together, i.e. `a=[0,3,6,9,12,15,18,0,4,8,12,16]`. The second line then prints a list with the entries of `a` in reverse order, with step size 2, in other words, it'll print `[16,8,0,15,9,3]`.

8.1.2 Question 2

Let's just go through this program line by line:

- We first have `i=0`, such that `j in range(1)`. The starting value of `j` is thus 0, and `tritotal` is incremented by `A[0][0]` i.e. the first row (first index), first column (second index), which is 1. Thus, `tritotal` equals 1 now.
- Then we have `i=1`, such that `j in range(2)`. Thus `j` starts at 0, and `tritotal` is incremented by `A[1][0]`, i.e. second row, first column, which is 4. So, `tritotal=5` now. After that, `j` is incremented to 1, and `tritotal` is thus incremented by `A[1][1]`, i.e. second row, second column, which is 5. Thus, `tritoal=10` now. After this, the `j`-for-loop is exited.
- Then we have `i=2`, such that `j in range(3)`. Thus, `j` starts at 0, and `tritotal` is incremented by `A[2][0]`, i.e. 7, so that `tritotal=17`. Then `j` equals 1 and `tritotal` is incremented by `A[2][1]`, i.e. third row, second column, equal to 8. So, `tritotal=25`. Finally, `j` equals 2 and `tritotal` is incremented by `A[2][2]`, i.e. third row, third column of `A`, which is 9. Thus, `tritotal=34`. After this, the `j`-loop is exited, and then the `i`-loop is exited.

Thus, 34 is printed.

8.1.3 Question 3

Again, let's just go through it line by line. We start with `i=2`. Then, we set `swp` equal to `True`. Then, we have `for k in range(2,2)`, but `range(2,2)` is nothing (since 2 itself is not included, and that overrules the fact that it starts at 2). Thus, we continue to the `if` statement: `if swp` evaluates to `True`, so we print the current value of `i`, i.e. we print 2. Then, `i` is incremented to 3. `swp` is set to `True`, and now we have `k in range(2,3)`, thus we check `k=2`. We check if `3%2==0`, which is not true (as `3%2=1`, as `%` is the modulo, i.e. the result is the remainder after division). Thus, `swp` is *not* set to false, and as `k` is only in `range(2,3)`, the `k`-loop is exited. Then, as `swp` is still `True`, we print the current value of `i`, i.e. 3.

Now then, onto `i=4`. We again set `swp` to `True`, and then evaluate `k in range(2,4)`, i.e. `k` takes the values 2 and 3. We check if for either of them whether `i` is divisible by it, and if it's the case for at least one of them, we set `swp` equal to false. 4 is divisible by 2, so we set `swp` equal to `False`, and we *don't* print 4.

Now, you can continue with this until you're at `i=11`, but that'd take a bit of time, and although I'm normally of just playing it safe, it'd start getting being a bit of a waste of time. You should realise that we're essentially printing the prime numbers up to and including 11: we check for the numbers between 2 and `11` whether it is

divisible by any numbers between 2 and smaller than the number. This is just checking whether something is a prime. Thus, we print the prime numbers, and thus we print `2 3 5 7 11`.

8.1.4 Question 4

Let's first look at the main part of the code, outside of the function definition. `a=linspace(0,50,51)` simply creates a numpy array of `[0,1,...,49,50]` (note that for `np.linspace`, the endpoint, by exception, is *in*cluded. Now, `f(a)` simply squares all of these entries, so we now have an array of `[0,1,4,9,...,2401,2500]`. `cond = f(a)<30` then makes another array of the same size as `a`, but with a 1 in places where `f(a)` is smaller than 30, and 0 in places where it's equal to or larger than 30. In other words, it'll create an array `cond=[1,1,1,1,1,1,0,0,...,0,0]` as only the first six squares are smaller than 30. Multiplying this with `a` yields `[0,1,4,9,16,25,0,0,...,0,0]` such that the sum is 55. Thus, we print 55.

8.1.5 Question 5

Lines 3 and 4 speak for themselves really. `t=arange(0,10.1,0.1)` creates a numpy array starting at 0, ending at (not including) 10.1, with step size 0.1, i.e. `t=[0,0.1,...,9.9,10.0]`. `y=h0-0.5*g*t*t` then computes `y` for each entry of `t`, i.e. it'll create `y=[10,9.95,9.8,...,-480.05,-490]`. Now, `ft=t[y>=0.0]` will select those entries from `t` where the corresponding entry of `y` is larger or equal than 0. The exact time at which `y` equals 0 is easy to compute with basic dynamics[1]:

$$\begin{aligned} 0 &= 10 - 0.5 \cdot 10 \cdot t^2 = 10 - 5t^2 \\ t^2 &= 2 \\ t &= 1.414\,\text{s} \end{aligned}$$

Thus, all entries of `t` up until and including `1.4` will be included in `ft`; all entries after that have `y<0` and are thus ignored. Thus, `ft=[0,0.1,...,1.3,1.4]`. The maximum value in this array is `1.4` evidently, so we print `time = 1.4 s`.

8.2 Part II: Debug the program

8.2.1 Question 6

There should be a square bracket at the end of `sol = [-b/(2.0*a)`. Note that the question itself is a bit weird as if you run the program it doesn't actually give you an error.

8.2.2 Question 7

Answer F is correct. Answer A would be correct except it doesn't work. Answer B is not true, and C is just wrong as well. D doesn't make sense, and E would be partly true if you'd ask for a a polynomial for which the discriminant is not larger than 0. Answer F is correct, as when you write `print "D = ",D`, `D` has never been defined before: even if you put it aft the of the second print statement (where you call `solve(a,b,c,)`, it still wouldn't work: `D` is defined within the function itself, and is thus an inner-scope variable. You'd have to explicitly return the value of `D`, and then save this value of `D` in a variable named `D` before you'd be allowed to write this.

8.2.3 Question 8

Okay so once again it's a question that depends on Python2. The correct answer is B: if you write `float((x-a)/(b-a))`, it'll still perform the operation `(x-a)/(b-a)` as if they're integers (if you input them as integers). Thus, for

[1] Admittedly, basic dynamics is already difficult enough for you first years but it's the thought that counts.

the values that are passed to the function, we'd get `(5-3)/(10-3)=2/7=0`. Instead, you'd need to write `float(x-a)/(a-b)` to make sure everything happens with floats. Thus, answer B is correct, as that's the line that should be changed.

