PYTHON

A. SIMON

CONTENTS

- Where to find Python?
  - On your own windows machine you probably want this: `http://wiki.math.bme.hu/view/AnacondaInstall` With this, you'll get a graphical development environment called `spyder` that people seem to like.
  - If you use Linux, you already have Python installed. Install `VSCode` or `spyder3` and/or `ipython3`.

Date: February 27, 2025.

  - https://colab.research.google.com/ or https://cocalc.com. You need to register here, but get a nice a jupyter notebook.
  - https://sagecell.sagemath.org/, choose Python from the available languages. This is the worst but simplest choice.
- Reading material:
  - http://math.bme.hu/~asimon/info2/python.pdf (the newest version of the lecture notes (this document))
  - Wentworth &al., How to think like a computer scientist
- Exercises for the lab sessions are here:
  - http://math.bme.hu/~asimon/info2/pythex.pdf

1. Python from scratch

1.1. **Python as a calculator.**

If you start the Python interpreter `python3`, or, preferably, `ipython3` from a terminal, you get a prompt, such as this:

```
>>>
```

or this:

```
In [84]:
```

Here you can type Python commands and the interpreter will execute them and return the results (if any). This is the same as Sage's behaviour, so should be familiar for most of you.[1] For example:

```
Python 3.12.9 (main, Feb  4 2025, 00:00:00) [GCC 14.2.1 20240912 (Red Hat 14.2.
Type "help", "copyright", "credits" or "license" for more information.
>>> 2+3
5
```

or

```
>>> 2**3
8
```

or

```
>>> 2**3 == 8
True
>>> (123**13) % 13 == 123 % 13  #because of Fermat's little theorem
True
```

(Whatever is written after an `#` on a line is ignored by Python; it's a *comment* for the human reader, just like % in LaTeX.)

It's not just integers (`int`s) that we can work with, but various other types of data, too. For example:

[1] In a jupyter notebook you need to press `Ctrl+Enter` to execute your command.

```
>>> 3.14 #a floating point number (float)
3.14

>>> 'This is a string' #a string
'This is a string'

>>> [1, 2, 'Hello', 3.5] #a list (of ints, a string, and a float)
[1, 2, 'Hello', 3.5]

>>> [1, 2, 'Hello', 3.5][2] #the 2nd member of the list (indexing starts at 0)
'Hello'
```

We can also store values (that you type in or get as a result) in *variable*s. (The usual terminology is *assigning a value to* a variable.) You can think of a variable as a box with a name into which you put the value. (That name should only contain letters of the alphabet, numerals (but mustn't start with one), and the underscore (_) character.)

```
>>> my_first_var = 2**3
```

From now on, `my_first_var` will contain 8 until we change it (or quit the Python interpreter):

```
>>> my_first_var
8

>>> 2 * my_first_var
16
```

and I can write `my_first_var` whenever I mean 8.

1.2. **Python as a programming language.** Why not write the *literal value*[2] 8 (or `2**3`) directly? There are quite a few reasons (for example, we don't want to write, say, `[1, 2, 'Hello', 3.5]` every time we need it) but the most important by far is that Python is not just a calculator. Typically, I want to do something (say `Action`) with lots of different values, and the way to achieve that is to do what I want to do with the variable, changing it to the different values, instead of doing `Action` with each of the literal values. For example, one such action is `print`ing. If I want to see the square of a few numbers, I can do this:

```
>>> print(1**2)
1

>>> print(2**2)
4
```

[2]A literal value is a value that appears directly in the source code of a program.

```
>>> print(3**2)
9

>>> print(4**2)
16
```

but the following is much more practical:

```
>>> number = 1
>>> while (number < 5):
...     print(number ** 2)
...     number = number + 1
...
1
4
9
16
```

Before trying to understand how this worked, let's see why this is much better. The main reason is that we only had write how to do `Action` *once* (on the third line). The rest is just specifying for which values of the variable `number` we want to do it. One benefit is that if we decide to print more squares, we just have to change 4 to a bigger number. Or if we want to print only every third square, we only have to change the last line:

```
>>> number = 1
>>> while (number < 15):
...     print(number ** 2)
...     number = number + 3
...
1
16
49
100
169
```

This was an example of a *loop*, more specifically a `while` loop. Loops are what make variables not only useful but indispensable. (And loops are one of the two components that turn a calculator into a programming language.)

Think about how this could've been achieved with our individual `print`s! And it's not just a matter of convenience. There are ways for a program to get input from a user (see § 3!). Now what if `15` above was not a constant, but the result of a user input? How could we modify our list of `print`s to print the amount of squares the user wishes us to print?

The syntax of a while loop is this:

```
while condition:
   do_this
   ...
   do_that
```

This executes repeatedly everything in its *body* (the indented block[3], `do_this`,…, `do_that`) as long as *condition* holds, and then control goes to the part of the program (if any) that follows the body. Python knows where that is, because it is indented at most as far as the `while` keyword itself. For example,

```python
number = 1
while (number < 15):
    print(number ** 2)
    number = number + 3
print('Done')
```

```
1
16
49
100
169
Done
```

Remark 1.1. There is a huge difference between

```
>>> 42
42
```

and

```
>>> print(42)
42
```

The first returns a value, the second only prints one: `print()` is a built-in function, but one that is called only for its *side-effect*, namely printing its argument(s) on the console, not for its value (which is `None`). Compare this:

[3]Those `...`s you sometimes see at the beginning of lines are not part of the definition. They're the prompts of the command line interface (just like `>>>`) that show that it expects more lines.

```
>>> a = 42
>>> print(a)
42
```

with this:

```
>>> a = print(42)
42
>>> print(a)
None
```

In other words: printing is for giving information to a user (a.k.a. a human) only.[4]

Back to our looping example. There is another, often more convenient way of looping: the `for` loop. Here's how our first `while` loop above:

```
number = 1
while (number < 5):
    print(number ** 2)
    number = number + 1
```

can be written as a `for` loop:

```
>>> for number in range(1,5):
...     print(number ** 2)
...
1
4
9
16
```

and the second:

```
number = 1
while (number < 15):
    print(number ** 2)
    number = number + 3
```

can be rewritten as a `for` loop like this:

```
>>> for number in range(1,15,3):
...     print(number ** 2)
...
1
16
49
100
169
```

[4]At least for now. This changes when we use `print()` for writing in a file (see § 3!)

The nice thing about the `for` loop is that it can iterate over not just a range of numbers, but almost anything for which this make sense (these things are called *iterables*): the characters of a string, the lines of a file, records of a database table,... and, perhaps most commonly, the elements of a list. (What `range()` returns is not a list, but can be turned into a list using the function `list()`; for example, `list(range(1,15,3))` returns [1, 4, 7, 10, 13].) For example, here is one way to compute the product of the elements of a list:

```
>>> #I store the list in a variable, because I want to use it later.
>>> l = [4,2,5,9]
>>> product = 1
>>> for n in l:
...     product = product * n
...
>>> product
360
```

We could've done this with a `while` loop, too:

```
>>> product = 1
>>> index = 0
>>> while index < len(l):
...     product = product * l[index]
...     index = index + 1
...
>>> product
360
```

but that is much less elegant.

One technical detail about both kinds of loops: we can jump out of a loop early with the `break` statement, and also go immediately to the next iteration with `continue`. But to be able to show any meaningful examples of these, we need the other construction that turns a calculator into a full-blown programming langue: the *if* statement.

```
if condition:
   do_this_if
   ...
   do_that_if
```

which executes everything in its body (`do_this_if`,...,`do_that_if`) but only if *condition* holds; and the extenden version:

```
if condition:
   do_this_if
```

```
    ...
    do_that_if
else:
    do_this_if_not
    ...
    do_that_if_not
```

which executes the body of the `else` clause if the condition doesn't hold. For example:

```
>>> if 2<3:
...     print('OK')
...
OK
>>> if 3<2:
...     print('OK')
...
>>> if 3<2:
...     print('OK')
... else:
...     print('Not OK')
...
Not OK
```

Now that we have the `if` statement, we can illustrate what `break` and `continue` does.

Suppose that we only want to compute the product of the odd elements of a list. Here is one way to achieve this:

```
>>> l = [4,2,5,9]
>>> product = 1

>>> for n in l:
...     if n%2 == 0: #if n is even
...         continue #take the next element of the list
...                  #that is, skip the rest of the body of the loop
...     product = product * n
...
>>> product
45
```

Exercise 1.1. Could we change the loop to achieve the same effect without using `continue`?

What if we only want to take the product until we encounter an odd number? Here's where `break` helps:

```
>>> l = [4,2,5,9]
>>> product = 1

>>> for n in l:
...     if n%2 == 1: #if n is odd
...         break    #get out of the loop. NOW!
...     product = product * n
...
>>> product
8
```

The last important building block that we need is the ability to define (and call) functions. This is not *absolutely* necessary, but would be very hard to live without.

What problems are they supposed to solve? A few examples ago we computed the product of the members of a list. If we had to do that for one list, it's more than likely that we'll want to do it again with other lists of numbers. So what we do is "abstract away" the concrete list from that program, and give the whole thing a name (`product` seems like a good choice) to be able to refer to it. Here's the result:

```
>>> def product(l):
...     result = 1
...     for n in l:
...         result = result * n
...     return result
...
```

which can be used (*called*) like this:

```
>>> product([4,2,5,9])
360

>>> product([4,2,5,9,10])
3600
```

The first line of the definition says that the name of the function being defined is `product` and its only parameter (or argument) is `l`. This means that this function has to be called with one argument, which will be assigned to the variable (more specifically, the parameter) `l`, which can be used in the body (the indented block, as always) of the function. It is a *local variable*, which means that even if we have a variable of the same name outside the function definition, its value will be restored when the function returns. The same is true for `result` defined on the second line. Here's an example that shows this:

```
>>> a = 1
>>> b = 2
>>> def fun(a):
...     b = a
...     return b
...
>>> fun(42)
42
>>> a
1
>>> b
2
```

Why is defining `product()` as a function better than copying our old code that computed the product of a list with the list replaced by a new one every time we need it? Apart form the obvious reason (that code is just a few lines, but what if we're talking about another, which is a few thousand lines?), there is a decisive one: if we find out that there is a bug in our implementation of `product()`, we only need to correct it in one place, the definition of the function.

Here's the general syntax of a function definition:

def *name(parameter1,parameter2,...)*:
    do_this
    ...
    do_that

There might be one or more

    return *a_value*

statements in the body of the function. What it does is make the function return immediately, and give back (return) the

    a_value

to the caller. The function can return even without a `return` statement, but it will then return the value `None` which is as good as returning nothing. (`return` in itself, with no argument, has the same effect.) We have already seen that `print()` does this. There are lots of other examples where a function doesn't return anything but is still useful. For example, it might write to a file (or delete all our files), make a phone call, etc. Before showing another example of a function that doesn't return anything but may still be useful, we need to know that in Python, every object has a type, and not only can we ask what it is, but also if an object is of a certain type:

```
>>> type(1)
<class 'int'>
>>> isinstance(1,int)
True
>>> isinstance(1,list)
False
```

Now we're ready for the example of a potentially useful function that doesn't return anything:

```
>>> def show(arg):
...     print(arg, 'is of type', type(arg))
...     if isinstance(arg,list) or isinstance(arg,str):
...         print('It is of length', len(arg))
...     else:
...         if isinstance(arg,int):
...             print('It is', 'odd.' if arg%2 == 1 else 'even.')
...         else:
...             print("I can't tell you more about it.")
...
>>> show(['Hell', 'o', 12])
['Hell', 'o', 12] is of type <class 'list'>
It is of length 3
>>> show('Hello')
Hello is of type <class 'str'>
It is of length 5
>>> show(3)
3 is of type <class 'int'>
It is odd.
>>> show(3.14)
3.14 is of type <class 'float'>
I can't tell you more about it.
```

Here we used the `if` *expression*[5] (as opposed to the `if` *statement*), which returns its first argument (the one before the keyword `if`) if its second argument (the one between `if` and `else`) is `True`, and its third argument (the one after `else`) otherwise:

```
>>> 'Yes' if True else 'No'
'Yes'
```

[5]An expression is something which produces a value (so variables, literals and function calls, among other things, are expressions).

```
>>> 'Yes' if False else 'No'
'No'
```

So this variant of `if` produces a value depending on a condition. This is useful in at least two situations. First,

```
maximum = b if a < b else a
```

is much more concise than the equivalent

```
if a < b:
    maximum = b
else:
    maximum = a
```

Exercise 1.2. Rewrite the `show()` function in such a way that it doesn't use `if` expressions.

The second situation is in a `lambda`, or anonymous function. This is a special kind of function for those cases when we need a function only once. Here is an atypical example:

```
>>> (lambda x : x ** 2)(3)
9
```

but we will see typical ones later.[6]

A `lambda`'s body can only have one expression in it, and it returns the value of that expression (there's no need for a `return` statement). So the only way to make a decision in a `lambda` is with the help of an `if` expression, as in the following example:

```
>>> (lambda x : x**3 if x > 0 else -x**3)(-2)    #|x^3|
8
```

or even

```
>>> (lambda x : (x if x > 0 else -x)**3)(-2)     #|x|^3
8
```

1.3. Miscellaneous basics.

Indexing lists (and strings). We have seen that lists can be indexed with the operator `[]`:

```
>>> numbers = list(range(18))
>>> numbers
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]
```

[6]For the impatients: if `l` is a list of non-empty lists of numbers, then `sorted(l, key=lambda x : x[0])` will return `l` sorted by the magnitude of the first numbers of the lists.

```
>>> numbers[10]
10
```

But there's more to [] than that.

When the index `i` is negative, `len() - i` is used instead (so we count from the end, but this backward indexing starts at 1, not 0).

