# PGR Methods and Skills Bootcamp

## Scientific Programming in Python

Ana Matran-Fernandez & Viola Fanfani

27 November 2018

Institute for Analytics and Data Science
University of Essex

# Today's plan

- 9.00 – 10.45: *Scientific programming in Python (I): Scipy, NumPy*
- 11.15 – 13.00: Scientific programming in Python (II): Pandas, Seaborn
- 14.00 – 17.00: Python practice in the lab

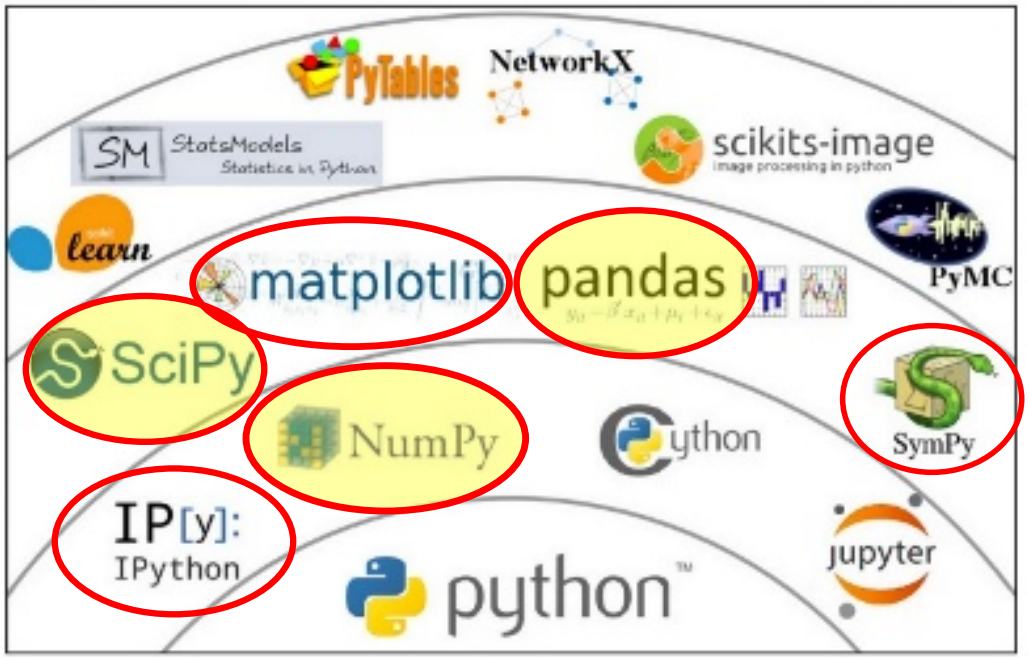
# Table of contents

# The Scipy ecosistem

# The Scientific Python Ecosystem

## PyData ecosystem

CONTINUUM ANALYTICS




ANACONDA

Image by Jake VanderPlas

3

# The Scientific Python Ecosystem

The SciPy ecosystem is a collection of open source software for scientific computing in Python.

Core packages:

- **NumPy**, the fundamental package for numerical computation. Creates and uses arrays, matrices and mathematical operators.
- **SciPy** library, a collection of numerical algorithms and domain-specific toolboxes, including signal processing, optimization, statistics and much more.
- **Matplotlib**, plotting package, basis for every advanced plotting libraries.

# The Scientific Python Ecosystem

The SciPy ecosystem is a collection of open source software for scientific computing in Python.

High-level libraries:

- **SciKit-Learn**, machine learning tools and algorithms.
- **Pandas** data analysis library. Provides structured data objects that make data-analysis of large datasets easier.
- **PyTables** to access data stored in the HDF5 format, minimises memory usage.
- **StatsModels** classes and functions for the estimation of many different statistical models, as well as for conducting statistical tests, and statistical data exploration.

# NumPy

# Why NumPy?

Short for Numerical Python, NumPy is the fundamental package required for high performance scientific computing and data analysis in the Python ecosystem.

It's the foundation on which nearly all of the higher-level tools, such as Pandas and scikit-learn, are built.

Many NumPy operations are implemented in C, making them super fast. For data science and modern machine learning tasks, this is an invaluable advantage.

# NumPy ndarray

NumPy provides an extension to the Python types, the `ndarray`, that is much more efficient at storing and manipulating numerical data than the built-in types.

ndarrays are n-dimensional arrays of homogeneous data types: being the same data-type they use the same memory and they are faster to use in operations. The exception: one can have arrays of (Python, including NumPy) objects, thereby allowing for arrays of different sized elements.

They have fixed sizes, every modification to the shape results in a new array being stored.

# Importing NumPy and creating an array

```
import numpy as np  # importing convention

a = np.array([1, 2, 3, 4, 5])
a.ndim  # number of dimensions of the array (1)
a.shape  # shape of the array (5,)

b = np.array([[3, 4, 5], [6, 7, 8]])
b.ndim  # 2
b.shape  # (2, 3)

# Some special arrays
np.zeros(5)  # 1-D array of 5 zeros
np.ones((6, 2, 3))  # 3-D array of one's
np.empty((2, 7))  # 2-D empty array
np.eye(4)  # 4-by-4 identity matrix
```

# Indexing

There are multiple ways of selecting specific rows and columns from a NumPy array.

The easiest way is slicing:

```
myArray[0]  # select first row
myArray[3, 5:7]  # specific elements from the 4th row
myArray[:2, 4, 26:30]
# and so on
```

What if we don't know which items exactly we want to access?

We may want to retrieve slices of the array based on a condition. We can do this using Boolean indexing.

# Boolean indexing

We can apply Boolean operators (which you saw in the lecture yesterday) to an ndarray to obtain a binary mask:
`mask = myArray != 0` returns a boolean array of the same shape of myArray, telling us all the elements that are not 0.

