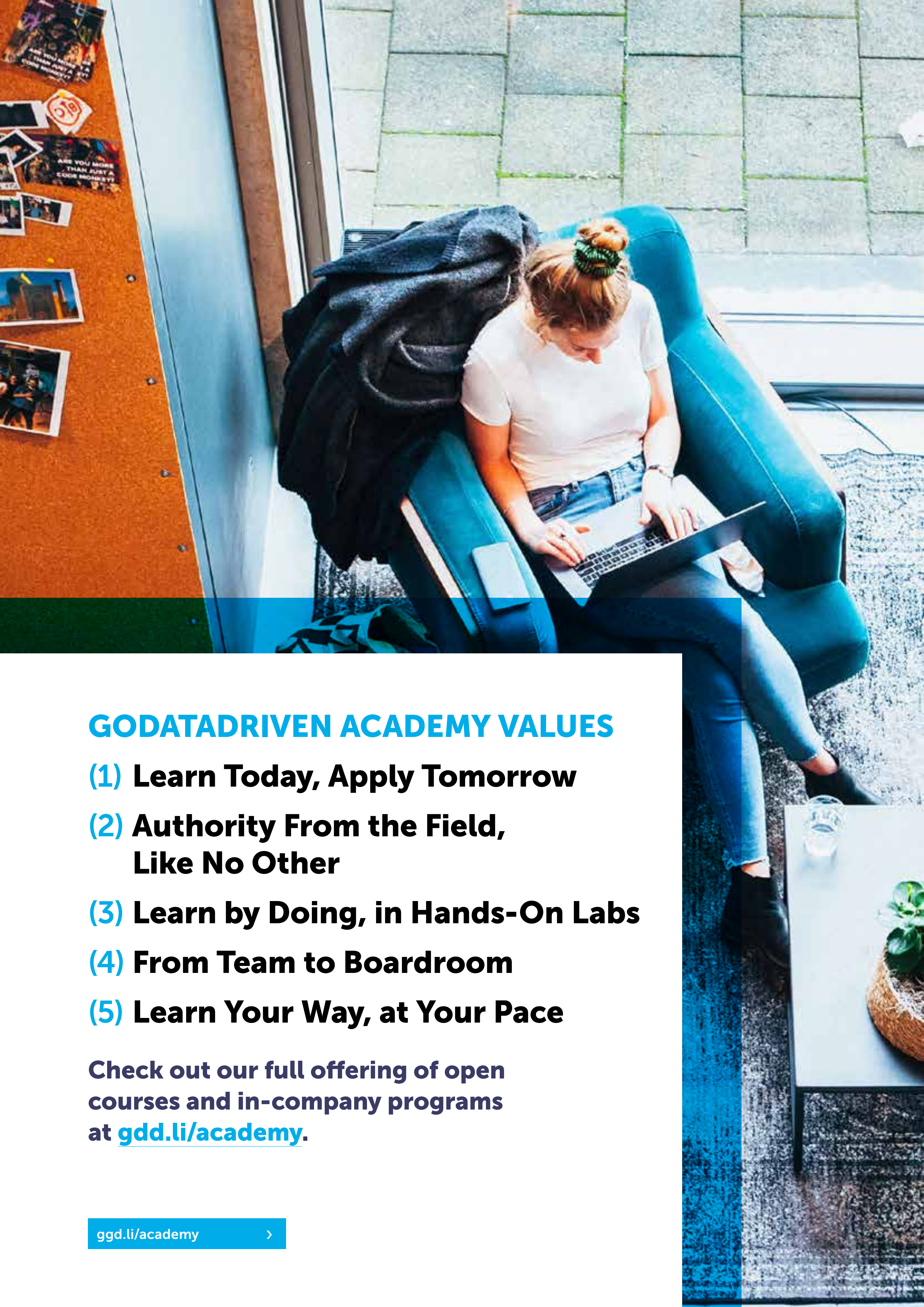
E-Book: Advanced Data Science with Python

BY GO DATA DRIVEN

GO DATA DRIVEN PROUDLY PART OF XEBIA



GODATADRIVEN ACADEMY VALUES

(1) **Learn Today, Apply Tomorrow**

(2) **Authority From the Field, Like No Other**

(3) **Learn by Doing, in Hands-On Labs**

(4) **From Team to Boardroom**

(5) **Learn Your Way, at Your Pace**

Check out our full offering of open courses and in-company programs at gdd.li/academy.

ggd.li/academy ›

INTRODUCTION

GoDataDriven Academy— Our Promise to You

At GoDataDriven, we believe professional development goes hand-in-hand with staying happy, motivated, confident, and relevant in your job. You trust in us to help you improve your data and AI skills, and we take that very seriously.