8.3 Part III: Complete the program

24 points for filling in the blanks is massive. Let's go through the things one-by-one:

a) Simply `np.arange(0,360,1)`.
b) They say either until one day or crash into spherical Earth: thus, we must complement it with `or h<0`.
c) This is arguably the hardest part of the calculations. How is θ ever defined? Well, consider the FBD of figure 8.1. The forces are defined positive in positive x and y direction. Now, we are already given that `Fgx = -Fg*cos(theta)`. So, we must define `theta` to be consistent with this: by defining it in the conventional way, so counterclockwise, measured from the positive x-axis, this formula would be consistent with the given definition of `Fgx`. Thus, we simply have `theta=arctan2(y,x)`, where I use `arctan2` as it's usually the safer option.
d) Then, how do we define `Fgy`? Well, like I said, I take `Fgy` to be positive in positive `y` direction. You'd be absolutely fine to do it positive in negative `y`-direction! However, then you need to take another minus sign into account for `ay` and it gets annoying. Now, if you look again at Figure 8.1, it's clear that we now must take `Fgy=-Fg*sin(theta)`: for $0 \leq \theta \leq 180°$, `Fg` will point in negative `y`-direction.
e) Simply `Fgx/msat`.
f) Simply `Fgy/msat`.
g) So we need to plot the spherical Earth in a red line. This is easily done using `plt.plot(xcir,ycir,'r')`.
h) So we need to set the `y`-axis range from 0 to `max(htab)`. Meanwhile, the x-axis should naturally run between 0 and `max(ttab)`. This is simply done with `plt.axis([0,max(ttab),0,max(htab)])`.

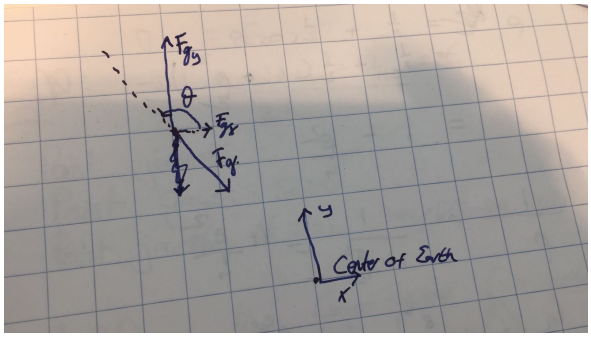

Figure 8.1: FBD.

8.4 Part IV

These were all pretty difficult generally, but once more, part of the difficulty lies in people trying to do it too fancy rather than just going for the easy solution that's just really lazy programming.

8.4.1 Question 10

Okay so you can write a program that actually solves the equation in some form or another. You can also take it easier, and limit yourself to writing a program that computes the volume given a given height h, and then check see what value h brings it closest to the desired 0.750 dm^3; due to the fact that only one digit is desired, this is very easily done.

So, we limit ourselves to writing a program where we define an input `h`, and then compute the corresponding volume. How do we do this? The solution is shown in the code below.

```
import math as m

h = 4.014

y = 16
ymin = 16-h
dy = 0.0001
V = 0

while y>ymin:
    r = 2 + y*m.sin(m.sinh(y/8))
    V = V + m.pi*r**2*dy
    y = y - dy

print(V/1000)
```

- Line 3 defines the line where we can input our guess for `h`.
- Line 5 makes sure we start integrating at `y=16`, whereas line 6 makes sure we stop integrating at `y=16-h`.
- Line 7 and 8 set the simulation parameters, so `dy=0.0001` and `V=0`.
- Line 10 defines the integration: we'll run for as long as `y>ymin`.
- Line 11 computes the radius of the cylinder for a given `y`.
- Line 12 updates the volume with $\pi r^2 \Delta y$.
- Line 13 updates `y`, taking into account that we're going to *smaller* values of `y`. This loop is continued until we reach `ymin`.

Just trying and guessing a few values leads to `h=4.014` being the correct answer; rounding to 1 digit yields `h=4.0`.

See, if you just do it the ugly way, it's not that hard actually.

8.4.2 *Question 11*

Again, a question that you can do super beautiful so that your parents will be very proud of you, or you can also just think about the beer you'll drink after the exam and get over with it quickly but successfully.

At a glance, there are probably two things that seem difficult: first, we must somehow count how many of each digit there are, and second we must then compare these numbers.

For the first task, I hope you remember from earlier exams that key to this is that you convert the numbers to strings. After that, you can loop through the strings and start counting somehow. Let's look at my solution below.

```
import numpy as np

total = 0
for number in range(10000,100000):
    istriplet = False
    string = str(number)
    lst = np.zeros(10)
    for digit in string:
        lst[int(digit)] = lst[int(digit)]+1
    newnumber = 3*number
    newstring = str(newnumber)
    newlst = np.zeros(10)
    for newdigit in newstring:
        newlst[int(newdigit)] = newlst[int(newdigit)]+1
    difference = lst-newlst
    if np.sum(np.abs(difference))<0.00001:
        istriplet = True

    if istriplet:
        print("Number is: ", number, " as its triplet is ", newnumber)
        total = total + number

print(total)
```

- In line 4, we initiate the for-loop; we have to loop through all 5-digit numbers, i.e. between 10000 and 100000.
- In line 5, we set our boolean `istriplet` equal to `False`.
- In line 6, we convert the number to a string. In line 7, we make a numpy array consisting of all zeros, of size 10, so that each entry corresponds to a digit between 0 and 9.
- In line 8-9, we count the number of each digit. The way to do it is actually remarkably simple: we convert the digit we're evaluating (which is originally a string, as we're looping through a string), back to an integer. We reserve the first entry in `lst` (so index 0) for digit 0, the second entry (index 1) for digit 1, etc. This way, we can extremely easily count digits, as everytime we see an integer `i`, we'll simply increment the entry in the list corresponding to index `i` by 1.
- Line 10-14 do essentially the same, but then for the number multiplied with 3.
- Line 15 computes the difference between the two numpy arrays: this is the reason why I used numpy arrays, as they allow for easy subtraction and stuff (whereas with lists subtraction is not possible).
- Line 16 computes the sum of the absolute differences between the two numpy arrays and checks whether it's sufficiently close to 0. If the number and triplet match with respect to how often each digit occurs, then the differences between the two arrays will be 0. However, if the number of occurences of each digit differ, then naturally the `differences` array will not consist of only zeros, as then obviously one (actually at least two) of the entries would cause a difference between the two arrays. Taking the absolute values of the differences is necessary, as otherwise obviously you'd always get that the sum of the differences is zero. Finally, I compare it with `<0.00001`, as rounding errors are always annoying so this way I just circumvent them if they'd be present (writing `==0.` is pretty bad practice as if any rounding occurs (which is very, very likely in regular computations), you're already fucked).
- If the difference is 0, then the `istriplet` is set to True; if this is the case, then the number will be printed together with its triplet, and the `total` will be incremented by the original number.
- At the end, the `total` is printed.