We can then do `myArray[mask]` to see only those values that were different from 0.

The `and` and `or` operators don't work with ndarrays. Instead, use `&` and `|`, respectively.

The results of these operations can be aggregated into a single value with the `any()` and `all()` methods.

We can also use the `np.where()` method to access specific values:

```
mask = np.where(myArray < 0)
myArray[mask] = 0
```

# Changing the Shape of the Array

We have seen how to visualise and select data from an existing ndarray, how can we add/delete rows/columns to it?

```
# Assume we have arrays of shapes:
array1.shape  # (20, 4)
array2.shape  # (10, 4)
array3.shape  # (20, 87)
np.hstack((array1, array3))  # stacks arrays horizontally; (20, 91)
np.vstack((array1, array2))  # stacks arrays vertically; (30, 4)
np.append(array1, array2)  # flattens both arrays and appends array2
                           # to the end of array1; (120,)
np.append(array1, array2, axis=0)  # (30, 4)

# Remove a row/column:
np.delete(array1, 2, axis=0)  # removes the 3rd row; (19, 4)
np.delete(array1, 2, axis=1)  # removes the 3rd column; (20, 3)
```

# Transposing and Reshaping

Transposing an array: `myArray.T` or `myArray.transpose()`

If myArray.ndim > 2, we can also permute its dimensions using transpose

```
# Assume we have:
array2d.shape #(20, 87)
array3d.shape #(10, 3, 28)

array2d.T  # (87, 20)
array3d.T  # (28, 3, 10)
array3d.transpose()  # (28, 3, 10)
array3d.transpose(1, 0, 2) # (3, 10, 28)
```

# Transposing and Reshaping

We can reshape our array:

```
# Assume we have:
array2d.shape #(20, 87)
array3d.shape #(10, 3, 28)

array2d.reshape((4, -1))  # (4, 435)
# -1 means we don't need to write this dimension;
# it's calculated automatically for us

# we can put the -1 anywhere
array2d.reshape((-1, 30)) # (58, 30)
# convert to 3D array
array2d.reshape((10, -1, 3))  # (10, 58, 3)

array3d.reshape((30, -1, 28))  # (30, 1, 28)
```

# Common operations

The power and beauty of NumPy are in its ability to vectorize operations. This means that we don't need to calculate values 1-by-1 using a for loop. E.g., `np.sqrt(myArray)`.

We can also obtain some summaries of the array:

```
myArray.mean() # average across the whole array
myArray.mean(axis=0) # average across columns
myArray.sum(axis=1) # aggregates all values across rows
myArray.std(axis=2) # standard deviation across the 3rd dimension

# How many values above 0
(myArray > 0).sum()

# Sorting values in place
myArray.sort()
# Keep the array as it is and return a sorted copy:
np.sort(myArray)
```

# NumPy as a Random Number Generator

The module **numpy.random** includes functions that we can use to generate random numbers.

```
# Array of shape (10, 23) of random numbers between [0, 1)
np.random.rand(10, 23)
# Array of 130 random numbers between [0, 50)
50 * np.random.rand(130)
# Random integers in [min, max)
np.random.randint(min, max)

# We can also draw numbers from a non-uniform distribution:
np.random.poisson(2.0)  # Poisson distribution with lambda=2
np.random.normal(1, 2, size=20)  # 20 values from a
#normal distribution with mean 0 and std 2

# If we set a seed, the numbers generated will be the same every time.
# Very important for debugging!!
np.random.seed(32)
# But very dangerous!!!!!!!!!!!
```

# Generating ranges

Two main functions in NumPy can be used to generate linear ranges of values: `np.arange` and `np.linspace`:

- `np.arange(start, stop, step)` generates values between [start, stop), with a `step` between consecutive values (by default, 1).
- `np.linspace(start, stop, num, endpoint=True)` returns `num` evenly spaced numbers over the specified interval. It's the preferred method when the step is not an integer.

These are very useful, for example, to create a time vector for plotting.

# Today's plan

- 9.00 – 10.45: Scientific programming in Python (I): NumPy, Scipy
- 11.15 – 13.00: ***Scientific programming in Python (II): Pandas, Seaborn***
- 14.00 – 17.00: Python practice in the lab

# Pandas

# What is Pandas?

Pandas is a Python Data Analysis Library that is extremely helpful to deal with **structured data**. It is probably the preferential library to deal with tables (.csv, .xlsx, .tsv ... ).

The explicit goal of pandas is to enable scientists to easily carry out statistical data analysis: data are managed in tables and they are built on Numpy datatypes, so using them with the Scipy ecosystem is straightforward.

Moreover, Pandas has been built to deal with large tables. Hence, it minimises memory usage and it has optimised methods to retrieve data fast.

# Data Structures

Series is a one-dimensional labeled array capable of holding any data type (integers, strings, floating point numbers, Python objects, etc.). The axis labels are collectively referred to as the index.

DataFrame is a 2-dimensional labeled data structure with columns of potentially different types. Like Series, DataFrame accepts many different kinds of input. Along with the data, you can optionally pass index (row labels) and columns (column labels) arguments. Each column of a DataFrame is a series with its own column label.

