Introduction to Python

Magnus Nielsen, SODAS, UCPH

Agenda

- Introduction
- Fundamental data types
- Operators
- Containers with indices
- Control flow
- Loops
- Reuseable code

Introduction

Once again - why?

Python can do stuff that Stata, R and SAS can do:

- Machine learning, statistics

Python can do stuff that Matlab can do:

- Arrays galore in `numpy`

Python is a general-purpose programming language:

- Automate the boring stuff in your life
- Web development, web scraping and user applications

Python has a huge community!

The internet is your best friend

With a huge community comes a huge amount of resources

Sites such as:

- Package documentation, e.g. pandas or sklearn
- Guides, e.g. W3Schools or on YouTube
- Q&A platforms such as StackOverflow
- GitHub, e.g. pandas or sklearn
- Books, e.g. aforementioned Automate the Boring Stuff with Python or Python for Data Analysis

Especially Python for Data Analysis may be interesting for this course

But all you *really* need is Google

Introduction to Python

Go through the Python tutorial at W3Schools

Just kidding, but there really are a ton of great guides out there!

The only way to get good at programming is simply to program!

A disclaimer

The amount of information in the presentation and exercises might be overwhelming

- View it as an introduction and a teaser

If you wish, you can continue preprocessing data in your favourite program and import the data into Python and go straight to machine learning

Embedded Python

When using Python, I will try to include both source code and the output

You can copy the code in the upper left corner

```python
print('hello world')
```

```
hello world
```

The source code might be hidden – but it’s still there

▶ Show code

```
hidden hello world
```

Python 101

Python makes heavy use of assigning variables

A variable is created when you assign a value to it using =

```
# Lines can be commented out with a #

# Variables can be assigned with =
var_1 = 'Example 1'

# Variables can be printed with the print() function
print(var_1)
```

```
Example 1
```

Python is case-sensitive

Some tips

`help()` gives information about objects

```python
help(print)
```

```
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

In PyCharm, hovering over an object also gives this information

In Jupyter Notebook, pressing `shift+tab` while inside parenthesis also gives this information

When you use a `.` in your code to call a method, PyCharm will suggest methods – to prompt this in Jupyter Notebook, press `tab`

Fundamental data types

The big three

String

- Words

Numeric

- Integers and floats

Boolean

- True and False

How to define them

Strings are defined with `''` or `""` - Multiline, raw and formatted strings also exist

Numeric are defined as numbers, with type dependent on delimiter

Booleans are defined as `True` or `False`

```
# strings
a_string = "I'm a string"
another_string = '2.5'
# numerical
an_int = 2
a_float = 2.5
# boolean
a_boolean = True

#confusion
print(another_string, a_float)
```

```
2.5 2.5
```

Strings that look like a `float`/`int` can cause confusion

Type conversion

You can convert between different types with `int()`, `float()`, `str()`, `bool()`

You can check a type with `type()`

```
a_float = 2.5
a_string = str(a_float)
an_int = int(a_float)
a_boolean = bool(a_float)

