wbs
WARWICK BUSINESS SCHOOL
THE UNIVERSITY OF WARWICK

For the Change Makers

Advanced Programming for Data Science

Day 1 (11th May)
Information Systems and Management
Warwick Business School
Term 3, 2018-2019

Agenda

- Day 1:
  - Data collection: web scraping using Requests and BeautifulSoup.
  - Data visualization: Seaborn
  - Data cleaning and manipulation: Numpy and Pandas *(regular expression)*.
- Day 2:
  - Conventional machine learning: clustering using Scikit-Learn
  - Deep neural network: classification and time series forecasting using Keras
  - Case study

Python basic

- What is a variable?
- Why do we use variable?
- How to name a variable?

Review: statement flow

- Sequential flow
- Conditional flow
- Loop flow

Conditional flow

- Format

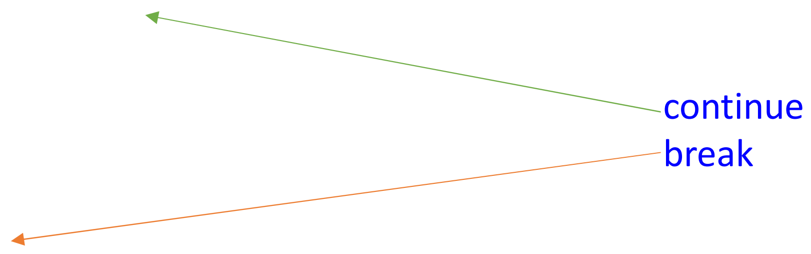
```
if condition:
        statement body
elif condition:
        statement body
…
else:
        statement body
```

x	and	y	Returns
True	and	True	
True	and	False	
False	and	True	
False	and	False	

x	or	y	Returns
True	or	True	
True	or	False	
False	or	True	
False	or	False	

Loop flow: while

```
while condition:
        statement body
statements
```



Loop flow: for

```
for iterator in iterable_object:
        statement body
```

List, tuple and string

- They are objects as a collection of sequence data, which is iterable.
- List is mutable while tuple and string are immutable.
- Indexing and slicing

Function

- A blackbox that contains a block of organized, reusable code.
- Better modularity and a high degree of code reusing.
- Parameter, argument and return value.

```
def function_name([parameters]):
        statement body
        [return]
```

- Function needs to be defined before called

Module

Install module: pip vs. conda

Import a module or functions:

import *module_name*

from *module_name* import *function_name*

Data Collection

Collecting publicly available data

- Increasing number of digital trace data that are publicly available online.
- Two major approaches to collect those data:
  1. Web scraping
  2. APIs
  3. Open data

Some basis of HTML

- Webpages are created using **HTML (Hypertext Markup Language)**, with CSS (Cascading Style Sheets) and JavaScript.
- HTML is a standard markup language (others like XML, LaTex) that uses tags to annotate the webpage content so they can be displayed as desired.

A simple html example

```
<!DOCTYPE html>
<html>
<head>
<title>A simple html example</title>
</head>
<body>
<p id="first1"> Zhewei </p>
<p id="last1"> Zhang </p>
</body>
</html>
```

A simple html example

```
<!DOCTYPE html>
<html>
<head>
<title>A simple html example</title>
</head>
<body>
<p class = "name" id="first1"> Zhewei </p>
<p class = "name" id="last1"> Zhang </p>
</body>
</html>
```

- Most tags will be used in pairs with opening tag and closing tag, i.e. <head> and </head>.
- Content between a pair of opening and closing tags will be displayed accordingly, which is also called element in a html file.

A simple html example

```
<!DOCTYPE html>
<html>
<head>
<title>A simple html example</title>
</head>
<body>
<p class = "name" id="first1"> Zhewei </p>
<p class = "name" id="last1"> Zhang </p>
</body>
</html>
```

- All HTML elements can have attributes that provide additional information about that element.
- Attributes are always specified in the opening tag.
- For web scraping, we normally deal with two kinds of attributes: id and class.
  - id can uniquely identify individual elements.
  - class can identify a set (class) of similar elements.