# Creating a DataFrame

Majorly there are two ways of creating a DataFrame:

1) Creating it using data from a dictionary:

```
d = {'one' : [1., 2., 3., 4.],
     'two' : [4., 3., 2., 1.]}
df = pd.DataFrame(d, index=['d', 'b', 'a'])

df = pd.DataFrame({'AAA' : [4,5,6,7],
                   'BBB' : [10,20,30,40],
                    'CCC' : [100,50,-30,-50]})

data = [(1,2.,'Hello'), (2,3.,"World")]
df=pd.DataFrame(data, index=['first', 'second'])
```

# Creating a DataFrame

2) Loading a table file

```
table = pd.read_table('data/table')

table = pd.read_table('data/iris.data.csv', sep=",")

table = pd.read_excel('tmp.xlsx')
```

# Visualising Data in the table

Pandas DataFrames provide a few methods for visualising their structure and data inside them.

```
df.head() # returns the first 5 lines of the table

df.tail() # returns the last 5 lines of the table

df.index() # returns the indices

df.columns()  # returns the columns

df.describe()   # returns synthetic values for each column,
                # like mean, sum...
```

# Indexing

There are multiple ways of selecting specific rows and columns from the table.

The easiest way, for someone that comes from usual Python/Numpy programming is to use slicing

```
df["column_1"] # for selecting column_1
```

```
df[0:3] # for selecting first 3 rows
```

but it is not recommended since it is not specific to pandas and there are more efficient ways of selecting data.

# Indexing

Pandas has other data access methods available:

- Selection by Label: with the `loc()` method, data can be selected using indices and columns names and it works with MultiIndex DataFrames too.

```
df.loc['20130102':'20130104',['A','B']]
df.loc[:,['A','B']]
```

- Selection by Position: with the `iloc()` method, data can be selected using integers in a more numpy-like way, but it is still efficient as the pandas loc indexing.

```
df.iloc[0:100,:]
df.iloc[0:3,1:5]
```

# Boolean indexing

Indexing with `loc` and `iloc` assumes that we know which rows and columns we want to access. Often times, we need to retrieve slices of the table based on a condition.

For this reason Pandas allows boolean indexing. With boolean operators applied to a DataFrame we obtain a binary mask: `mask = df > 0` returns a boolean DataFrame of the same size of df.

Allowed boolean operators are: | for *or*, & for *and*, $\sim$ for *not*.

The results of this operations can even be aggregated with the `any()`, `all()` methods.

# Setting Data

This map can be used to select rows/columns of the dataframe as follows:

```
df_selection = df[df < 0]
df_selection = df[ df.loc["column1"] > 0]
df_selection = df.where ( df<0, 0)
```

The first two selections return a sliced DataFrame only with the selected values.

The `where` method instead returns an object of same shape as the original DataFrame and whose corresponding entries are from df where the condition is True and otherwise are from other.

# Setting Data

We have seen how to visualise and select data from an existing DataFrame, how can we add rows/columns/values to it?

To set a single value in the DataFrame, Pandas recommend using `at` and `iat` that Access a single value for a row/column label pair.

```
df.iat[1, 2] = 10
```

With apply we can function along an axis of the DataFrame. It allows to conveniently iterate over the whole DataFrame without having to manually access ach row/column and allows using inline functions.

```
df.apply(lambda x: x**2, axis=1)
df.apply(np.sum, axis = 1)
```

Wih append we can extend a DataFrame adding a a row, while for columns we can just add a value or a Series to a column identifier.

# SQL like Operations

Since a DataFrame is essentially a table, Pandas provides some SQL-like functions to sort it, to group it, and to aggregate two different tables.

- `.sort_values(by=["column_1"])` sorts the table using the data in the specified column
- `.groupby(by=["column_1"])` returns an iterator where each key corresponds to the element we are grouping by and each item is the corresponding DataFrame
- with `.merge()` and `.join()` methods two DataFrames can be merged together using SQL-like rules.

Of course, there is much more to Pandas than what we covered, but these are fundamentals, necessary to have an idea of what Pandas is for. Important to mention is the fact that Pandas allows MultiIndexing ( a multi-tier index ) and manages TimeSeries.

This section has been built using the following resources:

Official Documentation:
http://pandas.pydata.org/pandas-docs/stable/

http://www.gregreda.com/2013/10/26/intro-to-pandas-data-structures/

# Seaborn

# What is Seaborn?

Seaborn is a library for making statistical graphics in Python. It is built on top of matplotlib and closely integrated with pandas data structures.

While matplotlib allows to draw an infinite number of plots with different options, Seaborn tends to be more restrictive, but at the same time greatly simplifies the visualisation of complex data.

From the seaborn project itself: ”If matplotlib “tries to make easy things easy and hard things possible”, seaborn tries to make a well-defined set of hard things easy too.”

# What is Seaborn?

Main functionality that seaborn offers:

- Specialized support for using **categorical variables** to show observations or aggregate statistics
- Options for visualizing **univariate or bivariate distributions** and for comparing them between subsets of data
- Automatic estimation and plotting of **linear regression** models for different kinds dependent variables
- **Multi-plot grids** that let you easily build complex visualizations

# What is Seaborn?

For matplotlib is important to understand the behaviour of pyplot objects and how to combine them is up to the developer. For Seaborn, instead, most of the work is in understanding what one needs to plot and to prepare the right Pandas DataFrame.

Take-home message: making nice plots is not futile and frivoulous. It helps you and others better understanding problems and results. Having a clear idea of what your data looks like at the beginning can spare you weeks of work.

For the rest of this presentation we are going to use Seaborn official tutorial as a guide to cover all the different types of plots that Seaborn offers.

