Motivation

The Age of Digital Astronomy

The transition from an analog to a digital world in the second half of the 20th century did not skip the field of astronomy. The *charge-coupled device* (CCD) has replaced the human eye and the photomultiplier tube as the recording device at the lower end of the telescope. Observations taken in Chile can be examined in France during the same night.

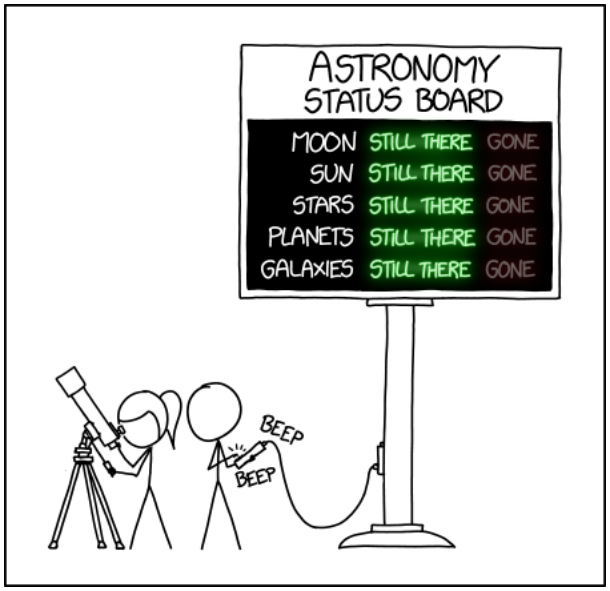

Figure 1: Astronomy reduced to the realm of booleans. Credit: xkcd

In addition to the now digital nature of astronomical observations, there is also a large increase in the amount of data. The next-generation of all-sky surveys will produce several terabytes of data per night each, such as the *Legacy Survey of Space and Time* (LSST) at the Vera C. Rubin observatory.

The field of "big data" brings new challenges to the profession of astronomer which require a completely new skillset: digital data processing and analysis. Astronomers in general do *not* have a computer science qualification. Nevertheless, the day-to-day work is often coding, and astronomers end up learning to code on their own.

This course aims to support you on your road to learn coding in general, and python in particular, by giving you an overview of what's ahead, how to progress efficiently, and potential shortcuts and dead-ends along the path. The lessons it contains come from experiences that I have made over my first 10 years of learning python on my own. It may seem daunting at first, but I assure you, it's an enjoyable journey.

Learn to code

Among the typical day-to-day work of an astronomer are writing publications and writing code for your own research. Meaning: Most of the time is spent programming.

Being skilled in a programming language makes your daily life easier and more enjoyable. In addition, the enjoyment you get out of programming increases with your skill. Remember, you code every day.

Once you know how to code, you want to learn how to do it properly. Learning to code is not a single activity, but it takes years and continuous throughout your career. To stay efficient, you have to keep up with the developments in your coding language.

Finally, coding is fun.

Learn to code python

There are many programming languages out there: Fortran, C, C#, C++, JavaScript, IDL, R, python, Julia, Go, and more. You can achieve the same tasks with most of them but some are better suited for certain tasks than others.

Here are some subjective reasons to choose python as your language of choice:

1. python is easy to read and write, which saves you time.

```python
colours = ["blue", "black", "red"]

for colour in colours:
    print(f"My favourite colour is {colour}")

if "yellow" in colours:
    print("Yellow is one of my favourite colours.")
```

2. There is a rich package ecosystem: most of the work has been done for you already.

   You want to analyse the data in a CSV file? The pandas package helps you out. You are observing stars and want to extract their absolute magnitudes from FITS images? The photutils package has functions for this. You have to convert the equatorial coordinates of a source to galactic ones while precessing the equinox of the coordinate frame from J1975 to J2000? Try astropy.coordinates.