Web scraping in two steps

1. Access the webpage and download the html file.

- **Requests**
- https://2.python-requests.org/en/master/user/quickstart/

2. Extract elements (data) from a webpage, which can be located by the tags and attributes.

- **BeautifulSoup**
- https://www.crummy.com/software/BeautifulSoup/

Fetch the webpage with Requests

- Requests is a HTTP library for Python that help you to make http request to the web server and fetch the webpage.
- You send a request to the web server asking for certain webpages.
- You can make **get** and **post** request.

```
import requests
url = "test.html"
result = requests.get(url)
```

Requests results

- The *`get()`* function will send a http request to retrieve the webpage specified. The only required argument for *`get()`* is the url address to the webpage.
- Returns a requests object that contains various information about the webpage retrieved with attributes:
  - status_code: 200 is good and 4xx (403,404,408,etc.) is bad.
  - encoding: how characters are coded digitally, utf-8 is generally the best.
  - text: the plain content of the webpage, including all markings.

Retrieve links with parameters

- Some webpages, such as search results, may require you to send necessary parameters.
- http://yourmainlink.com?name=python&format=ebook&lang=en
- You can pass those parameters by using a dictionary as the argument param.
- `payload = {'key1' : 'value1', 'key2' : 'value2'}`
- `payload = {'name' : 'python', 'format' : 'ebook', 'lang' : 'en'}`
- `result = requests.get(url,params=payload)`

Deal with limits

- Regardless legal issue, websites generally do not like scraping as it can take up the bandwidth. Therefore, many websites limit the number of requests you can make within certain time period. Your requests can be blocked after reaching the limit.
- So, be considerate, and pause a while if needed.
- proxy and headers can be provided as additional arguments to the *`get()`*.

Parsing your result with BeautifulSoup

- You need to parse the text retrieved by Requests: convert the plain text into structured data.
- BeautifulSoup is a parser library to create such tree-structured data by parsing HTML or XML files.

```
from bs4 import BeautifulSoup
url_html = result.text
url_content = BeautifulSoup(url_html, 'html.parser')
```

Navigate the parsed data

- Parsed result will be return as a BeautifulSoup object that has a range of methods to help you navigate and search the data.
    1. Using tag names directly.

    ```
    url_content.head # return the first matching element.
    ```

    2. Using *find()* and *find_all()*

    ```
    url_content.find_all('p') # find_all() returns a list
    ```

    3. Using select()

    url_content.select() # use CSS selectors instead of HTML tags.

Exercise

- Write a script to extract FTSE indices from London Stock Exchange.
- https://www.londonstockexchange.com/exchange/prices-and-markets/stocks/indices/ftse-indices.html

Other thoughts

- Check robots.txt before scraping the data. It is basically an instruction telling web robot Dos and Don'ts. This can be found by adding it to the url of the main site. For example, https://www.warwick.ac.uk/robots.txt.
- User agent spoofing. Sometimes, the web server will not response if no user agent is provided. You may manually create one or use some library (e.g. fake-useragent) to generate a random one and pass to the header argument.
- Set a timeout to avoid indefinitely waiting by passing a value (in seconds) to argument timeout.

Downloading data through APIs

- Many company provide APIs for developer to access their data.
- Many third parties also develop API-like libraries for data collection.
- Some considerations include:
  - Accuracy.
  - Ease of use.
  - Eligibility.
  - Limitation.
  - Cost.