# Basic Usage

Some of Seaborn functions only require data arrays as inputs:

`seaborn.distplot(x)` plots the distribution of the variable x
`seaborn.jointplot(x, y)` plots the bivariate distribution of x and y

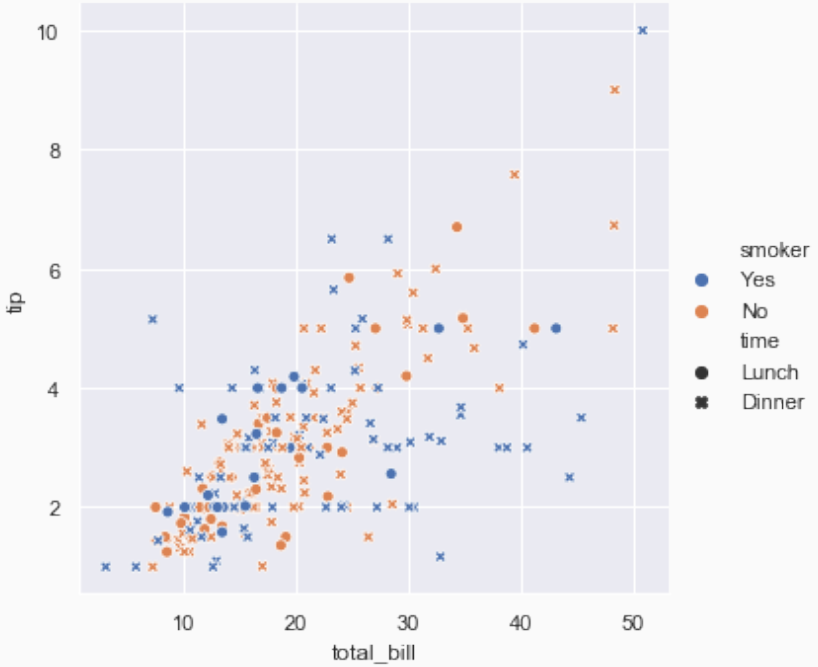
Others work only with DataFrames, specifying which columns to use, since they are automaticaly aggregating data:

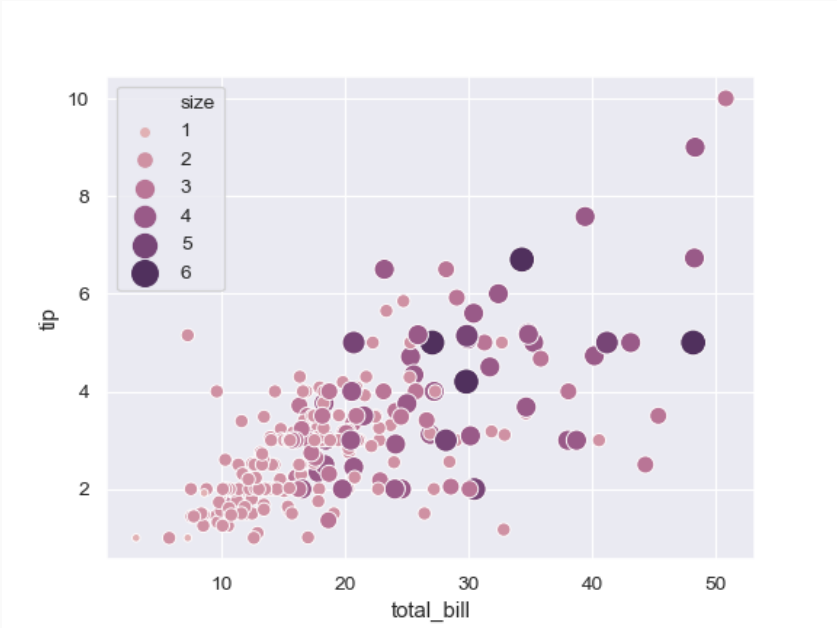
```
seaborn.catplot(x="column_x", y="column_y",
hue="kind", data=dataframe)
```

# Visualizing statistical relationships

```
seaborn.relplot(x="total_bill", y="tip", hue="smoker",
style="time", data=tips)
```

```
sns.scatterplot(x="total_bill", y="tip", hue="size",
size="size", sizes=(20, 200), hue_norm=(0, 7), legend="full",
data=tips)
```

