Writing Your First Package

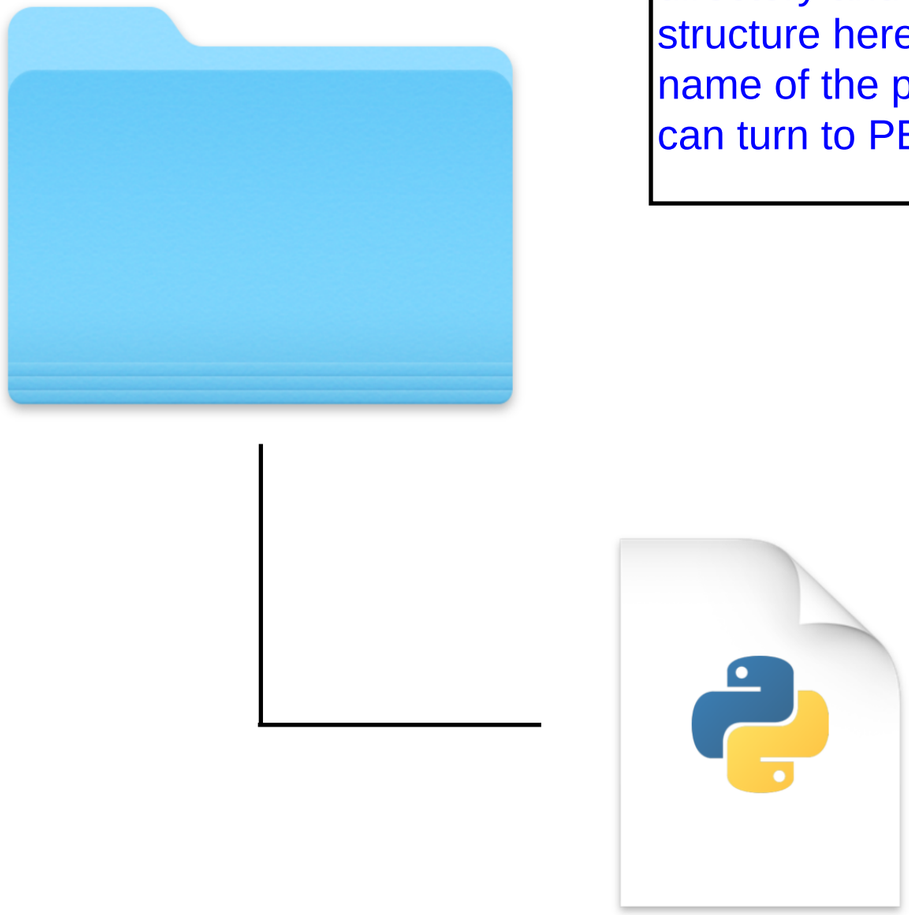
SOFTWARE ENGINEERING FOR DATA SCIENTISTS IN PYTHON

Adam Spannbauer
Machine Learning Engineer at Eastman

datacamp

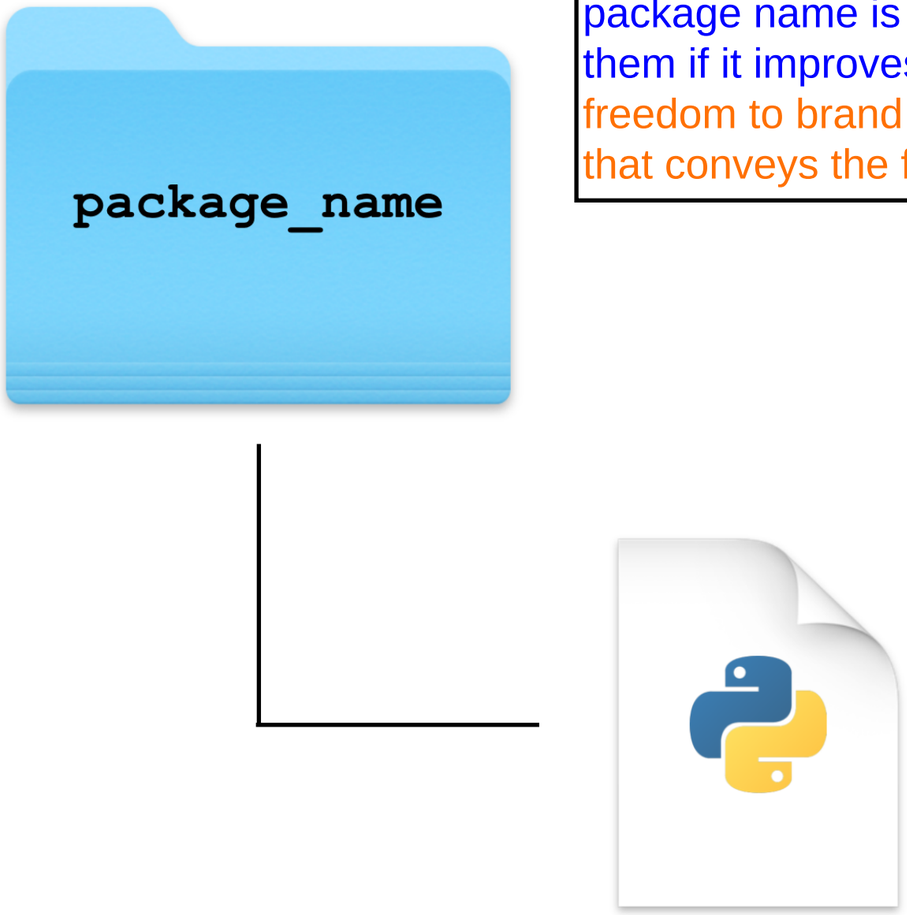
Package structure

2. Package structure
A minimal python package consists of 2 elements: a directory and a python file. You can see the basic structure here. The name of the directory will be the name of the package, but how should you name it? We can turn to PEP 8 again for some guidance.



