Elementary Algorithmic Programming in Python

Antonio Carzaniga

10 February 2023

Programming means telling a computer how to do something. It is not fundamentally different from telling *a person* how to do the same thing. In essence, you write step-by-step instructions. Think about the way you would write a recipe or the directions to a meeting place. Those are effectively "programs". With a person, you might write those programs in English (or whatever natural language that person understands). Here we deal with computer programs, so the language is going to be a bit different. Still, the general idea is the same.

For our purposes, here we use a language called Python. Our goal, however, is not to learn Python in great detail. Instead, we want to learn just enough of Python to be able to write relatively short programs that solve algorithmic or combinatorial problems. In fact, not only the programs will be short—never more than 50 lines, and most often less than 20 lines of code—but we will also intentionally limit ourselves to some very basic features of the Python language.

This is a *read-do* document. You must read the text slowly, a paragraph or even a sentence at a time. Don't rush through it or just skim it. Sometimes we go into technical details or even just random digressions that you should most likely skip. But those are relegated to footnotes. So, sure, skip the footnotes. But don't skip the main text.

As soon as you have at least a basic idea of the concepts described in the text, you should immediately put them into practice with the many examples and exercises listed in the text. You should also practice with other examples that you create and experiment with out of your own curiosity. This is what I mean when I say this is a read-do document. This is the best way to learn programming anyway—or anything else, really.

Preliminaries

I assume you have Python installed on your computer. Specifically, we will use Python-3. I also assume that you have some basic experience using a command shell and a text editor. You may also choose to use an integrated development environment (with an integrated editor) but that is not at all necessary.

In essence, you should be able to run Python as an interactive shell, edit a Python program (as a text file), and run a Python program as a standalone program or within an interactive Python shell. Let's review these preliminary skills one by one.

Running Python as an Interactive Command Shell

Run the `python3` command from a command shell, which you typically get by opening a *terminal* application. So, open a terminal application, and at the prompt type `python3`, and then enter that command with the return key (). You should see something like this:

```
$ python3
Python 3.10.6 (main, Nov 14 2022, 16:10:14) [GCC 11.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

You might get something slightly different. But if you get something completely different, like an error message `python3: command not found`, then you probably don't have Python-3 installed on your computer, or in any case your shell can't find it. If that is the case then you should ask your friendly teacher (yours truly) or a fellow student to help you out. Do it! Seriously, don't put this off. Continuing on, I'm assuming you managed to run `python3`.

Notice that there are *two* shells here. The first one is what I call a *command* shell, which is sometimes also called terminal shell because it's what you typically run when you open a terminal application, as I suggested here. So, really, there are three programs we need to distinguish here: the *terminal,* the *command shell,* and the *Python shell.*

All of this can be confusing. The terms are confusing, and it's also difficult to make sense of not one but three programs running within what looks like the same user interface in front of you. Let's try to do just that, starting with some basic and hopefully intuitive definitions. A *terminal* is a user-interface (UI) program: a program that you use to input data into other programs and to view their output data. A terminal typically provides a *text* UI, meaning that it takes input from what you type into the keyboard, and displays output in the form of text on the screen or window. Simple![1]

A *shell* is a program that interactively reads and interprets your input as commands and then executes those commands. So, in this sense, all shells are *command* shells. However, different types of shells differ in the way they interpret and execute commands. What we call *command shell* here uses a special language that we do not care to discuss here.[2]

In the most basic setup, we start with a terminal and command shell. The role of the terminal is to read input from the keyboard and pass it to the shell, and at the same time output onto the screen whatever the shell outputs. This first shell is the one taking commands like `ls`, `mkdir`, and `cd`, and also applications like `python3`, which is what we run here. The prompt of this command shell is a single `$` character (dollar sign). You might see a different prompt depending on your configuration and preferences. However, in these examples, I will use the single `$` character to identify the prompt of the command shell.

Notice that when the shell runs a command, that command takes over the input and output from and to the terminal application. So, if you type something into the terminal, that input doesn't go to the shell and instead goes to the command that the shell started. When that command terminates, the input and output are reconnected to the shell. It is possible for a command that you run through the shell to act itself as a shell, and therefore to read and run its own commands, which will temporarily

[1] Even in this most basic setup, things aren't as simple as it might seem. In fact, some time the output you see on the terminal does not come from the shell but instead comes from the terminal itself. In some cases, the terminal does not pass every keystroke to the shell, and instead allows you to input a whole line before shipping that whole line to the shell. In this case, the terminal itself displays or "echoes" the letters you type and even performs some basic line editing, for example deleting the last character when you press the backspace key, etc.

In other cases, the terminal is configured to ship every keystroke to the shell, which then takes care of implementing its command editing features. In fact, this is what happens when, with most modern shells, you can bring up previous commands and perform advanced editing, such as deleting words, swapping words, and much more. Notice that, when operating in this mode, the output sent to the terminal is not simply copied into the screen. Instead, the output to the terminal might contain sequences of bytes that are interpreted as special commands to the terminal.

So, well, it ain't so simple after all. Aren't you glad you took the time to read this whole footnote!?

[2] As a good hacker, you might be curious about this language, too. Feel free to feed your curiosity. Personally, I think it's worth your while. The more you learn about it, the more you'll realize that what looks like a primitive, awkward little program to some, is in fact an extremely powerful, sometimes even elegant, and arguably unsurpassed tool. As you become more and more proficient in the use of a command shell, you will enjoy the wonderful feeling of controlling your computer the same way a violin virtuoso plays a Stradivari.

take over the input and output of the terminal. Effectively, at any given time, you can have a stack of commands executing at the same time within the same terminal application.[3]

Here the command that acts as a second shell is the *Python* shell. Thus when we run `python3` from the command shell, we obtain the prompt >>> of the *Python* shell (three > characters one after the other). At this point, you can type *Python* commands, and the Python shell will execute them. For example:

```
>>> print('Ciao!')
Ciao!
>>>
```

Running a Python Program

First of all, you should create a directory within which to experiment with the following examples. I suggest you do that using a command shell, since the next steps will require you to run commands from a command shell anyway:

```
$ mkdir programming
$ cd programming
$
```

Now you are in your new `programming` directory, so go ahead and create a file called `ciao.py` containing the following text:

```
print('Ciao!')
print('This␣is␣your␣first␣Python␣program.')
```

Seriously, do it! Don't just read through this document. This is a *read-do* document. You read an example, and then you try it yourself right away.

So again: create and edit a file called `ciao.py`. To do that, you should use a good text editor. My favorite one is Emacs, but you should use whatever works well for you. Be warned though, that some editors will mess things up, because they don't write files in plain text format. So, make sure that the file you just edited and saved contains exactly those two lines of plain text. You can do that from the terminal shell with the `cat` command:

```
$ ls ciao.py
ciao.py
$ cat ciao.py
print('Ciao!')
print('This is your first Python program.')
```

If this is what you see, then you are now ready to *run* your Python program. From the same command shell:

[3] Needless to say, it's more complex than that. Many command shells allow you to run one or more commands "in the background". This means that the commands run but the control and input go right back to the shell. Notice that the output isn't a big problem, since the terminal (with the help of the operating system) can and does read and then display the output produced by multiple commands. That might create a mesh-up of multiple output streams that might well make no sense to the user. But that would not pose much of a technical problem for the terminal or the shell that invoked all those commands.

The input is a different matter. Somebody—presumably you!—has to decide whether what you type into the terminal should go to the shell or one of its running commands, and which one of them. In fact, when you invoke a command in the background, that command will be suspended as soon as it tries to read from the terminal input. And of course the shell allows you to selectively bring back to the foreground that or any other command running in the background thereby reconnecting them to the terminal input.

```
$ python3 ciao.py
Ciao!
This is your first Python program.
```

Reading a Python Program and Then Using Python Interactively

In many examples below, you will want to write a program, or more specifically an algorithm in the form of a Python *function*, and then run that function interactively, perhaps to test it with different arguments. Let's see how you can do that.

In a source file called `greetings.py` write a function `ciao(name)` as follows.