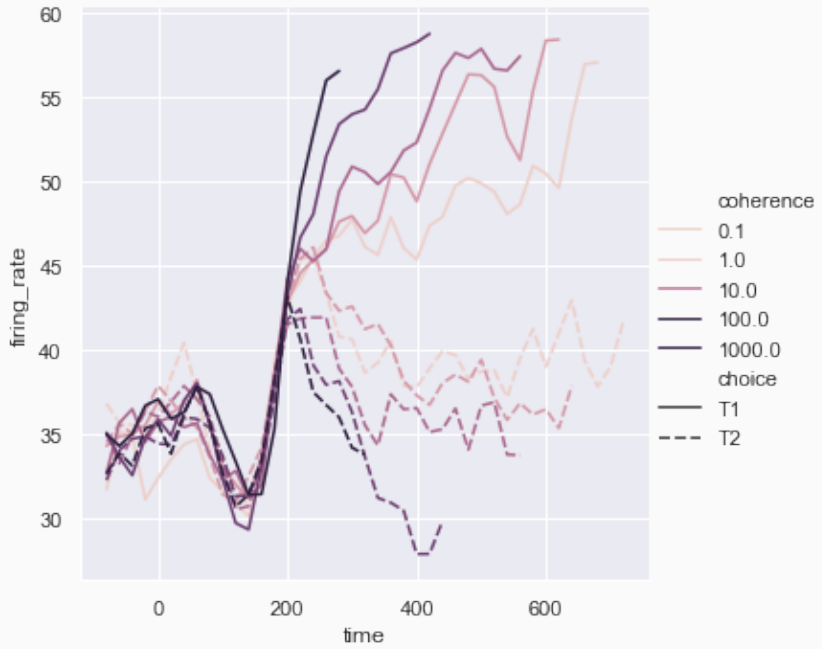


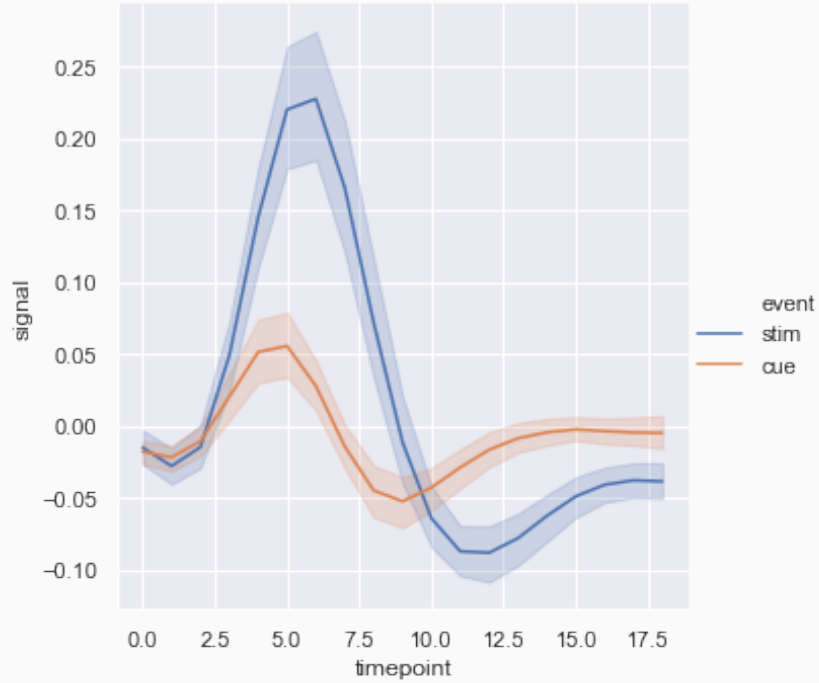



# Visualizing statistical relationships

```
sns.relplot(x="time", y="firing_rate",hue="coherence",
style="choice", palette=palette, kind="line", data=dots)
```

```
sns.relplot(x="timepoint", y="signal", hue="event", kind="line",
data=fmri);
```

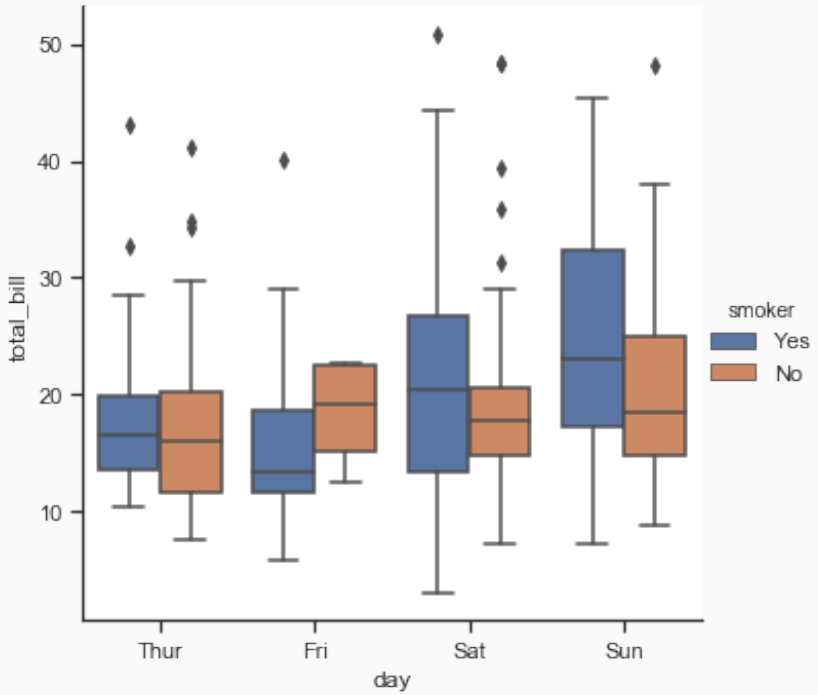


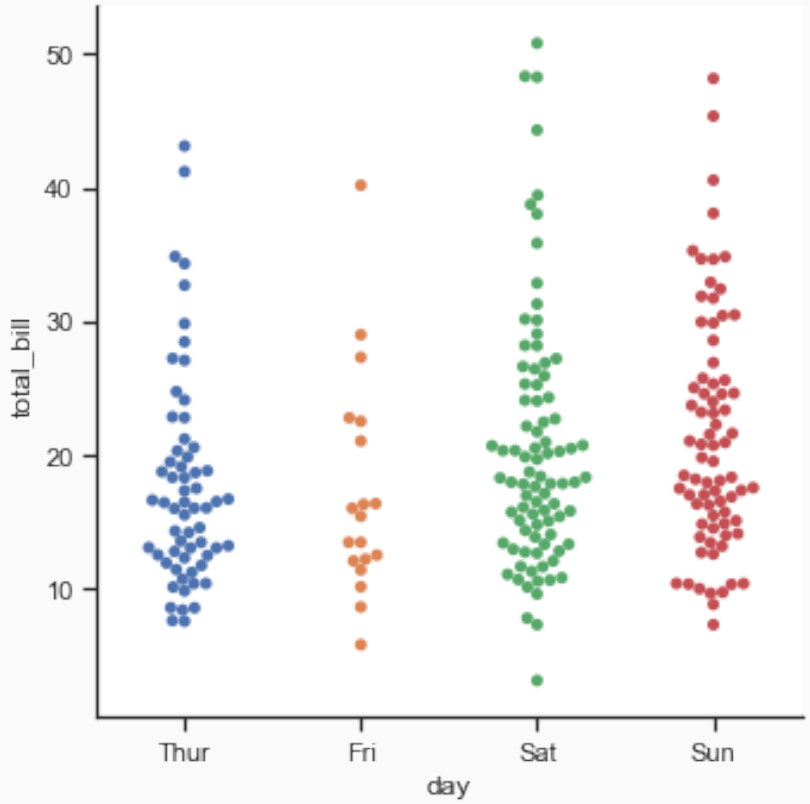



# Plotting with categorical data

```
sns.catplot(x="day", y="total_bill", hue="smoker", kind="box",
data=tips)
```

```
sns.catplot(x="day", y="total_bill", kind="swarm", data=tips)
```