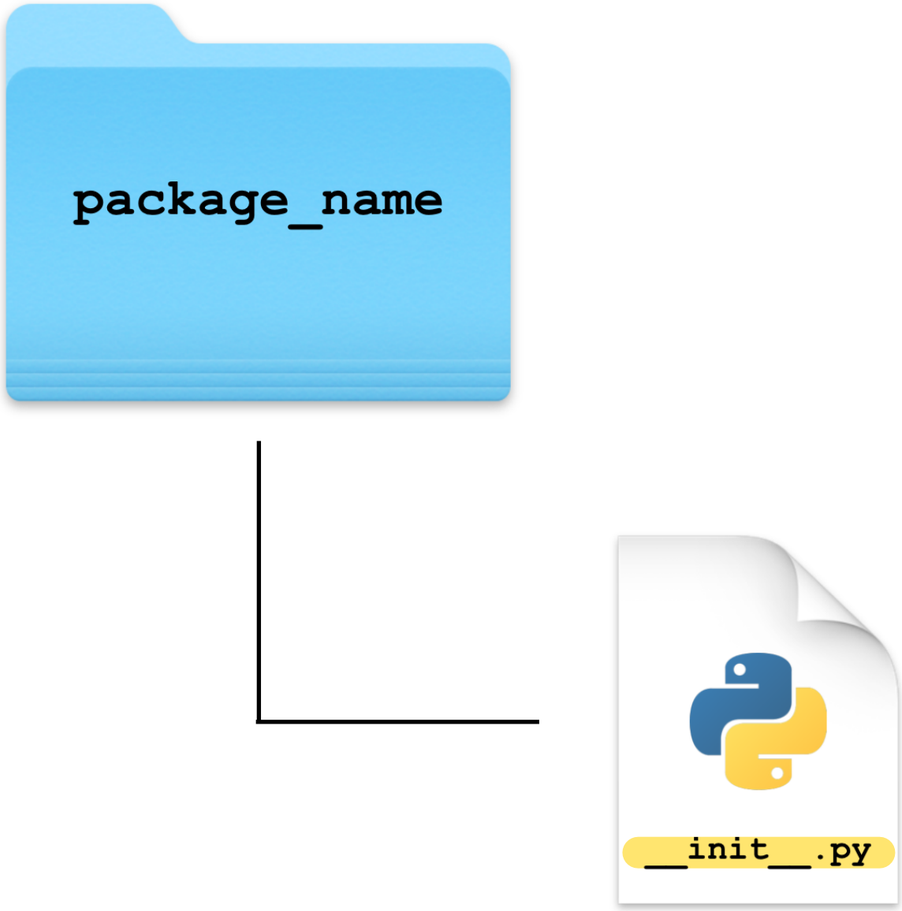
Package structure

3. Package structure
To paraphrase, PEP 8 states that packages should have short, all-lowercase names. The use of underscores in a package name is discouraged, but you can and should use them if it improves readability. Outside of this, you have the freedom to brand as you'd like. I'd suggest picking a name that conveys the functionality of the package.