Exercise 1.3. What would be the problem with starting at 0?

```
>>> numbers[-3]  #we can use this to find the 3rd member counting from the end
15
>>> numbers[len(numbers)-3] #instead of this
15
```

One can can extract not just individual members, but various *slice*s of a list:

```
>>> numbers[:10]  #the first 10 members
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> numbers[10:15]  #from the 10th up to (but not including) the 15th member
[10, 11, 12, 13, 14]
>>> numbers[15:]  #everything from the 15th member
[15, 16, 17]
>>> numbers[::2]  #only the ones with even indices
[0, 2, 4, 6, 8, 10, 12, 14, 16]
>>> numbers[1::2]  #only the ones with odd indices
[1, 3, 5, 7, 9, 11, 13, 15, 17]
```

Slices work with negative indices, too:

```
>>> numbers[:-10] #until the 10th from the end
[0, 1, 2, 3, 4, 5, 6, 7]
>>> numbers[-10:] #the last 10
[8, 9, 10, 11, 12, 13, 14, 15, 16, 17]
>>> numbers[-10:-3]  #the last 10 members except for the last 3
[8, 9, 10, 11, 12, 13, 14]
>>> numbers[:-3][-7:]  #the same: forget the last 3 and then take the last 7
[8, 9, 10, 11, 12, 13, 14]
>>> numbers[-10:][:7]  #the same: take the last 10 and then take its first 7
[8, 9, 10, 11, 12, 13, 14]
```

steps can also be negative:

```
>>> numbers[::-1]  #all of them, backwards
[17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>> numbers[::-2]  #every second of them, backwards
[17, 15, 13, 11, 9, 7, 5, 3, 1]
```

but then *start* and *end* are interchanged:

```
>>> numbers[4:1:-1] # from the 4th (incl.) to the 1st (excl.)
[4, 3, 2]
>>> numbers[4::-1]  # from the 4th, to the beginning
[4, 3, 2, 1, 0]
```

All these tricks work for strings, too. For example:

```
>>> s = 'abcdefgh'
>>> s[3]
'd'
>>> s[-3]
'f'
>>> s[5:]
'fgh'
>>> s[5::-1]
'fedcba'
```

But there is one feature of the indexing and slicing operators that don't apply to strings (because strings are *immutable*): they can be used for assignment, too:

```
>>> numbers
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]
>>> numbers[0]=-10
>>> numbers
[-10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]
>>> numbers[1:4] = [11,22,33]
>>> numbers
[-10, 11, 22, 33, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]
>>> numbers[-1:-4:-1] = [111,222,333]
>>> numbers
[-10, 11, 22, 33, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 333, 222, 111]
```

Exercise 1.4. What's the difference between `numbers[1] = [True]` and `numbers[1:2] = [True]`? Is one of these equivalent to `numbers[1] = True`?

List comprehension. Given a list (or in fact any iterable, such as a `range`) `l`, `[f(x) for x in l]` returns a list whose ith member is the result of `f` applied to the ith member of `l`. So for example

```
>>> [x/2 for x in range(10)]
[0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5]
```

We can filter `l` if we want:

```
>>> [x/2 for x in range(10) if x%2 == 0]
[0.0, 1.0, 2.0, 3.0, 4.0]
```

List comprehensions can of course be nested (and then should be read "outside in"):

```
>>> [[str(i)+j for j in 'abc'] for i in range(3)]
[['0a', '0b', '0c'], ['1a', '1b', '1c'], ['2a', '2b', '2c']]
```

Here we use the `str()` function, which returns a string representation of its argument, and the fact that the sum of two strings is their concatenation.

Variable unpacking (basic version). Besides lists, there is another data structure, `tuple`, that can hold objects in a sequential order. If you write

```
>>> 2+3, 2*3, 2**3
(5, 6, 8)
```

the result is not three separate object, it's one `tuple` (of three objects). Here's proof of that:

```
>>> a = 2+3, 2*3, 2**3
>>> a
(5, 6, 8)
>>> type(a)
<class 'tuple'>
```

But it's easy to unpack a tuple into its components:

```
>>> a, b = 2+3, 2*3
>>> a
5
>>> b
6
```

A useful consequence of this is that instead of writing

```
>>> a = 5
>>> b = 10
```

one can write

```
>>> a, b = 5, 10
```

The two are not completely equivalent, because in the second case, the two assignments happen in parallel. But that means, that we can exchange the values of two variables like this:

```
>>> a, b
(5, 10)
>>> a, b = b, a
>>> a, b
(10, 5)
```

instead of having to use a temporary variable, as in more primitive languages:

```
>>> temp = a ; a = b; b = temp
>>> a, b
(5, 10)
```

Methods. We'll frequently talk about *methods* without formally introducing them (until § 8). It's safe to think that they're just like ordinary functions but called in a peculiar manner. Instead of writing `upper('abc')` to get the uppercase version of `'abc'`, we write

```
>>> 'abc'.upper()
'ABC'
```

and say that `.upper()` is a method of strings. The latter means that this call only makes sense if what's before the dot is (or evaluates to) a string. Another example is `.append()`, which is a method of lists. It appends its argument to the list it's called on:

```
>>> l = [1,2,3]
>>> l.append(100)
>>> l
[1, 2, 3, 100]
```

Note that unlike `.upper()`, it doesn't return a value: it changes the object it's called on. For now, we can imagine that, say,

```
obj1.m(obj2, obj3)
```

is like the function call

```
m(obj1, obj2, obj3)
```

it's just that the type (class) of `obj1` and `m` have a special relationship.

Modules. We'll learn about modules later, for now it's enough to know that whenever the keyword `import` appears, it means that some extra functionality will be provided for the rest of the program. In each case it will be clear what. For example:

```
>>> import math
>>> math.sqrt(2), math.pi
(1.4142135623730951, 3.141592653589793)
```

Once `import`ed, we can get a lot of information about the `math` `modul` by writing

```
help(math)
```

1.4. **Some examples.**

Example. Define a function `repeats()` of one argument, a list of numbers, which returns `True` if two consecutive members of the list are equal, and `False` otherwise.

For example:

```
>>> repeats([])
False
>>> repeats(range(10))
False
>>> repeats([1,2,1,4])
False
>>> repeats([1,2,1,4,4])
True
>>> repeats([1,2,1,1,4])
True
```

```
def repeats(l):
    for i in range(1,len(l)):
        if l[i-1] == l[i]:
            return True
    return False
```

or

```
def repeats(l):
    if l == []: return False  #need to check because of the next line
    last = l[0]
    for i in l[1:]:
        if i == last:
            return True
        last = i
    return False
```

or

```
def repeats(l):
    for x, y  in zip(l, l[1:]):
        if x == y:
            return True
    return False
```

We haven't learned about the function, but you can look up its documentation (`help(zip)` or `zip?`) or wait until we encounter it on page 34.

Example. Write a function `squares()` of two arguments, so that `squares(m,n)` returns the list of squares between m and n.

```
def squares(m,n):
    return [i**2 for i in range(n+1) if m <= i**2 <= n]
```

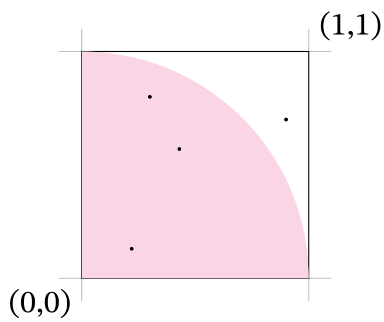
or

```
def squares(m,n):
    res = []
    i = isquare = 0 #this works as expected
```

```
    while isquare <= n:
        if isquare >= m:
            res.append(isquare)
        i = i + 1
        isquare = i**2
    return res
```
or, since $m \leq i^2 \leq n \iff \sqrt{m} \leq i \leq \sqrt{n} \iff \lceil \sqrt{m} \rceil \leq i < 1 + \lfloor \sqrt{n} \rfloor$ for $i \in \mathbb{N}$,

```
import math
def squares(m,n):
    return [i**2 for i in range(math.ceil(math.sqrt(m)),1+math.isqrt(n))]
```

Example. Write a function of one (integer) argument that computes an approximation to π by throwing darts randomly at the unit square with a quarter of the unit circle in it, and comparing the number of throws that landed in the quarter circle with the total number of throws (the argument of the function).



The function `random()` of the package `random` below returns a `float` in the interval $[0, 1]$. Since we use it a lot[7], we import it in such a way that it doesn't need to be qualified with the package name.

```
from random import random

def mcpi(n):  #Monte Carlo pi
    incircle = 0
    for _ in range(n):  #we don't care about the loop variable
        px,py = random(),random()
        if px ** 2 + py ** 2 <= 1:
            incircle += 1  #a concise way of incrementing incircle
                           #see below!
    return(4*incircle/n)
```

[7]twice

```
>>> mcpi(10) ; mcpi(10) ; mcpi(10 ** 3) ; mcpi(10 ** 6)
3.6
2.4
3.144
3.14164
```

`incircle += 1` in line 6 is an example of an in-place assignment; it does the same as `incircle = incircle + 1`. This works for all binary operations in Python, not just `+`: if `o` is a binary operation applicable to the values of `x` and `v`, where `x` is a variable (`v` may be a variable, a function call, a literal such as `42`, `'a literal string'`, etc.), then `x o= v` is equivalent to `x = x o v`. So for example,

```
>>> b = 3; b **= 2; b
9
```

because `b **= 2` is equivalent to `b = b ** 2`. Similarly:

```
>>> b = 14; b %= 3; b
2
```

2. Some useful details

We've already learned enough Python to be able to use it for writing simple programs. Here we collect (in no particular order) some basic features of the language that are useful but would've only served to distract us when we made our first steps.

Simple types. Some "simple" types (of which we have already encountered two) are listed in Table 1. Complex literals can be written as `a+bj`, where `a` and `b` are `int` or `float` literals.

```
Python 3.12.9 (main, Feb  4 2025, 00:00:00) [GCC 14.2.1 20240912 (Red Hat 14.2.
Type "help", "copyright", "credits" or "license" for more information.
>>> (1+1j)**2
2j
```

`j` in itself is just a variable name:

```
>>> j**2 == -1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'j' is not defined
>>> 1j**2 == -1
True
```

But if we want complex numbers, we probably want Sage (or SymPy), too.

A warning: dividing an `int` with another results in a `float`, even if the second `int` divides the first:

```
>>> 2/2, type(2/2)
(1.0, <class 'float'>)
```

And since the precision of `float`s are limited, this can lead to some unexpected results:

```
>>> 18530201888518410 / 2
9265100944259204.0
```

There are ways to work around such problems. If you know that the result should be an `int`, you can use `//` instead of `/`. If you don't, you can write

```
x//y if x%y == 0 else x/y
```

For example,

```
>>> [x//3 if x%3 == 0 else x/3 for x in range(5,10)]
[1.6666666666666667, 2, 2.3333333333333335, 2.6666666666666665, 3]
```

The function `divmod()`, which returns the pair `(x//y, x%y)`, can also be useful in this context.

Finally, we have seen `None`, the value that functions which don't return a value return. (It has other uses, too). It is of type `NoneType`.

Complex types. A not so simple type is `str`ing. We've met it already, and will learn more about it in Section 4; for the time being it's enough to know that a string literal is whatever is written between quotation marks of various kinds, most importantly `'` and `"`. Another important "complex" type is `list`, but what we have learned about lists will be enough for us for a while, assuming we haven't forgotten about the `.append()` method (see 1.3 and 1.4), which is often used for accumulating objects in a list. Finally, there are tuples which we've seen in connection with variable unpacking on page 15.

There are more to come.

Type	**Description**
`int`	integer
`float`	floating point number
`complex`	complex number
`bool`	boolean (`True` and `False`)
`NoneType`	`None` (null value)

TABLE 1. Some simple types

Generalized booleans. The condition in an `if` or `while` statement or in an `if` expression is usually a `bool`ean, but can be of other type, too, as shown in the following examples:

```
>>> if 0: print('Yes')
...
>>> if 10: print('Yes')
...
Yes
>>> if []: print('Yes')
...
>>> if [0]: print('Yes')
...
Yes
>>> if '': print('Yes')
...
>>> if 'nonempty string': print('Yes')
...
Yes
>>> if None: print('Yes')
...
```

More on `if`. We know everything there is to know about the `if` statement, except that it can optionally have one or more `elif` clauses:

```
if cond_1:
   do_this_1
   ...
elif cond_2:
   do_this_2
   ...
else:
   do_this_3
   ...
do_this_no_matter_what
```

does the same as, but is more compact than

```
if cond_1:
   do_this_1
   ...
else:
   if cond_2:
       do_this_2
       ...
```

```
    else:
        do_this_3
        ...
do_this_no_matter_what
```

For example:

```
>>> for i in range(-5,26,10): #i.e. [-5,5,15,25]:
...     if i<0:
...         print('negative')
...     elif i<=10:
...         print('small')
...     elif i<=20:
...         print('medium')
...     else:
...         print('big')
...
negative
small
medium
big
```

Exercise 2.1. Do the same without `elif`!

Binary operators and relations. See Table 2 for an (incomplete) list of binary operators. One noteworthy change from Sage is that ^ no longer means exponentiation.

For binary relations `R` and `S`, `x R y S z` means `x R y and y S z`; for example

```
>>> 3 <= 4 > 2 < 10
True
```

This lets us get away with fewer `and`s.

More on loops. Both `for` and `while` loops have extended versions that are useful when `break` is used in the body. Here's the syntax for `for` loops, but it's similar for `while`:

```
for variable in iterable:
    do_this
    ...
    do_that
else:
    do_this_too
    ...
    do_that_too
```

This works as before (the body (`do_this ... do_that`) will be executed with *variable* bound to successive values of the *iterable*), but the body of the `else` clause will only be executed if the loop terminates normally, not by executing a `break`. (The `while` loop's `else` clause behaves similarly.) Here's an example where this is useful. Suppose we have a `list` of numbers and we want to print the first number that is divisible by 7, or print that there is no such number.

```
numbers = [2, 5, 10, 14, 21, 35, 42, 51]

for n in numbers:
    if n % 7 == 0:
        print(n, "is divisible by 7")
        break
else:
    print("No number in the list is divisible by 7")
```

```
14 is divisible by 7
```

We can't just write the last `print()` statement after the loop, because then it would be executed even if there were a member of the `list` that is divisible by 7. Without an `else` clause we'd have to keep track of what's going on with an extra variable:

```
numbers = [2, 5, 10, 14, 21, 35, 42, 51]
broke = False

for n in numbers:
    if n % 7 == 0:
        print(n, "is divisible by 7")
        broke = True
```

Operators and relations	**Description**
`or`	boolean or
`and`	boolean and
`not`	boolean not
`in`, `not in`	membership
`is`, `is not`	identity test
`<`, `<=`, `>`, `>=`, `==`, `!=`	comparison
`+`, `-`	addition, subtraction
`*`, `/`, `//`, `%`	multiplication, division, truncating division, remainder
`**`	exponentiation

TABLE 2. Binary operators and relations

```
        break
if not(broke):
    print("No number in the list is divisible by 7")
```

```
14 is divisible by 7
```

This is not only ugly, but error prone, for what if this whole piece of code is part of a larger one that also happens to use a variable named `broke`? But in cases like this, we may be better off writing and calling a function, and use `return` instead of `break`. This will also prevent the execution of commands after the loop when they shouldn't be executed.

