Introduction to Python

Olivier Mattelaer
UCLouvain
CP3 & CISM

Goal of this lecture

- Help you to decide if you want to use Python for your project
- Give you the python syntax such that you can read python code (and write simple one)

What do we cover

- Basic data structure in Python
    - ➡ Advanced one/class will be for the next lecture
- Control Flow
- Function
- My python favorite trick
- Modules/Packages

What is Python

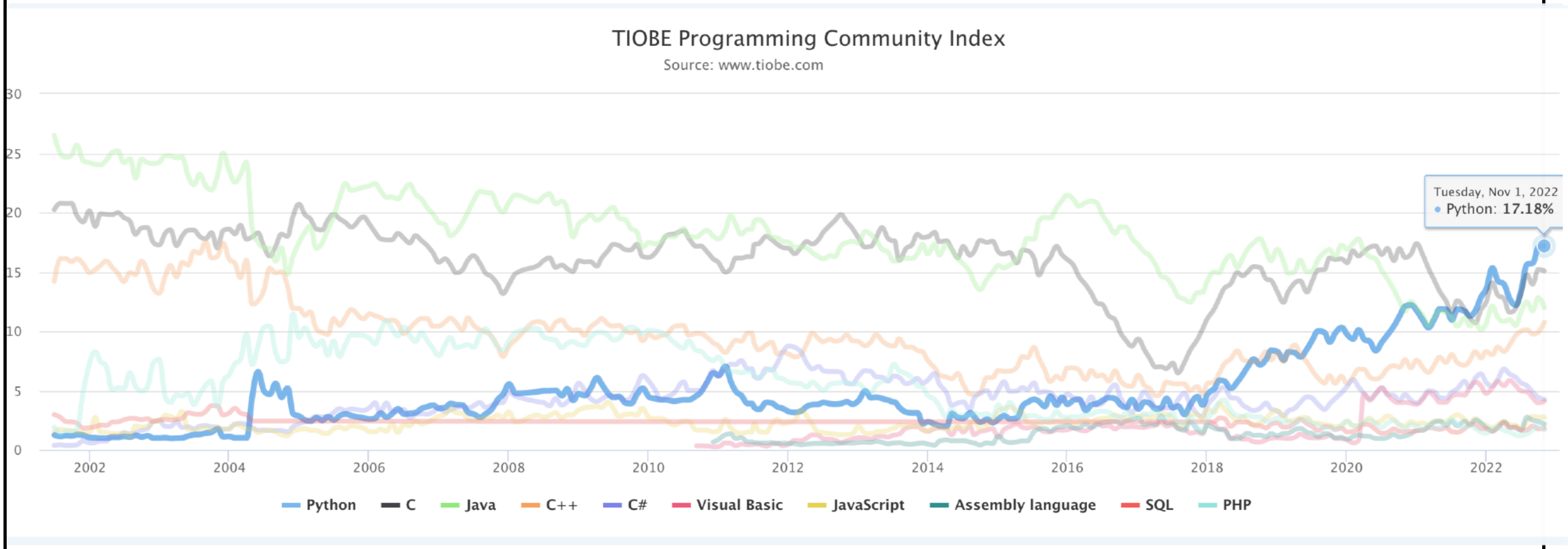
- Python is object-oriented
- Python is Interpreted (executed line by line)
    - High portability
    - Usually lower performance than compiled languages
- Python is High(er)-level (than C or Fortran)
    - Lots of high-level modules and functions
- Python is dynamically-typed and strong-typed
    - no need to explicitly define the type of a variable
    - variable types are not automatically changed (and should not)

Why Python?

- Easy to learn
  - Python code is usually easy to read, syntax tends to be short and simple
  - The Python interpreter lets you try and play
  - Help is included in the interpreter
  - Huge community
- Straight to the point
  - Many tasks can be delegated to modules, so that you only focus on things specific to your needs
- Fast
  - A lot of Python modules are written in C, so the heavy lifting is fast
  - Python itself can be made faster in many ways (there's a session on that)
- Hugely popular

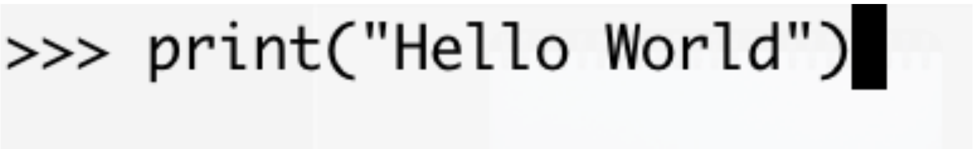
Why Python?

Tiobe Ranking



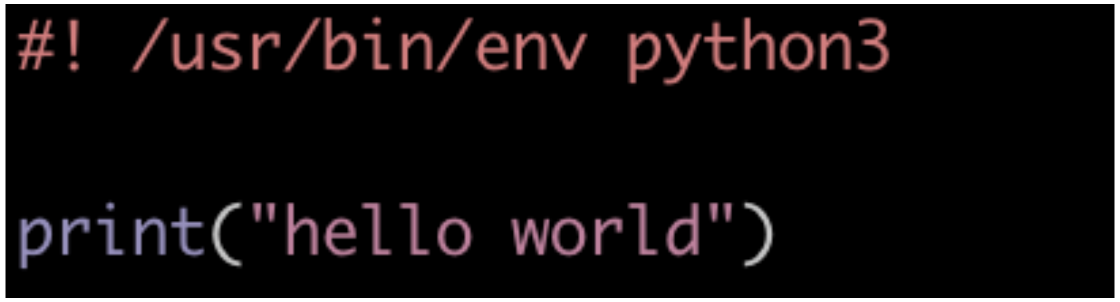
- Python is currently #1
- Strong rise since 2018
  - ➡ Python for machine learning

Hello World



```
>>> print("Hello World")
```

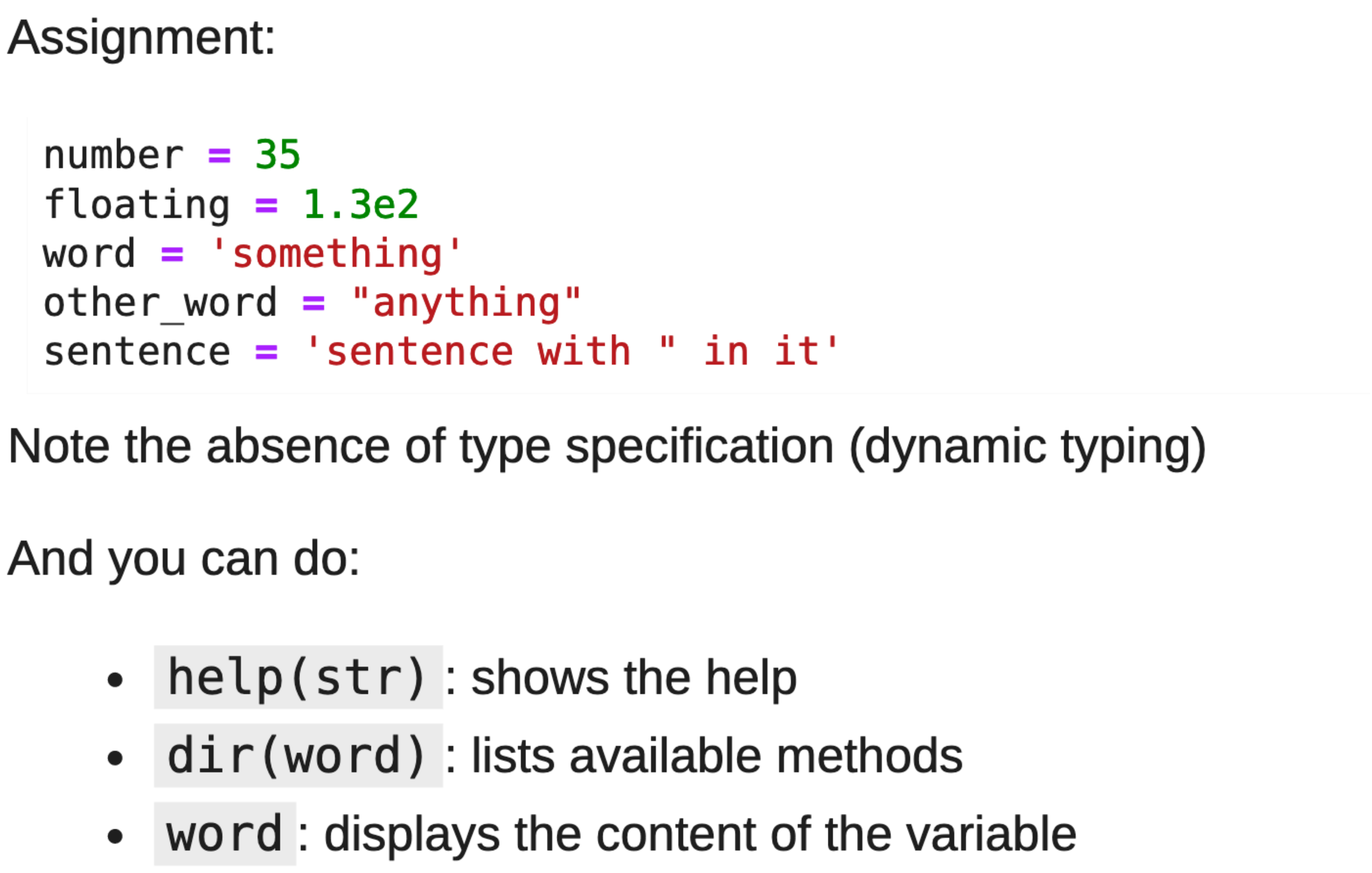
- You can start a terminal
  - ➡ python3
- Write the line in a file
  - ➡ python3 ./myfile.py
- Add a shebang to your file



```
#! /usr/bin/env python3

print("hello world")
```

  - ➡ ./myfile.py
- JupyterHub

Variable



Assignment:

```
number = 35
floating = 1.3e2
word = 'something'
other_word = "anything"
sentence = 'sentence with " in it'
```

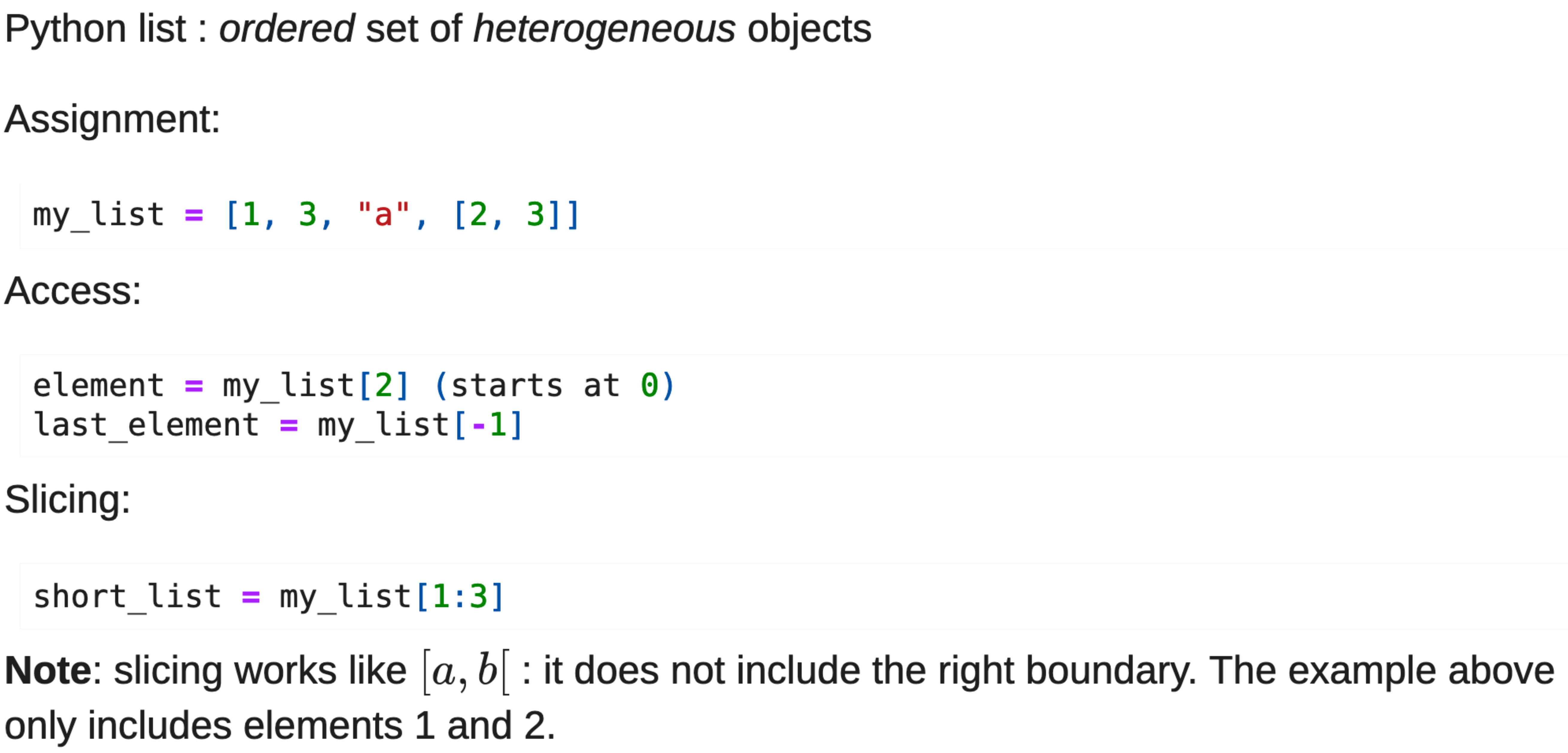
Note the absence of type specification (dynamic typing)

And you can do:

- `help(str)` : shows the help
- `dir(word)` : lists available methods
- `word` : displays the content of the variable

Basic Python Data Structure

List



Python list : *ordered* set of *heterogeneous* objects

Assignment:

```
my_list = [1, 3, "a", [2, 3]]
```

Access:

```
element = my_list[2] (starts at 0)
last_element = my_list[-1]
```

Slicing:

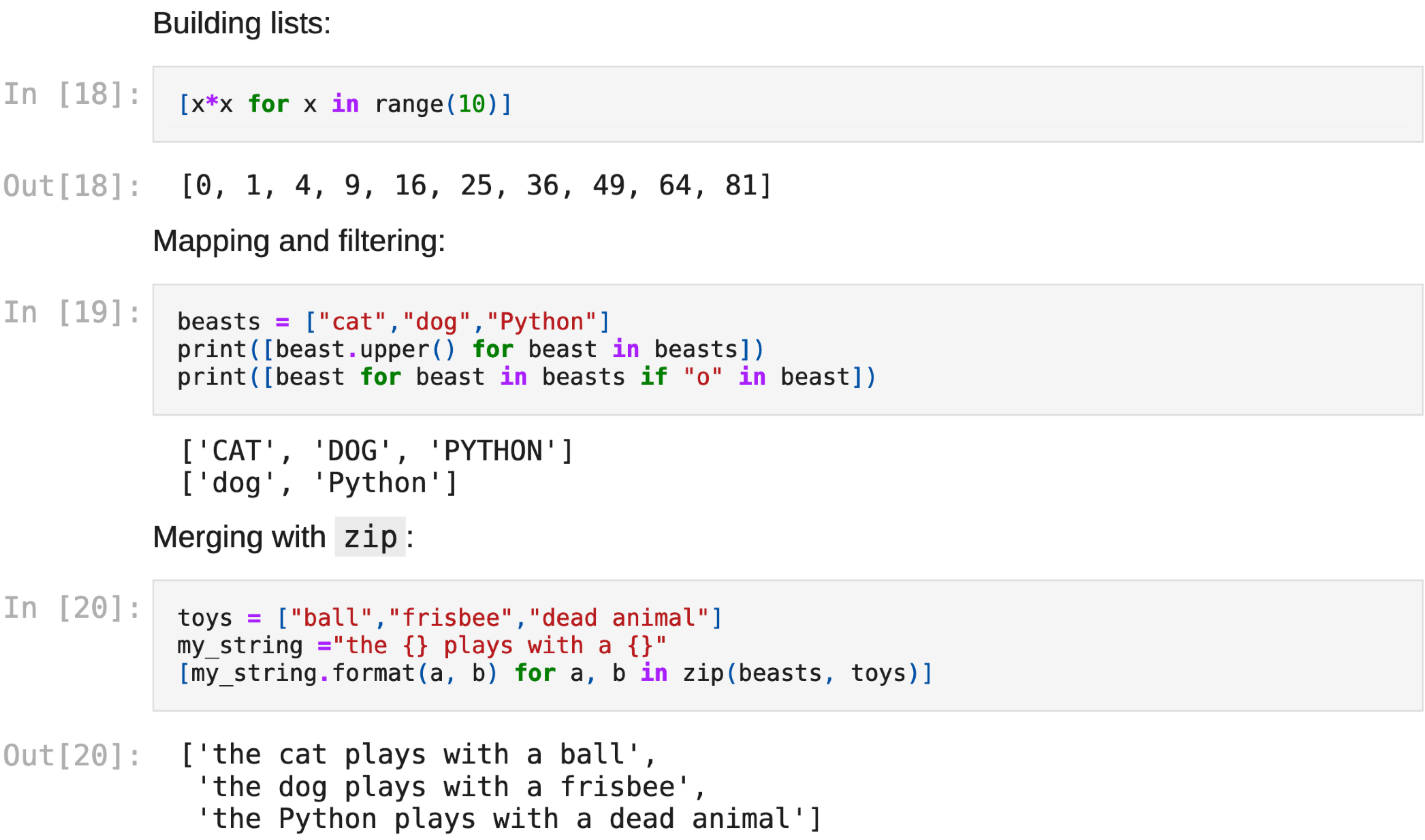
```
short_list = my_list[1:3]
```

Note: slicing works like $[a, b[$: it does not include the right boundary. The example above only includes elements 1 and 2.

Add element to a list:

```
short_list.append(10)
```

Comprehension list



Building lists:

```
In [18]: [x*x for x in range(10)]

Out[18]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Mapping and filtering:

```
In [19]: beasts = ["cat","dog","Python"]
         print([beast.upper() for beast in beasts])
         print([beast for beast in beasts if "o" in beast])

         ['CAT', 'DOG', 'PYTHON']
         ['dog', 'Python']
```

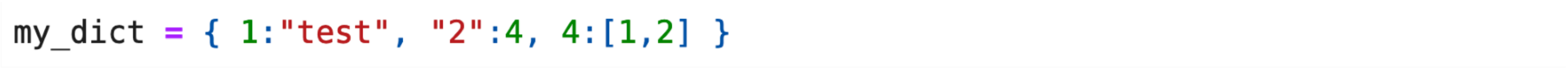
Merging with `zip`:

```
In [20]: toys = ["ball","frisbee","dead animal"]
         my_string ="the {} plays with a {}"
         [my_string.format(a, b) for a, b in zip(beasts, toys)]

Out[20]: ['the cat plays with a ball',
          'the dog plays with a frisbee',
          'the Python plays with a dead animal']
```

Dictionary

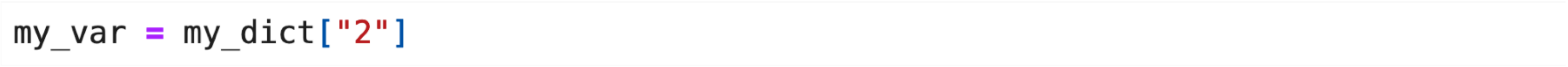
Python dict: *ordered heterogeneous* list of (key -> value) *pairs*

Assignment:



```
my_dict = { 1:"test", "2":4, 4:[1,2] }
```

Access:

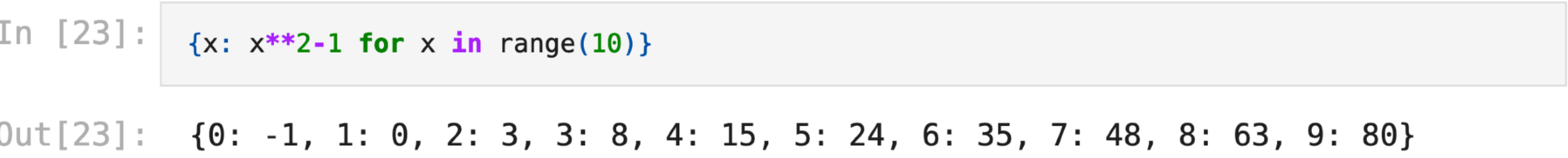


```
my_var = my_dict["2"]
```

Key of the Dictionary need to be immutable element:

So not list/dictionary/…

Dict comprehensions work too:



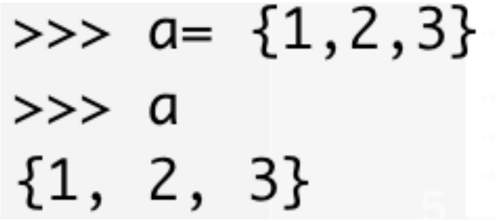
```
In [23]:  {x: x**2-1 for x in range(10)}

Out[23]:  {0: -1, 1: 0, 2: 3, 3: 8, 4: 15, 5: 24, 6: 35, 7: 48, 8: 63, 9: 80}
```

Tuple

- Immutable ordered element
  - ➡ A = (1, 2, [])
  - ➡ Can be used for dict
- Access as for list
  - ➡ A[0] returns 1
- Can not change the content
  - ➡ A[0] = 2 crashes (TypeError: 'tuple' object does not support item assignment)

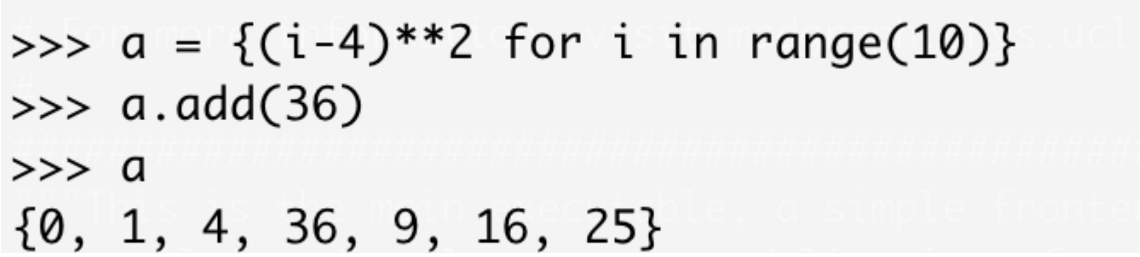
Set

- Unordered and unique element (all element are immutable



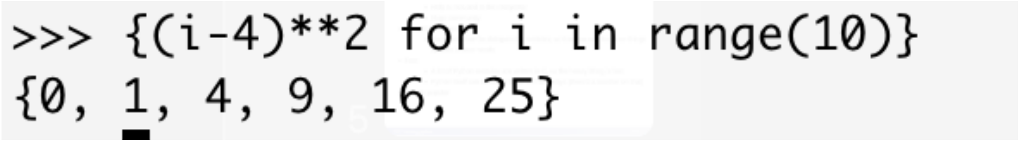
```
>>> a= {1,2,3}
>>> a
{1, 2, 3}
```

- Order are not preserve
  - ➡ Can change from one run to the next
- Can add element



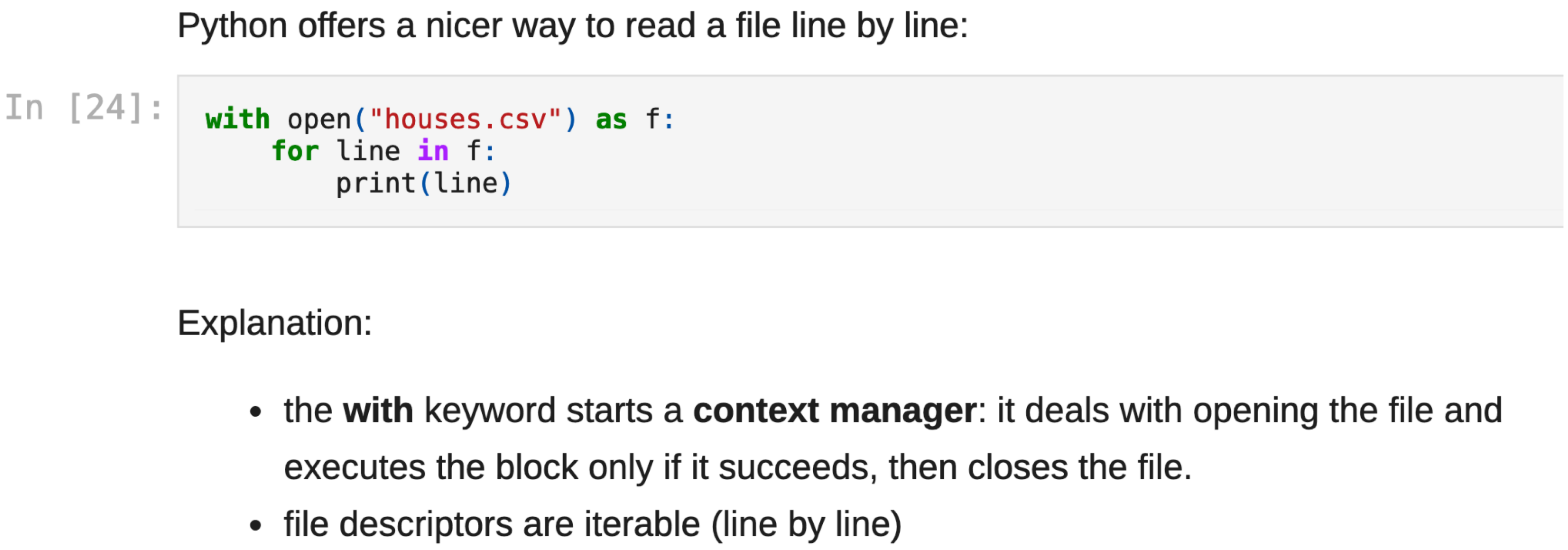
```
>>> a = {(i-4)**2 for i in range(10)}
>>> a.add(36)
>>> a
{0, 1, 4, 36, 9, 16, 25}
```

- Comprehension set



```
>>> {(i-4)**2 for i in range(10)}
{0, 1, 4, 9, 16, 25}
```

Files



Python offers a nicer way to read a file line by line:

```
In [24]:

with open("houses.csv") as f:
    for line in f:
        print(line)
```

Explanation:

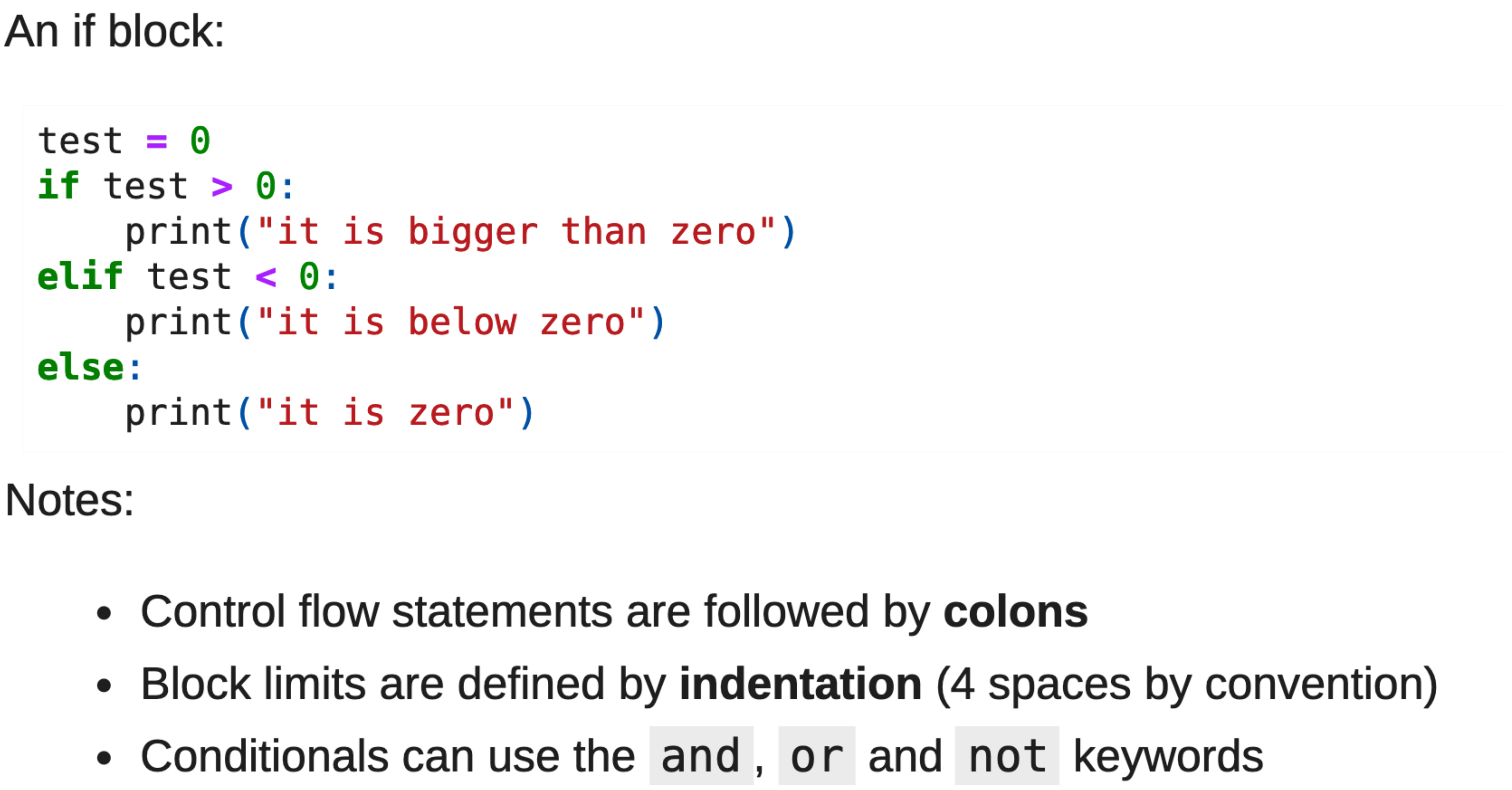
- the **with** keyword starts a **context manager**: it deals with opening the file and executes the block only if it succeeds, then closes the file.
- file descriptors are iterable (line by line)

You can also read the full file with

text = f.read()
all_lines = f.readlines()

Function and Flow

If statement



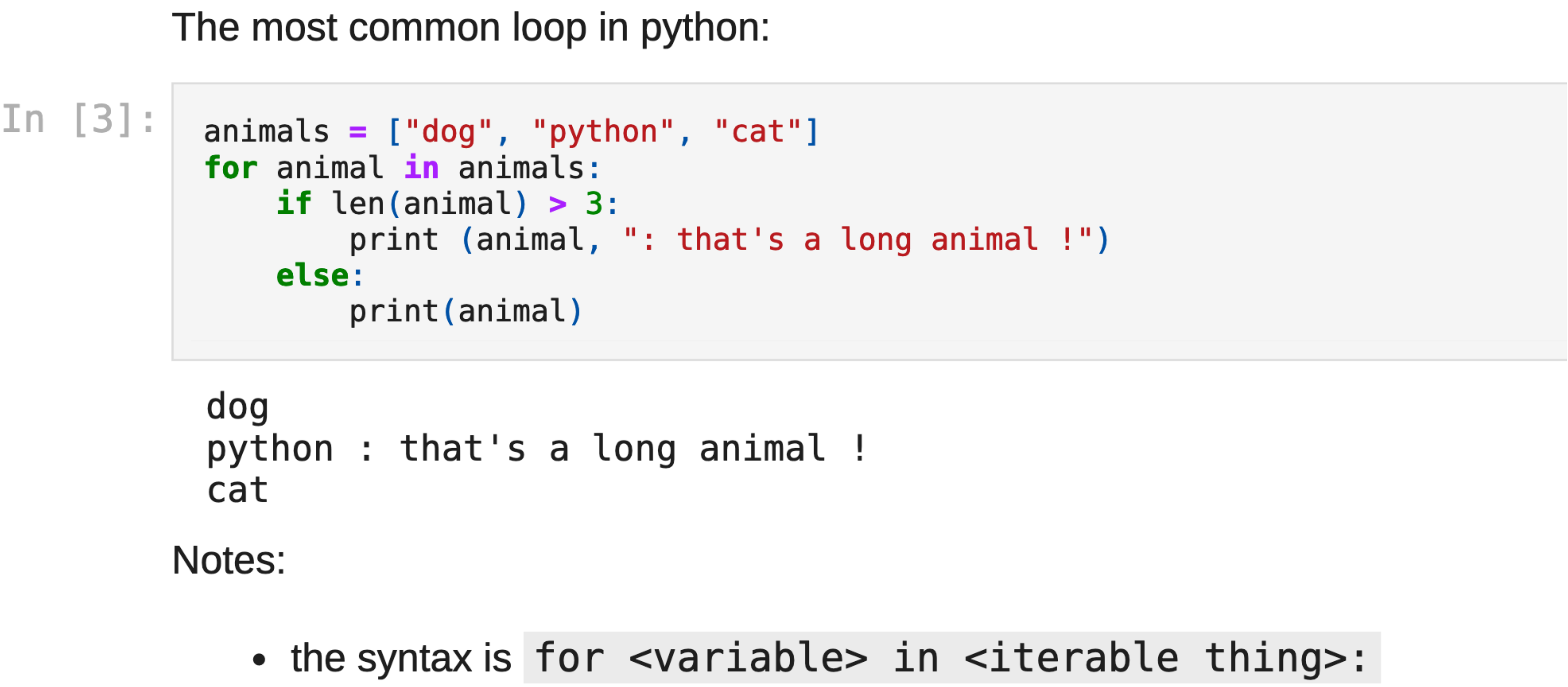
An if block:

```python
test = 0
if test > 0:
    print("it is bigger than zero")
elif test < 0:
    print("it is below zero")
else:
    print("it is zero")
```

Notes:

- Control flow statements are followed by **colons**
- Block limits are defined by **indentation** (4 spaces by convention)
- Conditionals can use the `and`, `or` and `not` keywords

For loop



The most common loop in python:

In [3]:

```
animals = ["dog", "python", "cat"]
for animal in animals:
    if len(animal) > 3:
        print (animal, ": that's a long animal !")
    else:
        print(animal)
```

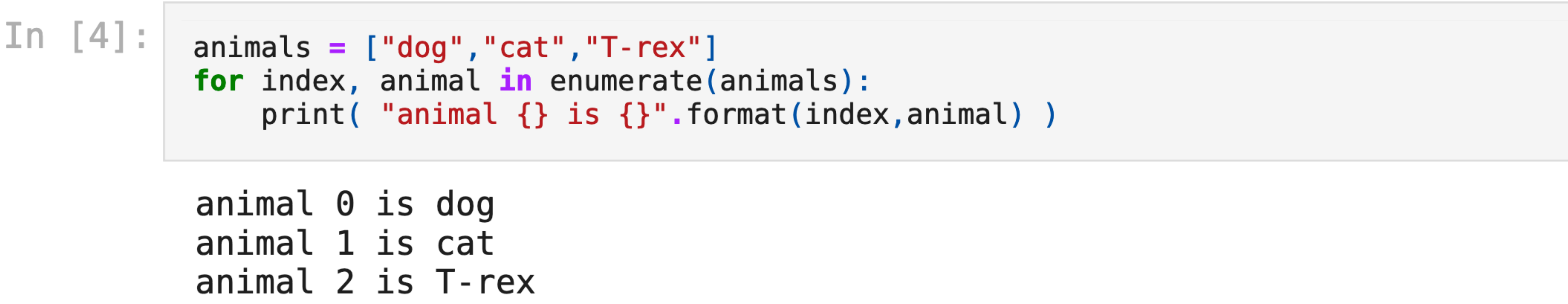
```
dog
python : that's a long animal !
cat
```

Notes:

- the syntax is `for <variable> in <iterable thing>:`

More on for loop

What if i need the index ?

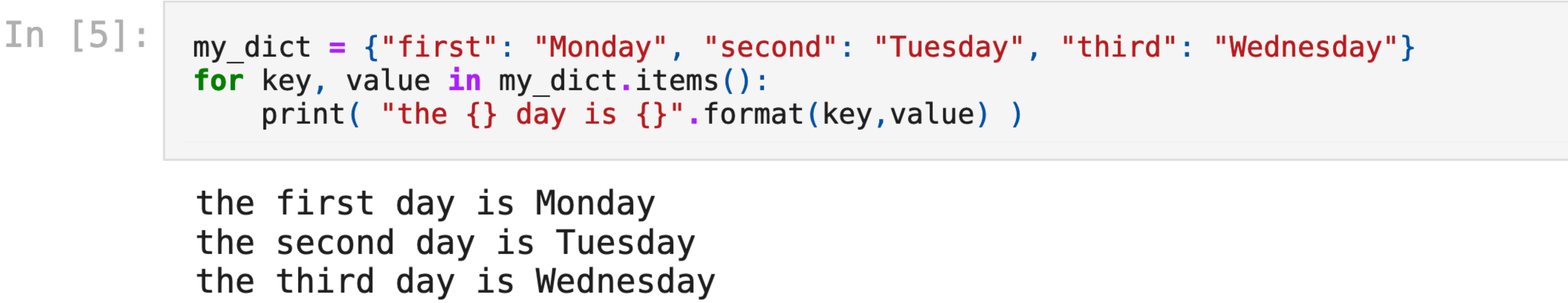


In [4]:

```python
animals = ["dog","cat","T-rex"]
for index, animal in enumerate(animals):
    print( "animal {} is {}".format(index,animal) )
```

```
animal 0 is dog
animal 1 is cat
animal 2 is T-rex
```

What about dictionaries ?



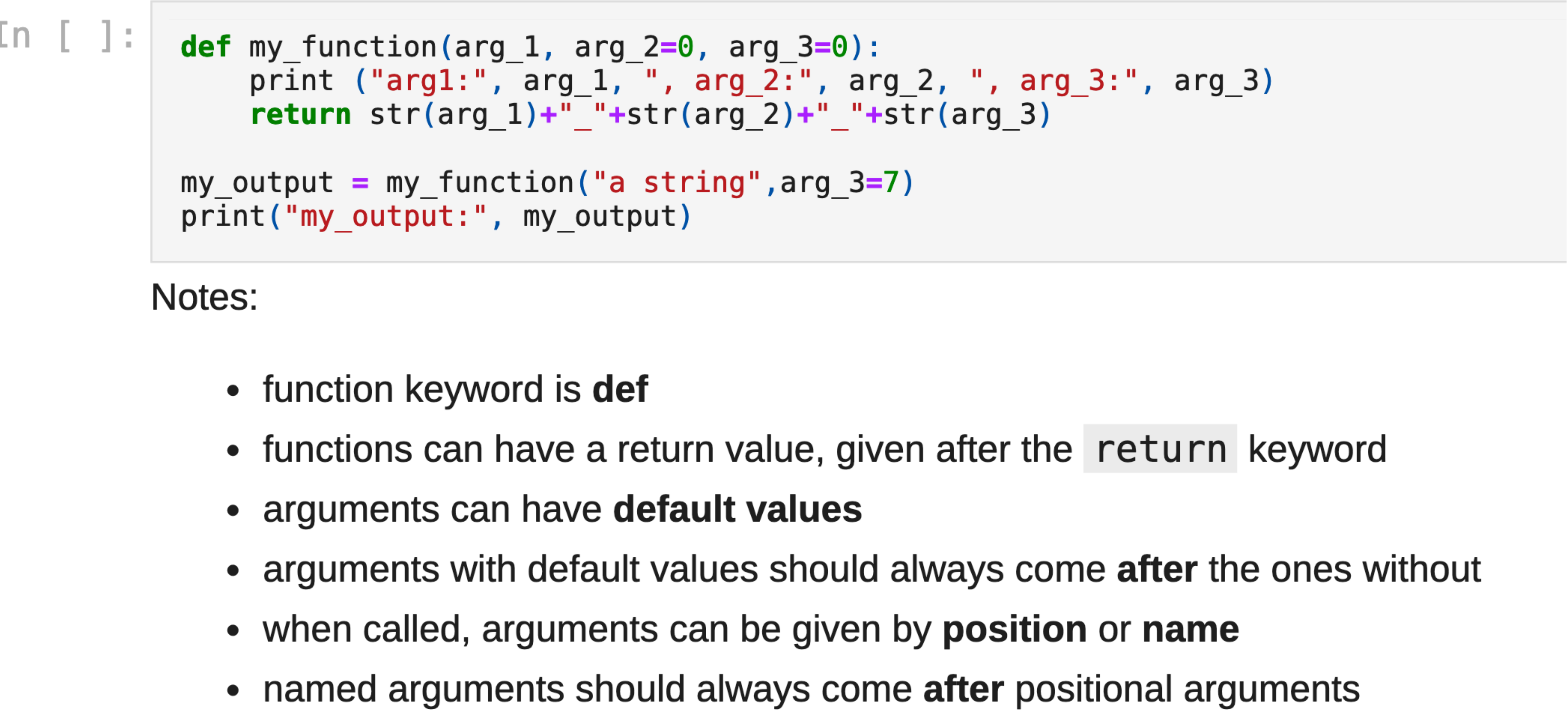
In [5]:

```python
my_dict = {"first": "Monday", "second": "Tuesday", "third": "Wednesday"}
for key, value in my_dict.items():
    print( "the {} day is {}".format(key,value) )
```

```
the first day is Monday
the second day is Tuesday
the third day is Wednesday
```

(More on string formatting very soon)

Functions



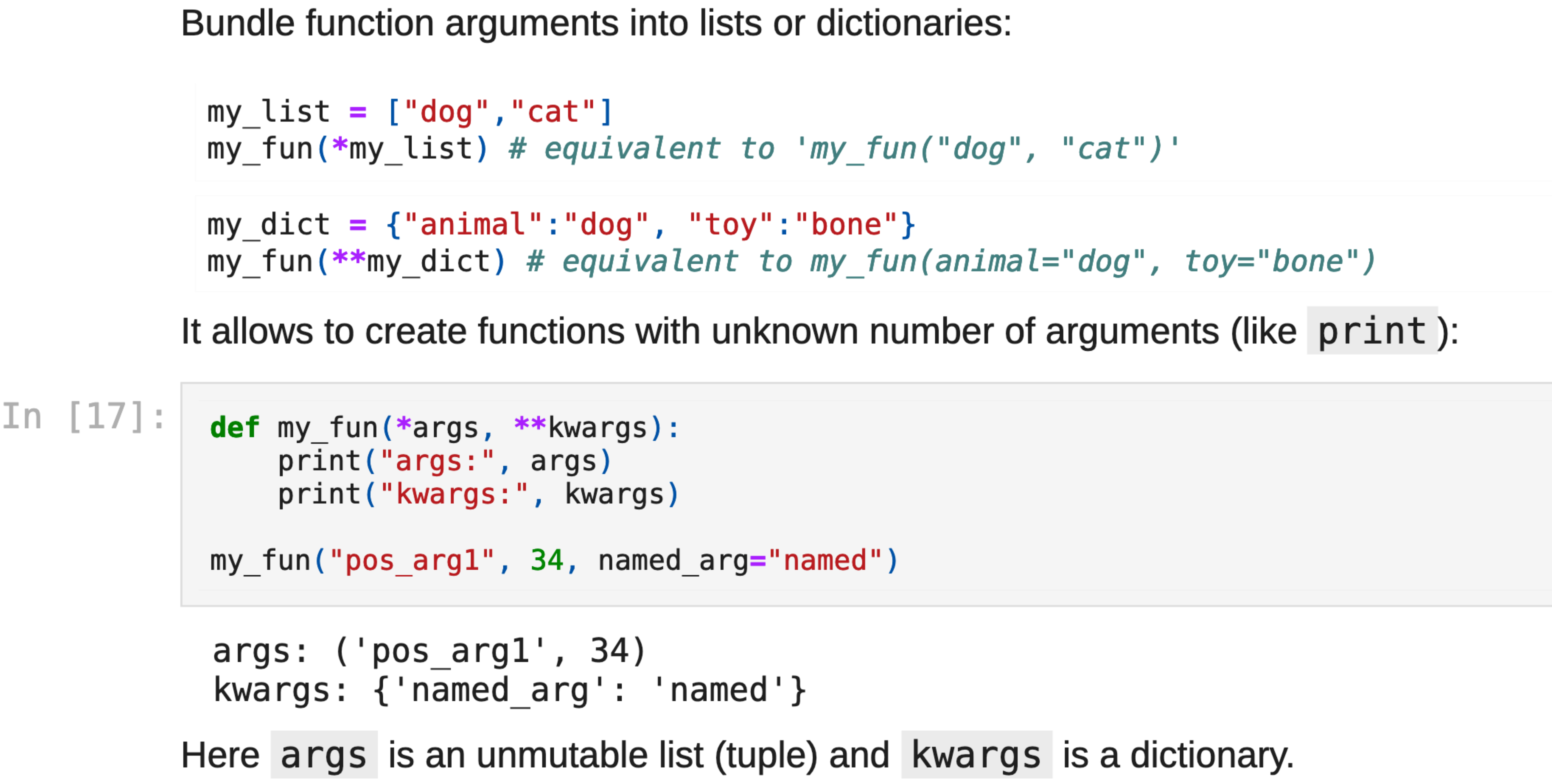
In [ ]:

```
def my_function(arg_1, arg_2=0, arg_3=0):
    print ("arg1:", arg_1, ", arg_2:", arg_2, ", arg_3:", arg_3)
    return str(arg_1)+"_"+str(arg_2)+"_"+str(arg_3)

my_output = my_function("a string",arg_3=7)
print("my_output:", my_output)
```

Notes:

- function keyword is **def**
- functions can have a return value, given after the `return` keyword
- arguments can have **default values**
- arguments with default values should always come **after** the ones without
- when called, arguments can be given by **position** or **name**
- named arguments should always come **after** positional arguments

Function/packing



Bundle function arguments into lists or dictionaries:

```
my_list = ["dog","cat"]
my_fun(*my_list) # equivalent to 'my_fun("dog", "cat")'
```

```
my_dict = {"animal":"dog", "toy":"bone"}
my_fun(**my_dict) # equivalent to my_fun(animal="dog", toy="bone")
```

It allows to create functions with unknown number of arguments (like `print`):

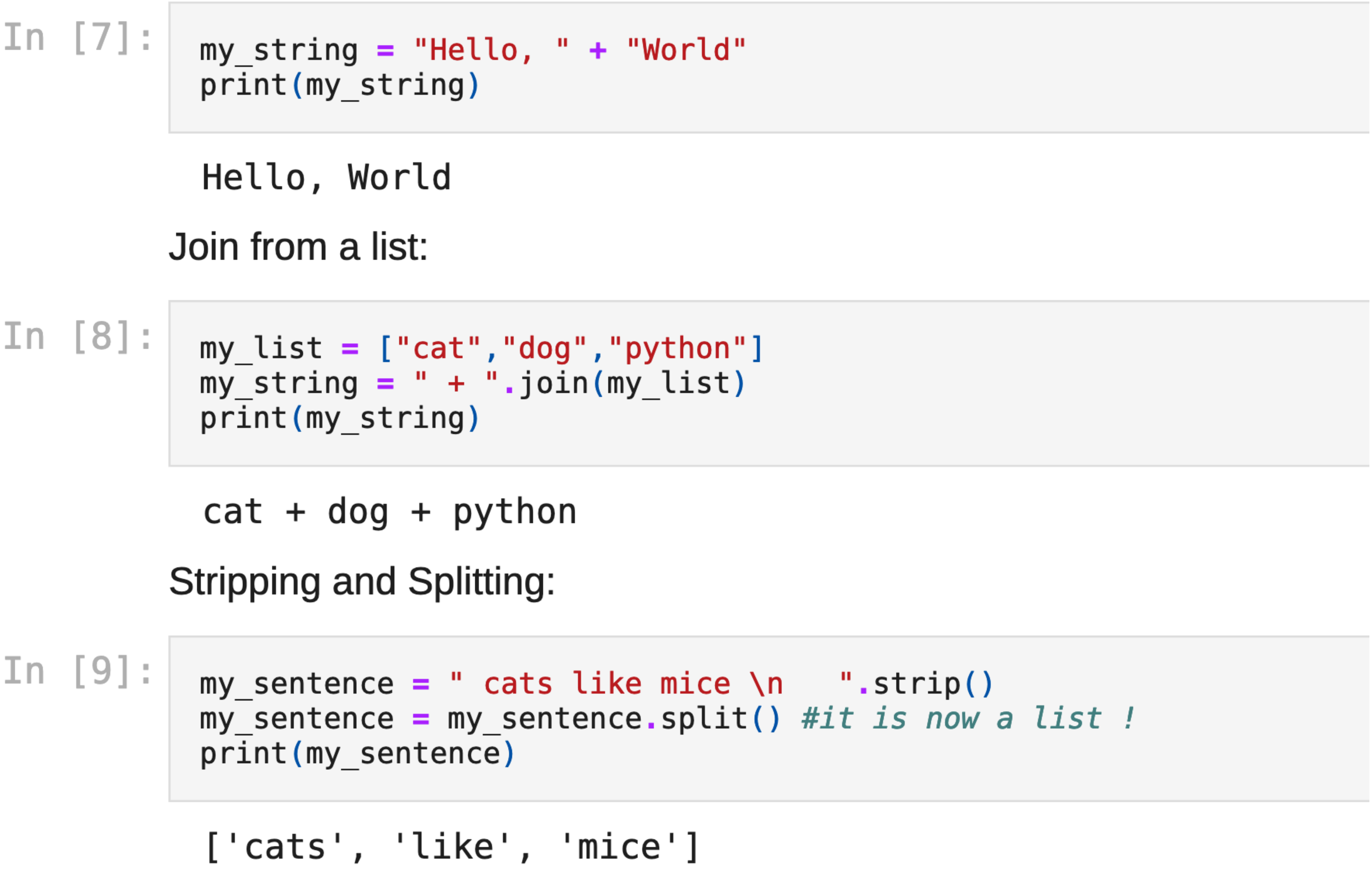
```
In [17]: def my_fun(*args, **kwargs):
             print("args:", args)
             print("kwargs:", kwargs)

         my_fun("pos_arg1", 34, named_arg="named")
```

```
args: ('pos_arg1', 34)
kwargs: {'named_arg': 'named'}
```

Here `args` is an unmutable list (tuple) and `kwargs` is a dictionary.

String Formating

String manipulation



In [7]:
```
my_string = "Hello, " + "World"
print(my_string)
```

```
Hello, World
```

Join from a list:

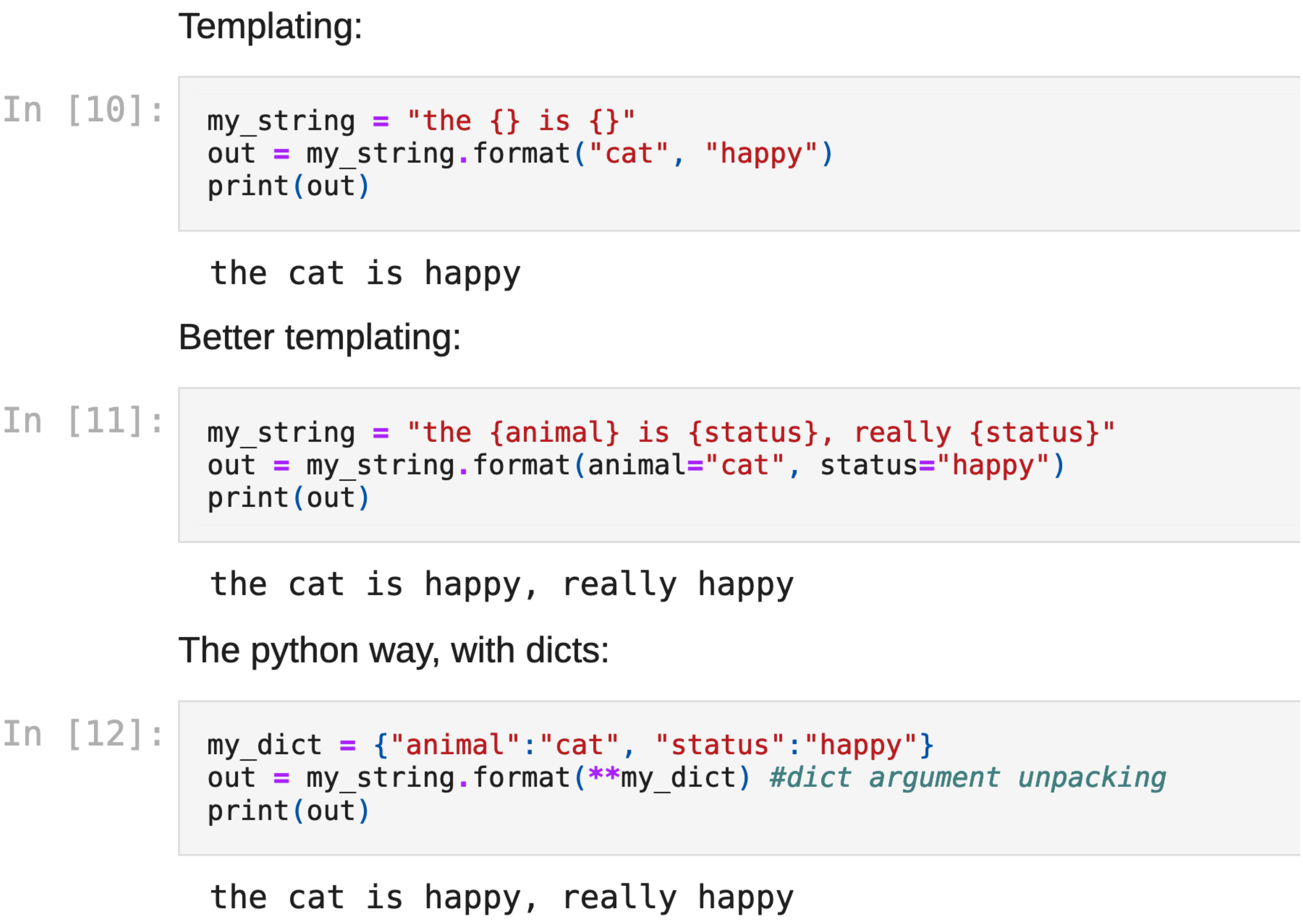
In [8]:
```
my_list = ["cat","dog","python"]
my_string = " + ".join(my_list)
print(my_string)
```

```
cat + dog + python
```

Stripping and Splitting:

In [9]:
```
my_sentence = " cats like mice \n   ".strip()
my_sentence = my_sentence.split() #it is now a list !
print(my_sentence)
```

```
['cats', 'like', 'mice']
```



Templating:

In [10]:

```
my_string = "the {} is {}"
out = my_string.format("cat", "happy")
print(out)
```

```
the cat is happy
```

Better templating:

In [11]:

```
my_string = "the {animal} is {status}, really {status}"
out = my_string.format(animal="cat", status="happy")
print(out)
```

```
the cat is happy, really happy
```

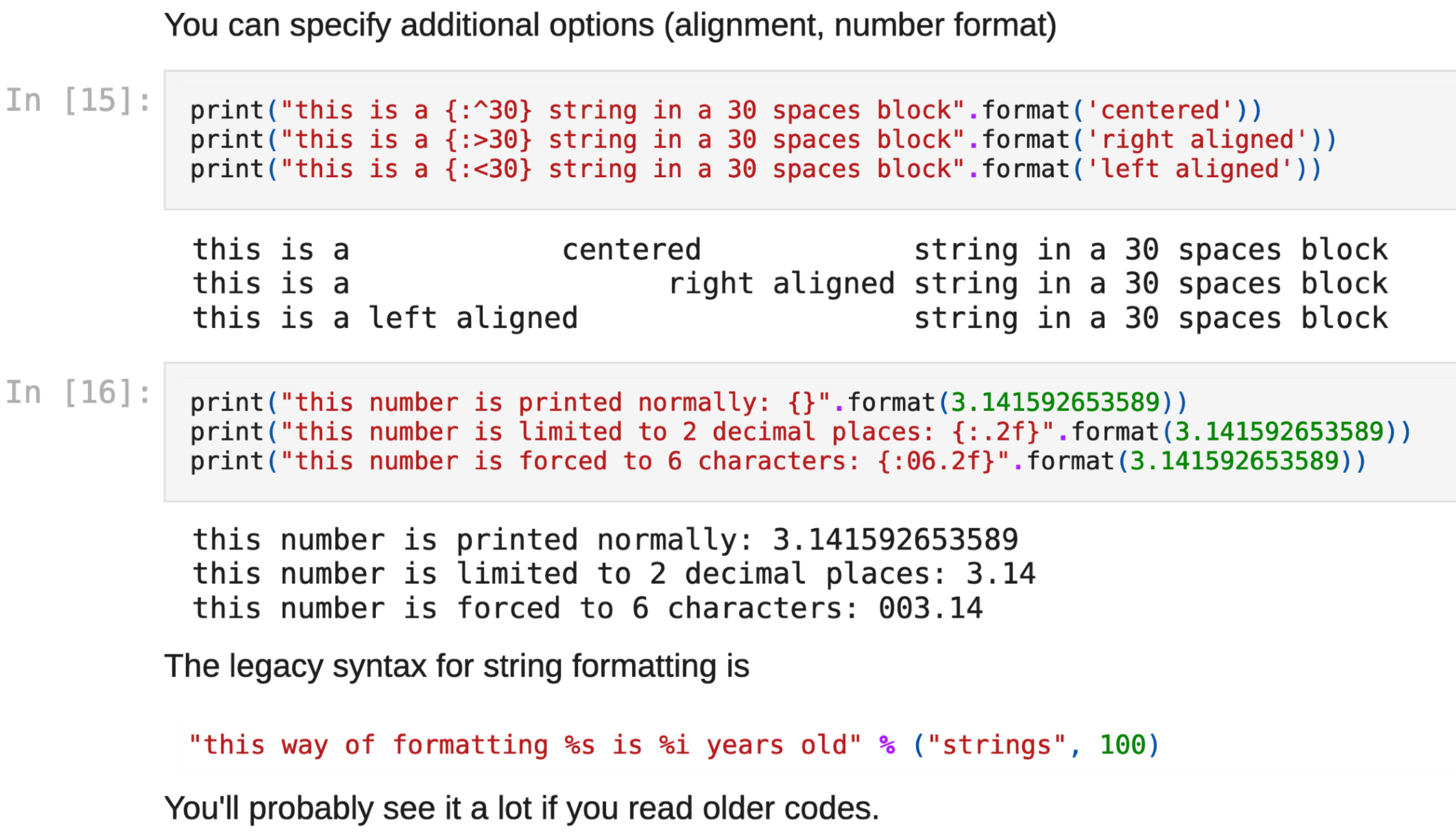
The python way, with dicts:

In [12]:

```
my_dict = {"animal":"cat", "status":"happy"}
out = my_string.format(**my_dict) #dict argument unpacking
print(out)
```

```
the cat is happy, really happy
```

Strings, final notes



You can specify additional options (alignment, number format)

In [15]:

```
print("this is a {:^30} string in a 30 spaces block".format('centered'))
print("this is a {:>30} string in a 30 spaces block".format('right aligned'))
print("this is a {:<30} string in a 30 spaces block".format('left aligned'))
```

```
this is a            centered            string in a 30 spaces block
this is a                  right aligned string in a 30 spaces block
this is a left aligned                   string in a 30 spaces block
```

In [16]:

```
print("this number is printed normally: {}".format(3.141592653589))
print("this number is limited to 2 decimal places: {:.2f}".format(3.141592653589))
print("this number is forced to 6 characters: {:06.2f}".format(3.141592653589))
```

```
this number is printed normally: 3.141592653589
this number is limited to 2 decimal places: 3.14
this number is forced to 6 characters: 003.14
```

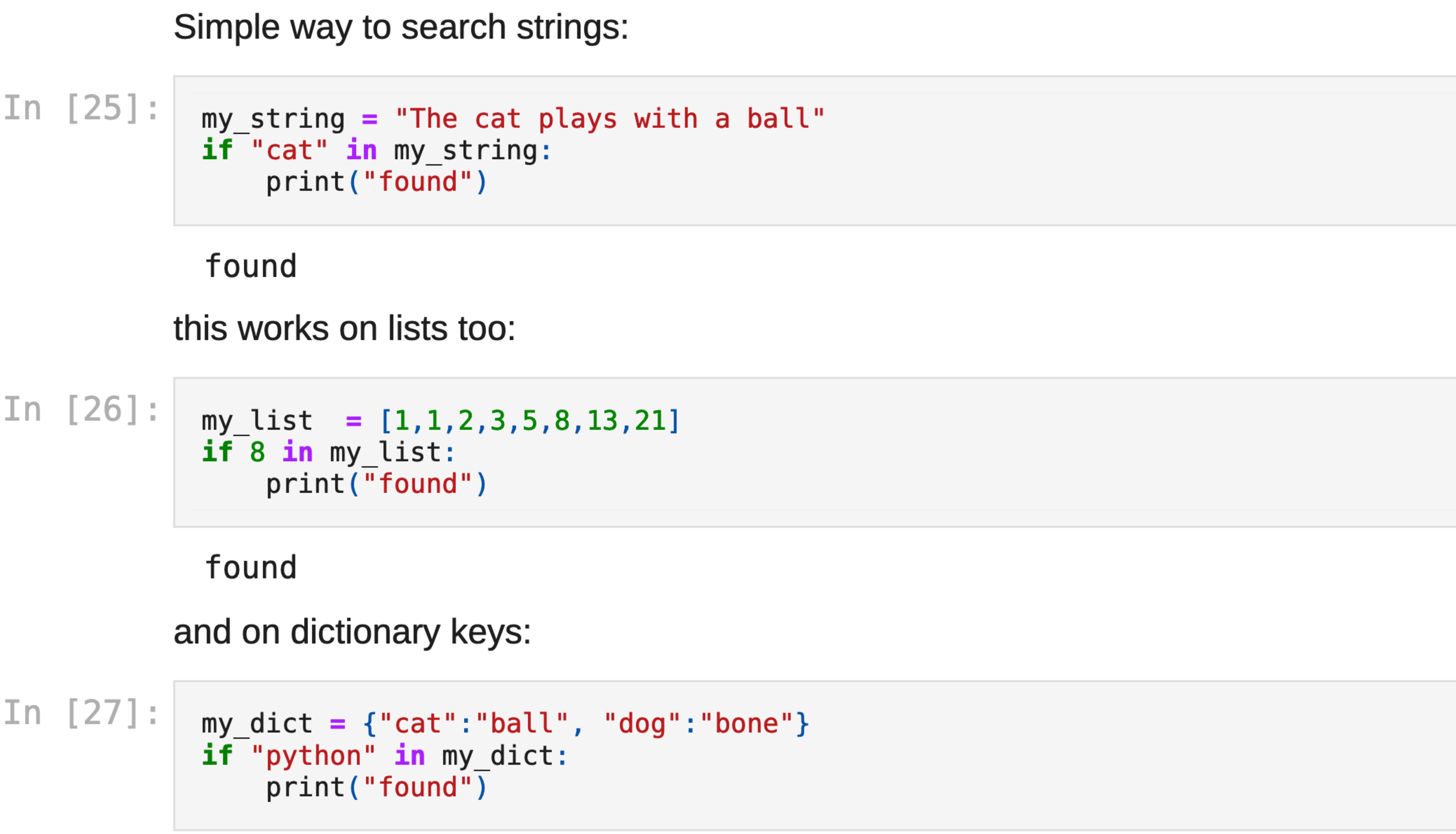
The legacy syntax for string formatting is

```
"this way of formatting %s is %i years old" % ("strings", 100)
```

You'll probably see it a lot if you read older codes.

Now, you know Python!
Let me present some cool stuff!

Favorite features I



Simple way to search strings:

```
In [25]:
my_string = "The cat plays with a ball"
if "cat" in my_string:
    print("found")
```

```
found
```

this works on lists too:

```
In [26]:
my_list  = [1,1,2,3,5,8,13,21]
if 8 in my_list:
    print("found")
```

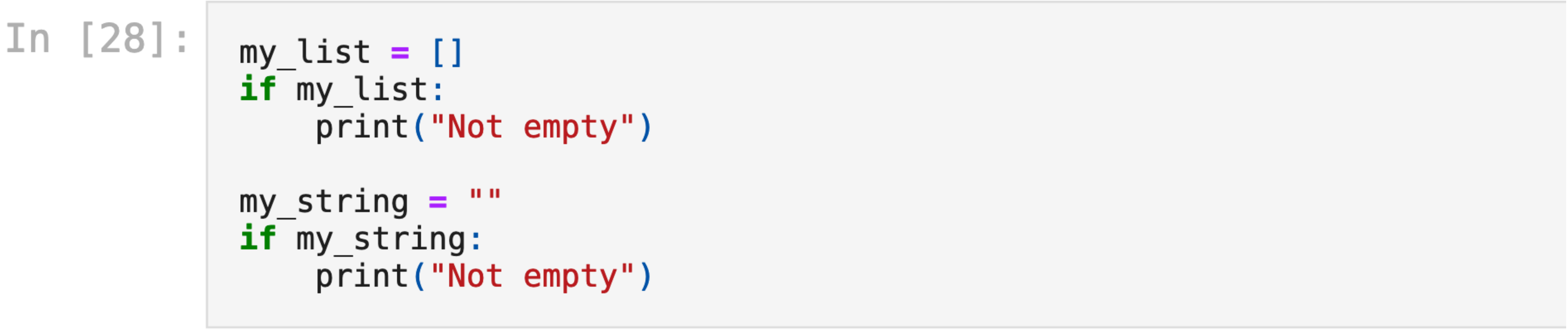
```
found
```

and on dictionary keys:

```
In [27]:
my_dict = {"cat":"ball", "dog":"bone"}
if "python" in my_dict:
    print("found")
```

Favorite Features 2

- Everything is True or False:

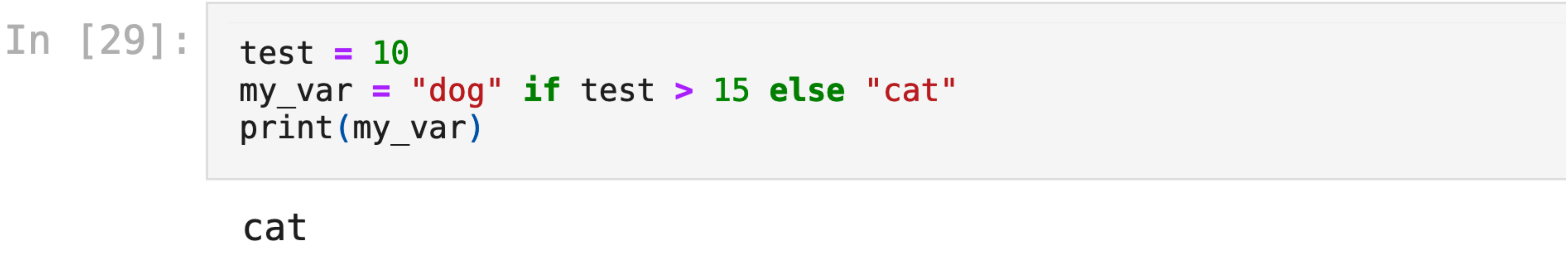


In [28]:

```python
my_list = []
if my_list:
    print("Not empty")

my_string = ""
if my_string:
    print("Not empty")
```

In general, empty iterables are False, non-empty are True

- The useful and very readable ternary operator:

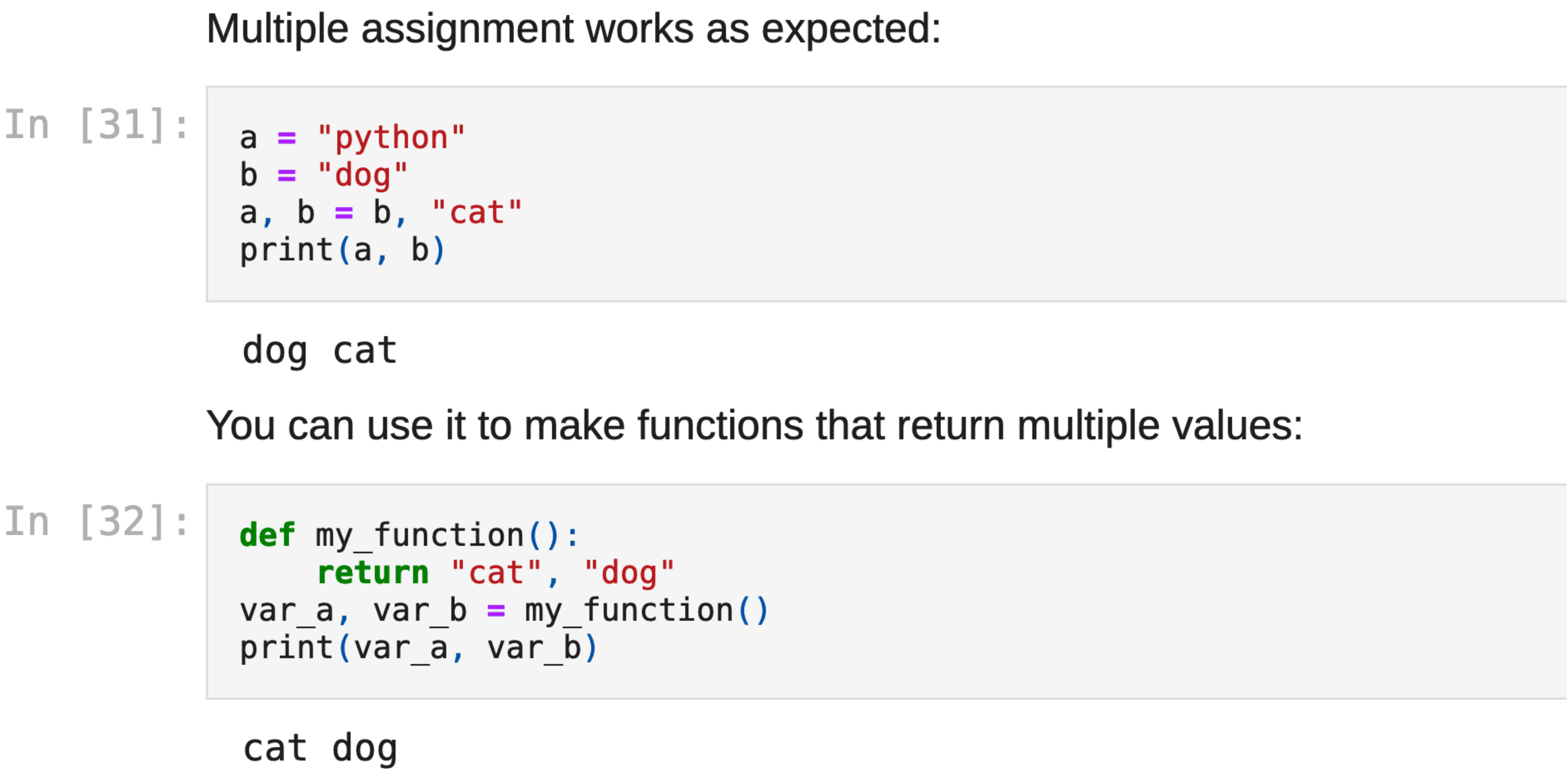


In [29]:

```python
test = 10
my_var = "dog" if test > 15 else "cat"
print(my_var)
```

```
cat
```

Favorite Features 3



Multiple assignment works as expected:

In [31]:

```python
a = "python"
b = "dog"
a, b = b, "cat"
print(a, b)
```

```
dog cat
```

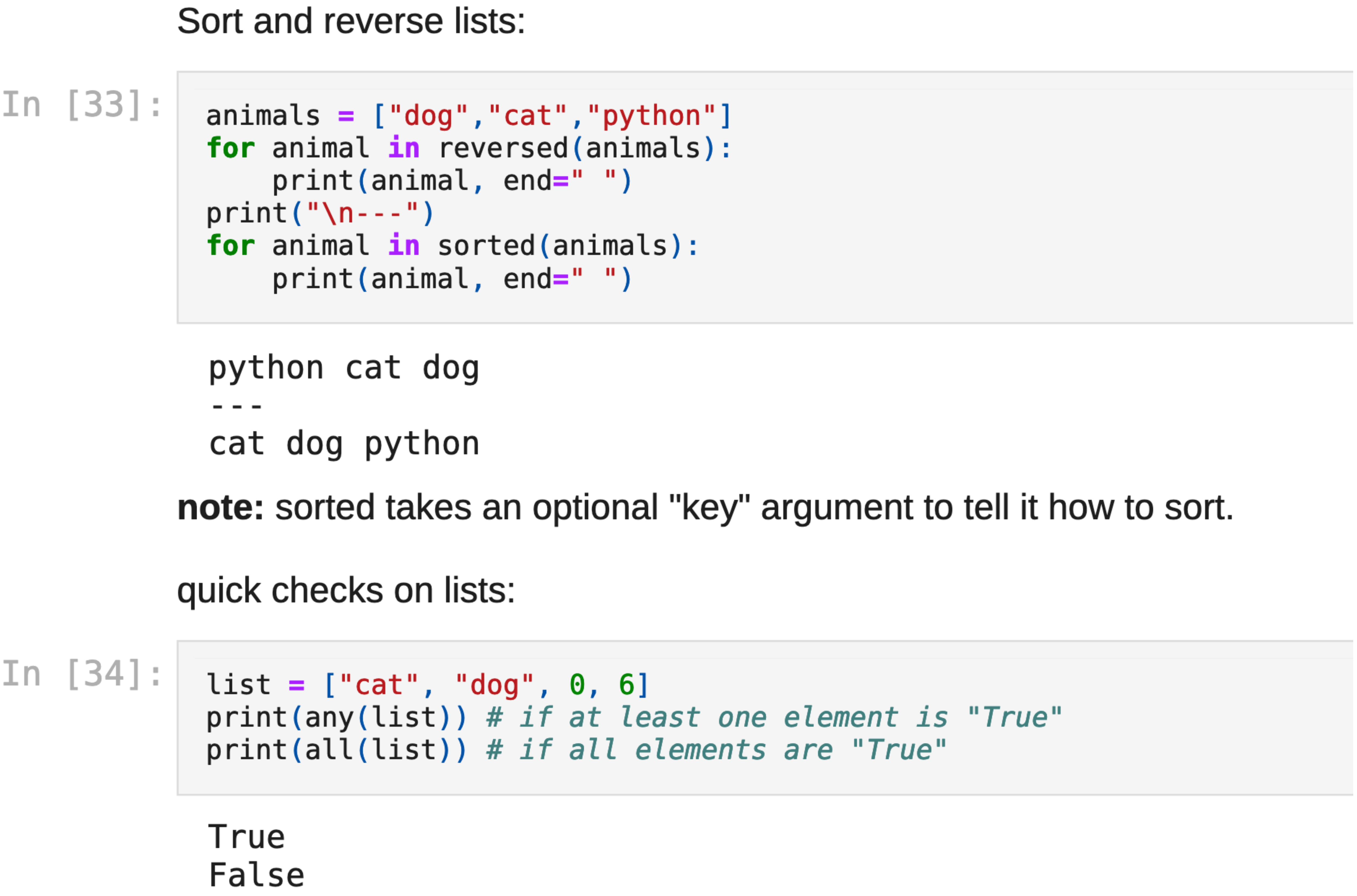
You can use it to make functions that return multiple values:

In [32]:

```python
def my_function():
    return "cat", "dog"
var_a, var_b = my_function()
print(var_a, var_b)
```

```
cat dog
```

Favorite Features 4



Sort and reverse lists:

In [33]:

```
animals = ["dog","cat","python"]
for animal in reversed(animals):
    print(animal, end=" ")
print("\n---")
for animal in sorted(animals):
    print(animal, end=" ")
```

```
python cat dog
---
cat dog python
```

note: sorted takes an optional "key" argument to tell it how to sort.

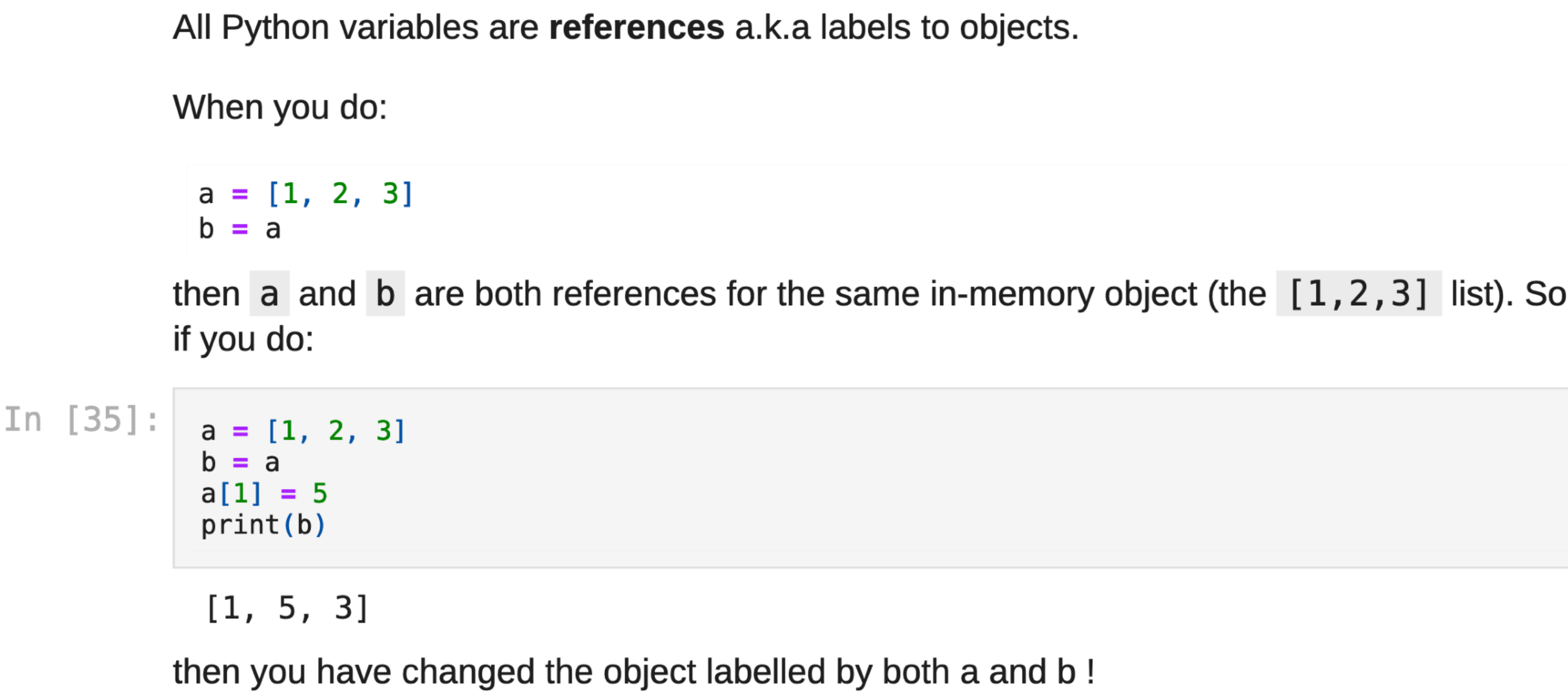
quick checks on lists:

In [34]:

```
list = ["cat", "dog", 0, 6]
print(any(list)) # if at least one element is "True"
print(all(list)) # if all elements are "True"
```

```
True
False
```

Python has “funny” behaviour



All Python variables are **references** a.k.a labels to objects.

When you do:

```
a = [1, 2, 3]
b = a
```

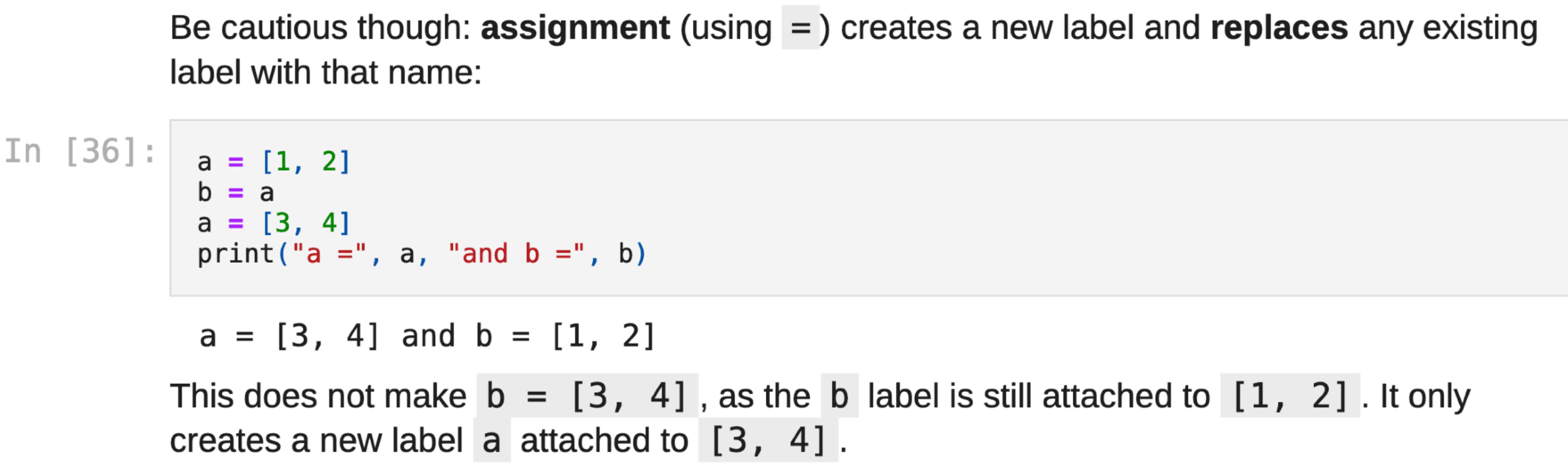
then `a` and `b` are both references for the same in-memory object (the `[1,2,3]` list). So if you do:

```
In [35]:

a = [1, 2, 3]
b = a
a[1] = 5
print(b)
```

```
[1, 5, 3]
```

then you have changed the object labelled by both a and b !

Python variables



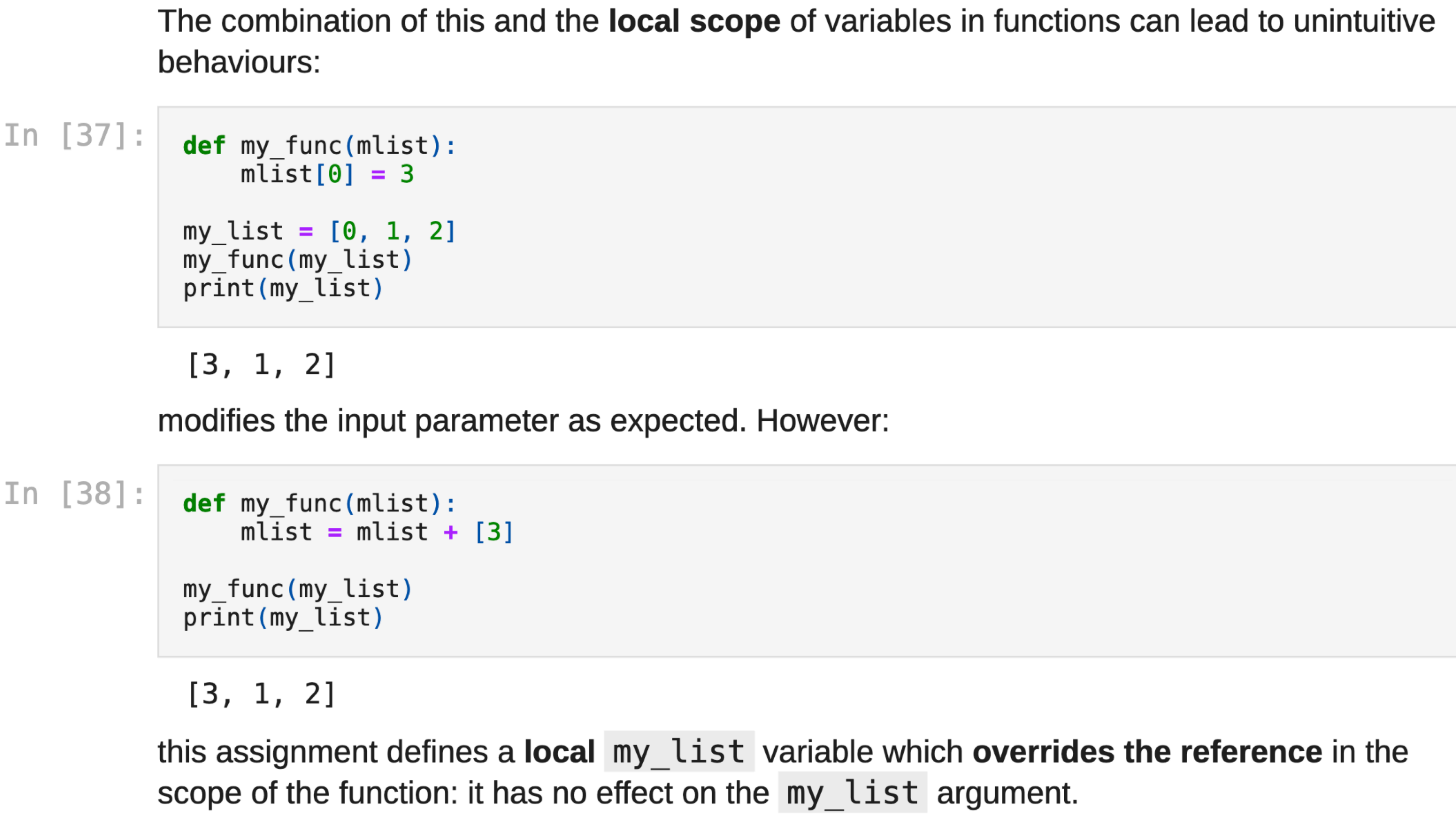
Be cautious though: **assignment** (using `=`) creates a new label and **replaces** any existing label with that name:

In [36]:
```
a = [1, 2]
b = a
a = [3, 4]
print("a =", a, "and b =", b)
```

```
a = [3, 4] and b = [1, 2]
```

This does not make `b = [3, 4]`, as the `b` label is still attached to `[1, 2]`. It only creates a new label `a` attached to `[3, 4]`.

Python variables: pitfalls



The combination of this and the **local scope** of variables in functions can lead to unintuitive behaviours:

In [37]:

```
def my_func(mlist):
    mlist[0] = 3

my_list = [0, 1, 2]
my_func(my_list)
print(my_list)
```

```
[3, 1, 2]
```

modifies the input parameter as expected. However:

In [38]:

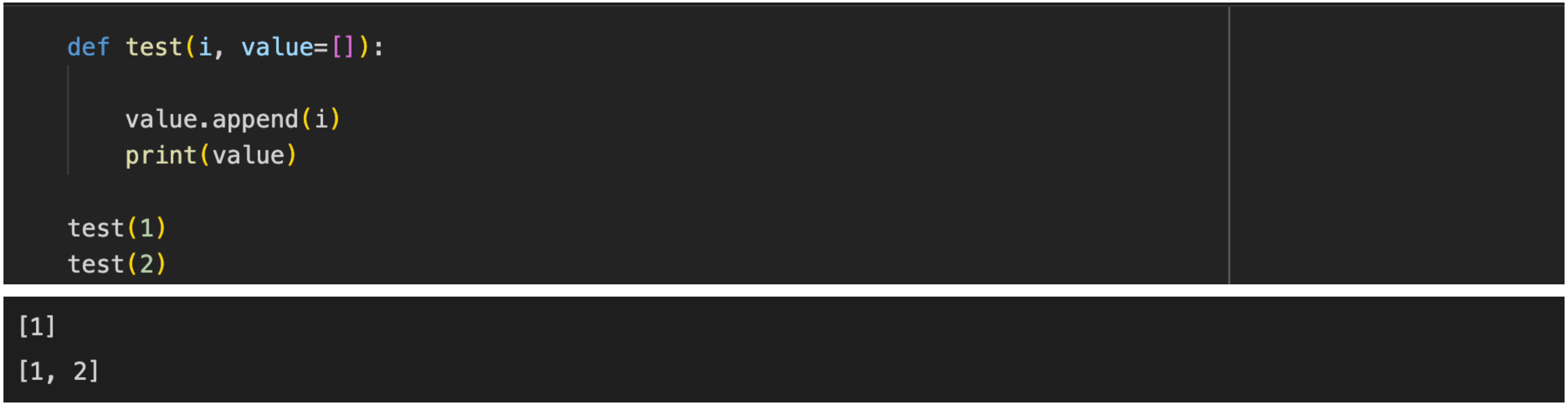
```
def my_func(mlist):
    mlist = mlist + [3]

my_func(my_list)
print(my_list)
```

```
[3, 1, 2]
```

this assignment defines a **local** `my_list` variable which **overrides the reference** in the scope of the function: it has no effect on the `my_list` argument.

Non immutable default value

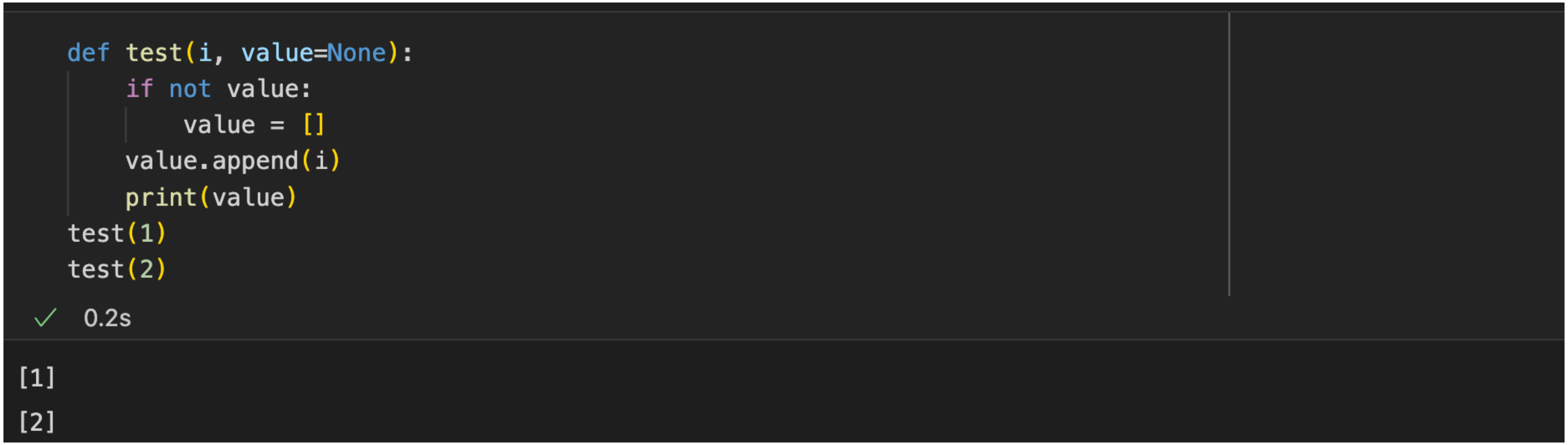
- The default value is not reset after each function call



```python
def test(i, value=[]):

    value.append(i)
    print(value)

test(1)
test(2)
```

```
[1]
[1, 2]
```

- Rather use:



```python
def test(i, value=None):
    if not value:
        value = []
    value.append(i)
    print(value)
test(1)
test(2)
```

✓ 0.2s

```
[1]
[2]
```

COOL but I need …

Random number

Parser (like csv, ini file,…)

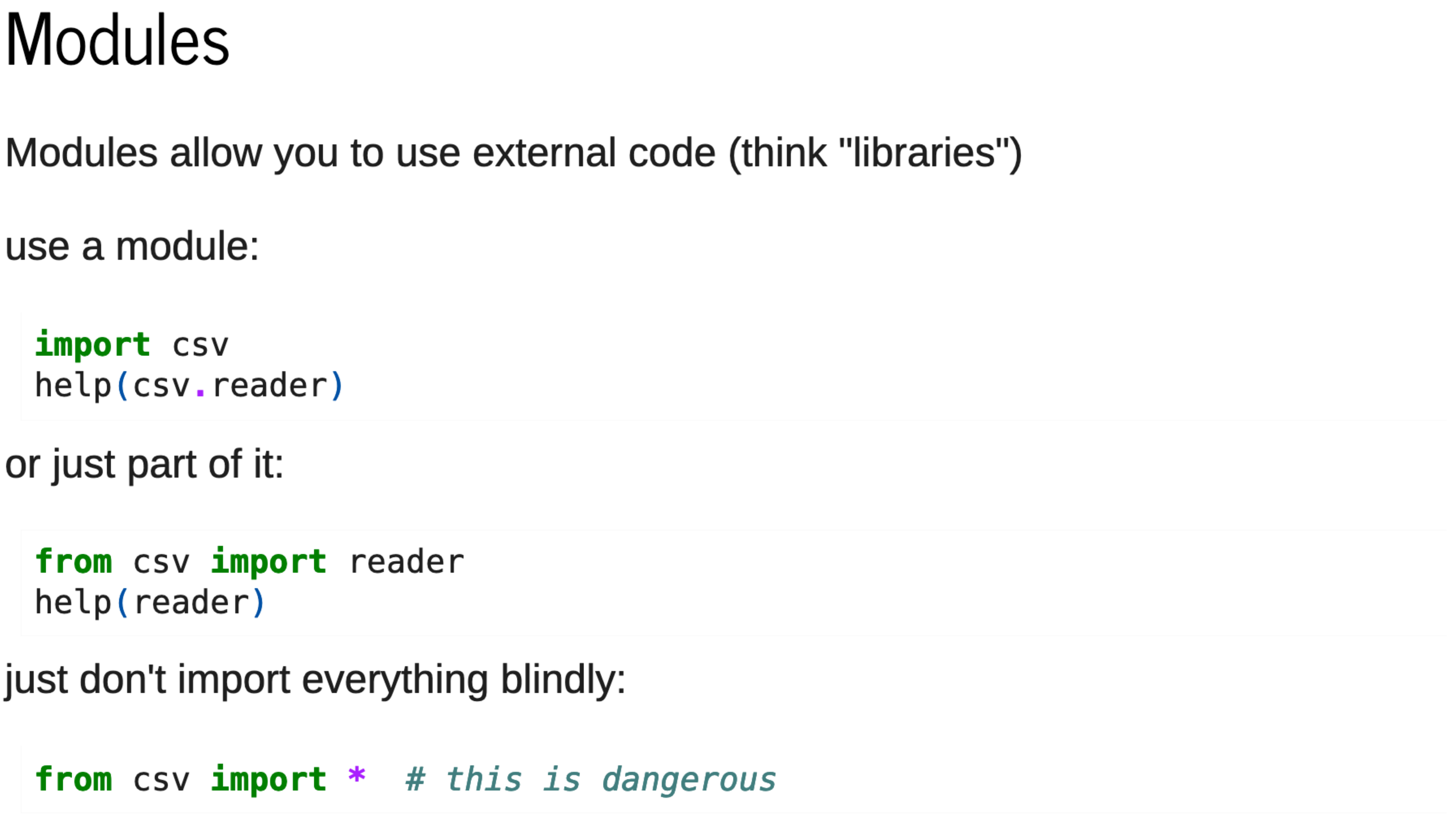
Iterators,

Efficient numerical computation,

Symbolic computation,

Plot

… (name it)



Modules

Modules allow you to use external code (think "libraries")

use a module:

```python
import csv
help(csv.reader)
```

or just part of it:

```python
from csv import reader
help(reader)
```

just don't import everything blindly:

```python
from csv import *  # this is dangerous
```

Module example : csv

csv is a **core module**: it is distributed by default with Python

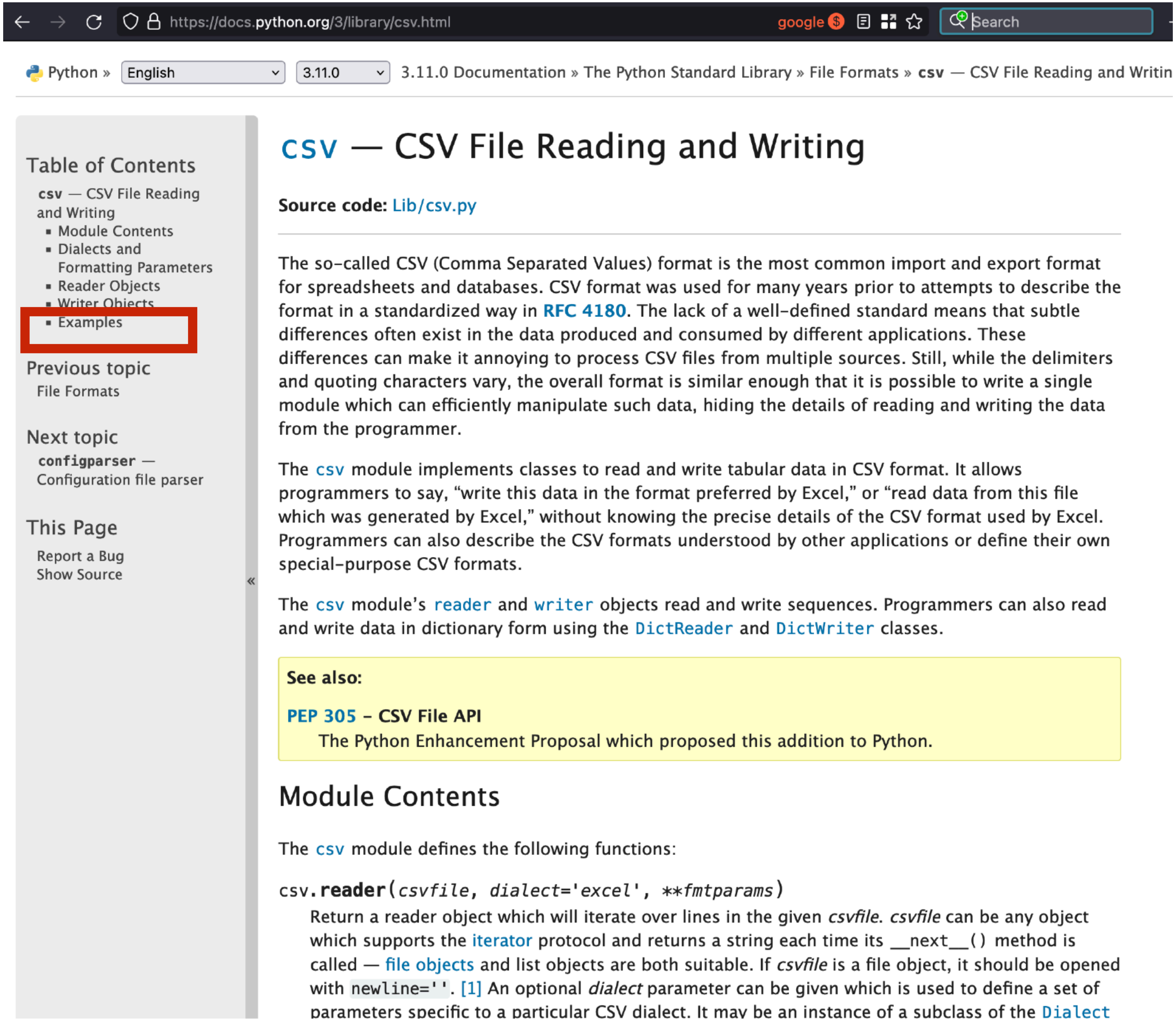
In [39]:

```python
import csv
with open('my_file.csv') as csvfile:
    reader = csv.DictReader(csvfile)
    for row in reader:
        print("row:", row)
        print("the {animal} plays with a {toy}".format(**row))
```

```
row: {'animal': 'dog', 'toy': 'bone'}
the dog plays with a bone
row: {'animal': 'cat', 'toy': 'ball'}
the cat plays with a ball
```

- `DictReader` is an object from the csv package
- `reader` is an iterator built by DictReader
- `reader` gives dictionaries, for instance `{"animal":"dog", "toy":"bone"}` and affects them to the `row` reference
- keys names are taken from the first line of the csv file

See Documentation



https://docs.python.org/3/library/csv.html

3.11.0 Documentation » The Python Standard Library » File Formats » **csv** — CSV File Reading and Writin

Table of Contents

csv — CSV File Reading and Writing
- Module Contents
- Dialects and Formatting Parameters
- Reader Objects
- Writer Objects
- Examples

Previous topic

File Formats

Next topic

configparser — Configuration file parser

This Page

Report a Bug
Show Source

`csv` — CSV File Reading and Writing

Source code: Lib/csv.py

The so-called CSV (Comma Separated Values) format is the most common import and export format for spreadsheets and databases. CSV format was used for many years prior to attempts to describe the format in a standardized way in **RFC 4180**. The lack of a well-defined standard means that subtle differences often exist in the data produced and consumed by different applications. These differences can make it annoying to process CSV files from multiple sources. Still, while the delimiters and quoting characters vary, the overall format is similar enough that it is possible to write a single module which can efficiently manipulate such data, hiding the details of reading and writing the data from the programmer.

The `csv` module implements classes to read and write tabular data in CSV format. It allows programmers to say, “write this data in the format preferred by Excel,” or “read data from this file which was generated by Excel,” without knowing the precise details of the CSV format used by Excel. Programmers can also describe the CSV formats understood by other applications or define their own special-purpose CSV formats.

The `csv` module’s `reader` and `writer` objects read and write sequences. Programmers can also read and write data in dictionary form using the `DictReader` and `DictWriter` classes.

See also:

PEP 305 – CSV File API
The Python Enhancement Proposal which proposed this addition to Python.

Module Contents

The `csv` module defines the following functions:

`csv.`**`reader`**`(`*`csvfile, dialect='excel', **fmtparams`*`)`

Return a reader object which will iterate over lines in the given *csvfile*. *csvfile* can be any object which supports the iterator protocol and returns a string each time its `__next__()` method is called — file objects and list objects are both suitable. If *csvfile* is a file object, it should be opened with `newline=''`. [1] An optional *dialect* parameter can be given which is used to define a set of parameters specific to a particular CSV dialect. It may be an instance of a subclass of the `Dialect`

Interacting with the OS and filesystem:

- sys:
  - provides access to arguments (argc, argv), useful sys.exit()
- os:
  - access to environment variables
  - navigate folder structure
  - create and remove folders
  - access file properties
- glob:
  - allows you to use the wildcards * and ? to get file lists
- argparse:
  - easily build command-line arguments systems
  - provide script usage and help to user

Enhanced versions of good things

- itertools: advanced iteration tools
  - cycle: repeat sequence ad nauseam
  - chain: join lists or other iterators
  - compress: select elements from one list using another as filter
  - …
- collections: smart collections
  - defaultDict: dictionary with default value for missing keys (powerful!)
  - Counter: count occurrences of elements in lists
  - ...
- re: regular expressions
  - because honestly "in" is not always enough

Utilities

- copy:
  - sometimes you don't want to reference the same object with a and b
- time:
  - manage time and date objects
  - deal with timezones and date/time formats
  - includes time.sleep()
- pickle:
  - allows to save any python object as a string and import it later
- json:
  - read and write in the most standard data format on the web
- requests:
  - access urls, retrieve remote files

Basics for science

- numpy:
    - linear algebra
    - fast treatement of large sets of numbers
- matplotlib:
    - standard library for plotting
- scipy:
    - optimization
    - integration
    - differential equations
    - statistics
    - ...
- pandas:
    - data analysis

Installing modules

The standard package manager is **pip**:

- Search for a package:

```
pip search BeautifulSoup # famous html parser
```

- Install a package:

```
pip install BeautifulSoup  # use "--user" to install in home
```

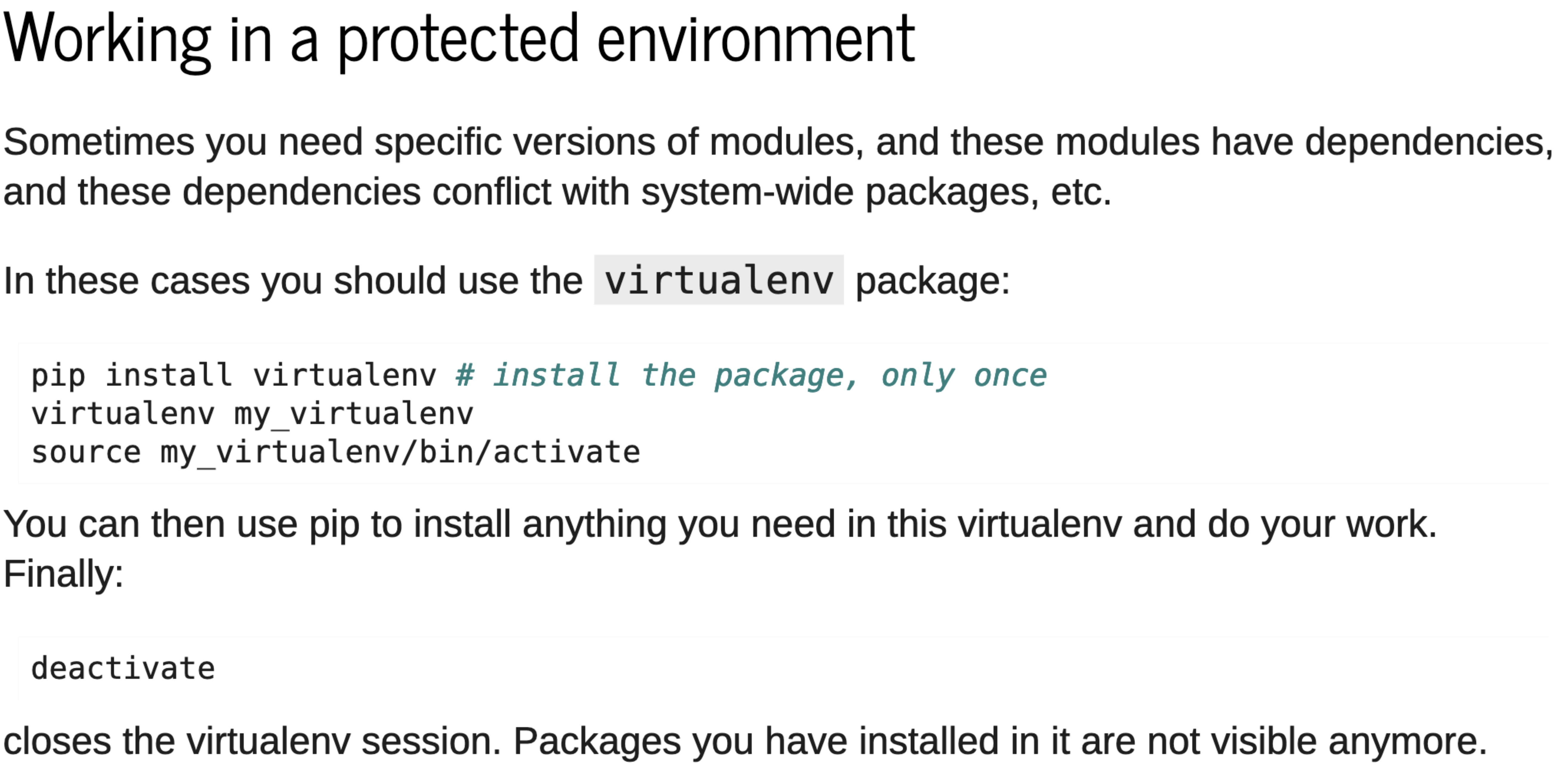
- Upgrade to latest version:

```
pip install --upgrade BeautifulSoup
```

- Remove a package:

```
pip uninstall BeautifulSoup
```

Dependencies nightmare



Working in a protected environment

Sometimes you need specific versions of modules, and these modules have dependencies, and these dependencies conflict with system-wide packages, etc.

In these cases you should use the `virtualenv` package:

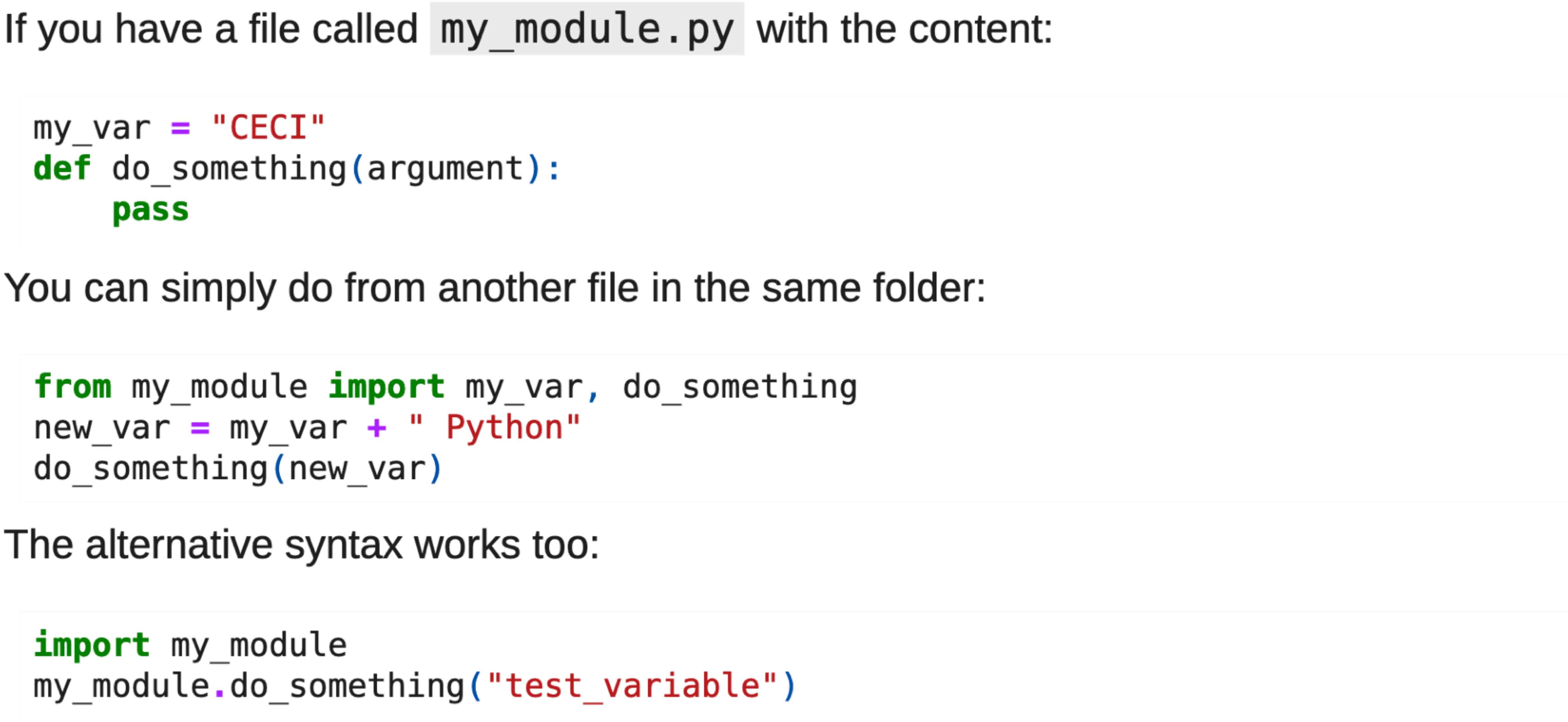
```
pip install virtualenv # install the package, only once
virtualenv my_virtualenv
source my_virtualenv/bin/activate
```

You can then use pip to install anything you need in this virtualenv and do your work. Finally:

```
deactivate
```

closes the virtualenv session. Packages you have installed in it are not visible anymore.

Python files are modules



If you have a file called `my_module.py` with the content:

```
my_var = "CECI"
def do_something(argument):
    pass
```

You can simply do from another file in the same folder:

```
from my_module import my_var, do_something
new_var = my_var + " Python"
do_something(new_var)
```

The alternative syntax works too:

```
import my_module
my_module.do_something("test_variable")
```

Importing scripts

You know you can import any file as a module. This allows to debug in the interpreter by using:

```python
import my_file
```

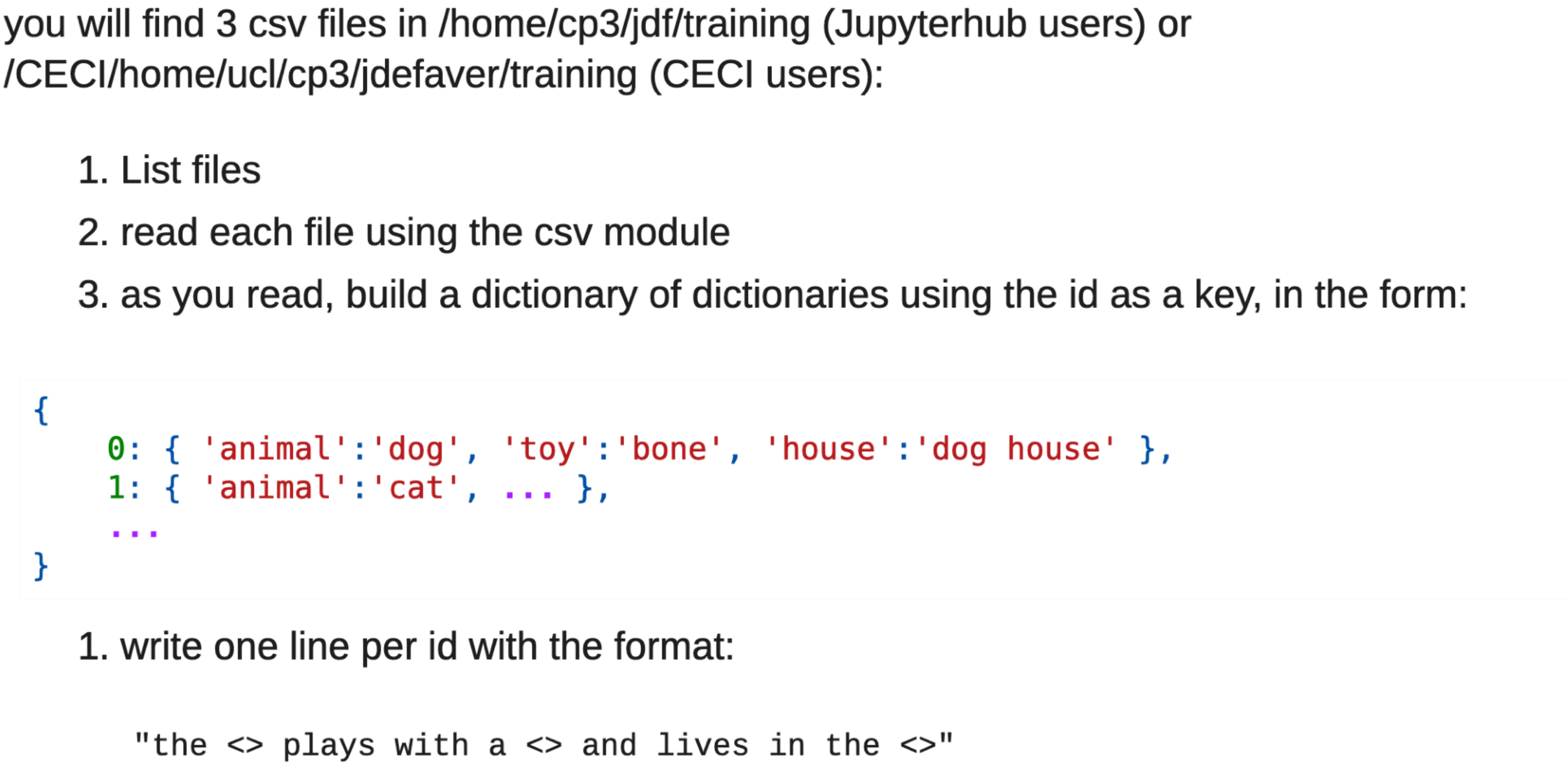
to access functions and objects. But doing this runs the whole content of `my_file.py` which is not what you want.

You can avoid that by putting the code to be executed only when the script is run (not imported) inside a block like this:

```python
def my_function():
    ...

if __name__ == '__main__': # that's two underscores
    print(my_function())   # put main code here
```

That way the "print" will not be called when you import my_file, only when you run `python my_file.py`

Exercise



you will find 3 csv files in /home/cp3/jdf/training (Jupyterhub users) or /CECI/home/ucl/cp3/jdefaver/training (CECI users):

1. List files
2. read each file using the csv module
3. as you read, build a dictionary of dictionaries using the id as a key, in the form:

```
{
    0: { 'animal':'dog', 'toy':'bone', 'house':'dog house' },
    1: { 'animal':'cat', ... },
    ...
}
```

1. write one line per id with the format:

```
"the <> plays with a <> and lives in the <>"
```