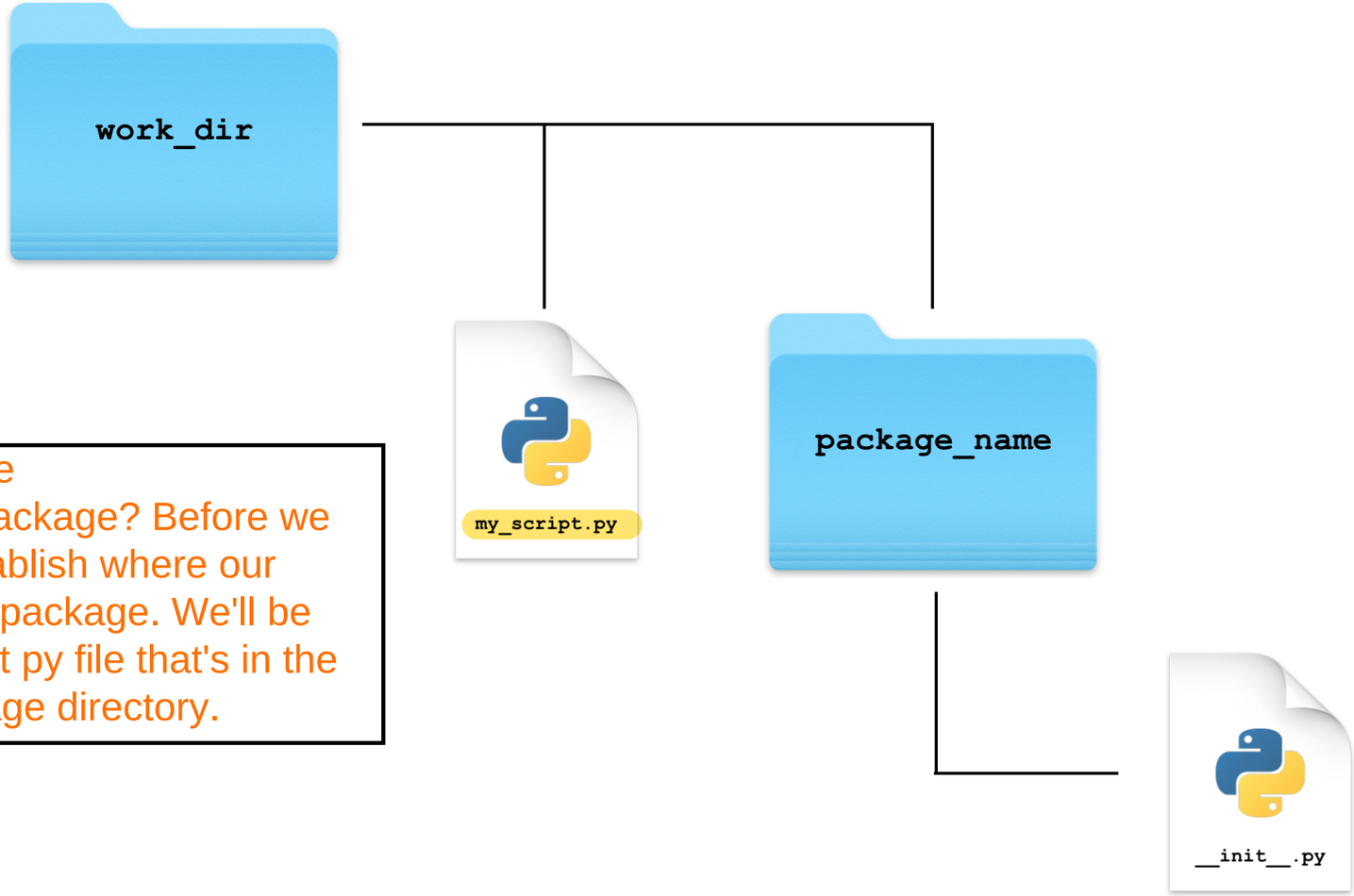


Package structure

4. Package structure
The file in our newly branded directory doesn't have any flexibility in naming. We must name it underscore underscore init underscore underscore dot py. This file lets python know that the directory we created is a package. And that's it! With this structure we've created a package that we can import just like we would import numpy or any other package. Note that as of Python version 3-point-3 any directory can be imported as if it were a package without error even if it doesn't follow this structure. Earlier versions of Python would throw an error if you tried to import an improperly formatted package. Even though a directory might import without error, you will run into issues if you do not follow this structure.