3. It has a large user base, meaning it will continue to progress and develop.

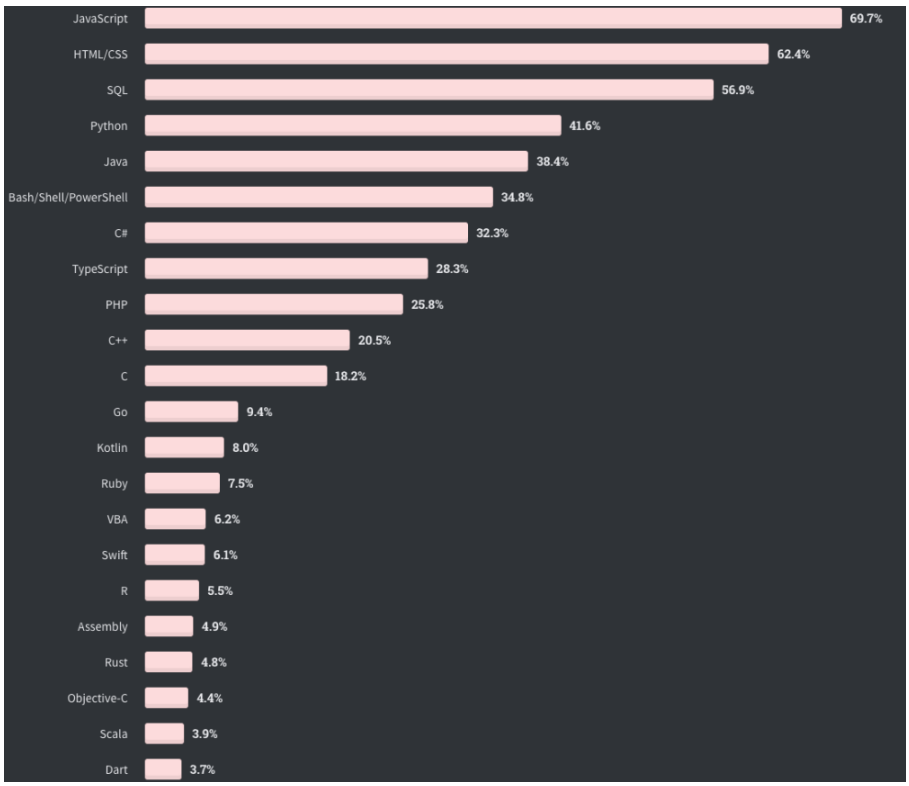

Figure 2: Results of the 2020 *StackOverflow Developer Survey* on the most used programming languages by their user base (65,000 responses, multiple answers possible).

1. It's free. It's accessible. All the code you have can be inspected on your computer.

A common criticism of python is that it is slow compared to languages like FORTRAN, C, C++. This is objectively correct and can be understood when comparing the inner workings of the languages. For me, this is secondary: my time is more valuable than the CPU time. If I can write code quicker, I don't mind if it takes longer to execute. And it's really not that slow.

Scope of this Course

Learning to code in general and a language in particular takes a lot of time and effort. The aim of this course is therefore not to teach you how to code in python. Instead, I want to provide you with an overview of what there is to learn, what to focus on in the beginning, and how to learn efficiently *on your own*.

In the next sections, we will cover:

- A minimal introduction to python. We need a basic vocabulary to talk about aspects of the

language.
- python on your system: what is installed? How do you execute python files?
- When things go wrong: understanding error messages.
- The python standard library
- Highlights of third-party python packages
- Tools for coding: editors, jupyter notebooks, ipython
- Best Practices for coding python
- Some challenging exercises

Not included are

- How to code python: this is not possible in 3h, though learning the syntax is not difficult.
- How to navigate the command line, tools like awk, grep, sed. This is an optional skillset which I recommend but you can do without for now.

A minimal introduction to python

This course does not intend to teach you python syntax. Nevertheless, we will need a basic vocabulary when discussing aspects of the language, so here is a brief overview of the most minimal python basics.

What is python?

There are three key properties to define python. They may not influence the way you code in the beginning but it's good to know about them anyway.

1. python is an interpreted language: Each line is executed as it is passed to the python interpreter. Lines at the top of a script are executed first. The alternative is a compiled language such as C++, where a compiler verifies the code before compiling it into an executable, which is then run by the user.
2. python is a high-level language: The programmer does not have to worry about common questions such as memory allocation and variable typing. This is one of the reasons why python is quick to learn.
3. python is object-oriented. This refers to the structure of python code, which revolves around building classes with exposed high-level interfaces while the inner workings are hidden. This is a conceptual difference to other programming languages which follow other paradigms.