```
def ciao(name):
   print('Ciao', name)
   print('Have␣fun␣with␣Emacs!')
```

We're getting ahead of ourselves defining "functions" here, but that's okay. Just make sure the spacing is correct though. The spaces before the two `print` commands are important, as we will see in detail later. For now just make sure those two lines have the same number of initial spaces (indentation).

Now, we could run this `greetings.py` program the same way we did with `ciao.py` in the example above. However, that is really not what we want. In fact, if we run it as a program, we don't see any output. Try it yourself:

```
$ python3 greetings.py
$
```

And this is correct. The `greetings.py` program defines a function `ciao(name)` but does not use it. Instead, what we want is to use that function with some arguments. For example, we might want to call the `ciao` function as follows, from a Python shell:

```
>>> ciao("Antonio")
Ciao Antonio
Have fun with Emacs!
>>>
```

In other words, we want to run a Python shell that reads the definition of the `ciao` function from `greetings.py`, and then processes the Python command `ciao("Antonio")`. The way to do that is to run Python with the `-i` option, which makes it run in "interactive" mode:

```
$ python3 -i greetings.py
>>> ciao("antonio")
Ciao antonio
Have fun with Emacs!
>>> ciao("mamma")
Ciao mamma
Have fun with Emacs!
```

Notice that we invoke Python from the command shell, which starts the Python shell from which we can invoke the function defined in `greetings.py`. To exit the Python shell and therefore return to the command shell, you can invoke the `exit()` function.

```
>>> ciao("mamma")
Ciao mamma
Have fun with Emacs!
>>> exit()
$
```

Basic Arithmetic Expressions

Python supports numbers and basic arithmetic. Try it yourself using the interactive Python shell:

```
>>> 1 + 1
2
>>> 34 - 11
23
>>> 3 * 7
21
```

Easy. Makes sense. But did you try it yourself? Seriously, do it! As I said before, this is a *read-do* document. You should try every single example yourself. In fact, once you get the gist of it, you should invent and try more and more examples.

So, now you know that Python can read numbers, and can add, subtract, and multiply numbers. What's next? Division. There are in fact three fundamental division operations we need to know about:

```
>>> 5 / 2
2.5
>>> 6 / 3
2.0
```

The division operator `/` divides its two operands, as you would imagine, and always returns a floating point number. However, sometimes, especially in programming algorithms that work on *discrete* objects and structures, we want an integer division:

```
>>> 13 // 5
2
>>> 13 % 5
3
```

The integer-division operator `//` does exactly that: `a // b` gives you the *quotient* of a over b, meaning the number of times that b fits entirely in a. Correspondingly, `a % b` is the *remainder* of the integer division of a over b. So, both the quotient and the remainder are whole numbers, also known as *integers*.

We won't need any more numeric operations in the Algorithms course, really. Although of course you can combine those operators in long expressions, possibly grouping sub-expressions using parentheses. For example:

```
>>> 10 - (3 + 1)
6
>>> 27 % (2*(7 - 2))
7
```

Variables

Python can memorize values for you, so that you can use them again later. For example, you can memorize the result of an arithmetic expression:

```
>>> x = 10 + 2*5
>>>
```

The command `x = 10 + 2*5` is an *assignment*. It looks like a mathematical equation, since it contains an equal sign (=), but that's not what it is. What the equal sign really says is: first, compute the value of the expression on the right-hand side of the =, and then store that value in a memory cell called x. So, the command `x = 10 + 2*5` does not itself produce an output or a result, but it does change the memory of the computer. In fact, now you can read the value of x:

```
>>> x = 10 + 2*5
>>> x
20
```

Notice that x is now a valid command that simply returns (reads) the value 20 stored in the memory cell named x. The Python shell reads and prints that value.

To reemphasize the idea of an assignment, as opposed to a mathematical equation, try now the following command

```
>>> x = x + 1
>>>
```

Once again, the command `x = x + 1` is not an equation—for which there would be no solution anyway—but rather an assignment, which says: compute `x + 1`, and then store the resulting value into a memory location called x. Now, `x + 1` is an expression that reads a value from x, and the assignment then stores the value of the whole expression into x. You might think that that is weird or that it would cause some kind of circular dependency. But that's perfectly okay. The value initially stored in x is 20 (from the examples above), so `x + 1` is 21, which is what is stored in x after the assignment command. In fact, try these commands now:

```
>>> x
21
>>> x + 5
26
>>> x
21
>>> x = x + 5
>>> x
26
```

You can of course have many variables, and you can use them however you like in expressions. Now you have x whose current value is 26. What happens if you then run the following command?

```
>>> y = x - 21
```

Simple, you now have two variables—two memory cells—one named x that contains value 26, and another one named y that contains value 5. Now you can write something like this:

```
>>> y = x + y - 1
```

What is the effect of that command? We know that it's an assignment that first computes the value of `x + y - 1` and then stores that value into y. To compute `x + y - 1`, Python reads the value of the x memory cell, then reads the value of the y memory cell, adds the two values, and then subtracts 1. The result of that expression is 30, which Python then stores in the y memory cell.

If an expression refers to a variable name that was never assigned, Python prints an error message, as in this example:

```
>>> y = z + x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'z' is not defined
```

Exercise

What is the content of Python's memory after the following commands?