Importing a local package



5. Importing a local package
So how do we import our package? Before we look at some code let's establish where our script is relative to our new package. We'll be working in the my_script dot py file that's in the same location as our package directory.

Importing a local package

```
import my_package
help(my_package)
```

```
Help on package my_package:
NAME
    my_package

PACKAGE CONTENTS
FILE
    ~/work_dir/my_package/__init__.py
```

6. Importing a local package
With all the setup we've done so far, importing the package is a breeze. At the top of our script, we can use import my_package. Just for an added bonus, let's check out what happens if we try to call help on our package. We get some minimal information with python letting us know that it is indeed a package and additionally we see the location of our __init__ dot py file. As we previously covered, it's up to the developer to add in useful documentation to be printed whenever a user calls help. Later we will be going over how we can add this documentation as well as implementing some functionality to make our package more useful.

Let's Practice

SOFTWARE ENGINEERING FOR DATA SCIENTISTS IN PYTHON

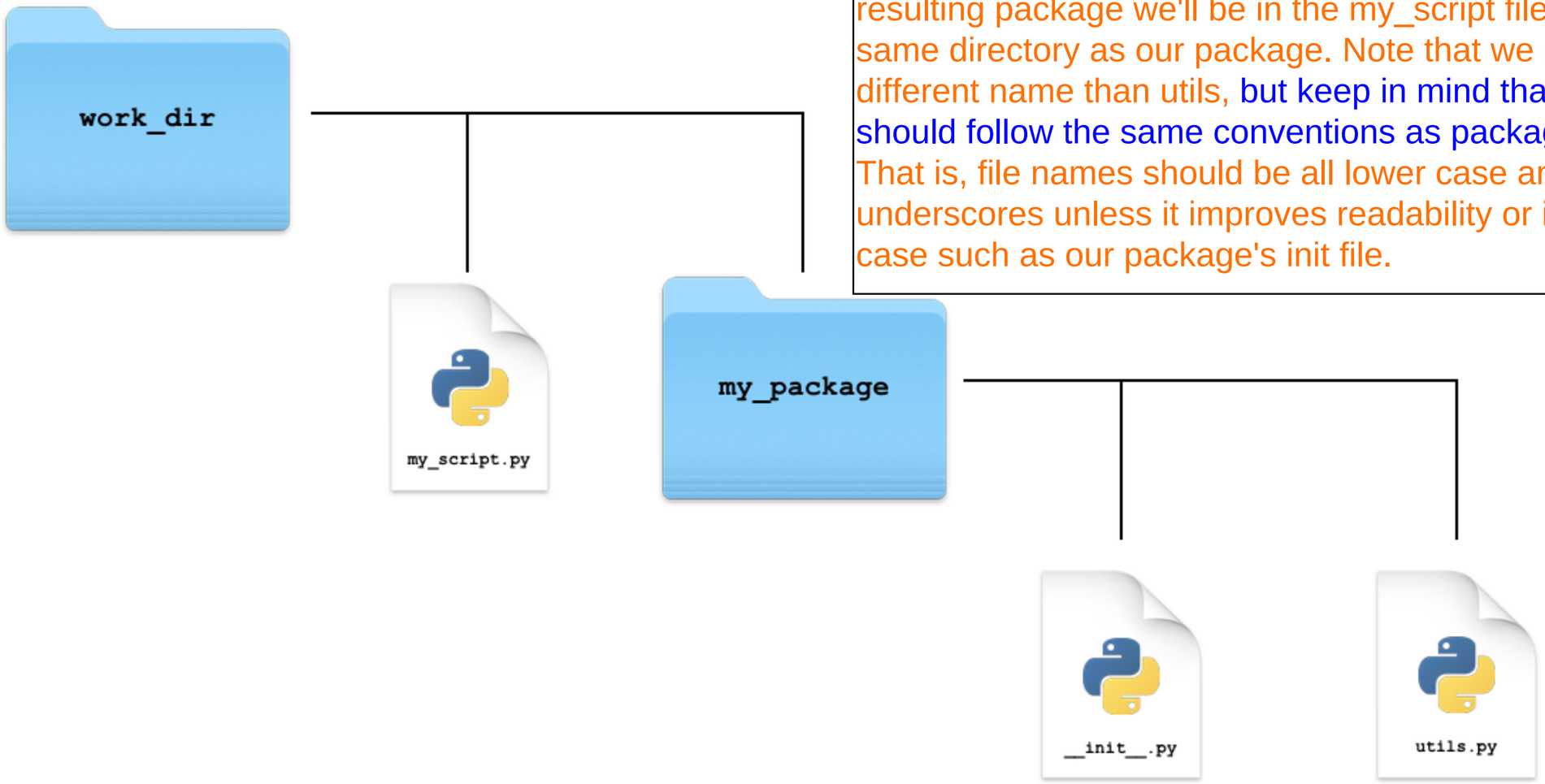
Adding Functionality to Packages

SOFTWARE ENGINEERING FOR DATA SCIENTISTS IN PYTHON

Adam Spannbauer
Machine Learning Engineer at Eastman

datacamp

Package structure



2. Package structure
To start, let's again look at the file structure we'll be using. Here we've added a file to our package's directory named utils dot py. Again, when we import and work with our resulting package we'll be in the my_script file that's in the same directory as our package. Note that we can choose a different name than utils, but keep in mind that file names should follow the same conventions as package naming. That is, file names should be all lower case and avoid underscores unless it improves readability or if it's a special case such as our package's init file.