Data Types

python allows you have to store data in variables and to run operations on these variables. Several *data types* are supported. A subset of these types are the following:

```
x = 4    # int    (integer)
y = 4.3  # float  (floating point value)
z = "hello world"  # str  (string)
today_is_thursday = True  # bool (boolean)
chars_and_numbers = ["a", "b", "c", "c", 1, 2, 3]  # list
phonebook = {"Alice": 61234567, "Bob": 68765432}  # dict (dictionary)
first_four_of_alphabet = {"a", "b", "c", "d"}  # set
```

To get the type of a variable, use the `type()` function:

```
>>> type(y)
<class 'float'>
```

The >>> notation means that this command is run the in the interactive python interpreter as opposed to in a python script. We will get to this in a bit.

A rather unique property of python is that *dynamic typing:* you can change the type of a variable at any point.

```
a = 3
a = "now I'm a string"
```

As mentioned above, this is one of the reasons that python is easy to write, you do not have to worry about *typing* your variables. A variable containing **`float`** can later contain a `dict` without any issue. On the other hand, this is one of the reasons python is slower to execute than other languages: the interpreter does not know what a variable contains before reading it.

Control Flow

Control flow refers to the order of execution of your python commands. The most basic two cases are the **`if`**-clause and the **`for`**-loop.

The **`if`**-clause is executed if its condition is met. The **`else`**-clause is an optional appendix which is executed if the condition is not met.

```
today_is_thursday = True

if today_is_thursday:
    print("Today is not Friday.")
else:
    print("Today might be Friday.")
```

Note the syntax here: the two clauses (**if** and **else**) are terminated with colons. To mark the span of the **if**-clause, indentation is used (other languages wrap the code to be executed into { and } for example). The indentation has to be a multiple of 2 spaces (4 being the most common case) and it has to be consistently used throughout the script.

```
today_is_thursday = True

if today_is_thursday:
    print("Today is not Friday.")

else:
    print("Today might be Friday.")

    print("I will go to the beach.")  # executed if today_is_thursday
       is False

print("I will study some python.")    # always executed
```

The second basic control flow method is the **for** loop. A common pattern is to iterate over the elements in a `list`.

```
weekdays = ["monday", "tuesday", "wednesday", "thursday", "friday"]

for day in weekdays:

    if day == "tuesday":
        print("Today is Tuesday.")

    else:
        print(f"Today is not Tuesday. It is {day}.")
```

Over the execution of the **for**-loop, the `day` variable stores the value of the elements of the list one-by-one. The last line shows you how to include a variable value into a string, using the `f-string` syntax.

Where to continue

Here are some resources you can use to learn `python` syntax.

- Official Tutorial: Extensive, often too detailed for a beginner
- Codecademy's course on python3: Interactive, step-by-step guide to the language. This is how I learnt the basics.
- Automate the Boring Stuff with Python: Popular beginner's guide to the language

A great resource for learning more about the inner workings of python and keeping up-to-date with developments are the python conferences (called "PyCon"s) taking place all over the world. The talks are regularly uploaded on various youtube channels such as this one from PyCon US.

python on your system

python2 versus python3

When researching python, you may still come across documentation which refers to python2. The 2 refers to the major version of the python distribution which is used. python2 and python3 code are different and incompatible in some key areas (such as the print() function). Do not bother with any documentation referring to python2: it is outdated and no longer supported (end-of-life was 01/01/2020). In fact, when I started coding python ten years ago, I already started with python3.

python on your system

On the laptops provided by the university, you will have python installed. To check which version is installed, open a command line and execute:

```
1 $ python3 --version
```

Note: do not type the $, it is a common way to indicate that a command is to be run in the command line.

This will echo the python version installed on your system. I like to stay on the latest stable release (currently 3.9). I would recommend you run a python version >=3.7. Note that you may have several versions of python installed on your system. Each version has it's specific executable à la python3.7, python3.8. Only one version will be linked ("aliased") to the python3 and python executables. To avoid confusion, on system with multiple python versions installed, I use the full python executable name to ensure that I run the intended version. In these notes, I assume that python3 is the executable we want to launch.

python scripts usually carry the .py suffix. When executing a script, python may create a directory called __pycache__/, which serves to speed up the execution in later runs. Feel free to leave it or delete it.