This results in 13 triplets: 10035, 10350, 12375, 14247, 14724, 23751, 24147, 24714, 24750, 24876, 24975, 27585, 28575. The sum of these is 265104.

8.4.3 *Question 12*

Okay personally I really don't like questions where I have to input strings, so I'm just gonna include the most stupid-ass way to do this (but it's still correct so use it as inspiration how you can sometimes just completely disregard the way they'd want you to solve the question and just hardcode everything). The code is shown below and looks super long, but it's really easy:

```
sentences = '...
    ARTHURNOWSTANDASIDEWORTHYADVERSARYBLACKKNIGHTTISBUTASCRATCHARTHURASCRATCHYOURARMSOFF...
    '

lst = []
for letter in sentences:
    if letter == 'A':
        lst.append(letter)

for letter in sentences:
    if letter == 'E':
        lst.append(letter)

for letter in sentences:
    if letter == 'I':
        lst.append(letter)

for letter in sentences:
    if letter == 'O':
        lst.append(letter)

for letter in sentences:
    if letter == 'U':
        lst.append(letter)

for letter in sentences:
    if letter == 'B':
```

```
            lst.append(letter)
    for letter in sentences:
        if letter == 'C':
            lst.append(letter)

    for letter in sentences:
        if letter == 'D':
            lst.append(letter)

    for letter in sentences:
        if letter == 'F':
            lst.append(letter)

    for letter in sentences:
        if letter == 'G':
            lst.append(letter)

    for letter in sentences:
        if letter == 'H':
            lst.append(letter)

    for letter in sentences:
        if letter == 'J':
            lst.append(letter)

    for letter in sentences:
        if letter == 'K':
            lst.append(letter)

    for letter in sentences:
        if letter == 'L':
            lst.append(letter)

    for letter in sentences:
        if letter == 'M':
            lst.append(letter)

    for letter in sentences:
        if letter == 'N':
            lst.append(letter)

    for letter in sentences:
        if letter == 'P':
            lst.append(letter)

    for letter in sentences:
        if letter == 'Q':
            lst.append(letter)

    for letter in sentences:
        if letter == 'R':
            lst.append(letter)

    for letter in sentences:
        if letter == 'S':
            lst.append(letter)

    for letter in sentences:
        if letter == 'T':
            lst.append(letter)

    for letter in sentences:
        if letter == 'V':
            lst.append(letter)

    for letter in sentences:
        if letter == 'W':
            lst.append(letter)

    for letter in sentences:
        if letter == 'X':
            lst.append(letter)
```

```
    for letter in sentences:
        if letter == 'Y':
            lst.append(letter)


    for letter in sentences:
        if letter == 'Z':
            lst.append(letter)




    print(lst[44], lst[49], lst[16], lst[37], lst[41], lst[72])
```

There are a few ugly things I'm doing:

- First off, they tell you to ignore spaces and symbols, and to make no distinction between upper and lower case letters. Therefore, why not just be lazy as fuck and type everything over as if it were capitals, and ignore the spaces and spaces in the meanwhile? And while we're at it, also remove the newline in between. This nicely gets rid of some issues.
- Secondly, it's arguably the most lazy-ass way of sorting the characters. However, it works, and is very simple: we first loop through all of the letters and check if it's an A. Then we loop through the entire list and check whether letters are an E, etc. etc. This is simply a lot of copy-pasting and changing some letters (and remembering the alphabet which is hard enough). It's honestly not that difficult. And you can boast to your friends from industrial design that you wrote a hundred-line code on the exam in just a few minutes!

The final result is then simply K N I G H T.

It's honestly not difficult at all if you do it like this, and why bother improving it if there's only one exercise left before you get that holiday?

8.4.4 *Question 13*

Arguably the most difficult one of the exam, not the least due to the fact they expect you to read like a full page of text that no one wants to read. Furthermore I personally don't really see how someone who's not good in Python is supposed to do this on their own, so I'll just present my own solution.

```
import itertools
import math as m

Delft = [["D",0,0]]
cities = [["A",37,39],["N",101,-22],["Z",118,56],["G",151,134],
          ["L",99,132],["R",8,-10],["U",52,9],["E",174,23],["B",28,-49]]

listofindices = range(9)

firstbest = 0
secondbest = 0
thirdbest = 0
firstbestdistance = 100000000000000
secondbestdistance = 100000000000000
thirdbestdistance = 100000000000000

for irte in itertools.permutations(listofindices):
    distance = 0
    distance = distance + m.sqrt(cities[irte[0]][1]**2+cities[irte[0]][2]**2)
    for i in range(8):
        indexcurrentcity = irte[i]
        indexnextcity = irte[i+1]
        distance = distance + m.sqrt((cities[indexcurrentcity][1]-
                                      cities[indexnextcity][1])**2
                                     +(cities[indexcurrentcity][2]
                                       -cities[indexnextcity][2])**2)
    distance = distance + m.sqrt(cities[irte[-1]][1]**2+cities[irte[-1]][2]**2)
```

```
        if distance < firstbestdistance and not (irte[::-1] == firstbest or irte[::-1] ...
        == secondbest or irte[::-1] == thirdbest):
            thirdbestdistance = secondbestdistance
            secondbestdistance = firstbestdistance
            firstbestdistance = distance
            thirdbest = secondbest
            secondbest = firstbest
            firstbest = irte
        elif distance < secondbestdistance and not (irte[::-1] == firstbest or irte...
        [::-1] == secondbest or irte[::-1] == thirdbest):
            thirdbestdistance = secondbestdistance
            secondbestdistance = distance
            thirdbest = secondbest
            secondbest = irte
        elif distance < thirdbestdistance and not (irte[::-1] == firstbest or irte...
        [::-1] == secondbest or irte[::-1] == thirdbest):
            thirdbestdistance = distance
            thirdbest = irte

    print("Best combination is: ")
    print("D - ")
    for i in range(9):
        print(cities[firstbest[i]][0], "-")
    print("D")
    print("The associated distance is: ", firstbestdistance)

    print("Second best combination is: ")
    print("D - ")
    for i in range(9):
        print(cities[secondbest[i]][0], "-")
    print("D")
    print("The associated distance is: ", secondbestdistance)

    print("Third best combination is: ")
    print("D - ")
    for i in range(9):
        print(cities[thirdbest[i]][0], "-")
    print("D")
    print("The associated distance is: ", thirdbestdistance)
```

The code works as follows: there are 9 cities that are not Delft.