Open datasets

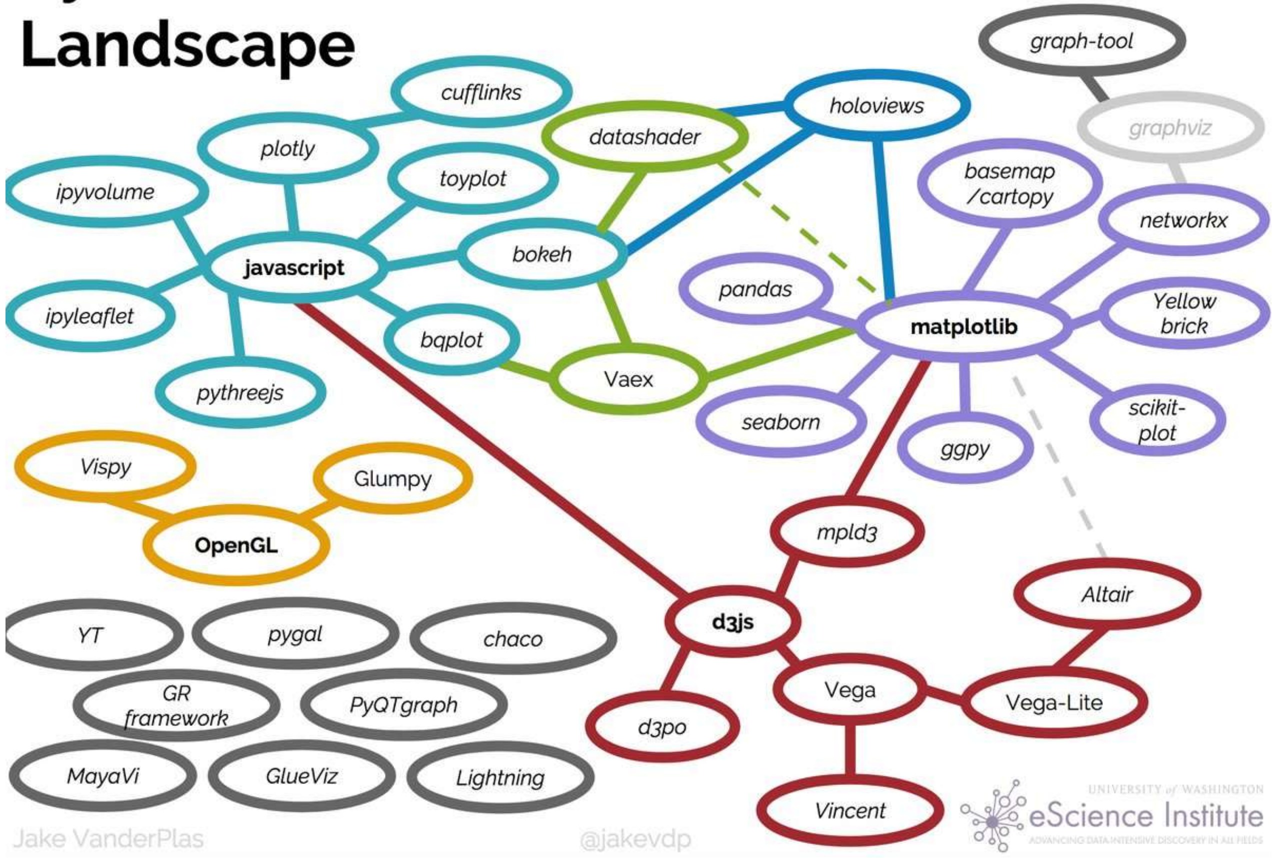
- There are many open datasets that you can use together with your internal dataset and collected dataset.
- https://toolbox.google.com/datasetsearch
- https://trends.google.com/trends/explore
- http://data.europa.eu/euodp/en/data/
- https://opencorporates.com/

Data Visualization in Python

Data visualization

- Eyeballing is a good way to have some quick initial understanding of your data. It is often used as the first step to start working on the data you just collected.
- There are tons of data visualization libraries for Python. Many data analysis libraries also provide built-in visualization functions.

Python's Visualization Landscape



Matplotlib and Seaborn

- **Matplotlib** is a plotting library for Python. It is the basis of many other libraries' visualization functionality. General and powerful yet a bit tricky sometimes.
- **Seaborn** is a data visualization library based on Matplotlib. It focuses on statistical visualization and provides easier, more user-friendly way to generate prettier graphs.
- We use build-in datasets for demonstration. You can explore these datasets here: https://github.com/mwaskom/seaborn-data

Tips dataset

total_bill	tip	sex	smoker	day	time	size
16.99	1.01	Female	No	Sun	Dinner	2
10.34	1.66	Male	No	Sun	Dinner	3
21.01	3.5	Male	No	Sun	Dinner	3

```
import seaborn as sns
tips = sns.load_dataset("tips")
```

`Function names`

Important things

Parameter names

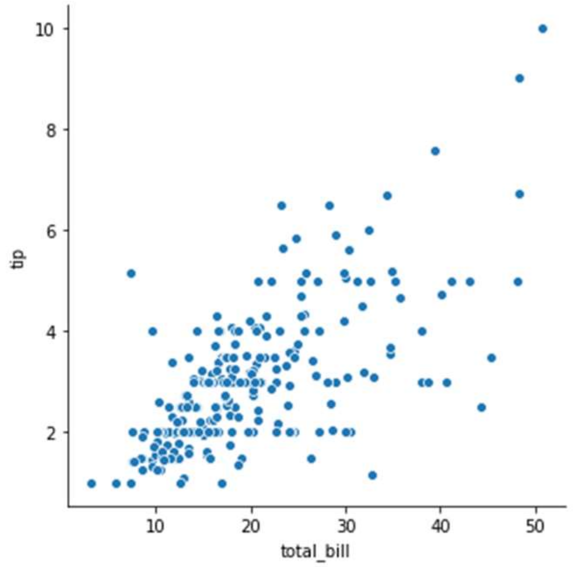
Values for argument

Scatter Plots

- Scatter plot is one of the most used visualization to explore the relationships between (two) variables.
- `relplot()` is a seaborn function that draws relational plots.

