

# Elementary Algorithmic Programming in Python

Antonio Carzaniga

21 February 2022

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 directions to go to some 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, and for our purposes we use a language called Python.

Our goal, however, is not to learn much about Python, but rather to learn 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. And as soon as you have at least a basic idea of the concepts described in the text, you 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 the best way to learn programming anyway—or anything else, really.

## Preliminaries

I assume you have Python installed on your computer. Specifically, I assume you have Python-3. I also assume that you have some basic experience in 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 by example.

## Running Python as an Interactive Command Shell

Run Python as an interactive shell by opening a command shell (typically through a terminal application) and by running the command `python3`. You should see something like this:

```
> python3
Python 3.8.5 (default, Jul 28 2020, 12:59:40)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

You might get something slightly different. But if you get something completely different, 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. 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 is what you typically run when you open a terminal application. Unfortunately, these names can be confusing. Essentially all shells are "command" shells, since their job is to read commands from a user interface (the terminal), execute those commands, and then write the resulting output on the same or another interface.

Anyway, I will distinguish the *command* shell from the *Python* shell. 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 (greater than). You may 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.

Then, from that command shell we run `python3` and obtain the prompt `>>>` of the *Python* shell, which consists of three `>` characters one after the other. This means that the command shell runs `python3`, which is the Python shell, that takes over the input and output and now starts prompting you to input commands.

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:

```
> 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 write an algorithm in the form of a Python *function*—and then run that program (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 called `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 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, 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.

## 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, one after the other, in the given order.

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.*

```
>>> 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, we'll 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 (or more) 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—that is, 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* is the code block after then `else` statement.

## 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 *one* 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 the block of instructions that follows the `for` instruction (indented) 20 times. In that block of instructions 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 :`, and that repeats the block of instructions that follows 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 is really nothing new, since it's exactly what we were doing before with `for i in range(20)`. You'd be right, it's exactly the same thing. But it shows how *while*-loops work: you test the condition of the `while` instruction, and if it's true then you execute 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 repeat the loop again and again. You stop the when you loop back to the `while` instruction and the condition turns out to be false.

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. Pretty weird eh? Can you input a number that makes the code loop forever? I bet you can't. But if you find one of those numbers, let me know!

Two more points about loops. What if we need to break out of a loop? As I said, the loop condition in the *while* instruction is evaluated when the execution goes back to that 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. For example:

```
n = int(input('choose_an_integer_n_>_1:_'))
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 say this is a read-do document? Do it. Try this code yourself!

## 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  $a_1, a_2, \dots, a_n$  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 to 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 we 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.

The Python language provides many other ways to cut or modify sequences. However, we intentionally do not consider them, since they often incur hidden complexities.

### 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
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?

## 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 an algorithm, or a method, or a subroutine—that we 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 return value of the function is determined by the execution of the `return` instruction, which also terminates the execution of the function and therefore the execution “returns” 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 below the definition that also invokes the `is_prime` function. This is perfectly okay, in the sense that the behavior is the intuitive and more conservative one: 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  $n$  is a private variable, 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*.

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, 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.

## More Exercises

There are three ways to learn computer programming: practicing, practicing, and practicing. So, get on with it!

### Minimum

Write a function `minimum(A)` that returns the minimum value in  $A$ . You may assume that the sequence contains at least one element.

#### Examples

```
>>> minimum([1,2,3])
1
>>> minimum([3,2,1])
1
>>> minimum([100,2,55,4,3,2,67,3])
2
>>> minimum([7])
7
```

### Position in Sequence

Write a function `position_in_sequence(A,x)` that returns the first position in which value  $x$  appears in  $A$ . Positions start from 0. Return  $-1$  if  $x$  does not appear in  $A$ .

#### Examples

```
>>> position_in_sequence([6, 7, 10, -2, 3], 7)
1
>>> position_in_sequence([3, 2, 1, 1, 3, 2, 2, 3, 1], 1)
2
>>> position_in_sequence([], 1)
-1
>>> position_in_sequence([2, 3, 4, 5], 1)
-1
>>> position_in_sequence([2, 3, 4, 5], 2)
0
```

### Count Lower

Write a function `count_lower(A,x)` that returns the number of elements in  $A$  whose value is less than  $x$ .

## Examples

```
>>> count_lower([0,7,10,-2,3], 7)
3
>>> count_lower([3, 2, 1, 1, 3, 2, 2, 3, 1], 2)
3
>>> count_lower([3, 2, 1, 1, 3, 2, 2, 3, 1], 10)
9
>>> count_lower([], 1000000000000000)
0
>>> count_lower([2, 3, 4, 5], 1)
0
```

## Check Sorted

Write a function `check_sorted(A)` that returns `True` if the given sequence `A` is sorted in non-decreasing order, or `False` otherwise.