- In line 5-6, there are 9 cities that are not Delft. These are stored in a two-dimensional list: the first dimension is the city, the second dimension contains the first letter of it, the x-position and the y-position.
- In line 8, we create a list of indices, similar to `idxlist = range(3)` as listed in the code. Essentially, what it does is that as we have 9 cities, our indices will range between 0 and (including) 8. `itertools` is able to shuffle these in all possible permutations, such that we are able to go through all possible permutations of cities.
- In line 10-15 we create variables in which we'll store the best performing orders of cities.
- In line 17, the hell breaks loose. `for irte in itertools.permutations(listofindices)` will shuffle the list `[0,1,2,3,4,5,6,7,8]` in all possible orders; `iter` is then one of those orders each time (e.g. it could equal `[5,2,3,6,1,8,0,7,4]`, meaning that the city 5 is visited first, then city 2, then city 3, etc.). Thus, the following calculations are done for each possible order of cities, and `iter` represents one possible order.
- In line 18-19, we compute the distance it takes to travel from Delft to the first city in the list. As Delft has coordinates $(0, 0)$, Pythagoras becomes really easy.
- In lines 20-26, we compute the distance from city i to city $i + 1$. We start looping at `i=0`. The current city then has the index `iter[0]`, i.e. the number listed first in the order of indices that's currently being analysed (so it'd be city 5, i.e. Rotterdam, if the 'random' order I listed above is being analysed. The next city has the index `iter[1]`, i.e. the second number listed in the order of indices that's currently being analysed (so it'd be city 2, i.e. Zwolle, if the 'random' order I listed above is being analysed). We then compute the distance, by simply looking up these indices in the `cities` list and looking for their x and y coordinates. Pythagoras is then easy.
  Then we continue to the next `i`. Now `indexcurrentcity` is Zwolle, and `indexnextcity` is 3, i.e. Groningen, etc. etc. and sum the distances, until our next city is the last index in the order.

- Finally, in line 27, we add the distance of the final city to Delft.
- In lines 28-34 we check whether the calculated distance is the best distance so far. If it is, then the third best distance is replaced with the second best distance, after which the second best distance is replaced with the previously best distance. Then the best distance becomes equal to the newly calculated distance. The same is done for the order: the third best order become the second best order, etc. etc.
- In lines 35-39 it is checked, if the distance is not better than the current best distance, whether the distance is better than the *second*-best distance. If so, then the third-best distance is replaced with the second-best distance, and the second-best distance is replaced with the current distance, etc. etc. Note that it is checked whether the order that's being analysed happens to be the reverse order of one of the ones that's already one of the best scores; if so, it should be ignored.
- In lines 40-42 it is checked whether the distance is better than the third-best distance, if it wasn't already better than the (second-)best distance. Note that it is checked whether the order that's being analysed happens to be the reverse order of one of the ones that's already one of the best scores; if so, it should be ignored.
- In lines 44-63, we start printing everything. We saved the best orders, and now can simply look up the corresponding first letter for them.

The results are:

- D-R-B-N-E-Z-G-L-A-U-D: 620.04 km
- D-R-U-A-L-G-Z-E-N-B-D: 627.80 km
- D-A-L-G-Z-E-N-U-B-R-D: 630.41 km

But honestly I have no idea who came up with the idea of including this question in the exam[2].

[2]If you're gonna include a question that's almost impossible to solve at least be like me and make it Christmas-themed or something (like I did for one of the bonus questions of the Dynamics exams, in case you ever wondered who came up with the idea of making such a question).


the
Flying
Sloth

9 August 2017

9.1 Part I: What will this print?

Egghhh no idea why some of the codes here are so long.

9.1.1 Question 1

Let's just go through this code line-by-line as it's way too complicated to immediately see a pattern.

We start with `eat=0`. Then we loop through each entry `x` in `a`, so the first value of `x` is `-1`. At this time `eat` is not larger than 0 (as it is equal to 0), so we'll evaluate the `else`. We then check if `x<0`: this is definitely the case. Thus, `eat=--1=1`. We then go to the next `x`. This time, `eat>0`, so we subtract 1 from `eat`, making `eat=0`.

Then we go to `x=2`. This time, `eat>0` is not true, so we go to the `else`. `x<0` is also not true, so we simply print `x`, meaning we print 4.

After that, we go to `x=7`. Again, `eat>0` is not true and neither is `x<0`, so we print 7. We then go to `x=-2`, and this time `x<0` does evaluate to true. So, we make `eat=--2=2`, and then continue with `x=9`. However, now `eat>0`, so we simply subtract 1 from `eat`, making it 1. Then we take `x=3`, but again we simply end up subtracting 1 from `eat`, making it 0. Finally, we analyse `x=5`: now `eat>0` is no longer true, and neither is `x<0`, so we simply print 5.

In total, this means we print 4 7 5.

9.1.2 Question 2

Egghh so long but anyway. Line 1-5 just defines a poem that I fail to see the deeper meaning of. In line 7, the poem is split; this means that a list is created with each entry equal to one of the words of `poem` (as separation occurs whenever a space occurs, so `didn't` will also be stored as one word).

Then, what the hell are lines 8-10 doing? Well, first `tmp` is made equal to word 2, 7, 12, 17 etc. Then, in `parts` itself, word 2, 7, 12, 17 etc. are overwritten by words 4, 9, 14, 19 etc. Then, those words are again overwritten the words that were originally stored in `tmp`. So to conclude: word 2 is switched with word 4; word 7 is switched with word 9, etc. etc. Thus, we now have the poem:

> much to his dad and mum's dismay
> horace ate himself one day
> he didn't stop to say his grace
> he just sat down and ate his face

Then, look carefully at the for-loop: what is the if-statement ever trying to achieve? Whether it's true or not the output is exactly the same. So simply the entire poem is printed, and you can read the poem as above.