Exercise 2.2. Write a function `first_divisible(numbers,d)` that prints the first number in the list of numbers `numbers` that is divisible by d, or prints that there is no such number. For example,

```
>>> nums = [2, 5, 10, 14, 21, 35, 42, 51]
>>> first_divisible(nums, 7)
14 is divisible by 7

>>> first_divisible(nums, 13)
No number in the list is divisible by 13
```

Don't use `break` and `else`.

It's quite common that we want to iterate over something but also keep track of where we are. To help with this, Python provides `enumerate()`, which can be used like this:

```
>>> for index, number in enumerate(range(10,15)):
...     print(index, number)
...
0 10
1 11
2 12
3 13
4 14
```

This is much more concise than what we would have to write otherwise:

```
index = 0
for number in range(10,15):
    print(index, number)
    index += 1
```

With an optional argument, `enumerate()` can start counting from integers other than 0:

```
>>> for index, number in enumerate(range(97,107),1):
...     print(index,chr(number))
```

```
...
1 a
2 b
3 c
4 d
5 e
6 f
7 g
8 h
9 i
10 j
```

It may seem strange that we have *two* loop variables, `index` and `number`. But this is just a case of variable unpacking, something we have seen on page 15. What happens here is that `enumerate()` returns collection of pairs:

```
>>> list(enumerate(range(10,15)))
[(0, 10), (1, 11), (2, 12), (3, 13), (4, 14)]
```

so on each round we get a pair whose first member is assigned to the variable `index`, and whose second member is assigned to the variable `number`.

3. I/O

One could go quite far with what we've learned so far. But one area where we'd soon feel constrained is the ability to provide data for our programs and to display their output in a usable form. This, especially the lack of our ability to utilize various data sources, is often fine for mathematically oriented (for example, typical Sage) programs, but not for real-life Python programs.

3.1. **User input and simple output.**

Asking for keyboard input from the user is the simplest way to get a small amount of data from the outside our programs (not known at the time they're written) with which to work. For a more substantial amount, it's reading from a text file, which will be covered in the next subsection[8].

The `input()` function returns whatever the user types until she presses RETURN, as a string. Here's an example:

```
x = int(input())
print(2*x)
```

[8]other methods include reading from a network socket or a database

This will wait until you enter a number, and will then print its double. The call to `int()` is there to convert the string representation (say `'9'`) of the input to an integer (`9`, in this case).

To make it more usable, we should call `input()` with the optional argument `prompt`, which will be displayed to the user. Try this version:

```
x = int(input('Enter an integer: '))
print (2*x)
```

With this, the user will see that there is something to be done.

Exercise 3.1. Write a program that asks for numbers, one after the other, and if the user presses RETURN without entering a number first, it returns the average of the numbers. So an interaction with your program should look something like this:

```
Enter a number: 1
Enter a number: 4
Enter a number: 12
Enter a number: 3
Enter a number:
The average is 5.0
```

For more complex inputs you may have to preprocess the string in other ways to make it usable for your program. For example, if we want a list of integers, we do this:

```
input_str = input('Please input a list of integers separated by spaces: ')
l = [int(i) for i in input_str.split()]
```

What the `.split()` method does here is to return the list of the words (substrings separated by whitespace) in the string. For example:

```
Python 3.12.9 (main, Feb  4 2025, 00:00:00) [GCC 14.2.1 20240912 (Red Hat 14.2.
Type "help", "copyright", "credits" or "license" for more information.
>>> '12  23 42     135  '.split()
['12', '23', '42', '135']
```

Returning to our doubling example: it's OK to return the double of the integer the user typed in, but it's better to tell her what the answer is an answer to, as in

```
The double of the number 21 is 42.
```

This will almost do that:

```
x = int(input('Enter an integer: '))
print ('The double of the number', x, 'is', 2*x, '.')
```

That's because, as we have seen before, `print()` accepts any number of arguments, and prints them, by default, separated by a space. For this reason the output will not exactly be what we wanted, but this:

```
The double of the number 21 is 42 .
```

There are other (arguably better) ways of circumventing this problem (see for example the discussion of f-strings later), but a simple one is to use the `sep` keyword argument to `print()`. This determines what gets printed between the various arguments, and is a space by default.

```
x = int(input('Enter an integer: '))
print ('The double of the number ', x, ' is ', 2*x, '.', sep='')
```

The value of `sep` can be any string. There is another useful keyword argument of `print()`, `end`, newline (`'\n'`) by default, which determines what is printed after all the arguments are. For example:

```
>>> for i in range(5): print(i,end='|')
...
0|1|2|3|4|
```

An ugly but simple workaround for getting rid of the last `sep` is to delete it after the loop with a `print('\b')`. `'\b'` is the backspace *backslash escape sequence*, just as `'\n'` is newline and `'\t'` is tabulator. It's best to avoid these, except possibly `'\n'`.

For our problem above,

```
>>> print('|'.join([str(i) for i in range(5)]))
0|1|2|3|4
```

is a better solution. This works because if `s` is a string and `l` is a list of strings, then `s.join(l)` is the concatenation of the strings in the list separated by `s`. For example,

```
>>> '<>'.join(['this', 'looks', 'strange'])
'this<>looks<>strange'
```

Exercise 3.2. Write a program that asks for numbers separated by spaces, and prints their average. So an interaction with your program should look something like this:

```
>>> Enter some numbers separated by spaces: 1 4 13 2
The average is  5.0
>>>
```

3.2. **Reading and writing files.** Suppose that

```
$ cat data.txt
one
two
three
very long
four
five
```

We can get the contents of `data.txt` from Python this way:

```
>>> with open('data.txt') as file:
...     for line in file:
...         print(line.rstrip())
...
one
two
three
very long
four
five
```

`with` opens a new block (see the colon at the end of the line and the indentation of the next lines); since it is followed by

```
open('data.txt') as file
```

what it does is open the file named `data.txt` in the current directory for reading, and assigns an iterable of the lines of the file to the variable `file`. The `.rstrip()` is there only to strip whitespace and newline from the end of each line. Try it without `.rstrip()` to see the difference!

To make us feel that our program actually does something, we may want to prepend each line with its line number in the output.

```
>>> with open('data.txt') as file:
...     for i, line in enumerate(file,1):
...         print(str(i)+': '+line.rstrip())
...
1: one
2: two
3: three
4: very long
5: four
6: five
```

(We've met `enumerate()` on page 24.)

There is a more primitive way to open a file (for reading or writing), namely with the function `open()`; so our first program above could've been written like this:

```
file = open('data.txt')
for line in file:
    print(line.strip())
file.close()
```

but the `with` construction guarantees that our file will be closed (there's no need to invoke the method `.close()`) once control leaves its body.

This is particularly important when writing files, because closing a file opened for writing ensures that all data sent to it is actually written to it. As an example of writing to a file, let's write to `out.txt` the lines of `data.txt` in reverse order:

```
>>> lines = []

>>> with open('data.txt') as file:
...     for line in file:
...         lines.append(line.rstrip())
...
>>> with open ('out.txt','wt') as out:
...     for line in lines[::-1]:
...         print(line,file=out)
...
>>> #we check the result by escaping back to the shell
>>> import os
>>> print(os.popen('cat out.txt').read())
five
four
very long
three
two
one
```

Here we opened the the file `out.txt` for writing in text mode; that's what `wt` means in the second argument to `open()`. Some important other possibilities are `rt` (read in text mode — the default), `rb` (read in binary mode) and `wb` (write in binary mode).

Another novelty here is that `print()` has a keyword argument `file`, which can be an open (for writing) file; in that case `print()` writes there instead of the standard output.

A shorter way to achieve the same result is this:

```
>>> with open('data.txt') as file:
...     with open ('out.txt','wt') as out:
...         for line in reversed(list(file)):
...             print(line.rstrip(),file=out)
...
```

This works because `list()` creates a list from an iterable, which `reversed()` can reverse. Instead of `reversed(list(file))` we could have written `list(file)[::-1]`.

4. Containers

4.1. `list`s, `tuple`s and `str`ings. We've encountered plenty of lists already, but there's a lot that can be done with them that we haven't covered yet. A subset of these are applicable to tuples and strings, too. This is not surprising, considering that both tuples and strings are a bit like lists, in that all of them are mappings from a proper initial segment of the natural numbers into all Python objects (in the case of lists and tuples) and characters (in the case of strings). That is, they are all *sequences*[9]. The main difference between lists and tuples is that tuples are immutable.

```
Python 3.12.9 (main, Feb  4 2025, 00:00:00) [GCC 14.2.1 20240912 (Red Hat 14.2.
Type "help", "copyright", "credits" or "license" for more information.
>>> l = list(range(0,10,2)) ; l  # one way of creating a list
[0, 2, 4, 6, 8]

>>> t = tuple(l) ; t  # one way of creating a tuple
(0, 2, 4, 6, 8)

>>> l[3]
6

>>> l[3] = 5 ; l[3]
5

>>> t[3]
6

>>> t[3] = 5  #not going to work
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> s = 'abcdef ghijk' ; s
'abcdef ghijk'

>>> s[3]
'd'

>>> s[3] = 'q'  #not going to work either
Traceback (most recent call last):
```

[9]as are `range`s, which are the return values of the function `range()`

`s[i]`	Element `i` of `s`
`s[i:j]`	A slice of `s`
`s[i:j:stride]`	An extended slice of `s`
`len(s)`	Length of `s`
`min(s)`	Minimum value in `s`
`max(s)`	Maximum value in `s`
`sum(s [,initial])`	Sum of items in `s` (not applicable to strings – use `.join()`)
`all(s)`	`True` iff all items in `s` are `True`
`any(s)`	`True` iff there is an item in `s` that is `True`
`x in s`	`True` iff `x` is a member of `s`

TABLE 3. Operations and functions on sequences

`s[i] = v`	Item assignment
`s[i:j] = v`	Slice assignment
`s[i:j:stride] = v`	Extended slice assignment
`del s[i]`	Item deletion
`del s[i:j]`	Slice deletion
`del s[i:j:stride]`	Extended slice deletion

TABLE 4. Operations applicable to lists

```
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

But other than this, everything that we have learned so far about lists (mostly various methods of indexing them) applies to tuples and strings, too. So do the following, which are new:

```
>>> l + l, t + t
([0, 2, 4, 5, 8, 0, 2, 4, 5, 8], (0, 2, 4, 6, 8, 0, 2, 4, 6, 8))
>>> l*2, t*2
([0, 2, 4, 5, 8, 0, 2, 4, 5, 8], (0, 2, 4, 6, 8, 0, 2, 4, 6, 8))
>>> s + s, 3*s
('abcdef ghijkabcdef ghijk', 'abcdef ghijkabcdef ghijkabcdef ghijk')
```

Exercise 4.1.* Why do you think the following works? Doesn't this contradict the fact that tuples are immutable?

```
>>> tup = ([0,1],[2]) ; tup[0][1] = 'a' ; tup
([0, 'a'], [2])
```

Hint: try it in Pythontutor!

Here are the list of methods applicable to lists, and to tuples (courtesy of IPython's `TAB`-completion[10]):

```
list.
  append()  count()   insert()  reverse()
  clear()   extend()  pop()     sort()
  copy()    index()   remove()

tuple.
  count() index()
```

Some of the list methods are understandably missing from tuples: for example, `l.sort()` sorts the list `l` *in place*, so, unless `l` is already sorted, it must do "item assignment" (to use the terminology of the error message above). [11] The same holds for `.reverse()`:

```
>>> l.reverse() ; l
[8, 5, 4, 2, 0]
>>> l.sort(); l
[0, 2, 4, 5, 8]
```

which doesn't mean we can't easily reverse a tuple or a string:

```
>>> t[::-1]
(8, 6, 4, 2, 0)
>>> s[::-1]
'kjihg fedcba'
```

but, unlike `reverse()`, this of course doesn't change the tuple or string itself.[12] `l.append(obj)`, as we have already seen, appends `obj` to the end of the list `l`, and `l.extend(obs)` extends `l` with the members of the iterable (list, tuple, string, ...) `obs`. This example should make clear the difference between the two:

[10] An alternative way of obtaining a list of them is `dir(list)`, or `dir(l)`, where `l` is a list. So for example `dir([])` works, too. If you want them together with their documentation, enter `help(list)` (or `help(l)` if `l` is a list) at the interpreter's prompt or a Jupyter notebook cell. But with either of these techniques (which of course work for other types, too), ignore the methods whose name starts with an underscore (_) character. We will see later why.

[11] There is a `sorted()` *function*, applicable to both lists, tuples and strings (and in fact, any iterable, and, in particular, any sequence). But whatever the type of its argument, it returns a list. `.reverse()` also has a function counterpart, `reversed()`, but it returns an iterable (technically, an *iterator*) that is not a list.

[12] We can write `t = t[::-1]`, thereby changing the value of `t` to a tuple that has the same members as `t`'s original value, but in reverse order — but this is nevertheless a *new* tuple under an old name (i.e., assigned to the same variable that held the original tuple). Try `print(id(l)) ; l.reverse() ; print(id(l))` and `print(id(t)) ; t = t[::-1] ; print(id(t))` to see the difference!