Adding functionality

working in `work_dir/my_package/utils.py`

```
def we_need_to_talk(break_up=False):
    """Helper for communicating with significant other"""
    if break_up:
        print("It's not you, it's me...")
    else:
        print('I <3 U!')
```

3. Adding functionality
With the right structure in place, the next step is to write some code. Here, in our utils dot py file, we write a function that prints one of two possible statements based on user input. Our utils file is known as a submodule and we can import and with a dot notation syntax of the form: package name dot submodule name dot function name. In this example, we call my_package dot utils dot we_need_to_talk.

working in `work_dir/my_script.py`

```
# Import utils submodule
import my_package.utils

# Decide to start seeing other people
my_package.utils.we_need_to_talk(break_up=True)
```

```
It's not you, it's me...
```

Importing functionality with __init__.py

working in `work_dir/my_package/__init__.py`

```
from .utils import we_need_to_talk
```

working in `work_dir/my_script.py`

```
# Import custom package
import my_package

# Realize you're with your soulmate
my_package.we_need_to_talk(break_up=False)
```

```
I <3 U!
```

4. Importing functionality with __init__.py
Alternatively, we can use our package's init file to make our utils' function more easily accessible by the user. To do this, we import our function in our init file as you see here. The dot notation we use when writing dot utils is known as a relative import and it must be used when packaging in Python versions 3 and above. Importing our function in the init file saves some typing when we want to import and use our function. We're now able to call my_package dot we_need_to_talk without including the additional reference to our utils submodule, importing the function in the init file took care of this reference for us.