9.1.3 Question 3

`lower()` makes the entire string lower case; `split()` splits it into a list of which the entries were originally separated by a space. `sort()` will then alphabetically sort them, in other words, the list will now look like `["captain", "hello", "is", "speaking", "this", "your"]`. Then, in line 3, we print the entries starting at the last entry (-1), up until the second-to-last entry (-3) (not including -3 itself), with step -1. In other words, we'll print `["your", "this"]`.

9.1.4 Question 4

Eghhh absolutely horrible looking code. But, we'll give it a try, once again by just going through it line-by-line.

We first take `i=0`. Then we take for `lastrow` to be the last entry of `tri`, which is simply 1 in this case. We make a new list, `newrow`, that's initially empty. Then we iterate over `j`: `lastrow` currently has a length of 1, so we actually have `for j in range(2)`. We start with `j=0`, such that the first statement evaluates to True (as `j==0` is true). Thus, we append `newrow` with a 1. Then we go to `j=1`, which is equal to `len(lastrow)` so again we append `newrow` with a 1.

Finally, we append `tri` with this `newrow`, such that `tri= [ [1], [1,1] ]`; note the fact that this list is two-dimensional. We then go to `i=1`. We now take `tri[-1]` once more; this time it'll equal the sublist `lastrow = [1,1]`. We create an empty `newrow`, and then loop `j in range(3)`, as the length of `lastrow` is clearly 2 now. We first take `j=0`, such that the first if-statement evaluates to `True`, and thus we append `newrow` with a 1. For `j=1`, the if-statement does not evaluate to `True` anymore, as `len(lastrow)=2`. Instead, we'll append `newrow` with `lastrow[0]+lastrow[1]`, so that it becomes 2. Finally, for `j=2`, we once more append `newrow` with a 1, such that `newrow` now consists of 1, 2, 1. We then append `tri` with this `newrow`, such that `tri` looks like `tri=[[1],[1,1],[1,2,1]]`.

Then, finally, what happens for `i=2`? Well, now `lastrow = [1,2,1]`, and we create again a new empty `newrow`. Then, `j` is in range `j in range(4)`, as the length of lastrow is now 3. For `j=0`, we append a 1 to `newrow`; for `j=1` we append `lastrow[0]+lastrow[1]=1+2=3`; for `j=2` we append `lastrow[1+lastrow[2]=2+1=3` and for `j=3` we append again a simple `1`. Thus, in the end, `newrow = [1,3,3,1]`, and `tri` becomes `[[1],[1,1,],[1,2,1],[1,3,3,1]]`.

9.1.5 Question 5

First, `a=[0.,0.5,1.,...,5.5,6.]` as `np.arange` does not include the endpoint (only `np.linspace` does). Then, `a[::2]` simply prints all values with even index, i.e. `[0.,1.,2.,3.,4.,5.,6.]`.

9.1.6 Question 6

Okok so the first few lines should be pretty straightforward. `nexty=y[1:]` simply creates a new array that contains all values of `y` except for the first. Similarly, `y=y[:-1]` copies all the values except for the last. Thus, essentially `y` and `nexty` are offset by one index, that is: `y[i]=nexty[i-1]`. Finally, `t=t[:-1]` does the same as the previous line, i.e. now `t=[0,0.01,0.02,...,14.98,14.99]` rather than including 15.00. Now, what does line 11 do? Well, `(y<0.0)` returns an array of True/False that's true whenever `y` is smaller than 0.0 and false if not; similarly, `(nexty>0.0)` creates an array that's true wherever `nexty` is larger than 0 and false elsewhere. Multiplying these two arrays is essentially an `and` statement: only if for the same index `y` smaller is than 0.0, and `newy` larger than 0.0, `True` is returned. Now, when is this the case? This is the case if the current value of `y` is still negative, but the next number would be positive. This is the case for `t=6.28` and `t=12.56`: sin(6.28) and sin(12.56) are still both smaller than 0, but sin(6.29) and sin(12.57) are both larger than 0. Thus, it'll print `[6.28 12.56]`.

9.2 Part II: Debug the programs

9.2.1 Question 7

Egghhh why is this exam so much reading. Anyway, the statements are:

A) Depending on whether you use Python2 or Python3. In Python2, you'd need to include `''` to make sure it's a string; in Python3 it's automatically assumed to be a string (so it's False).
B) True. It just does.
C) False. Just not true.

D) False. This is never specified to occur. Only lines that begin with a `C` are ignored.
E) False: it is actually checking whether the touch is *outside* the circle, due to the comparison `> r**2` rather than `< r**2`.

9.2.2 Question 8

The correct answer is E. There are no errors itself that cause Python to crash; however, note that `t=arange(0.0,0.001,10.0)` creates an linearly spaced array starting at 0.0, ending at 0.001 and with step size 10.: in otherwords, it'll just be `t=[0]`. Thus, the plots will be useless as only a single point is plotted, and no lines will be visible. Thus, answer E is correct.

9.2.3 Question 9

The correct answer is D. There are no actual errors in the program that cause a crash. However, look carefully at what is actually printed: we print `ad`, which is previously set equal to `None`. The fact that another variable `ad` is computed within the function definition is unimportant, as that `ad` is an *inner*-scope variable. Thus, we still have `ad = None`, and thus we do not actually print the coefficients but rather just `None`. Thus, answer D is correct.

9.3 Part III: Complete the program

9.3.1 Question 10

Let's just go through them one by one:

a) Five degree per step, from 5 to 85, just screams for `np.arange`. Indeed, we must write `np.arange(5,86,5)`, such that the 85 is included.
b) The gravity vector is simply 0 in x-direction, and -9.813 in y-direction. In other words, `gvec=np.array([0,-9.813])`.
c) Note that there is a multiplication sign in front of the `np.array`. Now, the velocity has a magnitude of `V0`; its horizontal component will thus be equal to $V_0 \cos\alpha$ whereas its vertical component will be equal to $V_0 \sin\alpha$. Thus, we gotta multiply with `np.array([np.cos(a),np.sin(a)])`.
d) The ball must remain above the ground, i.e. `while h>=0.0`.
e) This one is the hardest one. Note that first of all, the drag force itself should be multiplied with the magnitude `V**2`. However, we must also give it the correct direction, as currently it's just the magnitude. We can use the given hint for that, which says we should multiply with `-vvec/V`, where the minus sign already has been taken into account at the beginning of the line. Thus, we obtain `*V**2*vvec/V`.
f) Very easy, just `xall.append(xvec)`.
g) Also very easy. Note that `xall[:,0]` refers to all x-values; `xall[:,1]` will call all the y-values.