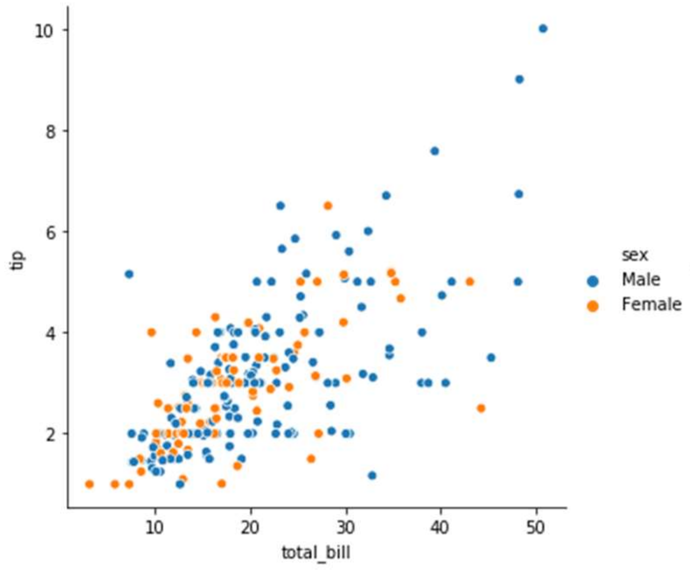
```
sns.relplot(x="total_bill",
y="tip", data=tips)
```

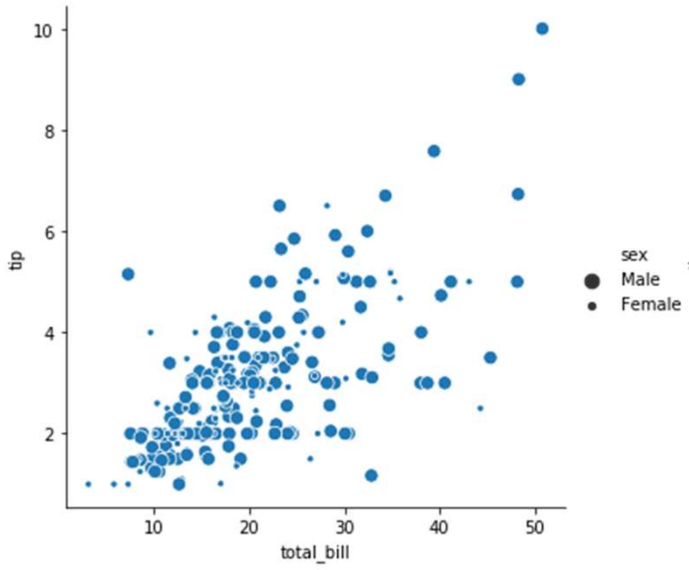
#three required arguments: x variable name, y variable name and the dataset name. *Values of x and y need to be numeric*.

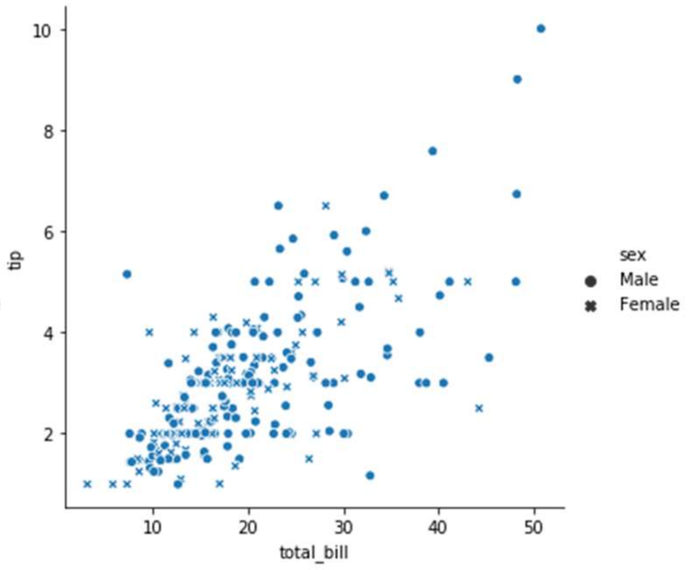


Optional arguments

- You may specify additional variables from the dataset to be used as the extra dimensions to group your data points, by passing the variable name to the arguments:
    1. hue: your data will be group in different colours.
    2. size: your data will be group in different size.
    3. style: your data will be group in different style.
    4. They can be used together.