print(a_float, type(a_float), a_string, type(a_string), an_int, type(an_int
```

```
2.5 <class 'float'> 2.5 <class 'str'> 2 <class 'int'> True <class 'bool'>
```

Some conversions are a bit odd, e.g. `bool()`, see more here

Note: `int()` always rounds down

Errors

Some things are not possible, and give an error

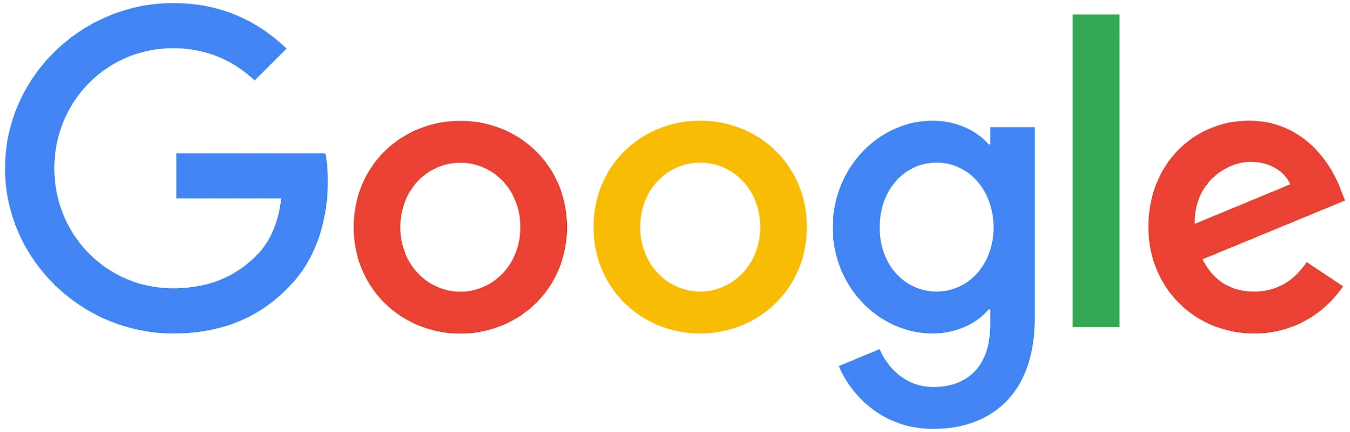
```
a_string = 'Error string'

int(a_string)
```

```
ValueError: invalid literal for int() with base 10: 'Error string'
```

The most important part of an error message (or stack trace) is usually the bottom (what went wrong) and the top (what part of the code started this)

Question

What do we do with this error message?



Before asking for help, try it out!

Operators

Basic operators

Some basic operators are:

- addition, `+`
- multiplication, `*`
- subtraction, `-`
- division, `/`
- power, `**`

```
print(2 + 2)
print(3 * 3)
print(7 / 2)
```

```
4
9
3.5
```

Comparisons

Python also supports comparisons, such as:

- equals, `==`
- not equals, `!=`
- smaller than, `<=`
- smaller than or equal, `<=`

These return boolean values (or errors)

```python
print(2 == 2)
print(3 <= 2)
print(7 != 2)
```

```
True
False
True
```

Combining booleans

Boolean values can combined using:

- the and operator - equivalent to &
- the or operator - equivalent to |

And can be negated with not

```python
print(True or False)
print(not (True | False))
print((1==1) and (2==1))
print(not ((1==1) & (2==1)))
```

```
True
False
False
True
```

Containers with indices

Three of the most fundamental composite data types are

- the *list*
- the *tuple*
- the *dictionary*

The list and tuple are accessed with numerical indices

The dictionary is accessed with indices chosen by the programmer (consists of `key:value` pairs)

These composite data types can contain other variables[1]

[1] Including other composite data types, such that they are nested!

Slicing with numerical

Numerical indices can be accessed using slices in as described here:

```
a[start:stop]        # items start through stop-1
a[start:]            # items start through the rest of the array
a[:stop]             # items from the beginning through stop-1
a[start:stop:step]   # start through not past stop, by step
```

Examples

```python
a_tuple = (1,2,3)
a_list = [1,2,3]
a_dict = {
  'key_1':'value_1',
  2: 73,
  'key_4': ['a', 'nested', 'list', [1,2,3]]
  }

print(a_tuple[0]) # Note that Python is 0-indexed!
print(a_list[:2])
print(a_list[0:3:2])
print(a_dict['key_4'])
```

```
1
[1, 2]
[1, 3]
['a', 'nested', 'list', [1, 2, 3]]
```

Control flow

What is control flow

Control flow means writing code that controls the way data or information flows through the program

In Python, this is (mainly) done using either

- conditional logic in if-else statements
- loops

Conditional logic

Essentially: If something is true, do something

Pseudo-code

```
if statement is true:
  do something
```

In the example above, the block called `code` is run if the condition called `statement` is True (the boolean value)

Python is designed to look like pseudo-code

```python
if 1 == 1:
  print('Hello')
```

```
Hello
```

What if the condition fails?

We introduce an alternative!