```
>>> l
[0, 2, 4, 5, 8]
>>> l.append(['a','b','c']) ;l
[0, 2, 4, 5, 8, ['a', 'b', 'c']]
>>> l.extend(['a','b','c']) ;l
[0, 2, 4, 5, 8, ['a', 'b', 'c'], 'a', 'b', 'c']
```

`.extend()` differs from `+` (concatenation) in two respects: first, `l.extend(obs)` modifies `l`, it doesn't create a new list, unlike concatenation. (That is why neither tuples, nor strings have this method.) And second, in `l.extend(obs)`, `obs` can be any iterable, not just a list, while the arguments of `+` must be of the same type.

```
>>> l = l[:5] ; l.extend('def') ; l
[0, 2, 4, 5, 8, 'd', 'e', 'f']
```

but

```
>>> l + 'def'    #not going to work
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "str") to list
```

Finally, the `index()` method returns the index of the first occurrence of its argument in the list, tuple or string (and throws a `ValueError` if it's not a member of the sequence):

```
>>> [2,0,1,'a',1,0].index(1)
2
```

And `count()` returns the number of occurrences of its argument in the sequence:

```
>>> [2,0,1,'a',1,0].count(1)
2
```

Although it is perfectly fine to access members of a tuple by indexing it, as in, say, `t[2]`, it's more common to access them by variable unpacking (see page 15):

```
>>> t
(0, 2, 4, 6, 8)
>>> a, _, c, _, e = t
>>> c, a
(4, 0)
```

(This works, but is used less with other kinds of sequences, too.) The underscore signals that we're not interested in (that is, don't want to assign to a variable) the corresponding value.

Variable unpacking features in a very common pattern: when traversing some iterable which consists of tuples. We've seen an example of this with `enumerate()` in Section 3.2. Here's an other: suppose we have three lists, the first containg names of goods, the second containing the corresponding unit prices, and the third the amounts stocked, and our task is to produce a list with (good, total value) pairs.

```
>>> goods = ['ball', 'table', 'racket', 'net']

>>> amounts = [570, 3, 12, 17]

>>> uprices = [0.13, 2000, 185, 23]

>>> [(good, a*up) for good, a, up in zip(goods, amounts, uprices)]
[('ball', 74.10000000000001), ('table', 6000), ('racket', 2220), ('net', 391)]
```

What's new here is `zip()`, which, according to the documentation:

> returns an iterable of *n*-length tuples, where *n* is the number of iterables passed as positional arguments to `zip()`. The *i*-th element in every tuple comes from the *i*-th iterable argument to `zip()`. This continues until the shortest argument is exhausted.

For example:

```
>>> list(zip([1, 2, 3, 4], ['a', 'b', 'c']))
[(1, 'a'), (2, 'b'), (3, 'c')]
```

or, in the example above,

```
>>> list(zip(goods, amounts, uprices))
[('ball', 570, 0.13), ('table', 3, 2000), ('racket', 12, 185), ('net', 17, 23)]
```

Very useful if we want to iterate parallel over more than one iterable.

Tuples can be written as literals, the same way as lists, but enclosed in parentheses instead of brackets:

```
>>> type((1,2,3))
<class 'tuple'>
```

In fact, it's the comma that is important, not the parentheses:

```
>>> x = 1,2,3 #we've been using this all the time
>>> x
(1, 2, 3)
>>> type(x)
<class 'tuple'>
```

There is a quirk though: for a tuple of length one, we need to signal to Python that it is a tuple, by writing a comma after its only member (since any expression can be surrounded by parentheses, they don't help here):

```
>>> (1) == 1
True
>>> type((1)), type((1,))
(<class 'int'>, <class 'tuple'>)
```

This is an ugly corner case which one should be aware of but probably never going to encounter.

Finally, a last word about variable unpacking: what if we don't know in advance the length of the tuple on the right hand side of an assignment? We should be able to indicate that some variable is there to receive all the members that don't get assigned to other variables. And "all the members" can only mean "some kind of collection of all the members". This can be done with a $*$ preceding the name of the variable, and the "kind of collection" is always a list:

```
>>> u,_,v,*w = list(range(10))
>>> u,w
(0, [3, 4, 5, 6, 7, 8, 9])

>>> _,_,v,*w,u = tuple(range(10))
>>> u,w
(9, [3, 4, 5, 6, 7, 8])

>>> u,_,v,*w = 'what goes where'
>>> u,w
('w', ['t', ' ', 'g', 'o', 'e', 's', ' ', 'w', 'h', 'e', 'r', 'e'])
```

We will see something similar when we learn about functions with variable number of arguments in §5.

4.1.1. *More on strings.* String literals can be written in four different ways:

```
>>> 'ab' == "ab" == """ab""" == '''ab'''
True
```

Each have their uses. For example,

```
>>> print("No I don't") #no need to escape '
No I don't
>>> print('"Yes," he said') #no need to escape "
"Yes," he said
>>> #not going to work:
>>> print("This is
  File "<stdin>", line 1
    print("This is
          ^
SyntaxError: unterminated string literal (detected at line 1)
```

```
>>> too long to fit in one line")
  File "<stdin>", line 1
    too long to fit in one line")
                                ^
SyntaxError: unterminated string literal (detected at line 1)
>>> #use this instead:
>>> print("This is \
... too long to fit in one line")
This is too long to fit in one line

>>> print("""This is
... too long to fit in one line""")   #handy for multiline strings
This is
too long to fit in one line
>>> print("This is \ntoo long to fit in one line") #but not strictly necessary
This is
too long to fit in one line
```

Methods applicable to strings:

```
str.
  capitalize()   encode()       format()       isalpha()
  casefold()     endswith()     format_map()   isascii()
  center()       expandtabs()   index()        isdecimal()      >
  count()        find()         isalnum()      isdigit()

  isidentifier() isspace()      ljust()        partition()
  islower()      istitle()      lower()        removeprefix()
< isnumeric()    isupper()      lstrip()       removesuffix()   >
  isprintable()  join()         maketrans()    replace()

  rfind()        rsplit()       startswith()   translate()
  rindex()       rstrip()       strip()        upper()
< rjust()        split()        swapcase()     zfill()
  rpartition()   splitlines()   title()
```

We have already met `.join()` in section 3.1 and `.rstrip()` in Section 3.2.

Some of the above methods are particularly useful. For example, `.replace()` replaces (by default, all) occurrences of a substring with another.

```
>>> s = "you think you can do it"
>>> s.replace("you","we")
'we think we can do it'
>>> s.replace("you","we",1)
'we think you can do it'
```

The third (optional) argument specifies the number of occurrences `.replace()` should replace at most. For more complex replacements, *regular expressions* should be used.

Another useful method is `.split()`, which we have met already in section 3.1. With no arguments, it splits the string into a list of words:

```
>>> s.split()
['you', 'think', 'you', 'can', 'do', 'it']
```

But with an argument, which is a string, it considers that string as the boundary of "words":

```
>>> s.split("ou")
['y', ' think y', ' can do it']
>>> [w.strip() for w in s.split("ou")]
['y', 'think y', 'can do it']
```

The method `.strip()` that was used here is the symmetric version of `.rstrip()`: with no argument, it strips the whitespace from each end of the string it is called on.

We have already met the `str` *function*, which usually returns a string representation of an object.

```
>>> 2**3+1
9
>>> str(2**3)+str(1)
'81'
```

Conversely, the function `int()` can be used to turn a string representation of an integer into an `int`, and the corresponding function for floating point numbers is `float()`[13].

```
>>> int("2")**int("3")+int("1")
9
>>> float("1.4142135623730951")**2
2.0000000000000004
```

If `int()` or `float()` cannot parse its string argument into an `int` or a `float`, it will throw a `ValueError`.

```
>>> int("nine")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'nine'
```

We'll see in Section 4.2.1 how we can deal with this situation. But how we *should* is a harder question and the answer depends on the context. (For example: should we silently return 0? Or 1? Or 42? Or return

[13]Both `int` and `float()` have other uses, too.

some number but warn the user? Or give her a chance to specify another number? Or just let her face the error?)

Being able to move between an object, a number, for example, and its string representation is important, among other things because text (file or message) is a very popular medium of communication. For example, every spreadsheet program (MS Excel, Openoffice Calc) can export and import sheets (tables) in text format (we'll meet this format, CSV, below). And the main protocol that the web uses (the Hypertext Transfer Protocol (HTTP)), or the popular JSON data interchange format, is text based. And since we often want to send/receive not just genuine text, such as names or newspaper articles, but also numbers, it's important to have functions that can recover a number from its text representation.[14]

f-strings. A piece of information one wants to display is almost always part constant, part variable. For example, even though the result a long computation may be 42, it's never a good idea to print just a number. Printing something like the following

```
The answer is: 42
Please enter your question:
```

is much more useful. This is especially true when the result consists of more numbers (and/or other types of data). For example, suppose we have a little "database" of goods, their amounts and unit prices.

```
>>> db = [
...     ('ball', 570, 0.13),
...     ('table', 3, 2000),
...     ('racket', 12, 185),
...     ('net', 17, 23)
... ]
```

If we want to present it, or something derived from it, as we did earlier:

```
>>> [(good, a*up) for good, a, up in db]
[('ball', 74.10000000000001), ('table', 6000), ('racket', 2220), ('net', 391)]
```

labeling the various items displayed is a mimimum requirement, unless we are the only user of our program. At the very least, we want something like this:

```
Name: ball
Total price: 74.10000000000001
Name: table
Total price: 6000
```

[14]Of course, a newspaper article may also contain numbers, but it's safe to treat them as text, because we usually don't need to *use* them as numbers, e.g. square them.

```
Name: racket
Total price: 2220
Name: net
Total price: 391
```

Every line here consist of two parts: a constant string, for example `Name:` and a variable (string representation of a) number, such as `6000`. We've encountered this problem already, and solved this by converting everything to string (with the function `str()`) and concatenating the results with `+`. But f-strings (*formatted string literals*) are much better suited to this task: they are strings with "holes" (the official name is *replacement fields*) in them, which are Python expressions enclosed in braces. What's inside these holes get evaluated at the time of printing. For example:

```
>>> f'1+1 = {1+1}'
'1+1 = 2'
>>> f'The first record of our database is {db[0]}'
"The first record of our database is ('ball', 570, 0.13)"
```

The `f` signifies that the string that follows isn't just any constant string, because Python code may be found in it between braces. With this, we can present our database in the desired form:

```
>>> for good, a, up in db:
...     print(f'Name: {good}\nTotal price: {a*up}')
...
Name: ball
Total price: 74.10000000000001
Name: table
Total price: 6000
Name: racket
Total price: 2220
Name: net
Total price: 391
```

There's much more to f-strings. Among other things, we have more control over how the data within braces is presented. For example, if we're bothered by the lots of decimals, we can write `{a*up:.2f}` in place of `{a*up}` (the part after the colon is called a *format specifier*) and this will ensure that the number will be written as a `float` with exactly 2 decimal places.

```
>>> for good, a, up in db:
...     print(f'Name: {good}\nTotal price: {a*up:.2f}')
...
Name: ball
```

```
Total price: 74.10
Name: table
Total price: 6000.00
Name: racket
Total price: 2220.00
Name: net
Total price: 391.00
```

We can also declare the width of a replacement field, which is useful for presenting data in tabular form[15]:

```
>>> def doit():
...     print(f'{"Name":20}Total price')
...     for good, a, up in db:
...         print(f'{good:20}{a*up:10.2f}')
...
>>> doit()
Name                Total price
ball                     74.10
table                  6000.00
racket                 2220.00
net                     391.00
```

What's new here is that `"Name":20` and `good:20` ensure that `"Name"` and `good` are printed in a column of width 20 characters, and because of `a*up:10.2f`, `a*up` is printed in a column of width 10 characters, right aligned, because it is a numeric field. We could've forced it to be left aligned with `a*up:<10.2f` and centered with `a*up:^10.2f`. Here are some more examples:

```
>>> ans = 42
>>> print(f'|{ans:7d}|'); print('|1234567|') # 'd' stands for 'decimal'
|     42|
|1234567|
>>> print(f'|{ans:<7d}|'); print('|1234567|')
|42     |
|1234567|
>>> print(f'|{ans:^7d}|'); print('|1234567|')
|  42   |
|1234567|
>>> print(f'|{ans:07d}|'); print('|1234567|') #padding by '0'
|0000042|
|1234567|
```

[15] There's no reason to define a function for this; I did it here for LaTeXnical reasons.

```
>>> print(f'|{ans:7b}|'); print('|1234567|') # 'b' stands for 'binary'
| 101010|
|1234567|
>>> print(f'|{ans:7.2f}|'); print('|1234567|')
|  42.00|
|1234567|
>>> print(f'|{ans:07.2f}|'); print('|1234567|')
|0042.00|
|1234567|
```

The official documentation of f-strings can be found here.

4.2. `dict`**s.** Like lists and tuples, `dict`ionaries hold a collection of objects, but unlike them, these objects are not indexed by natural numbers, but by keys (strings, numbers, tuples, ...), just like in real world dictionaries. So in the example above with goods and their values, it would make more sense to put the result of our computation in a dictionary, and not in a list of tuples, because then the total value of balls could be accessed simply by `total_values['ball']`. For this reason, we'll soon redo that example with a dictionary (and dictionary comprehension) in place of list and list comprehension.

But let's first see the basics of `dict`s.

```
>>> d = dict()
>>> type(d)
<class 'dict'>
>>> d
{}
>>> d['one']=1 ; d['two']=2; d
{'one': 1, 'two': 2}
>>> d['two']
2
>>> d.get('two')
2
>>> 'two' in d, 'three' in d
(True, False)
>>> d['three'] #not going to work
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'three'
>>> None == d.get('three')
True
>>> d.get('three',"can't find it")
"can't find it"
```

```
>>> d.get('two',"can't find it")
2
```

As you can see, the method `.get()` has a second, optional argument, `None` by default, which is returned in case the key (its first argument) is not present in the `dict`.

Other methods applicable to `dict`s, again, courtesy of IPython's `Tab`-completion:

```
dict.
 clear()      get()       pop()        update()
 copy()       items()     popitem()    values()
 fromkeys()   keys()      setdefault()
```

For example, `items()` and `values()` help iterating over the contents of a `dict`:

```
>>> #iterating over the keys
>>> for key in d:
...     print(key)
...
one
two
>>> #iterating over the values
>>> for val in d.values():
...     print(val)
...
1
2
>>> #iterating over the key-value pairs
>>> for key,val in d.items():
...     print(key, val)
...
one 1
two 2
```

Just like lists and tuples, `dict`s can be written as literals, and in exactly the same way they're printed. So `d` above could've been defined with `d = {'one': 1, 'two': 2}`.

Now we can come back to our inventory example.