## Examples

```
>>> check_sorted([1,2,3])
True
>>> check_sorted([1,3,2])
False
>>> check_sorted([])
True
>>> check_sorted([7])
True
>>> check_sorted([50,50,50])
True
>>> check_sorted([43,51,51])
True
>>> check_sorted([43,51,50,51,70])
False
```

## Log Base Two

Write a function `log_base_two(n)` that, given a positive integer `n`, returns the integer logarithm base-two of `n`, that is, the maximal integer `k` such that  $2^k \leq n$ .

## Examples

```
>>> log_base_two(1)
0
>>> log_base_two(2)
1
```



```
>>> log_base_two(5)
2
>>> log_base_two(1000)
9
```

## Maximal Difference

Write a function `maximal_difference(A)` that returns the maximal difference between any two elements of the given sequence `A`.

### Examples

```
>>> maximal_difference([2, 1, 5, 9, 4, 10, 8])
9
>>> maximal_difference([1])
0
>>> maximal_difference([1, 1, 1])
0
>>> maximal_difference([10,-3, 4, 11, 0, 9])
14
```

## Minimal Sum

Write a function `minimal_sum(A, x)` that returns `True` if there are some elements in `A` whose sum is greater or equal to `x`, or `False` otherwise.

### Examples

```
>>> minimal_sum([], 1)
False
>>> minimal_sum([1], 1)
True
>>> minimal_sum([3, 2], 4)
True
>>> minimal_sum([3, -2], 4)
False
>>> minimal_sum([2, 1, 5, -3, 9, 4], 20)
True
>>> minimal_sum([32,-3,10,7,-4,18,25], 50)
True
>>> minimal_sum([32,-3,10,7,-4,18,25], 94)
False
```

## Isolated Elements

Write a function `isolated_elements(A)` that prints, in order, all the elements that are different from all their adjacent elements, that is, each element that is different from the previous element (if that exists) and the next element (if that exists).

### Examples

```
>>> isolated_elements([])
>>> isolated_elements([1])
1
>>> isolated_elements([7,3])
7
3
>>> isolated_elements([2, 2])
>>> isolated_elements([2, 2, 3, 2, 3])
3
2
3
>>> isolated_elements([-2, 2, 2, 2, -2])
-2
-2
>>> isolated_elements([1,2,1,2,1,2,1,2])
1
2
1
2
1
2
1
2
1
2
```

## Repeated Elements

Write a function `repeated_adjacent_elements(A)` that prints, in order, all the elements that are repeated in sub-sequences of two or more elements. For every maximal subsequence of identical values, `repeated_adjacent_elements(A)` must print that value *only once*.

### Examples

```
>>> repeated_adjacent_elements([])
>>> repeated_adjacent_elements([1])
>>> repeated_adjacent_elements([1, 2])
>>> repeated_adjacent_elements([3, 3])
3
>>> repeated_adjacent_elements([1,-1,7,7,-1,7,1,7,7,7,7,2,2])
7
7
```

## Compression

Write a function `compress_sequence(A)` that prints a “compressed” version of the given sequence  $A$ . Compression is obtained by printing three or more consecutive elements with equal values with a line containing  $x * k$ , where  $x$  is the value of those elements and  $k$  is the number of consecutive equal elements.

### Examples

```
>>> compress_sequence([-1,1,1,1,7,7,7,7,5,5,1,1,4,1])
-1
1 * 3
7 * 4
5
5
1
1
4
1
```

## Maximal Increasing Subsequence

Write a function `maximal_increasing_subsequence(A)` that returns the maximal length of any strictly increasing sequence of contiguous elements of  $A$ . For  $A = -1, 1, 7, 5, -2, 1, 2, 7, 7, 5, 0, 1, 3, 4, 1$ , the result is 4, since there is at least one increasing subsequence of length 4 (e.g.,  $-2, 1, 2, 7$ ) but there is no increasing subsequence of length 5.

### Examples

```
>>> maximal_increasing_subsequence([-1,1,7,5,-2,1,2,7,7,5,5,1,1,4,1])
4
```

## Partition Even/Odd

Write a function `partition_even_odd(A)` that takes an array of integers  $A$  and sorts the elements of  $A$  so that all the even elements precede all the odd elements. The function must sort  $A$  *in-place*. This means that it must operate directly on  $A$  just by swapping elements, without using an additional array.

## Examples

```
>>> A = [-1,1,7,5,-2,1,2,7,7,5,5,1,1,4,1]
>>> partition_even_odd(A)
>>> print(A)
[-2,2,4,-1,1,7,5,1,7,7,5,5,1,1,1]
```

Notice that in general the solution is not unique. For example, the following result would be equally correct:

```
>>> A = [-1,1,7,5,-2,1,2,7,7,5,5,1,1,4,1]
>>> partition_even_odd(A)
>>> print(A)
[2,4,-2,5,1,5,1,7,7,7,-1,1,1,5,1]
```

## Like or dislike

Write a function `like_or_dislike(A)` that takes a sequence of ratings expressed with the strings 'like' and 'dislike'. The function must print 'like' if the most prevalent rating is 'like', or 'dislike' if the most prevalent rating is 'dislike', or 'undecided' otherwise.