9.4 Part IV: Write program

9.4.1 Question 11

Ayayayay numerical integration of ODEs is beautiful. See the beautifully simple code shown below.

```
import matplotlib.pyplot as plt

x = 0
y = 3
z = -10
tmax = 8
xlist = []
ylist = []
t = 0
dt = 0.001
```

```
    while t < tmax:
        dxdt = 10*(y-x)
        dydt = 28*x-y-x*z
        dzdt = x*y-8/3*z
        x = x + dxdt*dt
        y = y + dydt*dt
        z = z + dzdt*dt
        xlist.append(x)
        ylist.append(y)
        t = t + dt

    print(x, y, z)

    plt.plot(xlist,ylist)
    plt.show()
```

- In lines 3-10 we simply initialise the simulation, and already make lists in which we can store the x and y values.
- In lines 12-21 we compute the numerical solution using forward Euler. Every timestep, we first compute the derivatives. Do not immediately update `x` after computing `dxdt`! This makes your scheme inconsistent (that is, it computes something but it's the correct thing it's computing). Then we update the positions and save the `x` and `y` values. Finally, we print the final values of `x, y, z` and make a nice plot in the xy-plane; it is clear from the plot that $x = 5$ is crossed ten times. The final values of x, y and z are 1.0473, 1.5438 and 14.502, respectively.

9.4.2 *Question 12*

This is a tough one, but not of the level of difficulty that you shouldn't be able to do it. Once again, it is important to distinguish for yourself what the high-level tasks are that should be performed. We can think of the following:

- For a given number, execute the listed procedure.
- During listed procedure, ensure that the last sequence of divisions is taken track off and that the number of divisions by 2 is counted (without counting the one that happen before the final sequence of divisions by 2 is initiated).
- Compare the number of 2 divisions to previously obtained results.

Now, how do we do the first step? That's really easy to do, and is just a while loop (`while number != 1`), including an `if`-statement to determine whether it is odd or even. However, how do we then count the longest sequence of divisions by 2? Well, what you can just do is very simple: everytime you divide by 2, you increment some `counter` by 1. However, if you do the alternative, you simply reset this `counter` to 0 again.

So, that's a relatively straightforward way of counting how many 2 divisions occur at the end. However, how do we store our best-scoring numbers? How do we determine a-priori how long the longest sequence will be? Well, we can solve this as follows: we have a `bestcounter`, which we initially set equal to 0; this will contain the highest `counter` achieved by any number up to that point. Furthermore, we make a list `listofbestnumbers` in which we store our best numbers, that all have a counter equal to `bestcounter`.

Then, in each loop, we check whether the counter of the number we're considering is higher than the current `bestcounter`. In that case, we make a brand-new `listofbestnumbers`, containing only the current number we're considering. Furthermore, `bestcounter` is changed to the `counter` of the number we're considering.

If the current counter is *equal to* the current `bestcounter`, we'll simply add it to the list of the `listofbestnumbers`, as it scores equally well as the other numbers in that list.

All of this leads to the code shown below.

```
    bestcounter = 0
    listofbestnumbers = []
    for originalnumber in range(2,30000):
        number = originalnumber
        counter = 0
```

```
        while number != 1:
            if number%2 == 0:
                number = number/2
                counter = counter + 1
            else:
                number = 3*number + 1
                counter = 0
        if counter > bestcounter:
            listofbestnumbers = [originalnumber]
            bestcounter = counter
        elif counter == bestcounter:
            listofbestnumbers.append(originalnumber)

    print(listofbestnumbers, bestcounter)
```

At the end, we print the numbers 14563, 17529, 19417, 20455, 21845, 25889, 27273 and 29126. Furthermore, the highest counter was 16.

9.4.3 *Question 13*

First: please bear in mind that due to differences between Python3 and Python2, the answer suggested to verify your answer is not correct! Don't bother spending hours trying to find your mistake; your program should output a different value.

Now, onto the code itself. Once again, let's look at a grandscale what we have to do:

1. Generate the list of destinations. This can be done with the code they've given.
2. While the number of people remaining on the list remains larger than 0, the first 9 destinations should be taken from the list.
3. For these 9 destinations, the number of unique destinations needs to be counted, i.e. n_f.
4. For these 9 destinations, the highest destination needs to be searched, i.e. f_{max}.
5. The equation needs to be plugged in, and the total time should be summed up.

Point number 2 is arguable the hardest to visualise how to do it easily, and arguably it's something you need to have seen at once in your life. Let's look at the whole of my code first:

```
from random import randint, seed
import numpy as np

def fillhall(seedno, npeople):
    seed(seedno)
    return [randint(2,13) for i in range(npeople)]

numberofpeople = 100
waiting = fillhall(6,numberofpeople)

numberremaining = numberofpeople
i = 0
T = 0
while numberremaining > 0:
    takenaboard = waiting[i*9:min((i+1)*9,numberofpeople)]
    nf = np.size(np.unique(np.array(takenaboard)))
    fmax = max(takenaboard)
    T = T + 7  +(2.3+2.0)*fmax + 5*nf
    numberremaining = numberremaining - 9
    i = i+1
print(T)
```

- Lines 1-10 are basically just copied from the question, although I import `numpy` for another reason as I'll describe below.
- Line 11-13 initialise the while-loop: the `numberremaining` will be updated with each loop, so that `numberofpeople` may stay constant. `i` is a counter that I'll explain very soon, and `T` will be the time measure.
- Line 14 sets the while-loop and speaks for itself really.

