Python solution manual: 2017-2018 edition

Sam van Elsloo

June 2018
Version 1.0

Contents

1 Preface

As long as this message appears on top of the preface, this solution manual is still being updated. I'll still add numerous old exams on the dropboxfolder and add their solutions to this solution manual[1]. But I figured maybe you'd already want to see how the old exams look like so that's why I'm publishing this preliminary version of the summary.

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[2]. 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[3], 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 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
```

[1]I still have three full exams plus a few exams of which approximately half of the exam is still representative, so I'll upload them asap.

[2]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.

[3]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.

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.

1.2 *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[4]. 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[5], but rather the fastest, safest way of doing it on the exam.

[4]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.

[5]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

1.3 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[6]. 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.

  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[7]. 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

capable with Python but writing the most efficient programs isn't a requirement for passing the exam.

[6]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.

[7]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.

genuinely quite sad how no one shows up to the lectures: half the people leave after the first half of the first lecture[8]. 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[9]. 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 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.

[8]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.

[9]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.

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:

```python
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:

```python
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?

```python
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 $\left[10^{-1} \quad 10^{-0.5} \quad 10^{0} \quad 10^{0.5} \quad 10^{1}\right]$ 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

```python
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:

```python
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:

```
import random as rnd
rnd.seed(0)

row = rnd.randint(0,2)
col = rnd.randint(0,2)
print(row, col)

row = rnd.randint(0,2)
col = rnd.randint(0,2)
print(row, col)

row = rnd.randint(0,2)
col = rnd.randint(0,2)
print(row, col)

row = rnd.randint(0,2)
col = rnd.randint(0,2)
print(row, col)

row = rnd.randint(0,2)
col = rnd.randint(0,2)
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.