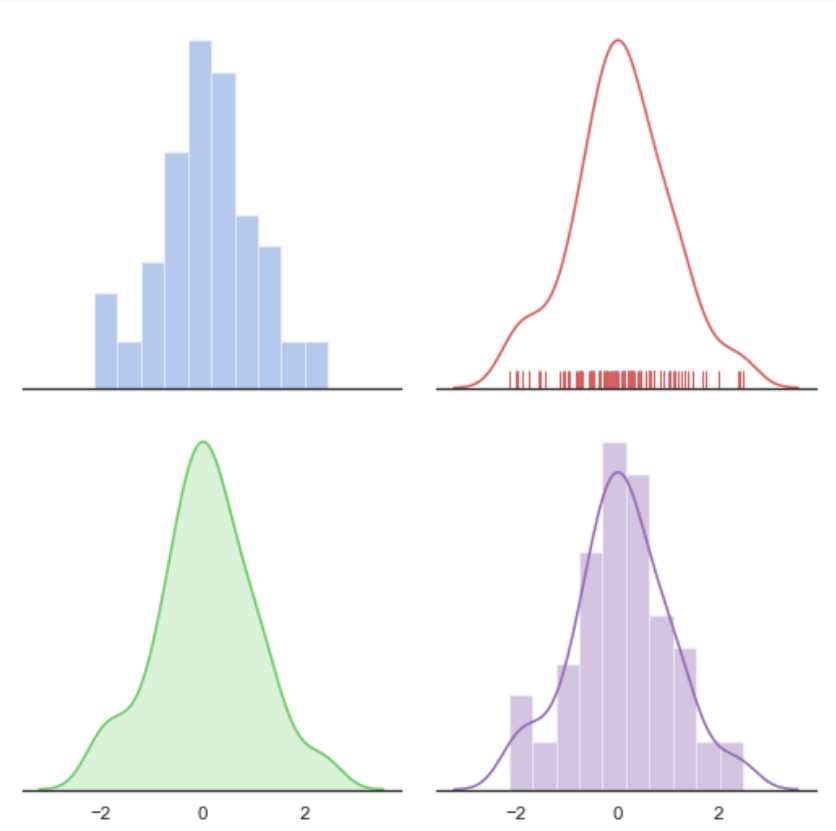


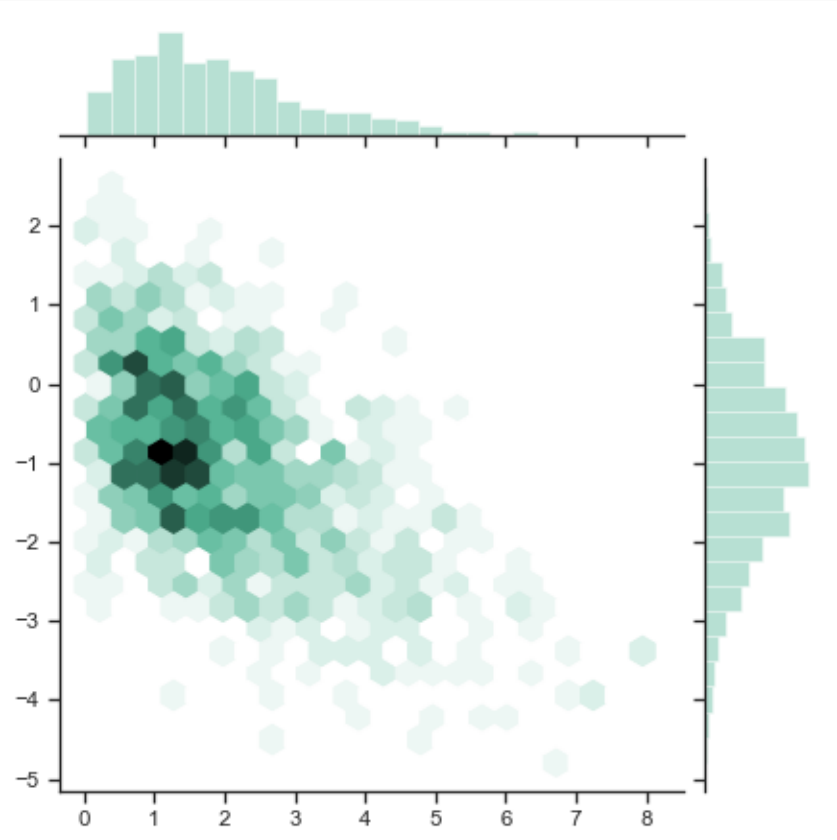



# Visualizing the distribution of a dataset

```
sns.distplot(d, kde=False, color="b", ax=axes[0, 0]) ...

sns.jointplot(x, y, kind="hex", color="4CB391")
```