So we strive to provide you with the best learning experience possible, by adhering to five promises:

(1) Learn Today & Apply Tomorrow
We design our programs so you can apply your newly acquired knowledge right out of the classroom. That way you can immediately increase your business value. We push the content of every training far beyond textbooks and theory. Applicability is one of our core values.

(2) Authority From the Field, Like No Other
We don't come from an abstract background; we are practitioners as much as we are teachers. All of our trainers work as consultants in the data and AI field, supporting top enterprises like Booking.com, ING, bol.com, Randstad, and Heineken. They solve challenges like yours every day, so you benefit directly from their experience.

(3) Learn by Doing, in Hands-On Labs
You learn best by doing. That's why, in every training, you can develop your skills and craft in our hands-on labs. We provide both the theory and context to get you up and running fast. All of our courses have a 50/50 split between theory and hands-on labs.

(4) From Team to Boardroom
Built through years of working with the top enterprises in Europe, GoDataDriven has the expertise to turn your data-driven ambition into reality. But becoming data-driven impacts your whole organization. That's why we deliver a wide range of programs suitable for all shapes and sizes—from individual teams to global workforces, as well as the boardroom.

(5) Learn Your Way At Your Pace
We deliver our curriculum through various training formats—classroom, in-company, online, or a combination. You choose the format that fits your purpose and preferred method of learning—or ask one of our academy advisors to guide you.

gdd.li/academy ›

Proudly Part of Xebia Group

GoDataDriven is proudly part of Xebia Group, an international consulting and training company, specialized in digital transformation. Xebia employs over 1,000 consultants worldwide. GoDataDriven and the GoDataDriven Academy both share Xebia's values:
(1) People First
(2) Sharing Knowledge
(3) Customer Intimacy
(4) Quality Without Compromise

Xebia

GO
DATA
DRIVEN

A Practical Guide to Using Setup.py

When you are using python professionally it pays to set up your projects in a consistent manner. This helps your collaborators quickly understand the
structure of a project, and makes it easier for them to set up the project on their machine. The key to setting up your project is the `setup.py` file. In this e-book I'll go into the details of this file.

Where we start

Here I assume that you already have a package that you want to set up.
This does not need to be a finished package - ideally you should create the `setup.py` long before your project is finished. It could even be an empty package; just make sure the package folder exists
and contains a file named **init.**py (which may be empty).

If you follow my colleague Henk's **structure**
for your project, your starting situation should look something like this:

```
example_project/
├── exampleproject/      Python package with source code.
|   ├── __init__.py      Make the folder a package.
|   └── example.py       Example module.
└── README.md            README with info of the project.
```

You may have other files or folders in your structure, for example
folders named `notebooks/`, `tests/` or `data/`, but these aren't required.

The case for a `setup.py`

Once you have created a package like this, then you are likely
to use some of the code in other places. For example, you might want

to do this in a notebook:

```
from exampleproject.example import example_function
```

This would work if your current working directory is `example_project/`, but in all other cases python will give you output like:

```
ModuleNotFoundError: No module named 'exampleproject'
```

You could tell python where to look for the package by setting the `PYTHONPATH` environment variable or adding the path to `sys.path`,
but that is far from ideal: it would require different actions on different platforms, and the path you need to set depends on the location of your code.
A much better way is to install your package using a `setup.py` and `pip`, since `pip` is the standard way to install all other packages, and it is bound it work the same on all platforms.

A minimal example

So what does a `setup.py` file look like? Here is a minimal example[0]:

```
from setuptools import setup, find_packages

setup(
    name='example',
    version='0.1.0',
    packages=find_packages(include=['exampleproject', 'exampleproject.*'])
)
```

Here we specify three things:

- The name of the package, which is the name that `pip` will use for your package. This does not have to be the same as the folder name the package lives in, although it may be confusing if it is not. An example of where the package name and the directory do not match is Scikit-Learn: you install it using `pip install scikit-learn`, while you use it by importing from `sklearn`.
- The version of your package. This is the version `pip` will report, and is used for example when you publish your package on **PyPI**[1].
- What packages to include; in our case this is just `exampleproject/`. Here we let `setuptools` figure this out automatically[2]. While you could in principle use `find_packages()`

without any arguments, this can potentially result in unwanted packages to be included. This can happen, for example,
if you included an `__init__.py` in your `tests/`
directory. Alternatively, you can also use the `exclude` argument to explicitly prevent the inclusion of tests in the package, but this is slightly
less robust.