```
>>> x = 2
>>> y = 1
>>> y = x
>>> x = y
```

Exercise

What is the content of Python's memory after the following commands?

```
>>> x = 1
>>> y = 1
>>> y = x - y
>>> x = y - x
```

Exercise

Write commands to swap the values of x and y, for example after the example above.

Exercise

What is the output of the following Python program?

```
x = 1
y = 2
x = y - x
y = x - y
print (x, y)
```

is the output different in the following program?

```
x = 1
y = 2
y = x - y
x = y - x
print (x, y)
```

Conditional Instructions

So far we have seen individual commands and sequences of commands in Python. Python executes a *sequence* of commands by executing each command in the given order, one after the other.

If this is all we had, we wouldn't be able to do much "programming", really. A program is most often supposed to behave differently in different situations. For example, say we have an accounting program where a positive balance must be printed as a "gain" while a negative number must be printed as a "loss", and a zero balance must be printed as "zero". In Python we would write a program like this:

```
def print_balance(b):
   if b > 0:
      print('gain:', b)
   elif b < 0:
      print('loss:', -b)
   else:
      print('zero')
```

Write this program in a source file called `print_balance.py` and try it with the interactive shell by running `python3 -i print_balance.py`. Once again, pay attention to the spacing. Make sure the indentation is consistent as in the code above. You can now try it with various arguments:

```
>>> print_balance(200)
gain: 200
>>> print_balance(-75)
loss: 75
>>> print_balance(10.5)
gain: 10.5
>>> print_balance(0.0)
zero
>>>
```

In Python you can write the conditions of the `if` command (and `elif`) in the intuitive way, meaning very much like you'd write conditions in English. For example, in English you might say, if it rains or it's cloudy, then we will play chess. And in Python, you'd say pretty much the same:

```
if weather == 'rain' or weather == 'cloud':
   play_chess()
```

Or you might say, if the temperature is greater or equal to 5 and less than or equal to 30, and if it's sunny, we'll play basketball. In Python:

```
if temperature >= 5 and temperature <= 30 and weather == 'sun':
   play_basketball()
```

Or you might say, if the temperature is less than 5 and it's not snowing, we'll go skiing:

```
if temperature < 5 and weather != 'snow':
   go_skiing()
```

Thus == means *equal to*, != means *not equal to*.

Code Blocks

By now you must have figured out that Python groups commands together by level of indentation. Incidentally, we used the term "command" here because we have worked primarily within an interactive Python shell, where those commands are executed immediately by the shell. However, those can just as well be part of a program. In that case we also call them "instructions" or "statements". Anyway, back to indentation and code blocks.

In programming, it is essential to group instructions together. A *program* is in fact a group of instructions. More specifically, a *function* is a group of instructions that has a name and some parameters. In our very first example, we define a function called `ciao` that takes an argument that the code of the function can refer to as `name`.

```
def ciao(name):
   print('Ciao', name)
   print('Have␣fun␣with␣Emacs!')
```

In other words, the instructions `print('Ciao', name)` and `print('Have fun with Emacs!')` are grouped together in sequence, and we give that sequence the name `ciao` and a parameter named `name`, so that later we can *invoke* that sequence of instructions with something like `ciao('Antonio')`.

That sequence of instructions is introduced by a semicolon (`:`) at the end of the `def` command, and consists of all the instructions indented by four spaces with respect to the `def` command.

Similarly, we have seen that an `if` statement introduces a code block, and so does a corresponding `else` statement, also with a semicolon at the end of the `if` and `else` commands. For example, you can write:

```
if n % 2 == 0:
   print(n, 'is␣even')
   print('so,␣I␣can␣split␣it␣in␣half.')
   print('half␣of',n,'is',n//2)
else:
   print(n, 'is␣odd')
   print('so,␣I␣can␣split␣in␣half',n,'+␣1')
   print('half␣of',n,'+␣1␣is',(n+1)//2)
```

At a high-level, this code says: if n is an even number—we check that by checking that the remainder of the integer division by 2 is zero—then do something, otherwise do something else. The *something* part is defined by the code block after the `if` statement. The *something else* part is the code block after the `else` statement.

In these examples, we use four spaces for the indentation of a block of instructions. In fact, you can use three or five or any other number of spaces. It is essential, however, that all the instructions in a block be indented by the same amount of spaces.

Exercise

Write a function `rectangle_area(x1,y1,x2,y2)` that, given the coordinates (x_1, y_1) and (x_2, y_2) of two opposite vertices of an axis-aligned rectangle, prints the area of the rectangle. An axis-aligned rectangle is such that its sides are parallel to the X or Y axis. You must not use any function or language features other than the ones we have seen so far.

Loops

We have seen sequences of instructions and conditional instructions. This is still not enough. We need to somehow instruct the computer to do something repeatedly. For example, say you want a program to print the word "ciao" a number of times. If you want to print "ciao" three times, you could write a sequence of instructions:

```
print('ciao')
print('ciao')
print('ciao')
```

But what if you want to print `'ciao'` *twenty* times? You wouldn't want to write the same instruction twenty times in your program, would you? Of course not. And in some cases you don't even know ahead of time how many times you need to repeat the print command. So, instead, you'd want to give the computer a single print instruction and then tell it to *repeat* that instruction twenty times, or any given number of times. We call these repetitions *loops*, because we think of the execution as a path through the program instructions, and in this case the execution path loops back to the same instructions a number of times. A simple way to write one such loop in Python is this:

```
for i in range(20):
   print('ciao')
```

This tells the computer to repeat 20 times the block of instructions that follows the `for` instruction (indented). In that block of instructions, which we also call the *body* of the loop, you also have the variable *i*. In fact, you might want to try and see what happens if you also print the value of that variable:

```
for i in range(20):
   print('ciao', i)
```

Anyway, this `for` instruction works if you know exactly how many times you need to print. In fact, it is even more generic than that. It works with any sequence that you can iterate on. So, in general you write `for x in` *sequence* `:`, followed by the body of the loop, and that repeats the body of the loop for every value x in the given sequence. For example:

```
for i in ['vanilla', 'chocolate', 'strawberry']:
   print('I␣love', i, 'ice␣cream!')
```

So, we now know how to loop over the elements of a given sequence. Very good. But this isn't enough either! What if you don't know ahead of time how many times you need to repeat some instructions? What if you need to run the instructions in the loop a number of times before you can decide to stop the loop? For example, you might want to repeat some action until you find something, or until you get to a certain target after some calculations. You can do this by writing a *while*-loop:

```
i = 0
while i < 20:
   print('ciao', i)
   i = i + 1
```

Well, you might say that this while-loop is really nothing new, since it's exactly what we were doing before with `for i in range(20)`. And you'd be right, it is exactly the same thing. In fact, the purpose of this first example was to show how while-loops work. It goes like this: you test the condition of the `while` instruction, and if it's true then you execute the body of the loop, that is, the block of instructions that follow the `while` instruction. At the end of that block of instructions, you loop back to the `while` instruction, again test its condition, and if it's true again, you again execute the body of the loop, and so on. You stop when you loop back to the `while` instruction and the condition turns out to be false, at which point the execution skips the body of the loops and continues with whatever instructions (if any) follow the body.

The above example is intentionally simple. In fact, we wrote it to mimic the previous for-loop. However, while-loops are a very general programming mechanism, more general than for-loops, or at least the type of for-loops that we see in this course.

Would you like to see a more interesting example? Here, run this program:

```
n = int(input('choose␣an␣integer␣n␣>␣0:␣'))
while n != 1:
   print(n)
   if n % 2 == 0:
      n = n // 2
   else:
      n = 3*n + 1
```

Did you run it? Seriously. I told you, this is a read-do document. Plus this example is a really good one. Trust me! In fact, try 27. Go ahead. I'll wait for you. (...) Pretty weird eh? Can you input a number (integer, greater than 0) that makes the code loop forever? I bet you can't. But if you find one of those numbers—or if you manage to prove that none exist—please do let me know![4]

Two more points about loops. What if we need to break out of a loop? This is useful in for-loops, which would otherwise keep repeating the body for all the elements in the `for` iteration. While-loops are more amenable to early termination, since you could write the termination condition in the `while` instruction. However, that condition is evaluated only when the execution completes the body and goes back to the `while` instruction. But sometimes it's just more convenient to check some condition and break from inside the loop. In Python you can do that with the `break` instruction, which works for both *while* and *for*-loops. For example:

```
n = int(input('choose␣an␣integer␣n␣>␣1:␣'))
```

[4] How do you even show that a program loops *forever?!* You could look for numbers that are part of a circular "orbit", starting from n and somehow returning back to n. That is at least something you can test with a program. But there could also be numbers whose orbit goes on to infinity. How do you test *that?* Anyway, just so you know, nobody ever found any number that goes into a circular orbit or that shoots up to infinity. And nobody could prove that such numbers do or do not exist. Isn't it surprising that we don't undestand the behavior of such a simple program?