## Examples

```
>>> like_or_dislike(['like'])
like
>>> like_or_dislike(['dislike','like','dislike'])
dislike
>>> like_or_dislike(['like','dislike'])
undecided
>>> like_or_dislike([])
undecided
```

## Stops and Inversions

A car moves along a road. For simplicity you may assume that the road is perfectly linear and that the position of the car is given as the distance from a certain reference point on the road. Write a function `stops_and_inversions(P)` that, given an array  $P$  of time-ordered positions of the car, prints the number of times the car stopped and the the number of times that the car inverted the direction of motion.  $P = p_1, p_2, p_3, \dots$  is time ordered in the sense that the car is at position  $p_1$  *before* it is at position  $p_2$ , it is at  $p_2$  before it is at  $p_3$ , and so on.

## Examples

```
>>> stops_and_inversions([])
stops: 0
inversions: 0
```

```

>>> stops_and_inversions([5,5,5])
stops: 0
inversions: 0
>>> stops_and_inversions([5,5,5,6,7,8])
stops: 0
inversions: 0
>>> stops_and_inversions([5,5,5,6,7,8,8])
stops: 1
inversions: 0
>>> stops_and_inversions([1,5,10,12,12,12,15,20,24])
stops: 1
inversions: 0
>>> stops_and_inversions([1,5,10,12,12,11,8,7])
stops: 1
inversions: 1
>>> stops_and_inversions([1,5,10,12,12,11,8,7,9,11,20,30])
stops: 1
inversions: 2
>>> stops_and_inversions([1,5,10,12,12,11,8,7,9,11,20,30,30,30,30,35,35])
stops: 3
inversions: 2

```

## Minimal Bounding Rectangle

Consider a sequence of  $n$  points  $p_1, p_2, \dots, p_n$  in a plane identified by their Cartesian coordinates. The coordinates are given in the form of two arrays  $X$  and  $Y$  both containing of  $n$  numbers, such that point  $p_i$  has coordinates  $X[i]$  and  $Y[i]$ .

Write a function called `minimal_bounding_rectangle_area(X,Y)` that, given two arrays of coordinates  $X$  and  $Y$  representing  $n$  points, returns the area of the smallest axis-aligned rectangle that covers all the  $n$  points. An axis-aligned rectangle is such that its sides are parallel to the  $X$  or  $Y$  axis.

### Examples

```

>>> minimal_bounding_rectangle_area([3,2,10,4],[5,5,3,2])
24
>>> minimal_bounding_rectangle_area([2],[89])
0
minimal_bounding_rectangle_area([-8,8,9,-9,4,-4,4,-9,7],[10,1,-1,-5,7,-7,-2,0,3])
306

```

## Distance Between Points

Write a function called `distance_between_points(X,Y,d)` that, given two arrays of coordinates  $X$  and  $Y$  representing  $n$  points, and a number  $d$ , return True if and only if two of the given points are at a distance  $d$  from each other, or False otherwise.

## Examples

```
>>> distance_between_points([3,2,1,10,5,4],[5,5,4,3,1,2],5)
True
>>> distance_between_points([3,2,1,10,5,4],[5,5,4,3,1,2],1)
True
>>> distance_between_points([3,2,1,10,5,4],[5,5,4,3,1,2],6)
False
>>> distance_between_points([3,2,1,10,5,4],[5,5,4,3,1,2],4)
False
```

## Find a Square

Write a function called `find_a_square(X,Y)` that, given two arrays of coordinates  $X$  and  $Y$  representing  $n$  points, and a number  $d$ , return `True` if and only if four of those points are the vertices of a square, or `False` otherwise.

## Examples

```
>>> find_a_square([0,1,2,3],[3,1,4,2])
True
>>> find_a_square([0,1,2,1],[3,1,3,5])
False
```

## Most Common Digit

Write a function called `most_common_digit(n)` that takes an integer  $n$ , and returns the most common digit in the decimal representation of  $n$ . If there are two or more most common digits, the function must return the smallest one.

## Examples

```
>>> most_common_digit(1969)
9
>>> most_common_digit(44272)
2
>>> most_common_digit(36223771538825331723)
3
>>> most_common_digit(0)
0
```

## Sums of Squares

Write a function called `sums_of_squares(n)` that prints all the pairs of squares that sum to  $n$ . That is, `sums_of_squares(n)` prints a line  $a^2 + b^2$  for all the distinct pairs  $a \leq b$  such that  $a^2 + b^2 = n$ .

## Examples

```
>>> sums_of_squares(25)
9 + 16
>>> sums_of_squares(21)
>>> sums_of_squares(125)
4 + 121
25 + 100
>>> sums_of_squares(178)
9 + 169
>>> sums_of_squares(179)
>>> sums_of_squares(17993241)
176400 + 17816841
```