tip
total_bill
sex
Male
Female


tip
total_bill
sex
Male
Female


tip
total_bill
sex
Male
Female

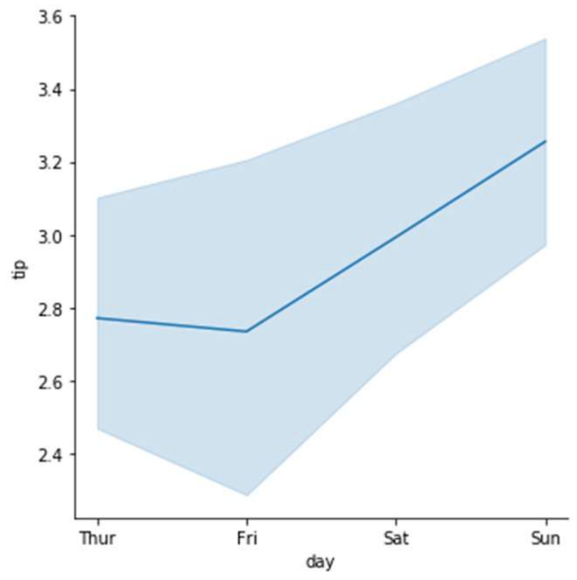
Exercise

- Draw a scatter plotter about the relationship between tip and total bill. Explore how different factors, such as smoker, day, time, size may affect this relationship.

Line plot

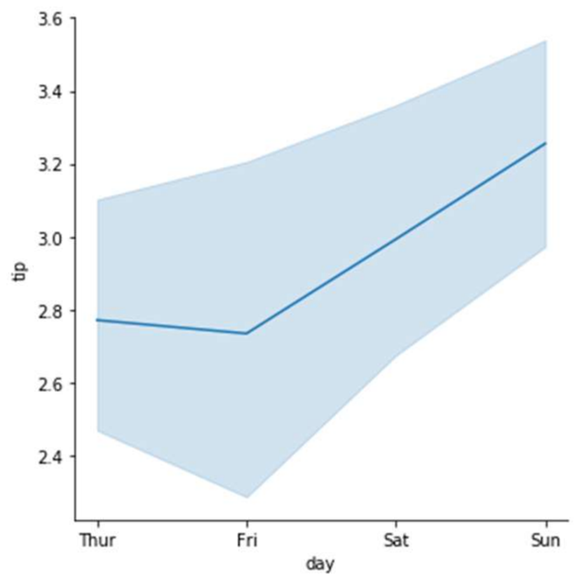
- When you have continuous variables, such as time, it can be a good idea to view the changes with line plot.
- An optional argument in *relplot()* is kind, with default value "scatter" for scatter plot, can be passed with value "line" to draw a line plot.

```
sns.relplot(x="day", y="tip",
kind='line', data=tips)
```