```
for i in range(2,n):
   if n % i == 0:
      print('composite')
      break
```

Similarly, sometimes it's useful to short-cut the body of a loop and go back immediately for the next iteration. For example:

```
n = int(input('choose␣an␣integer␣n␣>␣2:␣'))
C = [False]*n
for i in range(2,n):
   if C[i]:
      continue
   for j in range(i*i,n,i):
      C[j] = True
   print(i)
```

Did I not say this is a read-do document? Do it. Try this code yourself! And of course, you shouldn't just run the code. You should also try to understand it. So, do you have it? In particular, did you notice that weird expression `C = [False]*n`? As you can imagine, that creates an array containing n `False` values. We will discuss this expression a bit more in detail at the end of the next section.

Arrays

Computers are useful to process many values, not just one or two. So, how do you memorize many values? If you know you have three values, you can put the first one in a variable called x, the second one in another variable y, and the third one in z. But you can immediately see that that doesn't work for *many* values. In fact, it doesn't even work for a few values if you don't know how many you have.

What you want is what you often see in mathematical expressions where you have a collection of elements $a_1, a_2, \ldots, a_n$ and you refer to each element a_i by its numeric index i. In Python, you can do the same with an *array.* In fact, we have already seen an example of an array before, remember? You don't? Okay, no problem. It's a good exercise: go back through the text and see if you spot the array. *Do it!* Anyway, here's another one (in a Python shell):

```
>>> A = [ 2, 3, 5, 7, 11, 13, 17 ]
```

A is an array. An array represents a *sequence.* I should say, those are actually called "lists" in Python jargon. But they are really arrays—trust me, they are arrays—so that's what I'll call them. The nice thing about arrays is that you can access every element of the sequence by its position, as you do in those nice mathematical expressions. Except that for mathematicians the first element is in position 1, whereas in Python the first position is 0. So, this is how you access the first element:

```
>>> A = [ 2, 3, 5, 7, 11, 13, 17 ]
>>> A[0]
2
```

Notice that `A[0]` is really like the name of a variable (a memory cell) that stores the value in position 0 (the first one). And you can of course use another variable, say i, as the position in A. For example:

```
>>> i = 2
>>> A[i]
5
```

In fact, this is how you could print all the elements in A:

```
i = 0
while i < 7:
   print(A[i])
   i = i + 1
```

Here we loop from position 0 to position 6 (not 7) because we know that we have seven elements. But what if we don't know the length of an array ahead of time? In Python, we can use the predefined function `len(A)`. Try it yourself:

```
>>> A = ['chocolate', 'vanilla', 'strawberry']
>>> len(A)
3
>>> A = []
>>> len(A)
0
```

In fact, this is a more generic way to print the elements of an array `A`:

```
for i in range(len(A)):
   print(A[i])
```

or better yet, remember that a for-loop iterates through the elements of a *sequence*. And an array is definitely a sequence, so my preferred way to print the values in an array A is this:

```
for a in A:
   print(a)
```

Adding or Removing an Element at the End of an Array

For our purposes, there are only two ways to modify the length of an array. We can either add an element at the end, or we can remove the last element. For example, this is how we can read a sequence of numbers from the input:

```
A = []
while True:
   line = input()
   if line == '':
      break
   for x in line.split():
      A.append(int(x))
```

Here `A.append()` adds an element at the end of A, thereby increasing A's length by one. If instead we want to delete the element at the end of A, we write `del A[-1]` or `A.pop()`. `A.pop()` also returns the last element that is deleted.

Other Expressions Involving Arrays (Verboten!)

The Python language provides many other ways to cut, modify, or otherwise use arrays (and other sequence objects). However, we intentionally avoid all those operations because they most often incur hidden complexities and in some cases they are also a bit tricky. I even hesitate to mention some of those here because, you know, I don't want to be accused of corrupting young minds or anything. But it's a weird and complex world out there, so hold your nose, wear your helmet, and let's go into this dungeon just for once.

Python allows you to delete any element from an array, or even any subsequence of elements. For example, `del A[0]` removes the first element from array A, and `del A[1:4]` removes the second, third, and fourth. Now, think of how you would remove some elements from an array and still end up with a nicely packed array of the remaining elements. That's right, you can't do that in constant time. In fact, these expressions hide a linear complexity. So don't use them! Similar sub-sequence expressions can be used to copy or replace parts of a sequence. For example, `B = A[1:3]` creates an array that contains the second and third element of A, and `A[:3] = [-1,-2]` replaces the first three elements of A with the subsequence $-1, -2$. Again, not to be used.

Python also provides a *search* operator: `x in A` and `x not in A` are Boolean expressions that are *true* if x appears or does not appear anywhere in A, respectively. Admittedly, these are nice and convenient operations, but again, they hide a linear complexity.[5]

If A and B are two arrays, the expression `A+B` creates a third array containing all the elements of A followed by all the elements of B. Again, not be used. Of course, this little expression hides a linear complexity. And it's also tricky. Like other operations that *copy* arrays or parts of them, this one performs a "shallow" copy, so if some of the elements happen to be mutable (e.g., other arrays) then you might get in trouble when you modify one of those elements.

If A is an array and n is an integer, then the expression `A*n` returns a new sequence obtained by concatenating n copies of A. For example, `[0]*3` is equivalent to `[0,0,0]`. And, you guessed it, we have the same deal here regarding complexity and the pitfalls of shallow-copy operations. This one is in fact particularly tricky. Say you want to create an $n \times m$ matrix initially filled with zeroes. You write `M=[[0]*m]*n`, and you get an array of arrays, which is what you wanted. However, you now want to set some element, say $M_{1,2}$ to some value, say 7, so you write `M[1][2] = 7`. But now—horror!—the third element of *every* row is set to 7. You created a monster! So, you should avoid this kind of operation. But you should also try to figure out what's happening. Here's a hint: You want an array of n distinct lines, each consisting of an array of m elements. However, what you actually get is an array of n *references to the same line* of m elements. We will explain this kind of side-effects in the next section when we talk about the way arrays (or other objects) are passed as arguments to functions. Notice that we used this kind of sequence multiplication expression in one of our earlier examples. So, yes, sometime we break our own rules.

Exercise

Write a Python function called `print_reverse(A)` that prints a given sequence A in reverse order. For example

```
>>> A = [ 2, 3, 5, 7, 11, 13, 17 ]
>>> print_reverse(A)
17
13
11
```

[5] The complexity of `x in A` is linear when A is a *sequence*. That includes arrays and strings. However, the same expression works with other data structures, such as *sets*, in which case the complexity is logarithmic or constant.

```
7
5
3
2
```

Exercise

Write a Python function called `print_maximum(A)` that prints the maximal value in the given sequence A. For example

```
>>> A = [ 7, 21, 15, 1, 18, 31, 17 ]
>>> print_maximum(A)
31
```

What would you do if you are given an empty sequence?

Exercise

Write a Python function called `print_pairs(A)` that considers the given array A as a sequence of *pairs* of adjacent numbers, and then prints the sum of the numbers in each pair. If the length of A is odd, meaning that the last element has no companion number, then the functions should print that last value alone. For example:

```
>>> print_pairs([ 7, 21, 15, 1, 18, 31, 17 ])
28
16
49
17
```

Function Definitions

Once you have a program, you'll most likely want to run that program multiple times, possibly with many input values. You'll want to give that program a name and some arguments, to then invoke that program by name. In Python, you can do that by defining a *function.* In fact, we have done that already, for example with the `print_balance(b)` function:

```
def print_balance(b):
   if b > 0:
      print('gain:', b)
   elif b < 0:
      print('loss:', -b)
   else:
      print('zero')
```

This notion of a "function" is rather different from the notion of a function in mathematics. One notable difference in the example above is that the `print_balance(b)` function takes an argument (b) but, unlike a mathematical function, does not have a value.

For our purposes, a function is a *program*—or call it a method, a procedure, a subroutine, or simply an algorithm—that you can invoke by name. Such a function (program) may or may not take input arguments, and may or may not return a value. For example, the following function takes no arguments and returns a value.