```
>>> db
[('ball', 570, 0.13), ('table', 3, 2000), ('racket', 12, 185), ('net', 17, 23)]

>>> total_values = {good: a*up for good, a, up in db}
>>> total_values
```

```
{'ball': 74.10000000000001, 'table': 6000, 'racket': 2220, 'net': 391}
>>> total_values['ball']
74.10000000000001
```

This was a case of dictionary comprehension, which is very much like list comprehension, with parentheses replaced by braces, and which collects "key:value" pairs and not just any objects. Here's an other example of dictionary comprehension:

```
>>> words = 'some words are longer then others'.split(); words
['some', 'words', 'are', 'longer', 'then', 'others']
>>> lengths = {word:len(word) for word in words}; lengths
{'some': 4, 'words': 5, 'are': 3, 'longer': 6, 'then': 4, 'others': 6}
>>> lengths['are']
3
```

4.2.1. *Exceptions.* Programs sometimes run into situations they can't deal with. For example:

```
>>> 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

or

```
>>> 1 / int('two')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'two'
```

It's unlikely that we write `1/0` directly, but perhaps `0` or `'two'` came from user input that wasn't carefully checked. Such an error need not lead to the termination of our programs. We can use the `try: ... except: ...` construction to give us a chance to continue.

```
>>> try:
...     print(1/0)
... except:
...     print("Something is wrong: I assume you wanted 42")
...
Something is wrong: I assume you wanted 42
```

or

```
>>> try:
...     print(1 / int('two'))
... except:
...     print("Something is wrong: I assume you wanted 42")
```

```
...
Something is wrong: I assume you wanted 42
```

while of course

```
>>> try:
...     print(1/2)
... except:
...     print("Something is wrong: I assume you wanted 42")
...
0.5
```

This doesn't solve the problem, just hides it. But we can always give the user a second chance:

```
def divide():
    divisor = int(input("Enter the divisor: "))
    return 1 / divisor

try:
    print(divide())
except:
    print("This didn't work, sorry. Let's try again!")
    print(divide())
```

This is better, but the program (and hence the user) doesn't know what kind of problem it (she) is facing. If it did, perhaps it would take the appropriate measure. What's wrong with the input? Because `except:` catches everything, we can't even be sure that the problem has something to do with the input. Maybe what happened was that the computer ran out of memory, resulting in a `MemoryError`.

But if we look at how Python reported the errors above, before we caught it with `try: ... except: ...`, we see the type of the error (printed in red), and that is not just a name, but an object on which we can discriminate by writing it after `except`.

```
try:
    print(divide())
except ValueError:
    print("I want a NUMBER, not some junk!")
    print(divide())
except ZeroDivisionError:
    print("Can't divide by 0. Perhaps later, in version 2.0.")
    print(divide())
```

or even

```
while True:
    try:
        print(divide())
        break
    except ValueError:
        print("I want a NUMBER, not some junk!")
    except ZeroDivisionError:
        print("Can't divide by 0. Perhaps later, in version 2.0.")
```

to give the user as many chances as possible. Now an interaction with the user may look like this:

```
Enter the divisor: zero
I want a NUMBER, not some junk!
Enter the divisor: 0
Can't divide by 0. Perhaps later, in version 2.0.
Enter the divisor: 5
0.2
```

This is good, and solves the "the error may have a completely different origin" problem, too. For if a third kind of exception occurs, our exception handlers will not catch it, and we won't ask the user to reinput the number, which is good. But we can make this a little more elegant by saying goodbye before bailing out. Here is how:

```
while True:
    try:
        print(divide())
        break
    except ValueError:
        print("I want a NUMBER, not some junk!")
    except ZeroDivisionError:
        print("Can't divide by 0. Perhaps later, in version 2.0.")
    except:
        print("\nI don't know what happened and I can't deal with it. Sorry!")
        raise
```

```
Enter the divisor: two
I want a NUMBER, not some junk!
Enter the divisor:      #here I pressed Ctrl-d which means End-Of-File
I don't know what happened and I can't deal with it.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/tmp/mc.py", line 67, in <module>
  File "/tmp/mc.py", line 62, in divide
EOFError
```

```
>>>
```

Here the final `except` clause is activated if the exception is not of type `ValueError` or `ZeroDivisionError`; and `raise` raises the same exception that got us here. This is good practice because we don't want to hide information from the user (who, though not in this case, may be another part of our program).

For a different kind of example, suppose that we have to administer a popularity contest for actors. Since there is no catalogue of all actors, we need to keep track of the number of votes for an unknown number of people. We can put this data in a dictionary where the keys are the names of the actors who got at least one vote, and the values are the corresponding number of votes. But then we need to do something special (put the new key with value 1 in the dictionary) when an actor is voted for for the first time.

```
>>> votes = {} #empty dictionary

>>> def vote(name):
...     if name in votes:
...         votes[name]+=1
...     else:
...         votes[name]=1
...
>>> vote('Brad Pitt'); vote('Julia Roberts'); vote('Brad Pitt')
>>> votes
{'Brad Pitt': 2, 'Julia Roberts': 1}
```

Alternatively, we can omit checking every time whether a name is in the dictionary already, and instead, only put it in there if its absence leads to an error.

```
>>> votes = {} #empty dictionary

>>> def vote(name):
...     try:
...         votes[name]+=1
...     except KeyError:
...         votes[name]=1
...
>>> for n in ['Brad Pitt', 'Julia Roberts', 'Brad Pitt', 'Julia Roberts', \
... 'Chris Pratt', 'Julia Roberts', 'Michelle Yeoh']:
...     vote(n)
...
>>> for name, n in votes.items():
```

```
...     print(f'{name:20} {n:5d}')
...
Brad Pitt                2
Julia Roberts            3
Chris Pratt              1
Michelle Yeoh            1
```

This has nothing to do with handling exceptions, but if we want to see the two most popular actors, we can use the `.items()` method of `dict`s, which return all key-value pairs as an iterable; we can sort it (in reverse order) according to the number of votes:

```
>>> sorted(votes.items(), reverse = True, key=lambda kv: kv[1])[:2]
[('Julia Roberts', 3), ('Brad Pitt', 2)]
```

The `reverse` keyword argument makes `sorted()` sort in descending order. The `key` keyword argument (this has nothing to do with keys in a `dict`!) of `sorted()` lets us supply a function of one argument, which, given a member of the list to be sorted, returns the value on which the sorting should be based. In our case, it's the number of votes in the (name, number of votes) pair.

If we only care about the most popular actor, we can use the `max()` function (see table 4.1), which can also take a keyword argument `key`:

```
>>> max(votes.items(), key=lambda kv: kv[1])
('Julia Roberts', 3)
```

Remark 4.1. There are some fine points about exceptions that are good to be aware of.

- More than one kind of exception can be dealt with in the same `except` clause: we need to write them as a tuple. For example: `except (ValueError, ZeroDivisionError):`
- The `try: ... except: ...` block may have an `else:` clause, too. It will get executed if the `try:` succeeded.

  Exercise 4.2. What's the point of `else:`? Why not just write

  ```
  try:
      do_something
      do_this_if_all_went_well
  except:
      do_something_else
  ```

  instead of

  ```
  try:
      do_something
  except:
      do_something_else
  ```

```
else:
    do_this_if_all_went_well
```

- There may also be a `finally:` clause, which will get executed no matter what, and in particular, whether an exception occurred in the `try` clause or not.

```
while True:
    try:
        print(divide())
        break
    except ZeroDivisionError:
        print("Can't divide by 0. Perhaps later, in version 2.0.")
    finally:
        print("Finally!")
    print("At last")
```

  Here, if the input is correct, `"At last"` will not be printed (since control leaves `while` because of the `break`), but `"Finally!"` will, because of the "no matter what" rule.

When writing bigger programs it's important to be able to define and raise various exceptions. But there is one way (apart from *re*raising, which we have done before) to raise an exception that can be useful even in the simplest functions. It's done with the `assert` statement, whose first (and only mandatory) argument must be an expression that evaluates to a boolean (or something that can be cast to a boolean, see page 21 in Section 2). When this evaluates to `False`, an `AssertionError` exception is raised, and if the second, optional argument to `assert` is present, it is printed.

```
>>> for n in range(10):
...     assert n % 2 == 0
...     print("Everything is fine so far.")
...
Everything is fine so far.
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
AssertionError
>>> #we could handle it too, but we do this instead:

>>> for n in range(10):
...     assert n % 2 == 0, f"There is a problem: {n} is not even"
...     print("Everything is fine so far.")
...
Everything is fine so far.
```

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
AssertionError: There is a problem: 1 is not even
```

We don't want to handle an `AssertionError` because the goal is not for the program to keep running, but finding out that there is a problem. And in this case `assert`'s second, optional argument (the *assertion message*) is easier to use than a `try: ... except: ...` construct, which would only print out some informative text in the `except AssertionError:` branch anyway.

`assert`-s should not be relied on in a finished program, because checking them can be turned off.

4.3. `set`**s.** This is the last and least important kind of container. Its methods are:

```
add()          difference_update()   isdisjoint()  remove()
clear()        discard()             issubset()    symmetric_difference()
copy()         intersection()        issuperset()  symmetric_difference_update()
difference()   intersection_update() pop()         union()
                                                   update()
```

The syntax for literal sets is writing the set's elements separated by commas between braces:

```
>>> type({1,2,3})
<class 'set'>
```

The `set()` function, given an iterable as argument, also returns a set:

```
>>> set(range(5))
{0, 1, 2, 3, 4}
```

This makes it easy to remove duplicates from a list (provided the order is not important):

```
>>> list(set([1,3,2,3,1]))
[1, 2, 3]
```

But the usefulness of sets is limited by the fact that not every object can be put in a set:

```
>>> set([1,[2,3]])  #doesn't work
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

The error message means that Python cannot encode lists in such a way that identical lists get the same code. (Just think about what would/should happen if a list in a set changed. Should the code change, too?) And that would make it problematic to check whether a list is *in* the set or not. For the same reason, a `set` also cannot be put in a `set`, and neither can it be a key in a `dict`.

The names of most of the important methods speak for themselves. For example:

```
>>> s1 = {1,2,3}; s2 = {2,3,4} ; s1.intersection(s2)
{2, 3}
>>> s1.difference(s2)
{1}
>>> s1
{1, 2, 3}
```

But some don't:

```
>>> s1.difference_update(s2) ; s1
{1}
>>> s1.update(s2) ; s1  # why not union_update()?
{1, 2, 3, 4}
>>> s1.discard(5) ; s1
{1, 2, 3, 4}
>>> s1.discard(2) ; s1
{1, 3, 4}

>>> try:
...     s1.remove(1) ; s1
... except KeyError:
...     "Can't remove what's not there!"
...
{3, 4}
>>> try:
...     s1.remove(1) ; s1
... except KeyError:
...     "Can't remove what's not there!"
...
"Can't remove what's not there!"
```

As a data structure, `set` doesn't offer much over `dict` (with values all set to `None`). But the applicable methods listed above may come in handy.

4.4. **More on list comprehension.** We know that if `l` is a list, then

```
result = [expr for i in l]
```

is equivalent to

```
result = []
for i in l:
    result.append(expr)
```

and

```
result = [expr for x in l if c]
```

is equivalent to

```
result = []
for i in l:
    if c:
        result.append(expr)
```

For example,

```
>>> [i**2 for i in range(20) if i%2 == 1]
[1, 9, 25, 49, 81, 121, 169, 225, 289, 361]
```

But more generally,

```
result = [expr for i1 in l1 if c1
      for i2 in l2 if c2
      ...
      for iN in lN if cN ]
```

is equivalent to

```
result = []
for i1 in l1:
    if c1:
        for i2 in l2:
            if c2:
               ...
                for iN in lN:
                    if cN:
                       result.append(expr)
```

For example,

```
>>> [(i,j) for i in [1,2,3] for j in ['a','b']]
[(1, 'a'), (1, 'b'), (2, 'a'), (2, 'b'), (3, 'a'), (3, 'b')]
```

because it should be read "from left from right" unlike nested list comprehensions:

```
>>> [[(i,j) for i in [1,2,3]] for j in ['a','b']]
[[(1, 'a'), (2, 'a'), (3, 'a')], [(1, 'b'), (2, 'b'), (3, 'b')]]
```

which are read "outside in".