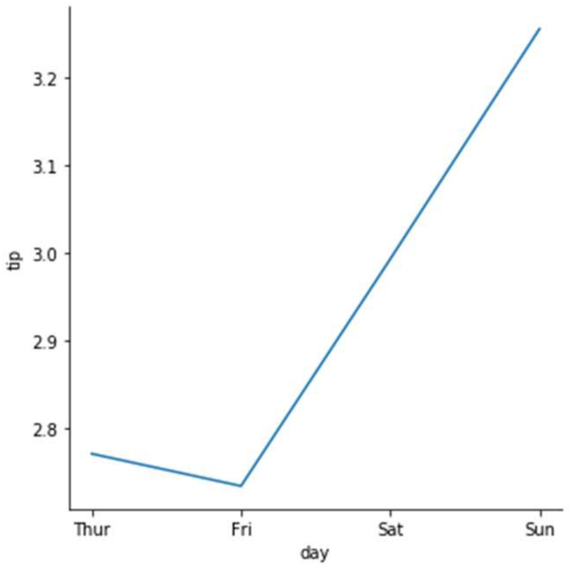


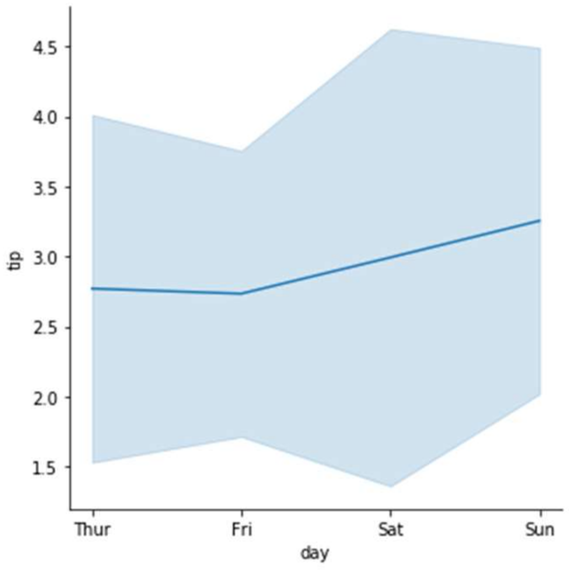
Line plot

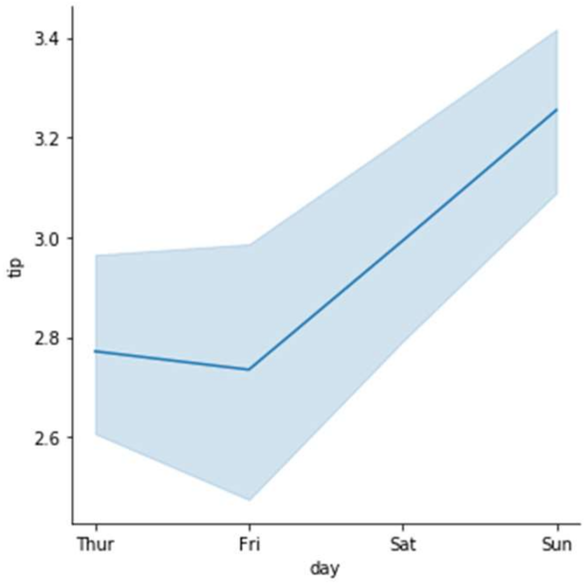
- By default, Seaborn aggregates the multiple measurements at each x value by plotting the mean and the 95% confidence interval around the mean
- You can change both by passing arguments:
    1. ci: value, none, or 'sd'.
    2. estimator: none or other pandas estimator.