```
def the_answer():
   x = 40
   y = 2
   return x + y
```

The value "returned" by the function is determined by the execution of a `return` instruction. The `return` instruction also terminates the execution of the function and "returns" the execution to the code that invoked the function. As an example, try the following code:

```
def is_prime(n):
   for i in range(2,n):
      if n % i == 0:
         return False
   return True

for i in range(2,100):
   if (is_prime(i)):
      print(i,'is␣prime')
   else:
      print(i,'is␣composite')
```

Notice that the code of the `is_prime` function in the example above uses a variable called i, as does the code that invokes the `is_prime` function (the code below the function definition). This is perfectly okay, in the sense that the behavior is the intuitive and more conservative one. That is, those are two separate variables, even if they have the same name, and so there is no conflict.

More specifically, the i variable, like all variables assigned within the `is_prime` function, is stored in a private memory area for each invocation of the function. That is, every time you invoke a function, even recursively, that invocation sets aside a private memory area for the execution of the code of the function. So, when the code assigns a variable, that variable does not refer to other variables with the same name elsewhere. Notice that the parameters of a function, such as the variable n in the function `is_prime(n)`, are treated exactly like any other private variable in the function.

One last but important point about function definitions. Despite the fact that local variables are private and do not conflict with variables assigned elsewhere with the same name, a function may still have side-effects. That is, a function may change the value of variables defined outside the code of the function. A classic example is a function that takes an array as an argument. Try this code:

```
def reset_to_zero(A):
   for i in range(len(A)):
      A[i] = 0

B = [1, 2, 3]
reset_to_zero(B)
print(B)
```

So, think about it, a function `prog1(n)` that takes a number n as a parameter can change the value of the variable n, but since that variable n is "private", that does not change the value of anything

outside the function. However, a function `prog2(A)` that takes an array *A* can still change the content of the array outside the function. Why? What's the difference? This is a tricky question, but it's also an important one. So, I don't want to dismiss it. I will try to be concise, which means that I can't be very precise. But I still want to describe things as they are *in essence.*

```
def prog1(n):
   n = 0

def prog2(A):
   A[0] = 0

n = 7
x = 7
prog1(x)
print(x,n)

B = [1, 2, 3]
prog2(B)
print(B)
```

So, what's the difference between `prog1(n)` and `prog2(A)` when *n* is a number and *A* is an array? Well, there is no difference! That is, Python treats the two functions and the two variables (parameters) in exactly the same way. The difference is in the nature of those variables. In fact, this is the nature of *all* variables in Python.

As we said initially, a variable is a space in memory that has a name—meaning that we can refer to by a certain name within some instructions in the program—and that can contain a value. This is true for both a variable `n = 7` containing the integer value 7, and an array `A = [1, 2, 3]`. However, the difference is that some "values" aren't values per-se, but rather *references* to other areas in memory that in turn contain other variables that can take values including references to yet-other areas in memory. That is the case for an array. So, *A* per-se is a variable just like *n*. But *A*'s value isn't the whole array. That is, *A* does not in itself contain the sequence `[1,2,3]`. And this shouldn't be surprising, since the sequence can be arbitrarily long. Instead, the sequence is stored somewhere else in memory, and *A* contains the *address* of that memory location.

So, let's see what happens when we invoke a function that takes an array as a parameter. The instruction `B = [1, 2, 3]` in the example above reserves some memory for the array itself, meaning a block of three variables (memory cells) containing the numeric values 1, 2, and 3, respectively, plus a variable called *B* that contains the address of that block of variables. So then we could write the expression `B[0]` that would refer to the first variable in the block of variables referenced by *B*.

When we then invoke the function `reset_to_zero(B)`, that executes the code in the function definition with parameter *A* initialized with—meaning, assigned to—the value of *B*. Think about what happens here: *B* is a variable whose *value* is the address of the block of variables holding the elements of the array; that value, meaning that address, is then assigned to the private parameter variable *A* in the execution of the `reset_to_zero` function. So, now `reset_to_zero` has a variable *A* that is distinct from *B*, but that contains a reference to the same block of variables as *B*. So, the expression `A[0]` within the code of the `reset_to_zero` function refers to exactly the same variable as `B[0]` would in the context of the invocation of the function. This is why the effect of the function, from the perspective of the caller code, is to change the value of the *B* array.