Executing python

As mentioned above, python is an interpreted language. This has two practical implications:

1. python will execute your code and stop the execution if it finds an error. Compiled languages will first validate the code and then run it.
2. python has an interactive interpreter, where each line is execute as you enter it.

There are two ways to run python code on your system:

1. Launch the interactive interpreter

```
$ python3
```

1. Run a python script by passing it's filepath to the python command

```
$ python3 my_script.py
```

Packages and modules

python has a lot of built-in functionality such as mathematical functions. These functions are separated into *modules*. Modules are python files which you can find and open on your own system. To use a function from a certain module, you have to **import** it.

```
>>> import math
```

Once you have imported a module, you can access the python objects it provides through the *dot notation*. Do not worry if you do not understand the syntax for now.

```
>>> math.pi
3.141592653589793
>>> math.sin(math.pi / 2)
1.0
```

Modules may further group functions into submodules, which are again accessed via the dot notation.

```
>>> import os.path
```

python has a *standard library* of modules which come pre-installed with your python distribution, such as the math and os modules. To extend your library, you can install third-party *packages* (not necessarily developed by the python core developers). A package is a collection of modules. To install a package, you use pip ("pip installs packages"). To install pip itself, execute

```
$ wget https://bootstrap.pypa.io/get-pip.py  # download an installer
    script
$ python3 get-pip.py # execute the installer script
$ rm get-pip.py # delete the installer script
```

Now, to install a package, run

```
$ python3 -m pip install [package_name]
```

A good first package to install is the numpy package, which contains many mathematical and numerical calculation functions for python.

```
$ python3 -m pip install numpy
```

Ensure that this worked by running

```
>>> import numpy
```

If a module is missing on your system, python will raise the ModuleNotFoundError on the **import** line.

When things go wrong

Understanding the error message

One of the first things you will need to learn is to understand python error messages and how to resolve them. Fortunately, they are mostly straight-forward.

```
>>> 4 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

The last line contains the type of the error and a brief error message. Most of the time, the errors have descriptive names like the one above (e.g. FileNotFoundError). Quite common are the ValueError (passing a variable with an invalid value), the TypeError (passing a variable with an invalid type), and the IndexError (for example, trying to access the fifth element in a list with four entries).

The *traceback* is a rundown of the error's origin. At the top, it will include the filename of your script and the line where the error occurred. In the case above, we are using the interpreter, hence the "file" is the standard input, and the error occurred in line 1. When we use an actual script, the traceback will contain more information.

```
$ python3 my_script.py
Traceback (most recent call last):
  File "my_script.py", line 4, in <module>
    x = math.sqrt(-3)
ValueError: math domain error
```

If the function we call itself calls more functions and the error occurs in the one of the nested functions, the traceback will still start with the offending line in our script. It will then print in a top-down manner each function which was called and from where, concluding with line which caused the actual error.

If the offending function call spans over multiple lines, only the first line will be printed. If you have edited the script while it was running (which is totally fine), the wrong line may be printed.

Fixing the error

Many times, the description of the error will provide enough information to understand its origin. If you do not understand what is going wrong, here are some steps to resolve the issue.

1. In the interactive interpreter, run

```
>>> help(function_name)  # function_name e.g. math.sqrt
```

to open the documentation of the function. It might expect different arguments from what you thought.

1. Right before the offending line in your script, insert the `breakpoint()` command. This will cause the python interpreter to stop the execution of the script at this line and drop you into the interactive interpreter, with all variables in their current state saved. You can then run any valid python command to inspect the current state of your variables, e.g. by typing the variable names, you get the current values. These values might be different from what you thought. Use `n` to execute the next line of code, `c` to continue, and `q` to quit.
2. Googling the error message. Typically, someone has had this issue before you and was nice enough to post the solution to a page like StackOverflow.

The standard library

The *standard library* refers to the classes and functions which are distributed with 'python' and maintained by the 'python' core developers. They likely present the best solution to the problem they aim to solve, so in general, you should prefer the *standard library* over someone else's modules.

The python standard library comes equipped with many modules for system management, network management, debugging, and more.

Some modules and functions which I use frequently are given below.

- os

This modules contains many functions for interacting with your operating system. I use is most to e.g. get a list of files in a directory and interact with their filepaths.