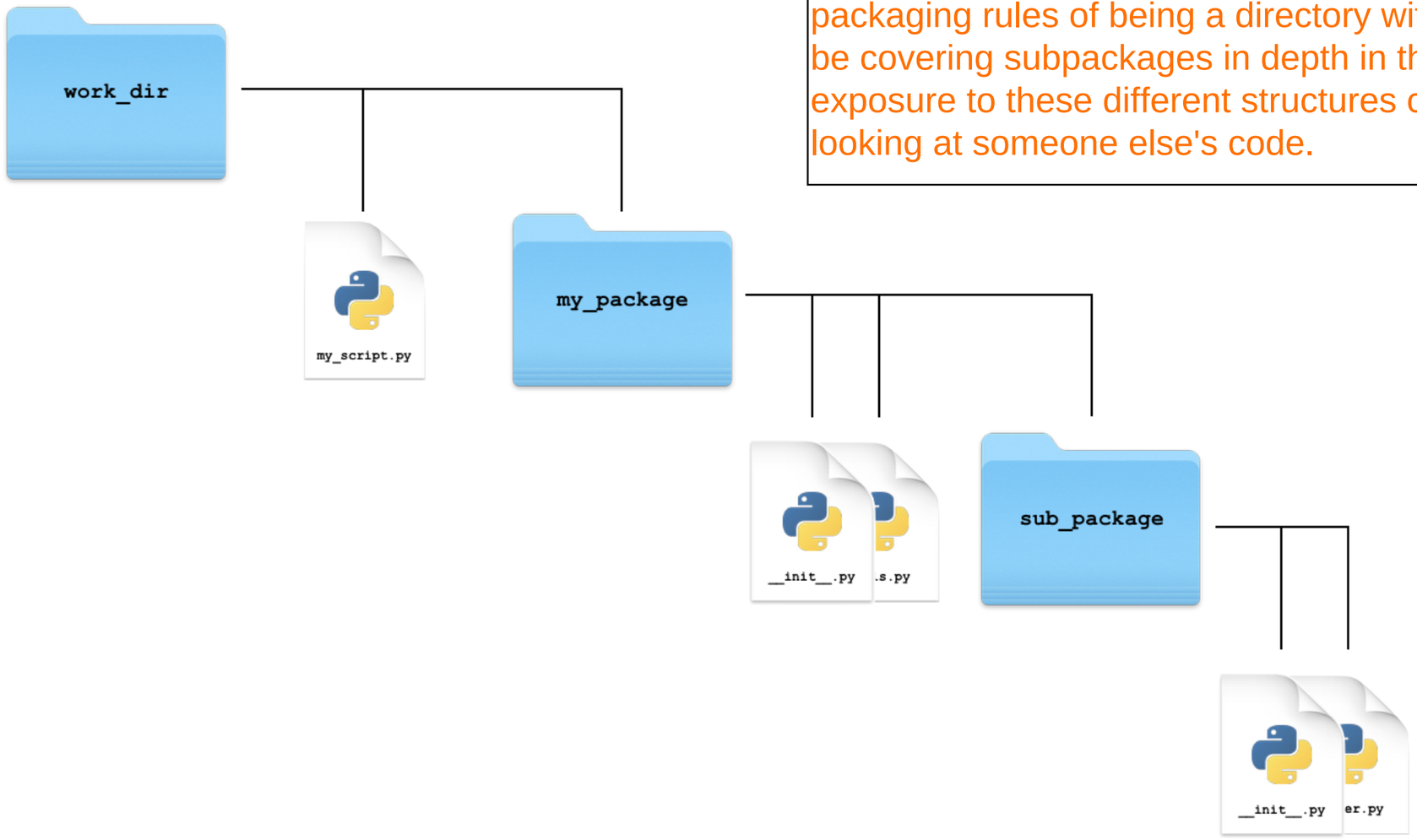
Extending package structure

5. Extending package structure

...When working with multiple submodules should you import them all in init? As a general rule, you should import your package's key functionality in your init file to make it directly and easily accessible. Less central functionality can be accessed by users through the longer submodule dot syntax we saw earlier. The decision of what is 'key' functionality is a gray area, and because of this, there isn't always a clear best way to organize your package. As a package developer, you'll need to use your judgment to decide what you think will give your users the best experience.

Extending package structure

6. Extending package structure
In addition to adding additional submodules, you can also build out packages inside your package! These are known as subpackages. Notice how the subpackage still follows the packaging rules of being a directory with an init file. We won't be covering subpackages in depth in this course but exposure to these different structures can be helpful when looking at someone else's code.