Here's another example that shows that in a list comprehension such as

```
[(i,j) for i in [1,2,3] for j in ['a','b']]
```

the inner loop (here `for j in ...`) the local variable established by the outer loop (`for i in ...`) is available:

```
>>> [j for i in range(3) for j in [range(3),range(3,6),range(6,8)][i]]
[0, 1, 2, 3, 4, 5, 6, 7]
```

It's worth abstracting away the essence of this in a function:

Exercise 4.3. Write a function `concatenate()` that concatenates the lists in the list that it is passed. For example:

```
>>> concatenate([list(range(3)),list(range(3,6)),list(range(6,8))])
[0, 1, 2, 3, 4, 5, 6, 7]
```

The original example with a condition:

```
>>> [(i,j) for i in [1,2,3] if i%2 == 1 for j in ['a','b']]
[(1, 'a'), (1, 'b'), (3, 'a'), (3, 'b')]
```

The concatenating example with multiple conditions:

```
>>> [j for i in range(3) if i%2 == 1
...  for j in [range(3), range(3,10), range(11,15)][i]
...  if j%2==0]
[4, 6, 8]
```

Example 4.1. Suppose that `l` is a list of lists of numbers that are all shorter than `len(l)`. The idea is that `l[i]` is the list of neighbours of `i`. Here is a function `n2(i,l)` that returns the list of all neighbours of neighbours of `i`:

```
>>> def n2(i,l):
...     assert i < len(l)
...     return [k for j in l[i] for k in l[j]]
...
```

and then for example

```
>>> n2(2,[[1,2,3],[1],[0],[2]])
[1, 2, 3]
```

5. Functions

We have been using and defining functions since forever. We know that a function definition looks like this:

```
def fun(par_1,par_2,...):
    statement_1
    statement_2
    ...
```

When a function defined this way is called with `fun(arg_1,arg_2,...)`, what happens is that the arguments `arg_1`, `arg_2`,...get evaluated (so for example, if `arg_1` is another function call, that function is called before `fun`), and then the statements in the body of the definition get evaluated, with `par_1` set to the value of `arg_1`, `par_2` set to the value of `arg_2`, etc. `par_1`, ...are the *parameters* of the function. In the body, these are local variables, so if there is a variable of the same name in the program, it is "shadowed" by the parameter. Its value cannot be seen or changed by the function. Assignment to a variable in the body makes that variable local, too. Here's the example that shows this.

```
Python 3.12.9 (main, Feb  4 2025, 00:00:00) [GCC 14.2.1 20240912 (Red Hat 14.2.
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 1 ; b = 2
>>> def some_fun(a):
...     b = a
...     print(f'''In the body of the function,
... a={a} and b={b} after the assignment''')
...
>>> some_fun(42)
In the body of the function,
a=42 and b=42 after the assignment
>>> a,b
(1, 2)
```

(This is a good place to introduce a neat trick of f-strings, very useful while debugging: instead of writing `expr={expr}`, as in the previous exmample, one can simply write `{expr=}` for any expression `expr`. And this is what we're going to do from now on.)

The global value of `b` is available in the body but only if we don't create a local variable with the same name:

```
>>> a = 1 ; b = 2
>>> def some_fun(a):
...     print(f'In the body of the function {a=} and {b=}')
...
>>> some_fun(42)
In the body of the function a=42 and b=2
>>> a,b
(1, 2)
```

but

```
>>> a = 1 ; b = 2
>>> def some_fun(a):
...     print(f'In the body of the function, {b=} before the assignment')
```

```
...     b = a
...     print(f'''In the body of the function,\
... {a=} and {b=} after the assignment''')
...
>>> some_fun(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in some_fun
UnboundLocalError: cannot access local variable 'b' where it is not associated
```

doesn't work because even if it happens later, Python knows that we have created a local variable `b` (but used it before having assigned a value to it). The fact that we never actually get to touch `b` doesn't change the fact that it's a local variable:

```
>>> a = 1 ; b = 2
>>> def some_fun(a):
...     print(f'In the body of the function, {b=} before the assignment')
...     if False:
...         b = a
...         print(f'''In the body of the function,\
... {a=} and {b=} after the assignment''')
...
>>> some_fun(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in some_fun
UnboundLocalError: cannot access local variable 'b' where it is not associated
```

The statement `b = a`, even if it is guaranteed not to be executed, is the givaway.

If, for some reason, we *did* want to change the value of the global variable `b` in the function, we could do it like this:

```
>>> a = 1 ; b = 2
>>> def some_fun(a):
...     global b
...     b = a
...     print(f'''In the body of the function
... {a=} and {b=} after the assignment''')
...
>>> some_fun(42)
In the body of the function
a=42 and b=42 after the assignment
```

```
>>> a,b
(1, 42)
```

but we shouldn't.

Remark 5.1. There's a way to change (usually inadvertently) the value of a global variable in a different way, as the following example shows. Suppose we want to define a function that sorts a list of numbers using the "bubble sort" algorithm. The idea of this algorithm is that whenever we find a pair of numbers in the wrong order, we reverse it.

```
>>> def bubble(lst):    #bad
...     for i in range(len(lst)):
...         for j in range(len(lst[i+1:])):
...             jj = i+1+j
...             if lst[jj]<lst[i]:
...                 lst[jj], lst[i] = lst[i], lst[jj]
...     return lst
...
>>> import random
>>> l = [random.randint(0,100) for _ in range(10)]
>>> l
[34, 73, 54, 23, 42, 59, 32, 26, 85, 32]
>>> bubble(l)
[23, 26, 32, 32, 34, 42, 54, 59, 73, 85]
```

The return value looks good, but there is a problem:

```
>>> l
[23, 26, 32, 32, 34, 42, 54, 59, 73, 85]
```

The function was not supposed to change its argument.[16] Here's the problem in a simpler context:

```
>>> def side_effect(a):
...     a=42
...
>>> b = 0; side_effect(b); b
0
>>> #so far, so good
>>> def side_effect(a):
...     a[0]=42
...
>>> b = [0]; side_effect(b); b
[42]
```

[16]When the purpose of a function or method is to have a side effect, such as changing its arguments, it's customary in Python for it to not return a value.

What's happened is not that after the call to the function a new object was assigned to `b` – that can't have happened, because there was no `global b` declaration in the body of the function. It's the old object itself, the one that is the value of `b`, that has changed, as shown on the picture. This is because for such complex values as lists (and all other

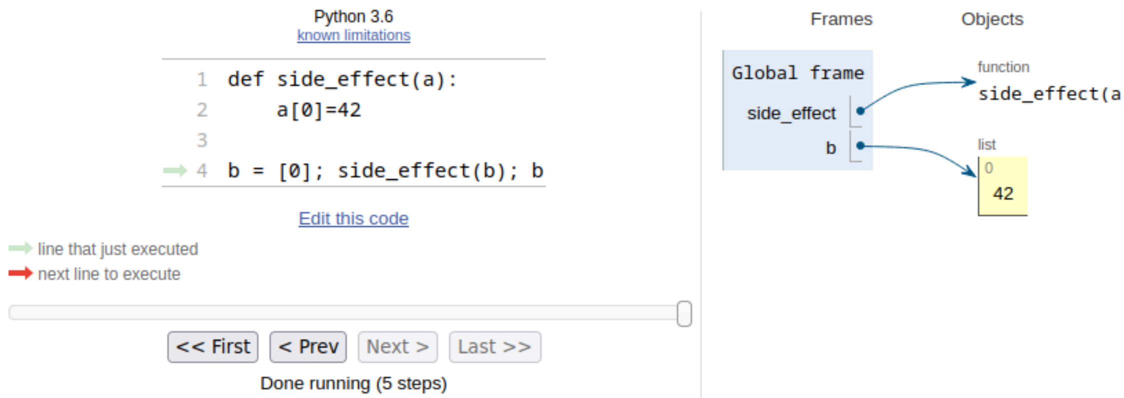


FIGURE 1. Memory (as shown by Pythontutor)

containers, except for strings), when they are assigned to a variable, what the variable contains is not the object itself, but a reference to it (in all likelihood its address in memory). And a list, being a mutable object,

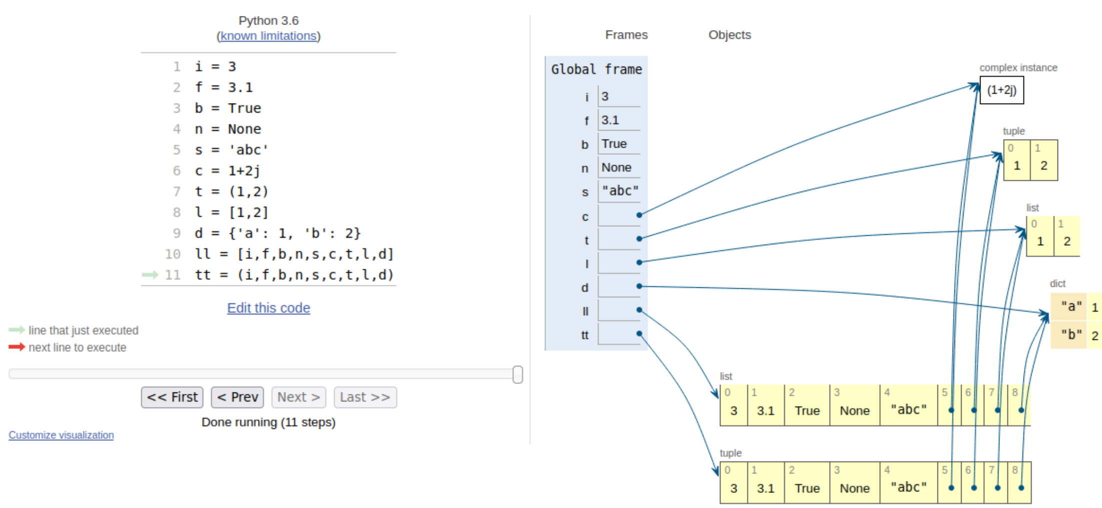


FIGURE 2. Memory (as shown by Pythontutor)

can change without its address having changed. This is what happens to the value of `b` in the example. The way to avoid this problem (and this usually, though not always, *is* a problem) is to make a copy of the complex object and mutate the copy:

```
>>> def side_effect(a):
...     a = a[:] #could use a.copy() or list(a) instead
...     a[0]=42
...
>>> b = [0]; side_effect(b); b
[0]
```

Here the variable `a` that is assigned to in the second line is a new local variable. (We could have given it any other name, but it's a good practice to use the name of the corresponding parameter.) And what is assigned to it is a brand new list.

But this isn't always enough:

```
>>> def side_effect(a):
...     a = a[:]
...     a[0][0]=42
...
>>> b = [[0]]; side_effect(b); b
[[42]]
```

The problem is that `a[:]` and `a.copy()` creates a *shallow copy* of the list. A shallow copy of a list is a new list, but if the original contained a list, then what it really contained is a reference to it, and that reference will be copied into the new list. So the same problem will crop up, only at another level. What we need here is a *deep copy*, which, instead of copying a reference (in any level) creates a new list (which is again a deep copy of the list the reference referred to) and write *that* into the newly created list. Something like this:

```
def deep_copy(l):
    return [deep_copy(i) for i in l] if isinstance(l, list) else l
```

would solve the problem as long as the value we pass into `side_effect` doesn't contain other kinds of complex values buried deep inside.

```
>>> def side_effect(a):
...     a = deep_copy(a)
...     a[0][0]=42
...
>>> b = [[0]]; side_effect(b); b
[[0]]
```

But the real solution is using the `deepcopy()` function from the `copy` module:

```
>>> import copy
>>> def side_effect(a):
...     a = copy.deepcopy(a)
...     a[0][0]=42
...
>>> b = [[0]]; side_effect(b); b
[[0]]
```

This takes care not just of lists but other complex values, too. For our bubble sort function, we don't need a deep copy, a shallow one will do, since the input list contains only numbers.

```
>>> def bubble(l):
...     l = l[:]
...     for i in range(len(l)):
...         for j in range(len(l[i+1:])):
...             jj = i+1+j
...             if l[jj]<l[i]:
...                 l[jj], l[i] = l[i], l[jj]
...     return l
...
>>> l = [random.randint(0,100) for _ in range(10)]
>>> l
[94, 14, 19, 58, 98, 33, 100, 58, 7, 18]
>>> bubble(l)
[7, 14, 18, 19, 33, 58, 58, 94, 98, 100]
>>> l
[94, 14, 19, 58, 98, 33, 100, 58, 7, 18]
```

A final word on the problem of deep vs. shallow copy: it can come up in other situations, not involving function calls, too, as the following example shows:

```
>>> b = [[0]] ; c = b ; c[0][0] = 42; b
[[42]]
```

And the solution is always the same:

```
>>> b = [[0]] ; c = copy.deepcopy(b) ; c[0][0] = 42; b
[[0]]
```

Keyword and optional arguments. In a call

```
>>> f(1,2)
x=1, y=2
```

to a function defined by

```
def f(x,y):
    print(f'{x=}, {y=}')
```

Python knows `1` should be assigned to `x` and `2` to `y` because of their respective positions. That's why these are sometimes called *positional parameters*. But we can be more explicit about what values to assign to which parameter with *keyword arguments*:

```
>>> f(y=2,x=1)
x=1, y=2
```

It is also possible to change the definition of the function so that the caller is forced to use some arguments as keyword arguments. For example,

```
>>> def f(x,*,y):
...     print(f'{x=}, {y=}')
...
>>> f(1,2) #wrong
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() takes 1 positional argument but 2 were given
>>> f(1) #wrong
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() missing 1 required keyword-only argument: 'y'
>>> f(1,y=2) #finally...
x=1, y=2
```

One can provide default values both for positional parameters:

```
>>> def f(x,y=3):
...     print(f'{x=}, {y=}')
...
>>> f(1,2)
x=1, y=2
>>> f(1)
x=1, y=3
```

and for keyword only parameters:

```
>>> def f(x,*,y=3):
...     print(f'{x=}, {y=}')
...
>>> f(1,y=2)
x=1, y=2
>>> f(1)
x=1, y=3
```

What we can't do is put non-default positional parameters after a default parameter:

```
>>> def f(x,y=3,z):   #wrong
  File "<stdin>", line 1
    def f(x,y=3,z):   #wrong
                 ^
SyntaxError: parameter without a default follows parameter with a default
```

This is only logical, for if the function definition started with

```
def f(x,y=3,z):
```

and we called `f` with two arguments (and we should be able to, because one of the three parameters has a default value), how should Python decide which parameter the second value should be assigned to? The parameter `y`, which is in the corresponding position, or `z`, which is a completely ad hoc choice but would result in all parameters getting a value? If you want to mix default and non-default parameters, use keyword arguments:

```
>>> def f(x,*,y=3,z):
...     print(f'{x=}, {y=}, {z=}')
...
>>> f(1,z=2)
x=1, y=3, z=2
```

It is also possible to define functions with variable number of arguments. Of course it doesn't make sense to write "variable number of parameters" in the definition. So we write just one, and mark it with an asterisk.[17] This signals to Python that this parameter should receive all the remaining (positional) arguments as a tuple. ("Remaining", because some might have been assigned to preceeding normal, positional parameters.) Suppose for example that we want to compute the average of an unknown number of numbers. Here's how we can do this:

```
>>> def avg(*nums):
...     return(sum(nums)/len(nums))
...
>>> avg(2,3)
2.5
>>> avg(2,3,5,9,6)
5.0
```