- In line 15, we define a new list `takenaboard` that is very easily set-up: in our first iteration, it'll take entries `waiting[0:9]`; in our second it'll take `waiting[9:18]`; in our third it'll take `waiting[18:27]`. This is very easily done by just incrementing `i` by 1 each iteration, as done in line 20. Note that for the stop-value I write `min((i+1)*9,numberofpeople)`; the reason for this is that if I have 100 people, then for a certain `i`, I will reach 108. This would yield me an error though, as `waiting` only has 100 entries. Thus, if it so happens that I don't end up nicely at the very end, I'll just automatically make it stop at the end of `waiting`, even if this means that I don't take 9 entries into `takenaboard`.
- In line 16, we count the number of unique destinations. We do this with use of several `numpy` functions. First, `np.unique` just returns a numpy array with *only* the unique values in it; however, you need to feed it a numpy array to work in the first place, hence I write `np.unique(np.array(takenaboard))`. The number of unique destinations is then simply the length of the numpy array; for numpy arrays, you need to call `np.size` for that.
- In line 17 I compute the highest destination.
- In line 18 I calculate the increase in time.
- In line 19 I subtract 9 people from the number of remaining people as we have taken 9 people on board.

Finally, we print `T`, which is equal to 1107.5 s.

10 Exam June 2015

10.1 Part I: what will this print?

10.1.1 Question 1

Line 1 will simply create `a=[6,5,...,-2,-3]`. Then, line 3 will transverse through this starting at the last entry, ending at the fourth-to-last entry, and with step size -2. In other words, it'll print `[-3,-1]`.

10.1.2 Question 2

Let's go through the while-loop line by line. First we make `idx` equal to 0. Then we print haiku entry 0, haiku entry -3 and punctuation entry 1 (as count is still 6, so `min(6-2,1)=1`), i.e. `i am integer,`. Then we decrement `count` by 2, so that `count` now equals 4. We then have increment `idx` to 1, and we print haiku entry 1, haiku entry -2 and punctuation entry 1, i.e. `proper numbers in order,`. Then we decrement `count` to 2. We now set `idx` equal to 2, and thus print haiku entry 2, haiku entry -1 and punctuation entry 0, such that we print `am i haiku now?`.

Thus, in total, we print the combination of what I wrote above.

10.1.3 Question 3

Let's go through this for-loop loop-by-loop: we first analyse the digit 0. If it is divisible by 2 (which it is), then we take the negative of `result`, and add 0 to it. This still results in 0, obviously.

Then we analyse the 1. This is not divisible by 2, so we ignore it. Then we get the 2. This is divisible by 2, so we take the negative value of the previous `result` (which was 0 so doesn't really matter) and add 2 to it, such that `result = 2`. The 3 is again ignored, but 4 is divisible by 2. Then we compute the new `result` as `result = -2+4=2`. Then, 5 is skipped but 6 is divisible by 2, so we get `result = -2 + 6 = 4`. 7 is skipped again, and 8 is the final digit divisible by 2, for which we get `result = -4 + 8 = 4`. Thus, the final value that's printed is 4.

10.1.4 Question 4

We first take `i=0` and `j=0`, for which `j*i` is zero obviously. This holds for the other values of `j` as well, so we go on with `i=1`. Then, for `j=0` the product is still 0, but for `j=1` and `j=2`, it equals 1 and 2 respectively. For `i=2`, we again have that for `j=1` and `j=2` it is nonzero and equals 2 and 4; for `i=3` it is 3 and 6, and for `i=4` it is 4 and 8. The sum of these numbers is 30, so we print 30.

10.1.5 Question 5

Let's go through it loop-by-loop. We first take `i=0`. Then `i%2=0`, so the while-statement is false as `j` is not smaller than 0. Thus, we immediately increment `n` by 6. Then we increase `i` to 1; now `i%2=1`, so `j` is smaller than this, so we increment both `n` and `j` by 1. After this, the while-loop is exited, and `n` is incremented by another 6, so that `n=13`.

Then we analyse `i=2`, for which `i%2=0` whereas `j` still equals 1, so the while-loop is ignored and we just increment `n` by 6, so that `n=19`. Finally, we consider `i=3`; now `i%2=1`, however, note that `j` is still 1 from the previous time! So, the while-loop is again discarded, as `j` still equals 1. Thus, we increment `n` one more time by 6, meaning we print 25.

10.1.6 Question 6

Note that `a=[0,2.5,5,7.5,10]` as `np.linspace` includes the endpoint, and that `b=[-1,-0.6,-0.2,0.2,0.6]`. Then, we print the sum of the entries in `b`, corresponding to those indices where the value of `a` is larger than 5.0. This is only the case for the last two entries of `a`, thus, in essence, were are only taking the last two values of `b` too. This means that the sum that's printed equals $0.2 + 0.6 = 0.8$.

10.2 Part II: Debug the program

10.2.1 Question 7

Answer B is correct. Line 2 should be indented.

10.2.2 Question 8

Once more a question that depends on Python 2/3 (at least partly). Please read it as `for j in range(i//2)`, i.e. that integer division occurs, as you can't write `for j in range(3.5)`. Now, note that although in the for-loop we are incrementing `i`, this will not affect the for-loop itself! When you write `for j in range(i//2)` as for-loop, then it'll evaluate `i//2` once at the very beginning, but if you change `i` later, it's not as if the range becomes longer (So you don't have to worry that first `j=0` and it's in `range(3)`, then `j=1` but since `i` is incremented by 2 it's in range `range(4)`, then `j=2` and `range(5)`, etc. It'll remain fixed `range(3)`). This means that it'll always read `for j in range(3)`, so the last value `j` will take is 2. Thus, the program will print 2, and answer C is correct.

10.2.3 Question 9

Let's go through all the statements:

A) False: just not true. You can import both.
B) False: nothing wrong with it.
C) True: since we write `from math import *`, it'll take the `math`-definition of `cos`. However, this definition only works on single values, and not on `numpy` arrays, for which you need `np.cos`. It'll indeed give a runtime error.
D) False: contradiction with what I wrote above.
E) False: again contradiction.
F) False: this is totally fine.
G) True: you also need to add `plt.show()`.

10.2.4 Question 10

Note that in Python2, the statement `x = i/100` would just make `x=0` each and every time. So, your plot would become rather useless and answer C would be correct. In Python3, answer B would be correct, since you should divide by 10 rather than 100 to get a plot from $x = -2$ to $x = 2$.