```
>>> import os
>>> dir_home = os.path.expanduser("~/")  # get the path to the home directory
>>> os.listdir(dir_home)  # list the contents of the home directory
...
```

Also check out the `pathlib` module, which is a more modern (i.e. object-oriented) approach to filepath handling.

- `collections.Counter()`

  The `collections` module has other useful functionality, but this one deserves its honourary mention. Given an iterable (e.g. a `list`), it returns a `dict` with the elements as keys and the number of times they appear in the iterable as values.

```
>>> from collections import Counter
>>> zoo = ["monkey", "elephant", "giraffe", "monkey", "tiger", "elephant", "monkey", "lion"]
>>> Counter(zoo)
Counter({"monkey": 3, "elephant": 2, "giraffe": 1, "tiger": 1, "lion": 1})
```

- `time.time()`

  Get the current system time. Useful to time the execution of your scripts.

- `sys.exit()`

  Used to gracefully exit a script.

Third-party packages

The majority of things you program in the beginning have been programmed by others, and probably better. As an example, I wrote a FITS image parser and editor before I discovered the `astropy` package.

Here is a overview of the most important packages outside the standard library.

numpy

`numpy` is likely the most used third-party package in `python`. It adds new data types as as the array and implements many maths and computing functions. `numpy` completely replaces the `math` module

in python in most applications. Under the hood, numpy offers speed by vectorizing many computations automatically.

The numpy.array is ubiquitous and frequently used as replacement for the list, as they behave like vectors:

```
>>> import numpy as np
>>> vector1 = np.array([1, 2, 3])
>>> vector2 = np.array([1, 2, 3])
>>> vector1 + vector2      # arrays
array([2, 4, 6])
>>> [1, 2, 3] + [1, 2, 3]  # lists
[1, 2, 3, 1, 2, 3]
```

matplotlib

matplotlib is the go-to package for creating figures in python. The figures you create can range from quick-look plots to publication-ready works-of-art. Watch out: creating and perfectioning figures is addictive.

Definitely check out the cheatsheets on their GitHub repository, made with matplotlib and LaTeX .

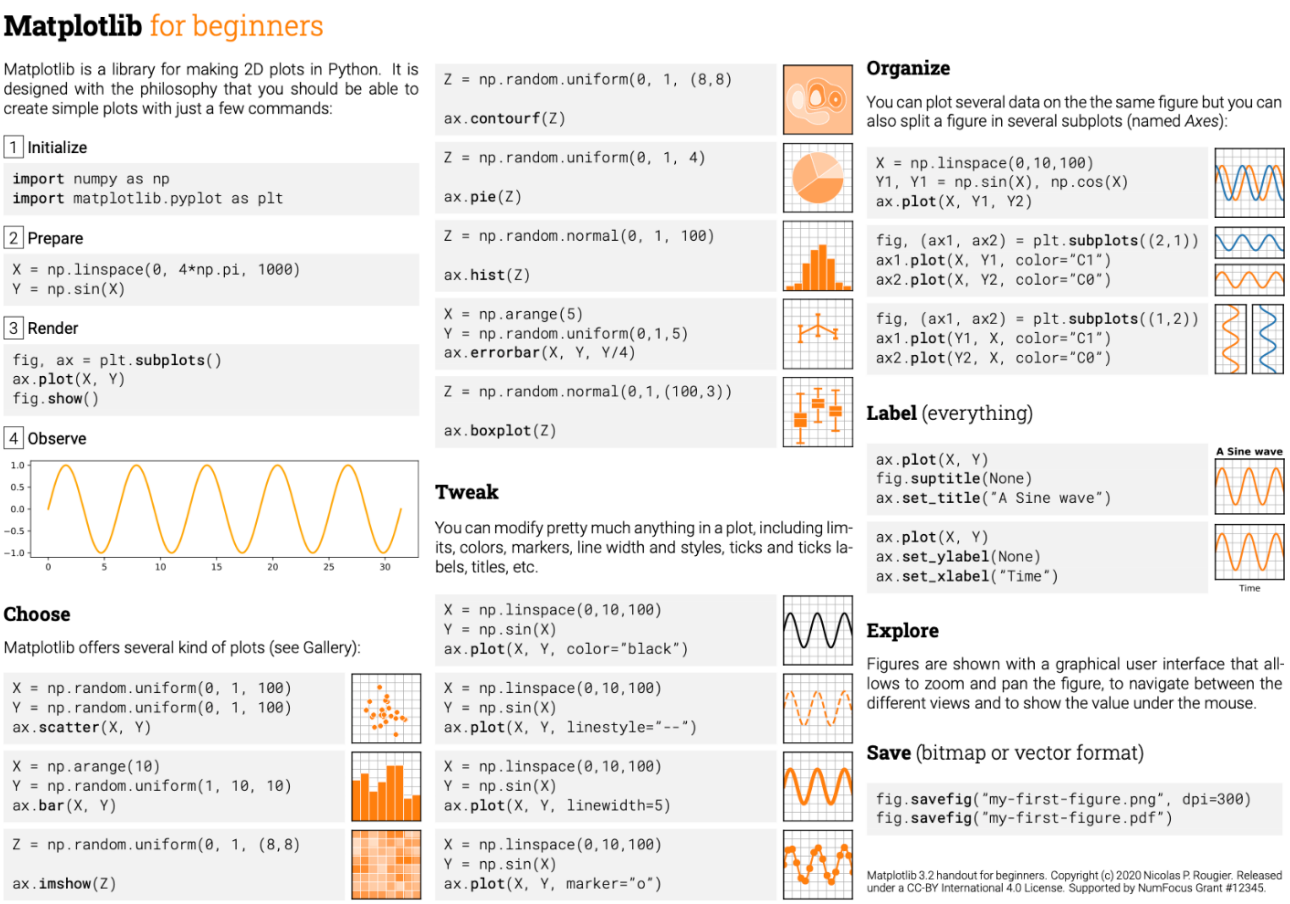

Figure 3: The *matplotlib for beginners* guide from Nicolas P. Rougier, one of several fantastic `matplotlib` cheat sheets.

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> x = np.linspace(0, 2*np.pi, 100)  # get 100 points evenly spaced
    between 0 and 2 pi
>>> plt.plot(x, np.sin(x), color="blue", label="Sine")
>>> plt.plot(x, np.cos(x), color="red", label="Cosine")
>>> plt.legend()
>>> plt.show()
```

pandas

`pandas` is useful whenever you have data in a table format, e.g. when reading in a `csv` file. It has some unintuitive quirks which you have to learn, then it becomes invaluable.

The `pandas.DataFrame()` is the core class.

scipy

All your higher mathematical needs: fast-fourier transform, curve fitting, integration, linear algebra, and more.

astropy

`astropy` is the astronomer's toolbox, from planning observations by computing source coordinates to analyzing the results with time series and other common astronomical models. There is plenty of functionality to read and edit `FITS` files.

Beware of the `astropy.Table` though, I never got used to them. Luckily, they have a `.to_pandas()` method, which returns a `pandas.DataFrame()`.

Best Practices

Best practices refer to code patterns and formats which could be written in many ways, but over the time it has become clear that there is a preferable way to do things. Most of them serve to make your code more readable and understandable.

Writing *pythonic* code

This comes with time and gaining knowledge of the standard library. A classic example:

```
colours = ["blue", "black", "red"]

i = 0  # don't do this

for colour in colours:
    i += 1 # don't do this
    print(f"{i}: My favourite colour is {colour}")
```

This should instead look like this.

```
colours = ["blue", "black", "red"]

for i, colour in enumerate(colours):  # do this instead
    print(f"{i}: My favourite colour is {colour}")
```

Indicators that you are not writing pythonic code (also called *code smell*):

- you are juggling lots of indices variables and it is becoming difficult to keep track of them

- you reach a high level of indentation
- you have many `if`-clauses

I *highly* recommend the following (and really all other talks) by Raymond Hettinger: Transforming Code into Beautiful, Idiomatic Python

Comment everything

Every third line I write is a comment. Comments tell me what the code I wrote is doing. It frequently occurs that I write code on Friday and when I come back on Monday into the office, I cannot explain what the code is supposed to do without studying it. Comment frequently, more for yourself than others. Avoid obvious comments: explain your code on a meta level, rather than in close detail.