Let's Practice

SOFTWARE ENGINEERING FOR DATA SCIENTISTS IN PYTHON

Making your package portable

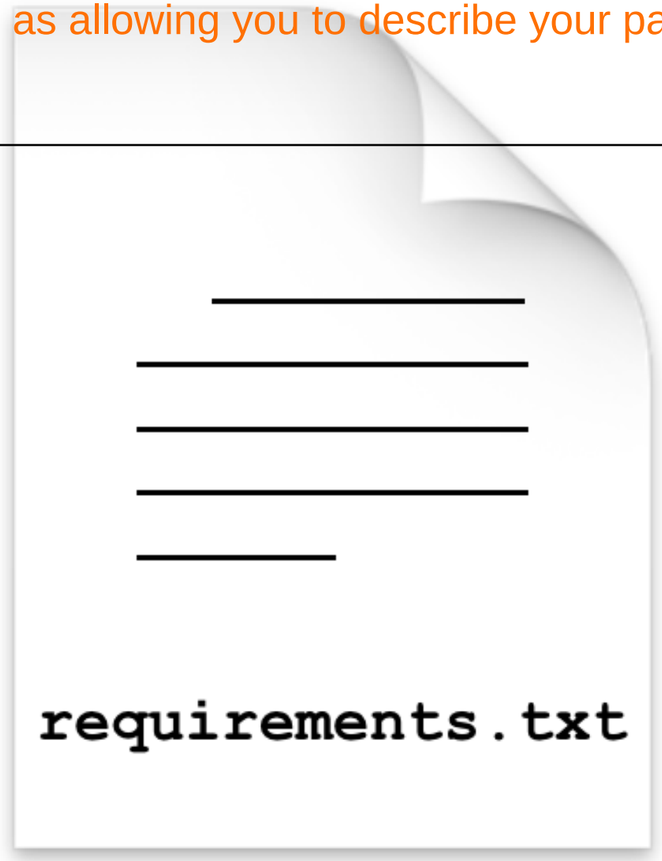
SOFTWARE ENGINEERING FOR DATA SCIENTISTS IN PYTHON

Adam Spannbauer
Machine Learning Engineer at Eastman

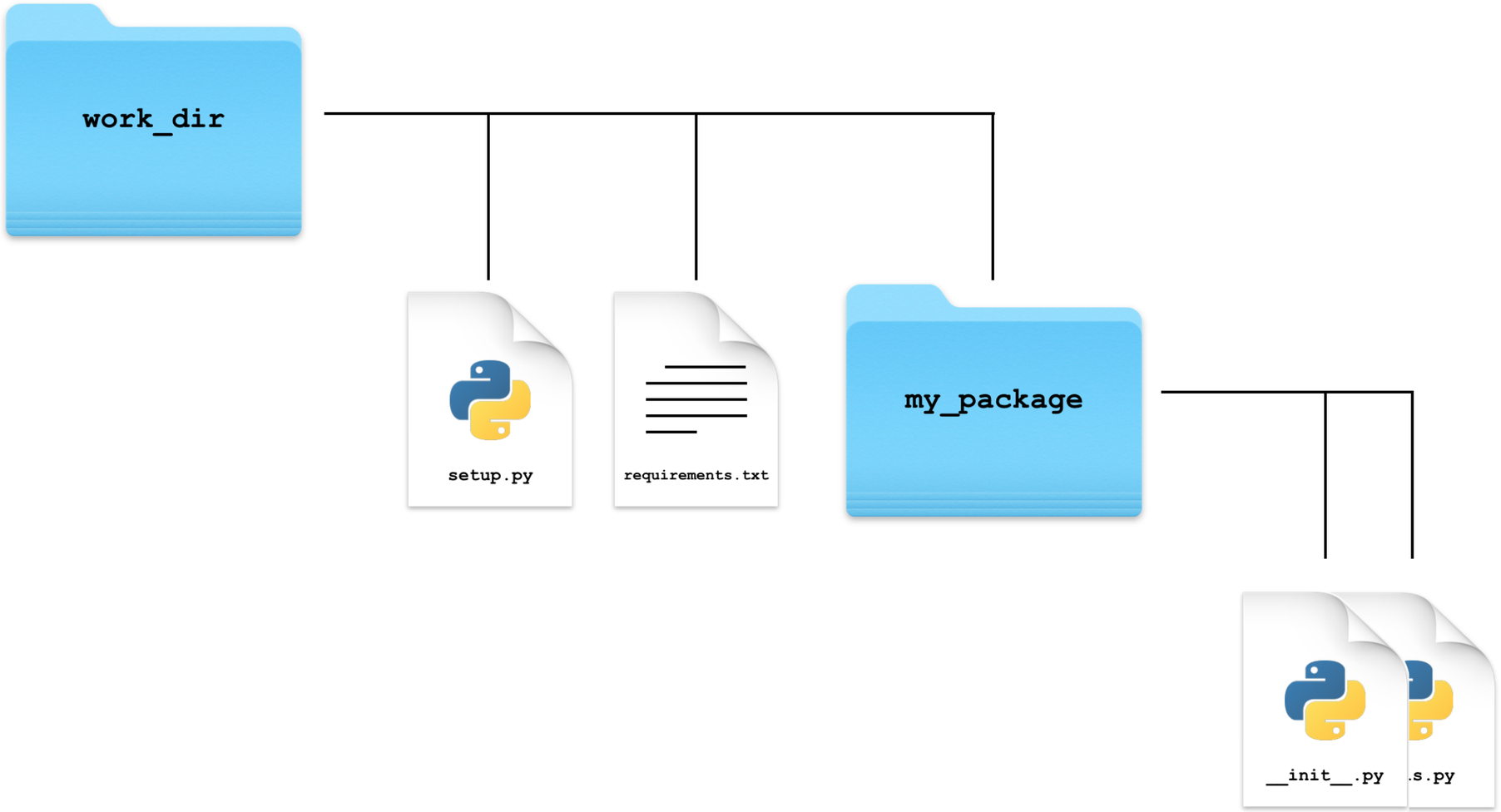
datacamp

Steps to portability

2. Steps to portability
The two main steps to sharing a python package are creating setup dot py and requirements dot txt. These two pieces provide information on how to install your package and recreate its required environment. These files list information about what dependencies you've used as well as allowing you to describe your package with additional metadata.



Portable package structure



Contents of requirements.txt

working in `work_dir/requirements.txt`

```
# Needed packages/versions
matplotlib
numpy==1.15.4
pycodestyle>=2.4.0
```

working with `terminal`

```
datacamp@server:~$ pip install -r requirements.txt
```

4. Contents of requirements.txt
A requirements file shows how to recreate the environment needed to properly use your package. This includes a list of python packages and optionally the version requirements for each package. Here we see 3 different ways to specify our requirements. If we don't have a reason to specify a version we can just list the package name as you see here for matplotlib. If version is important. We can mark a specific version by using a double equals, or mark a minimum version by using greater than or equal. Since open source packages are constantly evolving, specifying a version can be a big help to your users. To leverage our requirements file we can use this pip install command. This installs all the packages listed with respect the correct version. Note that we didn't actually install our package, we just recreated its environment.

Contents of setup.py

```
from setuptools import setup

setup(name='my_package',
      version='0.0.1',
      description='An example package for DataCamp.',
      author='Adam Spannbauer',
      author_email='spannbaueradam@gmail.com',
      packages=['my_package'],
      install_requires=['matplotlib',
                        'numpy==1.15.4',
                        'pycodestyle>=2.4.0'])
```

5. Contents of setup.py