```
if statement is true:
  do something
else:
  do something else
```

Which again looks similar in Python

```python
if 1 == 2:
  print('Hello')
else:
  print('1 does NOT equal 2')
```

```
1 does NOT equal 2
```

Python also supports `elif` (else-if)

Loops

When you want to do the same thing multiple times, loops are your best friend

Two types:

- For loops
- While loops

For loops

Do the same thing for each element in an *iterable* (e.g. a list)

```
for each element in iterable:
  do something
```

Once again, very similar

```
example_list = [1, 2, 3]

for i in example_list:
  print(i*i)
```

```
1
4
9
```

While loops

Do something while a statement holds

```
while statement is:
  do something
```

Commonly done with a counting variable, but not necessarily

```python
i = 0

while i < 3:
  print(i)
  i = i + 1
```

```
0
1
2
```

Make sure it terminates!

Reuseable code

It makes life easier

Reuse your own code

- self-defined functions

Reuse other's code

- built-in functions and packages

Reusing own code

Done using functions, which can be thought of as a recipe

You define:

- what the input is
- what it should do with
- what it should return

Extremely powerful!

You can also create your own packages/modules which you can import, but this is not covered.

How to define a function

The scaffold is as follows

```
def function_name(input_1, input_2, ..., input_k):
  something = do_something()
  return something
```

An example

```
def func_name(input_1, input_2):
  temporary_var = (input_1 + input_2)*2
  return temporary_var

func_name(2,3)
```

```
10
```

Python supports infinitely many inputs, default values and much, much more

How to use other’s code

Built-in functions

- So commonly used that they ship with Python and can be called immediately

Packages

- Contains functions
- Needs to be imported
- Sometimes also needs to be installed

Question

Do you know any built-in Python functions?

Built-in functions

Our dear friend `print()`!

But so many more:

```python
print('len is',len([1,2,3]))
print('sum is',sum([1,2,3]))
print('max is',max([1,2,3]))
print('abs is',abs(-1))
```

```
len is 3
sum is 6
max is 3
abs is 1
```

You won’t be able to remember everything, and once again Google is your best friend

Modules

Reusing other people's code is perhaps the most important part of Python!

- If you're doing a task, someone else probably has done it before
- You import the module (often with an alias) and then call functions from the module

Corresponds to reg, fixest, etable and so on

How to get them

Usually installed through `conda` or `pip`

If you need a specific module, Google “install module_name python”, e.g. for pandas it’s `conda install pandas`

- Install through the commmand-line interface (i.e. Anaconda Prompt)
- Install in notebook by prefacing with `%`, see here

In PyCharm, there’s a package manager window where you can search for packages

Some common packages

This will depend on the field you're operating in

- pandas, for loading data and data processing
- numpy, for numerical computations (matrices)
- matplotlib, for flexible plots
- seaborn, for quick plots
- sklearn, for machine learning

We will focus on pandas (this session) and sklearn (later sessions) due to time constraints

I will however shortly introduce the different modules

pandas

Series

First import

```
import pandas as pd
```

Most basic element is a Series (list / column)

```
series1 = pd.Series([1,2,3,4,5])
series2 = pd.Series(['a','b','c','d','e'])

print(series1)
print(series2)
```

```
0    1
1    2
2    3
3    4
4    5
dtype: int64
0    a
1    b
2    c
3    d
```

DataFrames

Series can be combined into DataFrames

```
data = {'column_1': series1, 'column_2': series2}
df = pd.DataFrame(data)

df.head()
```

	column_1	column_2
0	1	a
1	2	b
2	3	c
3	4	d
4	5	e

Tons of possibilities

The DataFrames are the main object in pandas

Usually loaded using `pd.read_csv` (dependent upon format, see list), but also offer support for `dta` or `SAS7BDAT`:

- `pd.read_stata`
- `pd.read_sas`

You will have time to work with pandas during the exercises

There are lots of guides online, e.g. in the documentation

numpy

Arrays

If you want to work with vectors and matrices, numpy is your friend!