It wouldn't make sense to have two such variadic parameters (which one would get which argument?), but positional parameters can preceed one. (`print()` is an example of a built-in function that has both variadic and keyword arguments. It prints all its non-keyword arguments, using

[17] This resembles variable unpacking with "starred variables" on page 35 in §4.

keyword arguments to decide how.) For example, suppose that sometimes we want to compute the geometric mean ($\sqrt[n]{a_1 a_2 \cdot \dots \cdot a_n}$), not the arithmetic one ($\frac{a_1+a_2+\dots+a_n}{n}$). Then a first, positional argument could receive the type of mean we want computed, and the rest of the arguments are the numbers themselves[18].

```
>>> from functools import reduce

>>> def prod(lst):
...     return reduce(lambda x, y: x*y,lst,1)
...
>>> #or
>>> def prod(lst):
...     res = 1
...     for i in lst: res*=i
...     return res
...
>>> def avg(typ, *nums):
...     return \
...         prod(nums)**(1/len(nums)) if typ == 'g' \
...            else sum(nums)/len(nums)
...
>>> avg('whatever',2,3,5,9,6)
5.0
>>> avg('g',2,3,5,9,6)
4.384327654865777
```

The first argument went into the parameter `typ`, and the rest into `nums`. This is fine for illustrating where arguments go if there are positional

[18] A note about `reduce`: if `f` is a function of two arguments, then for example

```
reduce(f,[a1,a2,a3,a4])
```

returns

```
f(f(f(a1,a2),a3),a4)
```

and if a last, optional argument is present, than that will be placed before the rest of the list (or be returned if the list is empty), so for example

```
reduce(f,[a1,a2,a3,a4],a)
```

returns

```
f(f(f(f(a,a1),a2),a3),a4)
```

parameters preceeding a variadic one, but it's not a very æstetic user interface. Since there is a sensible default here (we'd probably want arithmetic mean most of the time), `typ` should be made into a default parameter. But we can't put the variadic parameter after a default one, so let's do it the other way round!

```
>>> def avg(*nums, typ='a'):
...     return sum(nums)/len(nums) if typ == 'a' \
...         else prod(nums)**(1/len(nums))
...
>>> avg(2,3,5,9,6)
5.0
>>> avg(2,3,5,9,6,typ='g')
4.384327654865777
```

In this case Python knows where it should stop collecting arguments in `nums`, because it recognizes the keyword argument from the equality symbol. And it is a "mandatory" keyword argument, because it is after a variadic one. (The `*` above, which was used to force the subsequent parameters to be keyword parameters, can be thought of as a "dummy variadic parameter" accepting exactly zero arguments, whose only reason to exists is to force the subsequent parameters to be keyword parameters.)

There's a kind of inverse to variadic arguments: if `a` is a list or a tuple, `f(*a)` calls `f` with its members as arguments.[19] For example,

```
>>> (lambda x,y: x+y)(*[1,2])
3
```

and instead of

```
>>> avg(2,3,5,9,6)
5.0
```

we can call our `avg()` function like this:

```
>>> avg(*[2,3,5,9,6])
5.0
```

Example 5.1. This is an example that uses both variadic arguments and this "inverse". (It's a simplified version of Python's `map()` function.) Its first argument is a function of any number of arguments, and the rest of its arguments are that many lists. The result is the list of the result of applying the function to successive elements of the lists. For example,

```
>>> mymap(lambda x,y,z: (y,z,x), [1,2,3],[4,5,6],['a','b','c'])
[(4, 'a', 1), (5, 'b', 2), (6, 'c', 3)]
```

Here's the definition:

[19]This works for strings, too, but that seems utterly useless.

```
def mymap(fn, *lists):
    return [fn(*i) for i in  zip(*lists)]
```

We need `*lists` in the header, because we don't know in advance the arity of `fn` and hence the number of lists `mymap()` will be called with. In the body of the function `lists`'s value is a tuple of lists. We feed these lists as separate arguments (that's what the `*` does in `zip(*lists)`) to `zip()` (which, fortunately, is also a function that accepts any number of arguments), which returns a list (actually, an iterable, but that doesn't matter and shouldn't concern us now) of tuples: the first (which will be the value of `i` in the first iteration) contains the first members of all the lists that were the arguments of `mymap()`, the second tuple contains their second members, etc. And, in each iteration, `fn` will be called with the *members* of the value of `i` (and not the value itself), because of the asterisk in `fn(*i)`, so on the first iteration, the first members of the lists, on the second the second members of the lists, etc. And the result is the list of the results `fn` returns.

Anonymous functions. We've seen and used `lambda`s before, but a short overview of what they are and why they are useful doesn't hurt.

First of all, `lambda`s are not indispensable. (Very few constructs are.) They construct functions whose bodies consist of one expression only, which will be the return value of the constructed function. So, wherever

```
lambda v1,...,vn: expr
```

appears in our program, we can always write `f` instead, supposing we have also written

```
def f(v1,...,vn):
    return expr
```

before, and that `f` is not used otherwise as the name of a function.

And this shows two reasons why `lambda`s are useful. First, when we need a function only once, it's not just an overkill to give it a name, but potentially dangerous, too: we need to make sure that there is no function of the same name elsewhere in the program. The other reason is that we see immediately what our `lambda` does, there's no need to look up the definition of a function defined elsewhere. And the definition *must* often be elsewhere, because, unlike a `lambda`, a `def` is a statement, not an expression, so cannot be written where an expression is expected, such as in a list.

Exercise 5.1. Write a function computing the factorial of positive integers using `reduce()` from `functools`.

Higher order functions. Functions that take functions as arguments or return functions are called higher order functions.

One use of higher order functions is avoiding code duplication. For example, suppose we need to do various operations on lists of numbers. We could write functions for each of these:

```
>>> def inc_list(l):
...     return [x+1 for x in l]
...
>>> def double_list(l):
...     return [2*x for x in l]
...
>>> inc_list(list(range(10)))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> double_list(list(range(10)))
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

but these two functions are practically the same; the only difference is what they are doing with the members of the list. So it makes sense to turn *that* into an extra parameter:

```
>>> def process_list(fun,l):
...     return [fun(x) for x in l]
...
>>> process_list(lambda x: x+1, list(range(10)))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> process_list(lambda x: 2*x, list(range(10)))
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Actually, `process_list()` is a simplified version of `mymap()` defined in Example 5.1 on page 62, which in turn is a simplified version of Python's `map()` function. `map()` returns an iterable[20], not a list, but that can be converted to a list if that's what we want:

```
>>> list(map(lambda x: x**2,range(10)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Here's an example of a function that, besides having a function as an argument, returns a function:

```
>>> def compose(f1,f2):
...     return lambda x: (f1(f2(x)))
...
>>> compose(lambda x: x+1,lambda x: 2*x)(4)
9

>>> compose(lambda x: 2*x, lambda x: x+1)(4)
10
```

[20] a `map`, to be precise

```
>>> from math import sqrt

>>> compose(sqrt, lambda x: x+1)(1)
1.4142135623730951
```

Documentation. Since the body of a function is a series of statements and expressions which get evaluated in the order they are written, it doesn't make a difference in the behaviour of a function if a literal object (such as `42` or `"nice wheather, eh?"`) is included in this series. Now if that literal object happens to be a string, and it's inserted as the very first statement, then it's called a *documentation string*, and is stored in the `__doc__` attribute of the function.

```
>>> def fun():
...     """
...     This function does nothing, but does it well.
...     Usage: fun()
...     """
...     pass
...
>>> print(fun.__doc__)

    This function does nothing, but does it well.
    Usage: fun()
```

The docstring is retrievable by the various IDEs. For example, by `fun?` or `help(fun)` in IPython. But under the hood, it almost surely uses the `__doc__` attribute.

Documenting the functions we write this way is very good practice.

6. Modules

Some less used parts of Python, and all "third party" provided functionalities are not loaded by default. They are collected in *modules* that can be `import`ed in different ways.

(1) `import math` This imports everything in the module `math`; you can access them by prefixing their name by `math.` For example, `math.sqrt()`.
(2) `import math as mt` The same as before, but using `mt` as an *alias*. This just means that the function `sqrt()` of the module `math` is now accessible as `mt.sqrt()`.

(3) `from math import sqrt` This will not import the whole of `math`, just `sqrt`, but make it accessible without qualification, that is, by simply writing `sqrt`. You can import a list of functions (and classes, etc.) by listing them separated by commas. For example, `from math import sqrt, isqrt`

To learn the details of a module (what it's good for, what functions, classes, variables, etc. it provides), enter

```
>>> help("math")
```

at the command prompt (or `math?` in IPython). And if you're only interested in one function, say, `isqrt()`, of the module, enter

```
Python 3.12.9 (main, Feb  4 2025, 00:00:00) [GCC 14.2.1 20240912 (Red Hat 14.2.
Type "help", "copyright", "credits" or "license" for more information.
>>> help("math.isqrt")
Help on built-in function isqrt in math:

math.isqrt = isqrt(n, /)
    Return the integer part of the square root of the input.
```

(Or `math.isqrt?` in IPython.)

Of course, we can write and use our own modules, too. To create a module named `foo`, we need to create a file named `foo.py` and write the definitions of the functions, classes and whatever else we want to have in the module. If there's a problem importing it, check and perhaps change the list contained in the variable `path` in the `sys` module.

7. Debugging

Programming is debugging. It doesn't happen very often that a function or method, not to mention a whole program does what it needs to do the first time it's run. There are two cases: either it throws an exception and we end up with a more or less unintelligible stack trace (often this is the better outcome), or it runs with no errors and produces the wrong result (in this case we're lucky if we realize that the result is wrong). The first kind of problem is better because at least we know where to start looking for the error.

In either case, we have a few options for investigating.

(1) Judicious use of `print()` calls. For example, to "trace" what is going on when we call the function `factorial()`:

```
Python 3.12.9 (main, Feb  4 2025, 00:00:00) [GCC 14.2.1 20240912 (Red Hat
Type "help", "copyright", "credits" or "license" for more information.
>>> def factorial(n): return n if n <= 1 else n * factorial(n-1)
...
```

we may want to put in an extra `print()` to show what arguments it is called with:

```
>>> def factorial(n):
...     print('n:',n)
...     return n if n <= 1 else n * factorial(n-1)
...
>>> factorial(4)
n: 4
n: 3
n: 2
n: 1
24
```

Depending on what we want to understand about our function, this may or may not help. If it doesn't, we can try to get a more complete picture:

```
>>> def factorial(n):
...     res = n if n <= 1 else n * factorial(n-1)
...     print('in: ',n,'out: ',res)
...     return res
...
>>> factorial(4)
in:  1 out:  1
in:  2 out:  2
in:  3 out:  6
in:  4 out:  24
24
```

The good thing about this method is that it doesn't need any extra tools, and is very easy to use. The bad is that we have to remove all those `print()`s afterwards, and that there is no interactivity: if, by looking at the value of one variable, we find we also need to know about another, we have to change the function and start all over again. Nevertheless, there is anecdotal evidence showing that this is the most popular debugging method used by Python programmers.

(2) Tracing your function. If you enter this:

```
def trace(f):
    depth = 0
    def wrapper(*args,**kwargs):
        nonlocal depth
        depth += 1
        print(f'{depth:>{2*depth}}: {f.__name__}:', *args, kwargs or '')
        res = f(*args,**kwargs)
```

```
            print(f'{depth:>{2*depth}}: {f.__name__} returned: {res}')
            depth -= 1
            return res
        return wrapper
```

(the details are not important), or save it in a file named `trace.py` in your working directory and import it with

```
from trace import trace
```

then any function, whose definition is preceeded by `@trace` will be traced, as in the following example:

```
>>> @trace
... def fact(n):
...     return 1 if n<=1 else n*fact(n-1)
...
>>> fact(5)
 1: fact: 5
   2: fact: 4
     3: fact: 3
       4: fact: 2
         5: fact: 1
         5: fact returned: 1
       4: fact returned: 2
     3: fact returned: 6
   2: fact returned: 24
 1: fact returned: 120
120
```

For untracing, just redefine the function without the preceeding `@trace`:

```
>>> def fact(n):
...     return 1 if n<=1 else n*fact(n-1)
...
>>> fact(5)
120
```

The good thing about tracing is that there's no need to change the program. On the other hand, it only shows how functions are called and what they return.

(3) Using a debugger. This is the most versatile method, but you need to learn to use a separate software. Or more, because there are many. The standard one, `pdb` (*p*ython *d*ebugger) is always present, but it's not very user friendly. You can try it like this:

```
>>> def fact(n):
...     breakpoint()
...     return 1 if n<=1 else n*fact(n-1)
```

```
...
>>> fact(4)
> <stdin>(3)fact()
(Pdb)
```

This is pdb's prompt; you can ask for help, or type q to quit.

IPython has a version of pdb that has the same commands, but is more user friendly. For example, it is easier to enter (there's no need to change the function)[21]:

```
Python 3.9.10 (main, Jan 17 2022, 00:00:00)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.20.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: def fact(n):
   ...:     if n<=1:
   ...:         return 1
   ...:     else:
   ...:         return n*fact(n-1)
   ...:

In [2]: %debug fact(5)
NOTE: Enter 'c' at the ipdb>  prompt to continue execution.
> <string>(1)<module>()

ipdb> s
--Call--
```

We don't want to continue execution, because that would just run the function to completion, but want to "step in" the function, and that's what the s command does.

```
> <ipython-input-1-2334ab7e9b53>(1)fact()
----> 1 def fact(n):
      2     if n<=1:
      3         return 1
      4     else:
      5         return n*fact(n-1)
```

Here we can see another aspect of IPython being more user friendly than pdb: it shows a bit of context. (This is why I use a more verbose version of fact().)

Now I use n, which means "execute the *n*ext statement". In this context, s would do the same.

[21]It is also possible to enter the debugger when our program throws an exception, by entering %debug.