setup dot py is what tells pip how to install our actual package. Additionally, its info will be used by PyPi if you decide to publish. The contents of this in our case contains a single call to the setup function from the setuptools package. There are other options, but setuptools is one of the most common and powerful choices. Here we use just a few of the possible setup arguments. There are many more options available that you can read more about in the setuptools documentation. The argument's names make them fairly self-explanatory; for example, your package's name is assigned to name and so on. Some less obvious arguments in our example are install_requires and packages. packages in essence lists the location of all the init files in our package. Our package has a single init file and it's in the directory 'my_package'. As we saw before, more complex packages might include subpackages with their own init files, if this was the case we would also list their locations here. Until you start writing more complex packages, the contents of the packages list will likely be the same as the name argument. install_requires might look familiar, in the case of our package, the contents are the same as our requirements file.

install_requires vs requirements.txt

working in `work_dir/requirements.txt`

```
# Specify where to install requirements from
--index-url https://pypi.python.org/simple/

# Needed packages/versions
matplotlib
numpy==1.15.4
pycodestyle>=2.4.0
```

6. install_requires vs requirements.txt
An example of when install_requires can differ is if you want to specify where pip should download packages from. This can be specified in the requirements file as you see here. ..To read up more on the differences you can see the documentation linked here.

Documentation: **install_requires vs requirements files**

pip installing your package

```
datacamp@server:~/work_dir $ pip install .
```

```
Building wheels for collected packages: my-package
  Running setup.py bdist_wheel for my-package ... done
Successfully built my-package
Installing collected packages: my-package
Successfully installed my-package-0.0.1
```

7. pip installing your package
Now that we've completed our setup-dot-py, we can install our package with pip using pip install dot from inside the same directory as our package. This will install our package at an environment level so we can import it into any python script using the same environment.

Let's Practice

SOFTWARE ENGINEERING FOR DATA SCIENTISTS IN PYTHON