```python
import numpy as np
array_1 = np.array([1,2,3])
array_2 = np.array([3,2,1])
matrix = np.array([array_1, array_2])
print('matrix:')
print(matrix)
print('slice:', matrix[0,:]) # supports slicing
print('dot + transpose:', array_1 @ array_2.T) # and dot products, transpos
```

```
matrix:
[[1 2 3]
 [3 2 1]]
slice: [1 2 3]
dot + transpose: 10
```

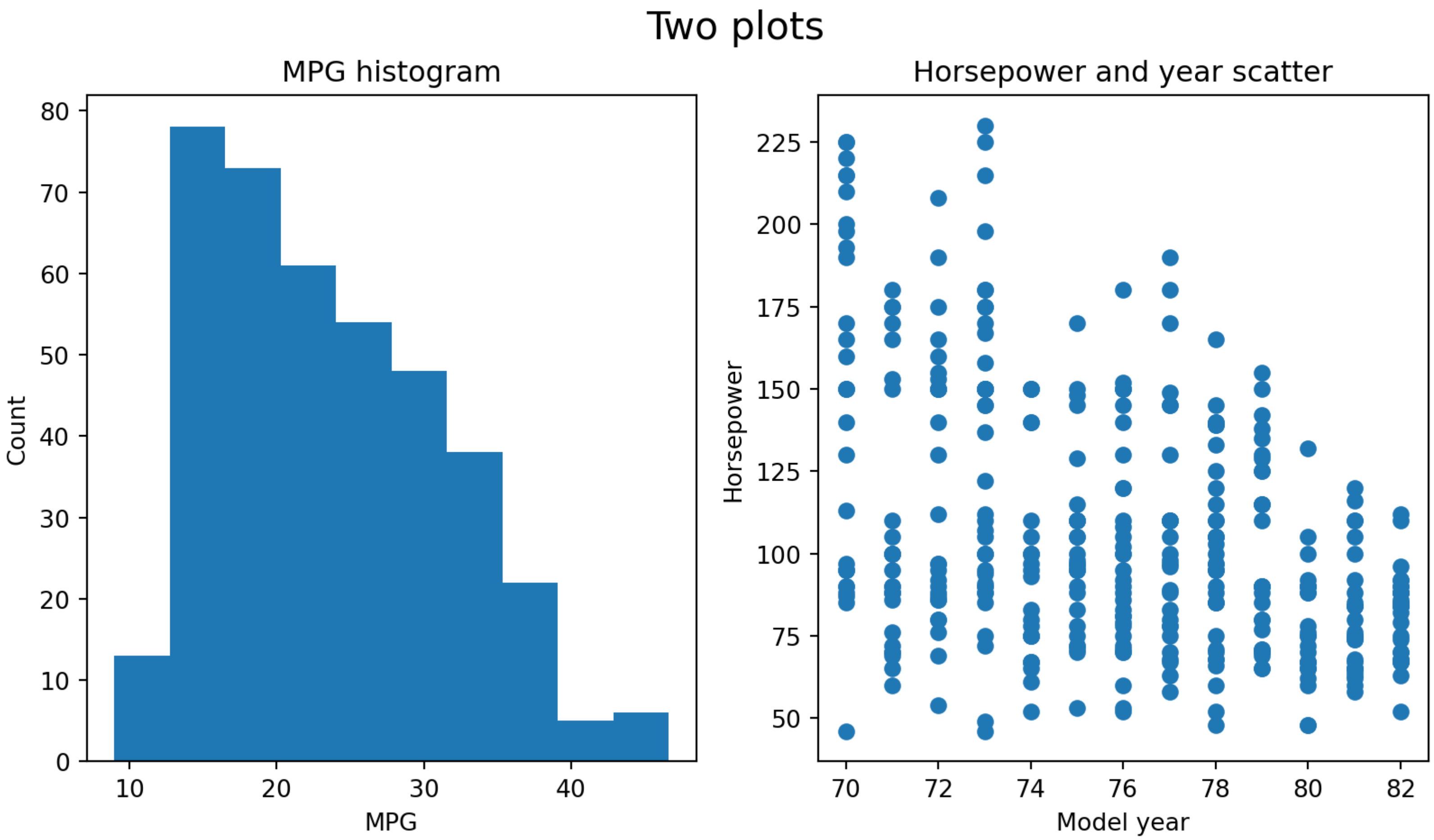