Now all that you need to do in order to install your package is to run the following from inside the `example_project/` directory[3]:

```
pip install -e .
```

The `.` here refers to the current working directory, which I assume to be the directory where the `setup.py` can be found. The `-e` flag specifies that we want to install in *editable mode*, which means
that when we edit the files in our package we do not need to re-install the package before the changes come into effect. You will need to either restart python or reload the package though!

When you edit information in the `setup.py` itself you will need to re-install the package in most cases, and also if you add new (sub)packages.
When in doubt, it can never hurt to re-install. Just run `pip install -e .` again.

Requirements

Most projects have some dependencies. You have most likely used
a **requirements.txt**
file before, or an **environment.yml**
if you are using `conda`. Now that you are creating a `setup.py`, you can specify your dependencies in the `install_requires` argument.
For example, for a typical data science project you may have:

```
setup(
    name='example',
    version='0.1.0',
    packages=find_packages(include=['exampleproject',
'exampleproject.*']),
    install_requires=[
        'PyYAML',
        'pandas==0.23.3',
        'numpy>=1.14.5',
        'matplotlib>=2.2.0,,
        'jupyter'
    ]
)
```

You may specify requirements without a version (`PyYAML`), pin a version
(`pandas==0.23.3`), specify a minimum
version (`'numpy>=1.14.5`) or set a range of versions (`matplotlib>=2.2.0,<3.0.0`).
These
requirements will automatically be installed by `pip` when you install your package.

Extras-require

Sometimes you may have dependencies that are only required in certain situations. As a
data scientist
I often make packages which I use to train a model. When I work on such a model
interactively
I may need to have `matplotlib` and `jupyter` installed in order to interactively work with
the
data and to create visualizations
of the performance of the model. On the other hand, if the model runs in production I do
not
want to install `matplotlib` nor `jupyter` on the machine (or container) where I train
or do inference. Luckily `setuptools` allows to specify optional dependencies in
`extras_require`:

```
setup(
    name='example',
    version='0.1.0',
    packages=find_packages(include=['exampleproject',
'exampleproject.*']),
    install_requires=[
        'PyYAML',
        'pandas==0.23.3',
        'numpy>=1.14.5'
    ],
    extras_require={
        'interactive': ['matplotlib>=2.2.0,, 'jupyter'],
    }
)
```

Now if we install the package normally (`pip install example` from PyPI or `pip
install -e .` locally)
it will only install the dependencies `PyYAML`, `pandas` and `numpy`. However, when we specify
that we want the optional `interactive` dependencies (`pip install
example[interactive]`
or `pip install -e .[interactive]`),
then `matplotlib` and `jupyter` will also be installed.

Scripts and entry points

The main use case of most python packages that you install from PyPI is to provide functionality
that can be used in other python code. In other words, you can `import` from those packages.
As a data scientist I often make packages that aren't meant to be used by other python code but
are meant to *do* something, for example to train a model. As such, I often have a python script that
I want to execute from the command line.

The best way[4] to expose functionality of your package to the command line is to define an `entry_point` as such:

```
setup(
    # ...,
    entry_points={
        'console_scripts': ['my-command=exampleproject.example:main']
    }
)
```

Now you can use the command `my-command` from the command line, which will in turn execute the `main`
function inside `exampleproject/example.py`. Do not forget to re-install - otherwise the command
will not be registered.

Tests

Whenever you write any code, I strongly encourage you to also write tests for this code. For testing
with python I suggest you use `pytest`. Of course you do not want to add `pytest` to your dependencies
in `install_requires`: it isn't required by the users of your package. In order to have it installed
automatically *when you run tests* you can add the following to your `setup.py`:

```
setup(
    # ...,
    setup_requires=['pytest-runner'],
    tests_require=['pytest'],
)
```

Additionally you will have to create a file named `setup.cfg` with the following contents:

```
[aliases]
test=pytest
```

Now you can simply run `python setup.py test` and `setuptools` will ensure the necessary dependencies
are installed and run `pytest` for you! Have a look **here** if
you want to provide arguments or set configuration options for `pytest`.

If you have any additional requirements for testing (e.g. `pytest-flask`) you can add them to `tests_require`.