```
ipdb> n
> <ipython-input-1-2334ab7e9b53>(2)fact()
      1 def fact(n):
----> 2     if n<=1:
      3         return 1
      4     else:
      5         return n*fact(n-1)

ipdb> n
> <ipython-input-1-2334ab7e9b53>(5)fact()
      2     if n<=1:
      3         return 1
      4     else:
----> 5         return n*fact(n-1)
      6
```

At this point, `s` is the good choice, because `n` would just execute line 5 and then return immediately. (Which is what we want when our code calls another function that we don't want to debug.)

```
ipdb> s
--Call--
> <ipython-input-1-2334ab7e9b53>(1)fact()
----> 1 def fact(n):
      2     if n<=1:
      3         return 1
      4     else:
      5         return n*fact(n-1)

ipdb> n
> <ipython-input-1-2334ab7e9b53>(2)fact()
      1 def fact(n):
----> 2     if n<=1:
      3         return 1
      4     else:
      5         return n*fact(n-1)

ipdb> !n
4
```

`!n` shows the value of the variable `n`. (The value of more than one variable can be queried by writing their names after the `!` separated by commas. See also the `display` command!) So we're in the second call into `fact()`. Arguments of the function can also be queried by the command `args`.

```
ipdb> help

Documented commands (type help <topic>):
========================================
EOF    cl         disable  interact  next    psource  rv           undisp
a      clear      display  j         p       q        s            unt
alias  commands   down     jump      pdef    quit     skip_hidden  until
args   condition  enable   l         pdoc    r        source       up
b      cont       exit     list      pfile   restart  step         w
break  continue   h        ll        pinfo   return   tbreak       whatis
bt     d          help     longlist  pinfo2  retval   u            where
c      debug      ignore   n         pp      run      unalias

ipdb> c

In [3]:
```

Pythontutor is also a kind of debugger; its strong point is that it shows what's happening with our variables, in a beautiful, graphical way.

8. The 45 minutes introduction to object oriented programming

Suppose we want a datatype for computing with matrices. We could easily represent matrices by nested lists: for example, by the list of rows, where a row is represented by the list of its members. So `[[1,0,0],[0,1,0],[0,0,1]]` would represent the 3×3 identity matrix. If `m` is such a matrix, we could get or set the jth member of its ith row by `m[i][j]` and `m[i][j] = v`.

This works, but the lack of abstraction (we need to deal with nested lists instead of matrices) leads to all kinds of difficulties, the most important being that if we ever come up with a better representation, we need to change all our code that deals with matrices. For example, we may find out later that we need to deal with sparse matrices (matrices where almost all elements are the same): representing them by nested lists is a waste of space.

One solution to this problem (and some others besides) is to define a matrix *class*.

```
class Mtx():

    def __init__(self, list):
```

```
        nc = len(list[0])
        #rows must be of equal length
        assert all([len(l) == nc for l in list[1:]]), 'dimension mismatch'
        self._list = list
        self._no_of_rows = len(list)
        self._no_of_columns = nc

    def no_of_rows(self):
        return self._no_of_rows

    def no_of_cols(self):
        return self._no_of_columns

    def get_row(self,rn):
        return self._list[rn]

    def get_col(self,cn):
        return [l[cn] for l in self._list]

    def get(self,r,c):
        return self._list[r][c]

    def set(self,r,c,value):
        self._list[r][c] = value

    def __repr__(self):
        return f'{self._no_of_rows} times {self._no_of_columns} \
        Mtx: {self._list}'

    def __str__(self):
        s = ""
        for i in self._list:
            s += str(i)+'\n'
        return s
```

(1) The definition of a class resembles the definition of a function: it's a block, the first line of which begins with a keyword (`class` in this case) that is followed by the name of the class and then a colon. Between the name of the class and the colon there may be a comma separated list of the names of other classes between parentheses. We will see examples of this later.

(2) In the body, the `def`s look exactly like function definitions, but these define *methods*, not functions.
(3) The first argument (its name doesn't matter, but it's customary to call it `self`) of a method will be bound to the instance of the class on which the method is invoked. That is, if `m` is an instance of `Mtx`, `m.get_row()` calls `.get_row()` with `self` bound to `m`. (The role of `.` in the call `m.get_row()` is the same as in for example `l.append(42)`.)
(4) The "magic" `.__init__()` method is run when an instance of `Mtx` (from now on: an `Mtx`) is created by calling `Mtx()`. (That's why it's magic: Python knows when to call it, we don't have to.) The newly created instance will be bound to the first argument of `.__init__()` and the arguments to `Mtx` (in this case, just one) to the rest.

   In our case, with the call

```
Python 3.12.9 (main, Feb  4 2025, 00:00:00) [GCC 14.2.1 20240912 (Red Hat
Type "help", "copyright", "credits" or "license" for more information.
>>> m = Mtx([[1,2],[3,4]])
```

the second argument to `.__init__()` will be `[[1,2],[3,4]]`, while the first is the new instance, which in this case is also assigned to the variable `m`.

   We can check that the value of `m` is really an `Mtx`:

```
>>> isinstance(m, Mtx)
True
```

(5) The `assert` statement in `.__init__()` stops the user making one possible error, with an informative error message:

```
>>> m = Mtx([[1,2,3],[4,5,6],[7,8]]) #should fail
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in __init__
AssertionError: dimension mismatch
```

(6) `_list`, `_no_of_rows` and `_no_of_columns` are *attributes* of the class. They hold instance-specific data (unlike methods, which belong to the class itself). The underscore signals that even though they can, they shouldn't be accessed (read or set) directly, that is, by anything other than methods of the class (or its *subclasses*, to be introduced later). So this:

```
>>> m._no_of_columns
2
```

is an example of what one should never do outside of the definition of a class. Invoking the accessor method:

```
>>> m.no_of_cols()
2
```

is the right way to get the number of columns.

(7) The two other magic methods, `.__str__()` and `.__repr__()` define how instances of the class will be printed and represented (for example, in the debugger). Here they (and `.set()`) are at work:

```
>>> m
2 times 2 Mtx: [[1, 2], [3, 4]]
>>> print(m)
[1, 2]
[3, 4]

>>> m.set(1,1,-5); print(m)
[1, 2]
[3, -5]
```

(8) Throughout, we use `assert` for checking that certain conditions hold. This, as have been mentioned earlier, is not such a good idea; but we don't know how to define exceptions in Python (actually, we're in the process of learning it, since exceptions are classes derived from the `Exception` class), so it's the best we can do.

To make our definition of `Mtx` useful, we should at least define methods `.add()` and `.prod()` for adding and multiplying them.

```
def add(self, other):
    assert isinstance(other,Mtx), 'only an Mtx can be added to an Mtx'
    assert (self.no_of_cols() == other.no_of_cols()
            and self.no_of_rows() == other.no_of_rows()),\
            'dimension mismatch'
    nr = self.no_of_rows()
    nc = self.no_of_cols()
    m = Mtx([nc * [0] for _ in range(nr)])
    for i in range(nr):
        for j in range(nc):
            m.set(i,j,self.get(i,j)+other.get(i,j))
    return m

def prod(self, other):
    assert isinstance(other,Mtx), 'an Mtx can only be multiplied by an Mtx'
    assert self.no_of_cols() == other.no_of_rows(),\
     f'''The number of columns ({self.no_of_cols()}) of {self}
```

```
         is not the same as the number of rows \
         ({other.no_of_rows()}) of {other}.'''
        nr = self.no_of_rows()
        nc = other.no_of_cols()
        onr = other.no_of_rows()
        m = Mtx([nc * [0] for _ in range(nr)])
        for i in range(nr):
            for j in range(nc):
                m.set(i,j,sum(
                 [self.get(i,k)*other.get(k,j) for k in range(onr)]))
        return m
```

This should be part of the class definition, otherwise Python wouldn't know which class they are supposed to be the methods of.

```
>>> m1 = Mtx([[1,2,3]]); m2 = Mtx([[1,2],[3,3],[2,1]])
>>> m1.add(42) #should fail
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 37, in add
AssertionError: only an Mtx can be added to an Mtx
>>> print(m1.add(m1))
[2, 4, 6]

>>> print(m2.prod(m1)) #should fail
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 49, in prod
AssertionError: The number of columns (2) of
[1, 2]
[3, 3]
[2, 1]
 is not the same as the number of rows (1) of
[1, 2, 3]
.
>>> print(m1.prod(m2))
[13, 11]
```

What else do we do with `Mtx`s beside adding and multiplying them? We could for example define a method that does scalar multiplication. (Do it!) We'd also want determinant, but there is a problem there: that makes sense only for square matrices. The same holds for powers. We could of course make `.power()` or `.determinant()` methods of `Mtx` and start their definition by checking that `_no_of_columns == _no_of_rows`;

but it's much better to create a specialized version, say `SqMtx` of `Mtx`, and define `.power()` and `.determinant()` as methods of `SqMtx`. `SqMtx` is called a *subclass* or *derived class* of `Mtx`, and `Mtx` is a *superclass* or *base class* of `SqMtx`.

A subclass, such as `SqMtx`, *inherits* everything (attributes, methods) from its superclass, except what is *overridden* in its definition, which, in `SqMtx`'s case, are the `.__init__()` and the `__repr__()` methods.

Here is how to do that (I leave `.determinant()` as a (nontrivial) exercise):

```
class SqMtx(Mtx):

    def __init__(self, list):
        super().__init__(list)
        assert self._no_of_columns == self._no_of_rows, \
        f'''The number of columns ({self._no_of_columns}) should be the same
        as the number of rows ({self._no_of_rows}).'''
        self._dim = self._no_of_rows

    def power(self,n):
        assert isinstance(n,int), 'The exponent should be an integer'
        res = self
        while n>1:
            res = res.prod(self)
            n -= 1
        return res

    def __repr__(self):
        return f'SqMtx of dimension {self._dim}: {self._list}'
>>> sm = SqMtx([[1,2],[3,4]])
>>> isinstance(sm, Mtx) #an SqMtx is an Mtx
True
>>> sm.add(sm) #that's why this works
2 times 2 Mtx: [[2, 4], [6, 8]]
>>> print(sm.power(1)); print(sm.power(3))
[1, 2]
[3, 4]

[37, 54]
[81, 118]
```

The first line of the definition declares the base class or classes of the new class. (If there are more, they are separated by commas.) The definition of the `__repr__()` method completely overrides the definition given in the base class. With `.__init__()`, the situation is similar, in that it overrides `Mtx`'s `.__init__()`. The difference is that it uses it, too. And the key to achieve this is the call to `super()`, which returns a reference to the superclass part of the object; so

```
super().__init__(list)
```

initializes an `Mtx`. Once that is done, we do the rest, the `SqMtx`-specific part of the work.

Appendix A. Standalone programs

Suppose someone finds one of our programs in Section 3.2 so useful that she wants to use it, too. What can we do to make it usable for her?

The first thing of course is that we need to define a function and make the filename an argument of it.

```
def cat(fn):
    with open(fn) as file:
        for line in file:
            print(line.rstrip())
```

In theory, we can send the user the file that contains this function definition. But in practice, we can't expect the users of our program to start a Python interpreter, load our program and invoke the function (`cat` in this case) that is its main entry point. We need to be able to deliver an executable file, or at least one that can be started with

```
python mycat
```

or perhaps

```
python mycat data.txt
```

from a terminal. (If we can deliver an executable, then the user can omit `python` from the above commands. But the way to do this delivery depends on the operating system.)

If `mycat.py` contains this:

```
import sys

def cat(fn):
    with open(fn) as file:
        for line in file:
            print(line.rstrip())

def main():
    if len(sys.argv) == 2:
        cat(sys.argv[1])
```

```
11     else:
12         raise SystemExit('Usage: '+ sys.argv[0] + ' [ filename ]')
13
14 if __name__ == '__main__':
15     main()
```

then

```
[simon@localhost tmp]$ python mycat.py data.txt
one
two
three
very long
four
five
[simon@localhost tmp]$ python mycat.py
Usage: mycat.py [ filename ]
[simon@localhost tmp]$
```

and if we include `#!/usr/bin/python` as the first line of mycat.py, then it can be invoked as `./mycat data.txt` on Linux. (`./` is not needed if `mycat.py` is in a directory that is a member of `$PATH` environment variable — but this has nothing to do with Python.)

The details:

- The role of the last two lines is to arrange that the `main()` function will be called if the program is run in one of the two ways above. The reason is that in this case the built-in variable `__name__` has the value `'__main__'`. (If it's `import`ed[22] in an other file with `import mycat`, its value is `mycat`.)
- `sys.argv` in lines 9, 10 and 12 is a list that contains the words of the invocation (except for `python`): so with

  ```
  [simon@localhost tmp]$ python mycat.py data.txt
  ```

  we get `sys.argv[0]==mycat.py` and `sys.argv[1]==data.txt`. That's how we can access the command line arguments.
- Line 12 raises an exception (signals that "something is wrong") and prints our message explaining the cause:

  ```
  [simon@localhost tmp]$ python mycat.py
  Usage: mycat.py [ filename ]
  [simon@localhost tmp]$ echo $?
  1
  ```

  In this case it looks as if we could have just printed the message with `print()`. But the fact that the result of `echo $?` is not 0

[22] See Section 6

shows that our program told the shell that it couldn't successfully terminate, which is potentially very useful [23] We'll learn a little more about exceptions in Section 4.2.1.

Exercise A.1. If you haven't done it yet, write a function `is_prime()` that returns `True` if its only argument is a prime, and `False` otherwise. Turn this into a standalone program `is_prime.py`! For example

```
[simon@localhost tmp]$ python is_prime.py 13
```

should print `True`, and if called by the wrong number of arguments, it should print a message explaining the correct invocation.

As at the beginning of this section, you will probably need the function `int()`, because the members of `sys.argv` are strings.

[23] `$ python backup.py && rm -r .` runs `python backup.py` and then, if it terminated normally, `rm -r .`. So it's very important that our programs signal to the shell their exit status. Here's a less dangerous example involving our `mycat.py`:

```
[simon@localhost tmp]$ python mycat.py data.txt && echo "mycat terminated successfully"
one
two
three
very long
four
five
mycat terminated successfully
[simon@localhost tmp]$ python mycat.py && echo "mycat terminated successfully"
Usage: mycat.py [ filename ]
[simon@localhost tmp]$
```