10.3 Part III: complete

10.3.1 Question 11

Let's just go through them one-by-one:

a) Very simple, just look at the line below: `np.array([0,10])` based on the comment next to it.

b) Probably the hardest line of the question. We must check two things: is the number of bounces 5 or less, and if it is larger than 5, we must stop as soon as the ball is on top. How do we do this? The first part is easy: we just check `while nbounce <= 5`. The second thing is a bit harder to translate to Python code. How do we know that we reach the top? After all, the maximum height it will bounce to reduces over time. For this, you have to think of properties that make the top unique; the most important property is that the velocity will start becoming in negative y-direction there. Thus, before the top `vball[1]` will be positive, and after the top `vball[1]` is negative. Now, we can't check `while nbounce <= 5 and vball[1]>0`: this would be invalidated as soon as the ball reaches its first top; rather, we must use an `or` statement, i.e. `while nbounce <= 5 or vball[1]>0`.
c) It may look a bit weird as to what exactly write here: if we multiply `fd` with the velocity vector, what else do we need to do? Well, we need to normalise the velocity vector, i.e. we need to divide by `np.linalg.norm(vball)`.
d) Simply `+vball*dt`.
e) We simply plot `xtab[:,0],xtab[:,1]`.

10.4 Part IV: write

10.4.1 Question 12

Very easy. Just look at the code below:

```
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(-7,7,0.001)
f = np.sin(x)*(0.003*x**4-0.1*x**3+x**2+4*x+3)
g = -10*np.arctan(x)

plt.plot(x,f,x,g)
plt.show()
```

It's super basic. We just plot them and then zoom in on the intersections. You'll find $x = -4.384$, $x = 0$, $x = 3.653$ and $x = 5.966$.

10.4.2 Question 13

Very similar to one of the first exams included in this solution manuals. We just get the code shown below.

```
import math as m

sum = 0
x = 0.4

for n in range(1,5001):
    sum = sum + 1/n**2*m.cos(n*m.pi*x)

print(sum)
```

We simply loop over all possible `n`; note that we start at `n=1` and note that we need to range to 5001 as 5000 itself is not included. The sum is straightforwardly computed, and we end up 0.065797.

10.4.3 Question 14

Already a bit more difficult.

We must loop through all numbers, and then check four things:

- Is it divisible by 7?
- Is it divisible by 11?

- Does it contain a 7?
- Does it contain two sequential 1s?

The basic premise of such a program is pretty simple: for each number, you define a variable `isjuf` that you initially set to `False` and then you check each of the four things; if one of them is satisfied you set `isjuf` to `True`. If `isjuf` stays `False` for a certain number, you save it in a list, so that at the end you can sum all the values in the list. Now, the first two are very easy to check, but how do we check the third, let alone the fourth? Well, once again you'll have to convert the number to a string, and then loop through the string to see if the digit is 1; to check whether it's an 11, we'll also take the next digit and see if both are 1. How does this translate to code? Look at the code below.

```
lst = []

for n in range(1001):
    isjuf = False
    if n%7==0:
        isjuf = True
    if n%11==0:
        isjuf = True
    string = str(n)
    for i in range(len(string)):
        if string[i] == '7':
            isjuf = True
    for i in range(len(string)-1):
        if string[i] == '1' and string[i+1] == '1':
            isjuf = True

    if isjuf == False:
        lst.append(n)

print(sum(lst))
```

Lines 1 - 8 speak for themselves really, note that `n in range(1001)` as we need to include 1000 itself. After that, we convert the number to a string. First, we loop through each index in the range of the length of the string, and then check whether the corresponding digit is equal to 7. Afterwards, we loop through each index in the range of the length of the string minus one: we then check whether the corresponding entry and the corresponding next entry is equal to 1. This is why the range is now minus 1: when we check the penultimate digit, we're already also calling the final digit in the string. Thus, we don't need to call this routine for the final digit itself, as it is already checked for the combination penultimate + final digit.

The correct answer is then 263791.

10.4.4 *Question 15*

The logic of the program is pretty simple: every time we update the x and y-values, then check if we run out of the box, or if not we'll check at least if we bounce against the mirrors, by simply checking whether the x or y values exceed the boundaries. The only real problem is how to reflect if we actually exceed the boundaries? Well, let's consider them just one-by-one:

- If we exceed the right boundary, i.e. if $x_{i+1} > 1.8$, then the new x-value should actually be $x = 1.8-(x_{i+1})$. With regards to ϕ: if we approach at angle ϕ, then we'll leave at an angle $\phi_{i+1} = \pi - \phi_i$ (just make a sketch yourself, with ϕ being the angle measured counterclockwise from the positive x-axis, to the direction the ball is currently pointing in).
- Similarly, if we exceed the left boundary, i.e. if $x_{i+1} < 0$, then we get $x = 0 - (x - 0)$, and again we get $\phi = \pi - \phi_i$ (again, draw your own sketch if you don't believe me).
- For the top boundary, i.e. if $y_{i+1} > 2$, we get in a very similar fashion to before that $y_{i+1} = 2 - (y - 2)$ (I hope you see the pattern: we compute the difference with respect to the boundary $(y - 2)$), and then simply subtract this from the boundary $y = 2$). However, ϕ is slightly different. Once again, if you make a sketch, you'll quickly realise that $\phi_{i+1} = 2\pi - \phi_i$ in this case.
- Finally, for the bottom boundary, i.e. if $y_{i+1} < -1$, we get $y_{i+1} = -1 - (y_{i+1} - -1)$.

Everytime we hit a barrier, we'll increase the counter `n` by 1. All of this leads to the code shown below:

```
lst = []
```

```
    for n in range(1001):
        isjuf = False
        if n%7==0:
            isjuf = True
        if n%11==0:
            isjuf = True
        string = str(n)
        for i in range(len(string)):
            if string[i] == '7':
                isjuf = True
        for i in range(len(string)-1):
            if string[i] == '1' and string[i+1] == '1':
                isjuf = True

        if isjuf == False:
            lst.append(n)

    print(sum(lst))
```

Lines 1-8 are very simple and just initialise everything. In line 10, we define a boolean that will contain whether we left the boundary or not, and we'll use this boolean to run the while-loop. There, we first update x and y as shown on the exam itself; we then check whether it's already outside gate. If it is, then the boolean is set to false, such that at the next loop we stop. If it's not outside, it first checks whether we're to the right of the right boundary; if it is, the position is corrected and we increment our counter `n` by 1. We do the same for the left, lower and upper boundary. Finally, for each loop, we append `xlist` and `ylist` with the newest x and y-values.

At the end, we print `n` and we plot the x- and y-coordinates, so that we get a nice graph of the path the thing travelled. It turns out that the ball is bounced 41 times from the walls.