Flake8

Personally I think it is a good idea to run **Flake8** to
check the formatting of your code. Just like with `pytest`, you do not want to add `flake8` to the
`install_requires` dependencies: it does not need to be installed in order to use your package. Instead, you can add it to `setup_requires`:

```
setup(
    # ...,
    setup_requires=['flake8']
)
```

Now you can simply run `python setup.py flake8`. Of course you can also pin the version
of `flake8` (or any other package) in `setup_requires`.

If you want to change some of the configuration parameters of Flake8 you can add a `[flake8]` section to
your `setup.cfg`. For example:

```
[flake8]
max-line-length=120
```

Package data

Sometimes you may want to include some non-python files in your package. These may for example be schema files or a small lookup table. Be aware that such files will be packaged together with your code, so it is in general a bad idea to include any large files.

Suppose we have a `schema.json` in our project, which we place in `exampleproject/data/schema.json`.
If we want to include this in our package, we must use the `package_data` argument of `setup`:

```
setup(
    # ...,
    package_data={'exampleproject': ['data/schema.json']}
)
```

This will make sure the file is included in the package. We can also choose to include all files based on a pattern, for example:

```
setup(
    # ...,
    package_data={'': ['*.json']}
)
```

This will add all `*.json` files in any package it encounters.

Now don't try to figure out the installed files' location yourself, as `pkg_resources` has some very handy convenience functions:

- `pkg_resources.resource_stream` will give you a stream of the file, much like the object you get when you call `open()`,
- `pkg_resources.resource_string` will give you the contents of the file as a string,
- `pkg_resources.resource_filename` will give you the filename of the file (and extract
  it into a temporary if it is included in a zipped package) for if the two options above do not suit your needs.

For example, we could read in our schema using:

```
from json import load
from pkg_resources import resource_stream

schema = load(resource_stream('exampleproject', 'data/schema.json'))
```

Metadata

If you are going to publish your package, then you probably want to give your potential users some more information about your package, including a description, the name of the author or maintainer, and the url to the package's home page. You can find a complete list of all allowed metadata in the `setuptools` **docs**.

Additionally, if you are going to publish to PyPI, then you may want to automatically **load the contents of your `README.md` into the `long_description`**, and provide **classifiers** to tell `pip` even more about your package.

Wrap-up

This e-book should be a good starting point to set up most of your python projects. If you want to read more about python packaging have a look at **the docs**. Here is an example `setup.py` which combines all parts shown in this e-book:

```
from setuptools import setup, find_packages

setup(
    name='example',
    version='0.1.0',
    description='Setting up a python package',
    author='Rogier van der Geer',
    author_email='rogiervandergeer@godatadriven.com',
    url='https://blog.godatadriven.com/setup-py',
    packages=find_packages(include=['exampleproject',
'exampleproject.*']),
    install_requires=[
        'PyYAML',
        'pandas==0.23.3',
        'numpy>=1.14.5'
    ],
    extras_require={'plotting': ['matplotlib>=2.2.0', 'jupyter']},
    setup_requires=['pytest-runner', 'flake8'],
    tests_require=['pytest'],
    entry_points={
        'console_scripts': ['my-command=exampleproject.example:main']
    },
    package_data={'exampleproject': ['data/schema.json']}
)
```

and the accompanying `setup.cfg`:

```
[aliases]
test=pytest

[flake8]
max-line-length=120
```

Improve your Python skills, learn from the experts!

At GoDataDriven we offer a host of Python courses from beginner to expert, taught by the very best professionals in the field. Join us and level up your Python game:

- **Python Essentials** - Great if you are just starting with Python.
- **Certified Data Science with Python Foundation** - Want to make the step up from data analysis and visualization to true data science? This is the right course.
- **Advanced Data Science with Python** - Learn to productionize your models like a pro and use Python for machine learning.

Footnotes

0: In this e-book I have used **setuptools**
to set up my example project. Alternatively
you could also use **distutils**,
which is the standard tool for packaging in python, but it lacks features
such as the `find_packages()` function and `entry_points`.
Since the use of setuptools is very common nowadays and many of its features
can be particularly useful, I suggest that you should use setuptools.

1: If you want the version of your package to also be available inside python,
have a look **here**.

2: You could also list your packages manually, but this is particularly error-prone.

3: Alternatively you could run `pythonsetup.pyinstall`, but using `pip` has
many benefits, among which are automatic installation of dependencies and the
ability to uninstall or update your package.

4: You could also use the `scripts` argument (see for
example **here**)
but as this requires you to create a python shell script it may not work
as well (or at all) on Windows.