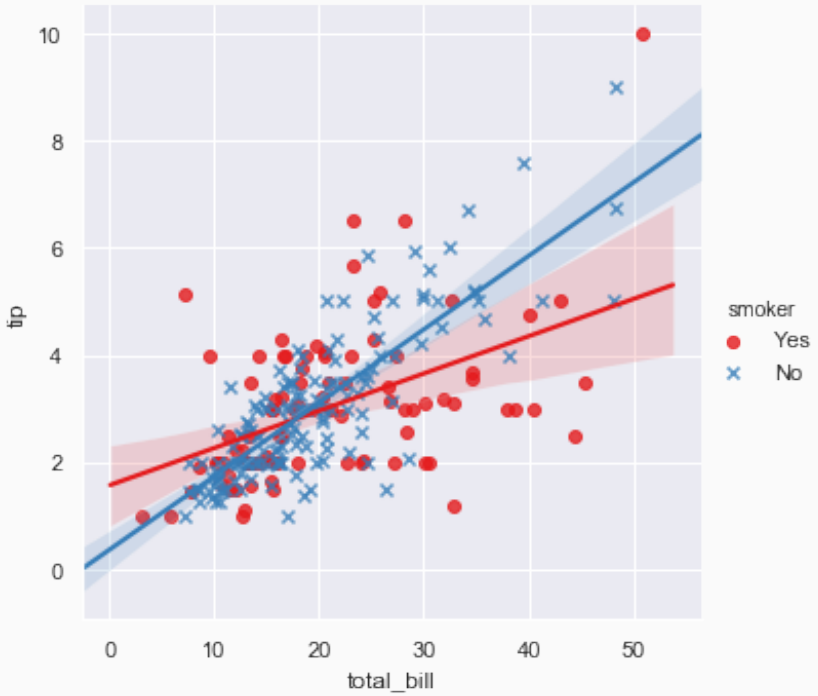


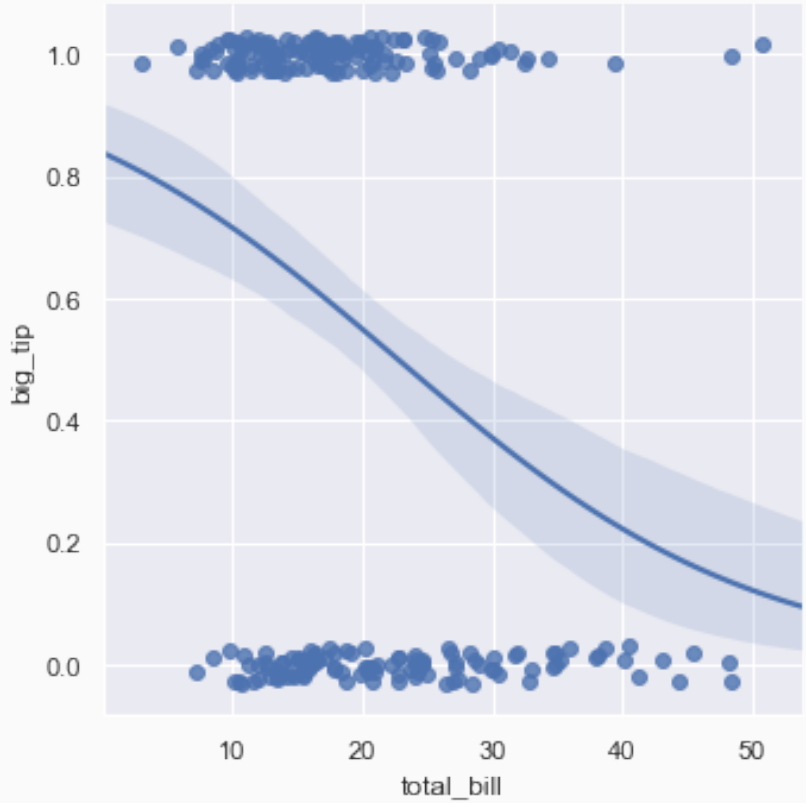



# Visualizing linear relationships

```
sns.lmplot(x="total_bill", y="tip", hue="smoker", data=tips,
markers=["o", "x"], palette="Set1")

sns.lmplot(x="total_bill", y="big_tip", data=tips,
logistic=True, y_jitter=.03)
```

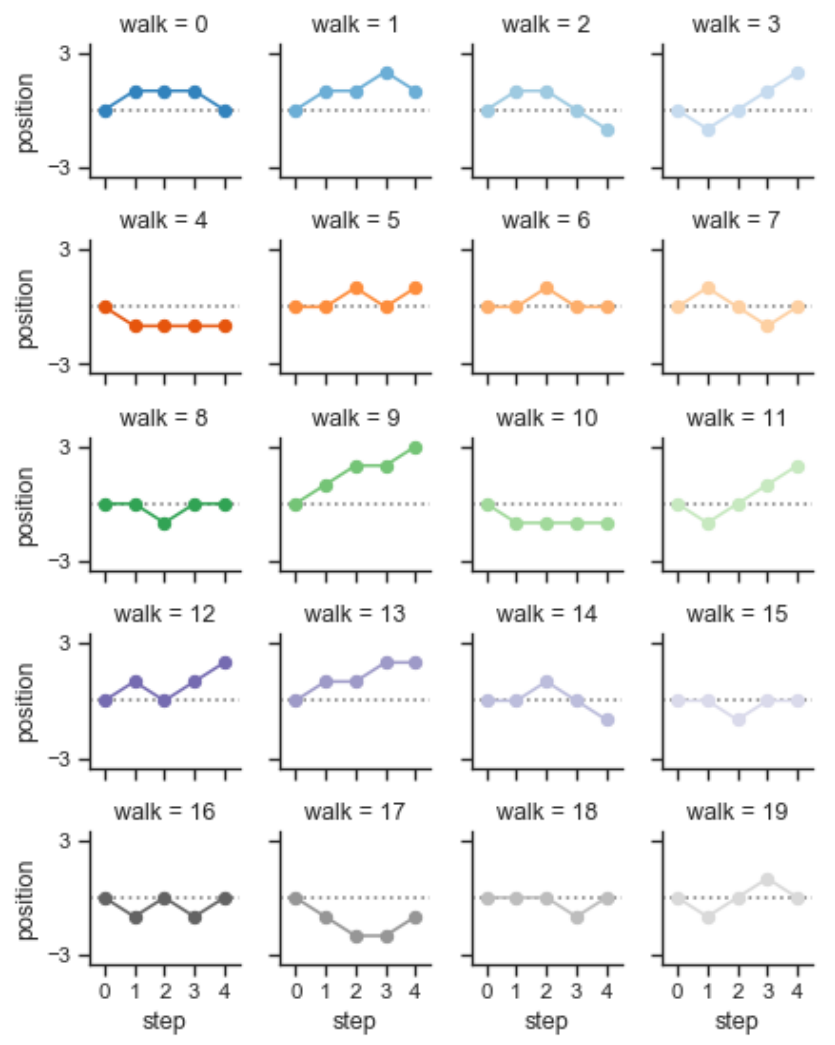


# Combining Multiple Plots

grid.map(plt.plot, ”step”, ”position”, marker=”o”)