Different confidence intervals

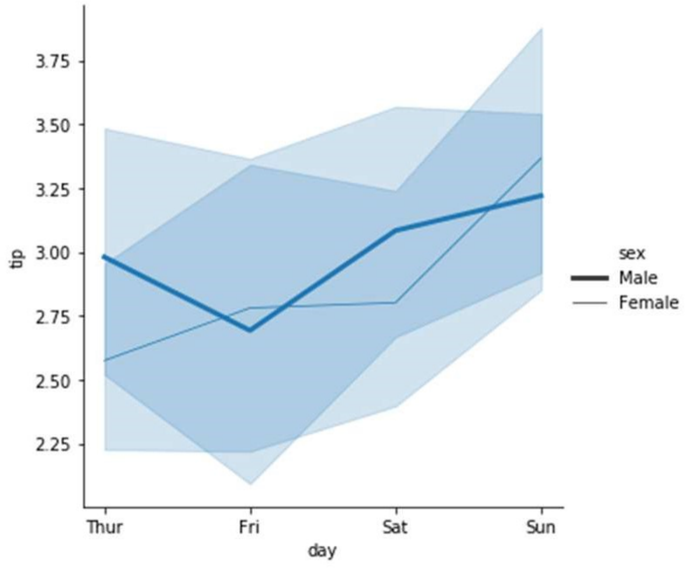


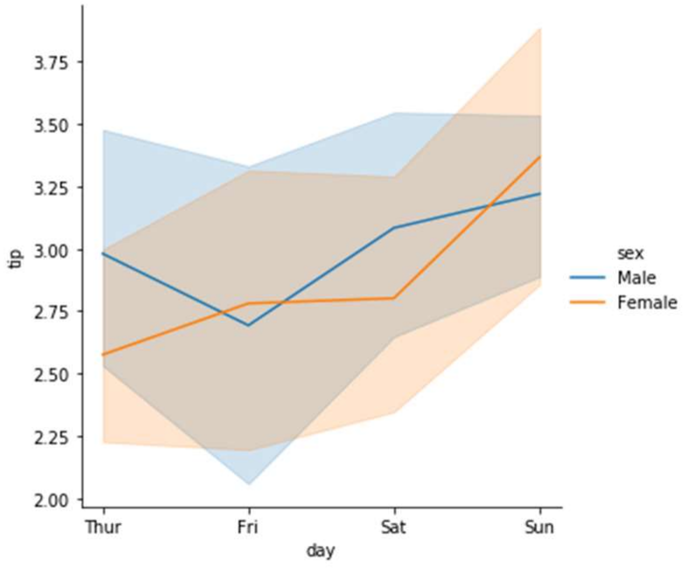


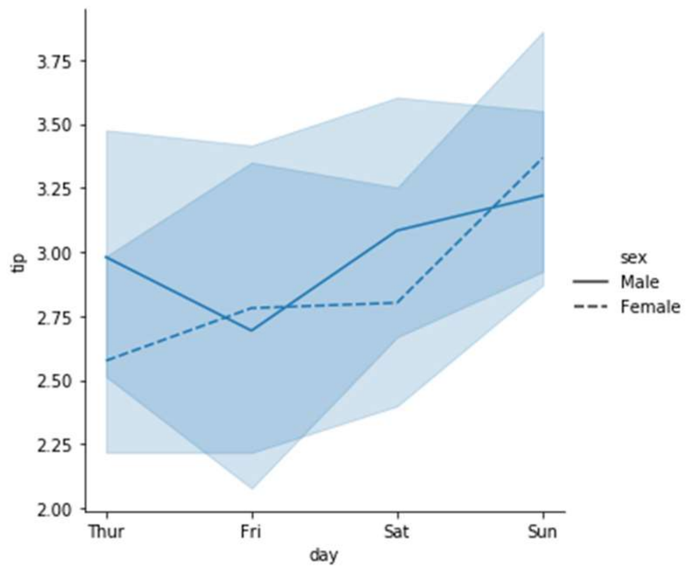


Line plotting by groups

- You can also create multiple lines based on the groups spitted by hue, size and style.