Only numeric data! Most matrix calculations are done under the hood (thank god!), so you probably won’t need this much

matplotlib

Flexible plots

Not always very intuitive (MATLAB-like syntax), but very flexible

▶ Show code



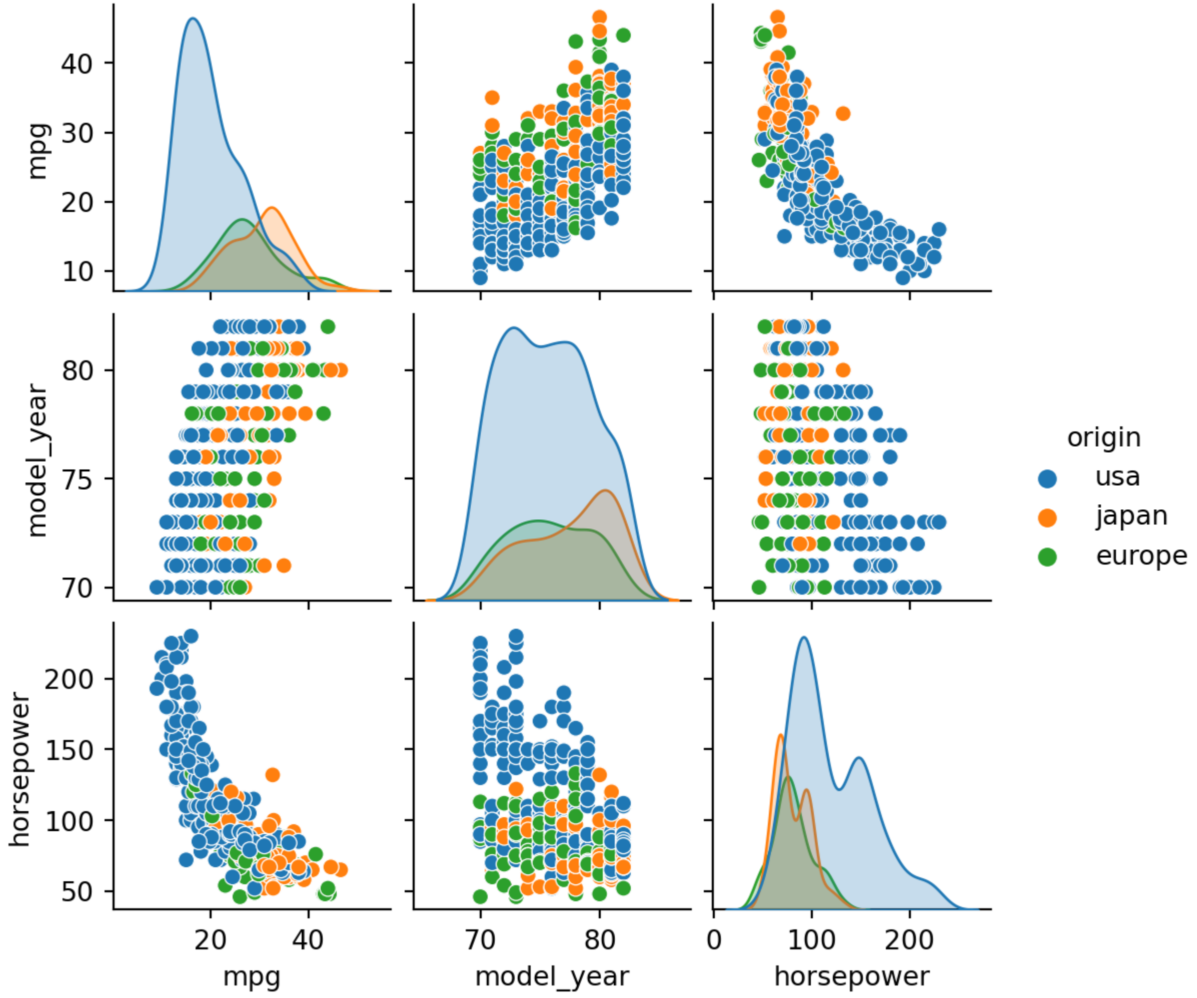
The figure (f) is the whole plot, whereas the axis (ax) contains the subplots, accessed through indices

seaborn