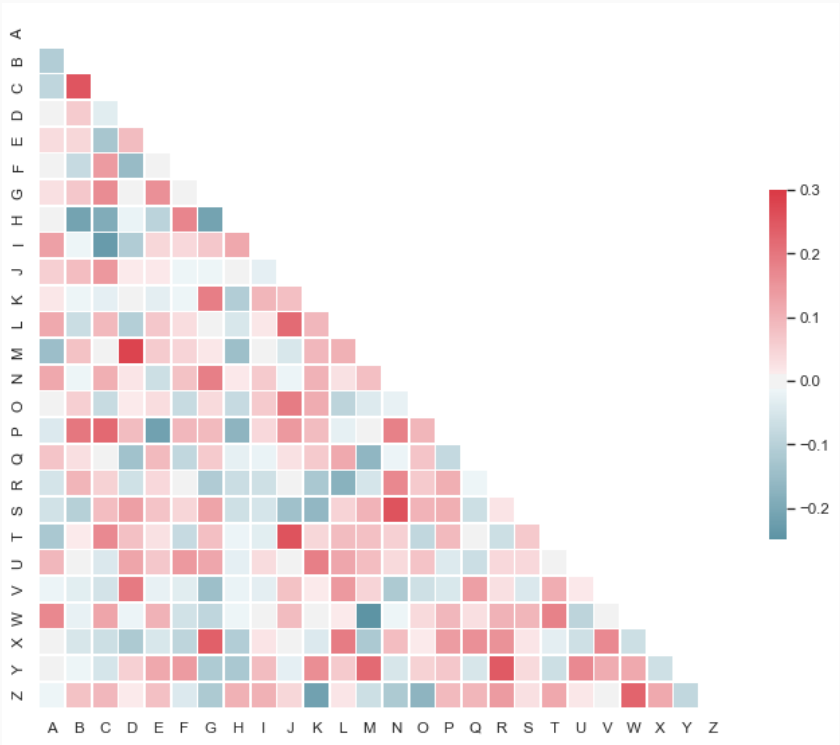
```
grid = sns.FacetGrid(df, col="walk", hue="walk", palette="tab20c", col_wrap=4, height=1.5)
grid.map(plt.plot, "step", "position", marker="o")
```

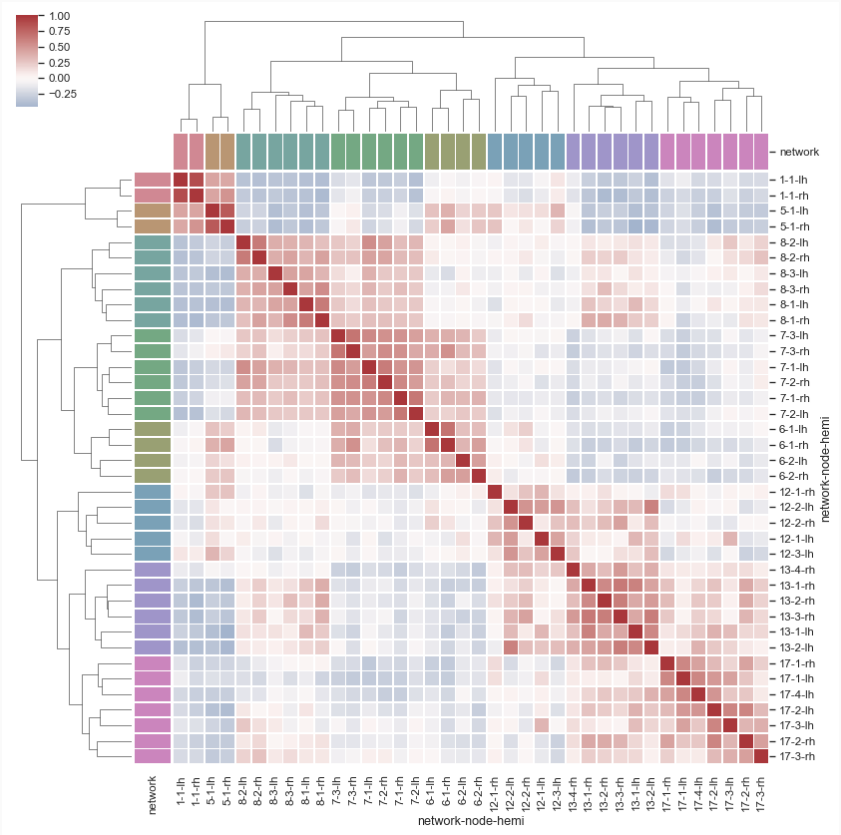


# Matrix plots

```
 sns.heatmap(corr, mask=mask, cmap=cmap, vmax=.3, center=0,
square=True, linewidths=.5, cbar_kws="shrink": .5)

sns.clustermap(df.corr(), center=0, cmap="vlag",
row_colors=network_colors, col_colors=network_colors,
linewidths=.75)
```



# Today's plan

- 9.00 – 10.45: Scientific programming in Python (I): Numpy, Scipy
- 11.15 – 13.00: Scientific programming in Python (II): Pandas, Seaborn
- 14.00 – 17.00: ***Python practice in the lab***