Get the habit of writing docstrings for your functions.

Figure 4: Comment your code but avoid redundant comments.

Meaningful variable names

Another useful rule to increase the readability of your scripts. Compare this

```
et = 50  # in s
```

to this

```
exposure_time = 50  # in s
exp_time = 50  # in s
```

While the former is quicker to type, the two latter are clear even when removed from the context of the script they appear in.

No magic numbers

Magic numbers are numbers which appear in your script without any explanation. They will confuse you at a later stage.

```
electrons = 5 * 30  # ?
```

versus

```
exposure_time = 30  # in s
rate = 5  # in electrons / s
electrons = rate * exposure_time
```

Proper formatting

Apart from the indentation, you are quite free to format your python code however you want. You can use ' or " as string delimiters, you can wrap variable assignments with whitespaces (`x = 4`) or you can use the shorter form (`x=4`). Nevertheless, there is a right way to format your code, and it is called PEP8. If you do not want to worry about code formatting, I recommend to use an autoformatter such as black, which will ensure that your code is always presentable. Many editors like the ones I listed above offer plugins which format your code on saving.

Here is a nice talk on the topic, once more by Raymond Hettinger.

Tools for developing python

Your editor

Choosing an editor for coding is a very subjective decision and a popular matter of discussion. I recommend you start with one of the editors below, as they provide several convenient features such as code completion, quick variable value inspection, and easy access to documentation.

Popular editors for writing python scripts are

- Atom
- Sublime Text 4
- Visual Studio Code
- PyCharm

Your interpreter

The default interactive python interpreter is quite dull. It is missing nice features like syntax highlighting and TAB-completion of function names. The ipython interpreter is a popular alternative. You can install it via pip.

```
$ python3 -m pip install ipython
```

jupyter and its notebooks

jupyter notebooks are popular tools for writing code in a format that's different to most editors. The input area is divided into cells which can contain different types of input (python code, markdown text, LaTeX). Each cell can be executed separately from other cells.

```
$ python3 -m pip install jupyter
$ jupyter notebook
```

Exercises

The two exercises below each have their own motivation. The first represents several tasks which you will encounter frequently in your daily work as researcher: simulating data, fitting data, and presenting the result.

The second task represents a standard introduction-to-coding task, the Fibonacci numbers. There is a twist: the last part of the task tells you to (1) notice that the first solution offers significant room for improvement and (2) to improve your first solution. Again, this represents a frequent aspect of your later work.

Neither of the tasks require you to come up with elaborate algorithms, they can be solved using functions provided either in the standard library or the third-party packages mentioned above.

Task 1: Simulating, fitting, and displaying data

- Take 100 random samples of a sine curve with an amplitude and a period of your choosing
- Add Gaussian noise to the samples. The noise should have a mean of 0 and a standard deviation equal to 10% of the sine curve signal
- Fit your simulated data with a sine curve
- Create a figure which displays the simulated data and the fitted sine curve
- In the figure, add the best fit parameters and their errors

Only continue reading if you need a hint.

Hint: numpy.random can be used to create the random data. To add the Gaussian noise, you can add a vector of 100 samples drawn from a Gaussian distribution to the vector of your sine curve data. For the fit, you can use scipy.optimize. The last part of the exercise requires the matplotlib package.

Task 2: The Fibonacci Numbers

- Define a function which accepts an integer n as argument and returns the value of the nth Fibonacci number F_n.
- Compute F_{40}. Print its value and the time it took to compute it.
- It takes a while. As Raymond Hettinger would say, "there must be a better way"! Get the computation time to less than one second.

Only continue reading if you need a hint.

Hint: You only need to add two lines of code (one of which is an import statement) to speed up the computation by a factor of 1e6.

Solutions

The solutions can be found on the GitHub repository of these notes in the exercises/ directory.