Quick and nice plots

Built on top of matplotlib – lots of powerful premade plots

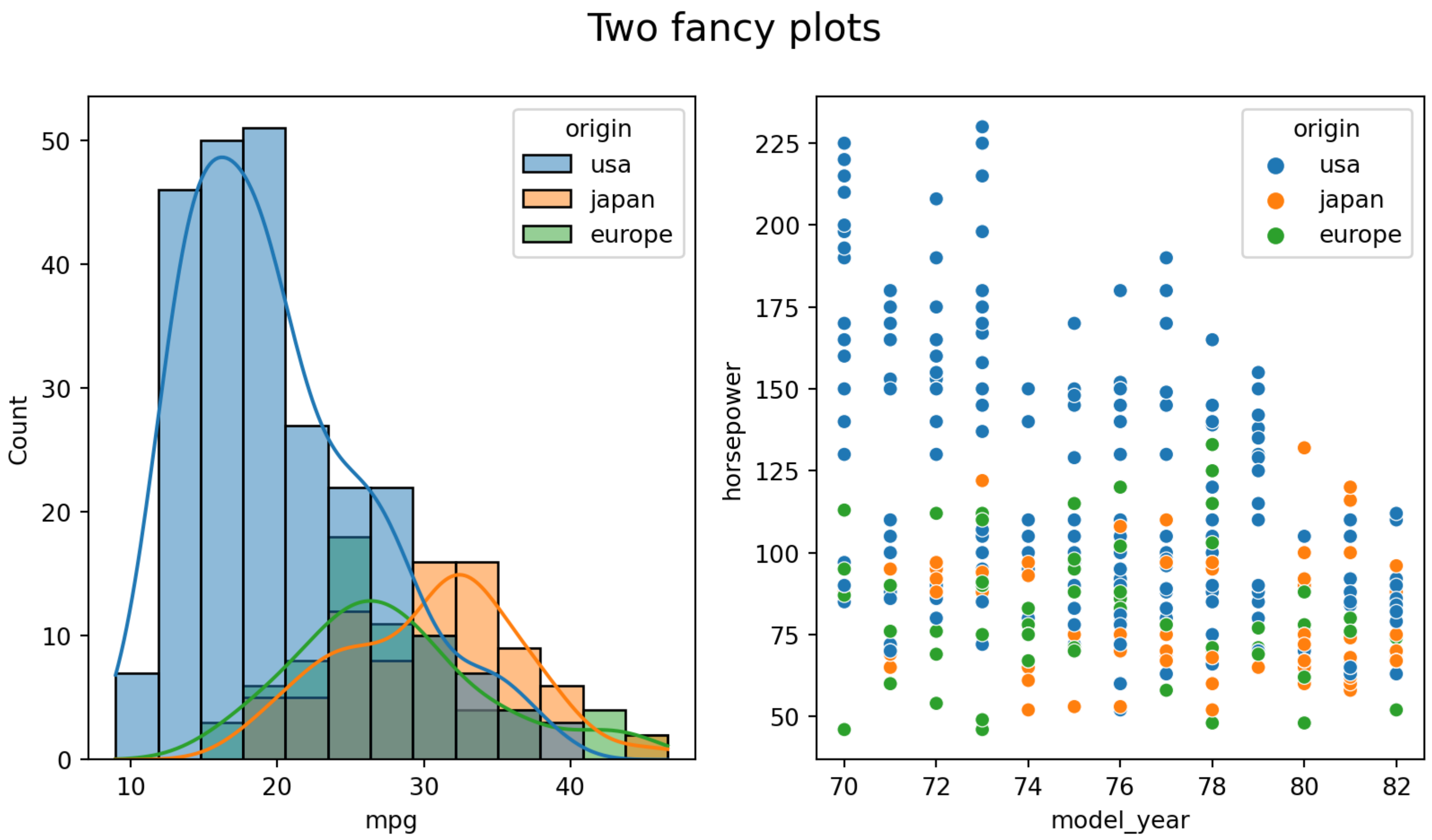
▶ Show code



The most powerful ones (like `pairplot()`) are not easy to post-process

plt and sns can be combined

▶ Show code



A large amount of different examples with code can be found online, e.g. here

sklearn

To be continued.. 👉 👉

To the exercises!