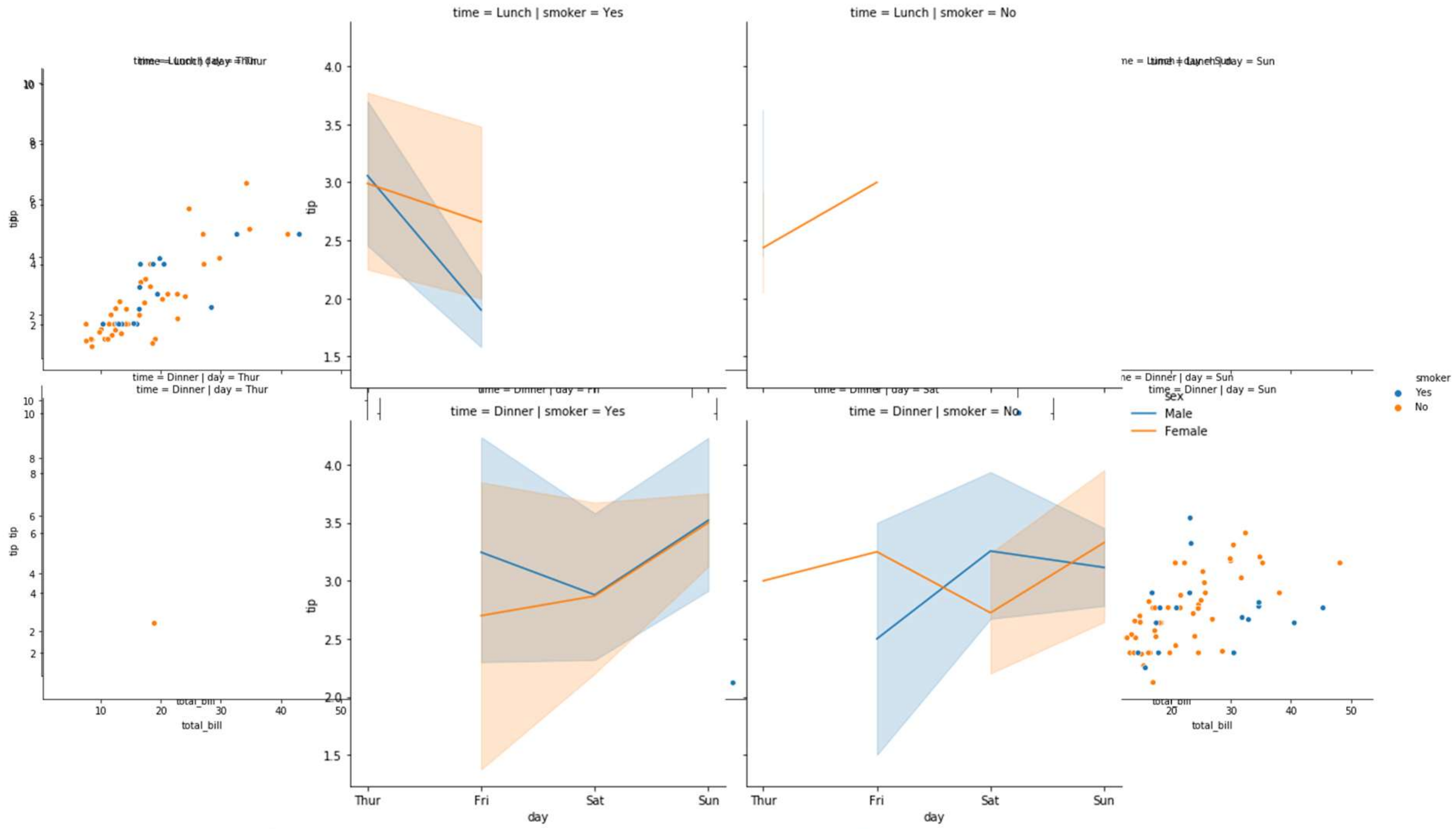




Showing multiple relationships

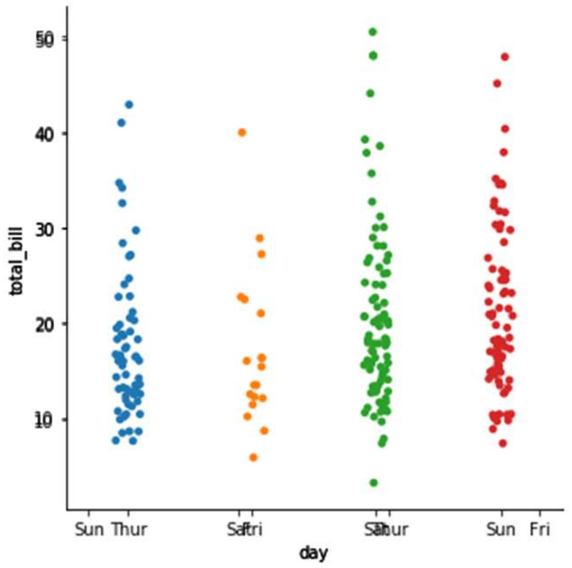
- Very often, when you explore your dataset, you would like to compare multiple relationships at once.
- You can create a grid of graphs and specify facets (grouping dimensions) by passing arguments:
  1. col: the variable used for splitting in horizontal direction.
  2. row: the variable used for splitting in vertical direction.

```
sns.relplot(x="total_bill", y="tip", col='day',
row='time', data=tips)
```

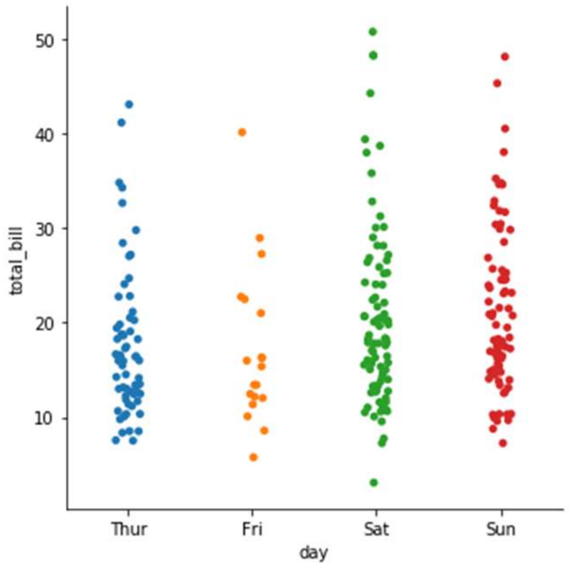

time = Lunch | smoker = Yes
time = Lunch | smoker = No
time = Dinner | smoker = Yes
time = Dinner | smoker = No
time = Lunch | day = Thur
time = Dinner | day = Thur
time = Lunch | day = Sun
time = Dinner | day = Sun
tip
day
total_bill
Thur
Fri
Sat
Sun
sex
Male
Female
smoker
Yes
No

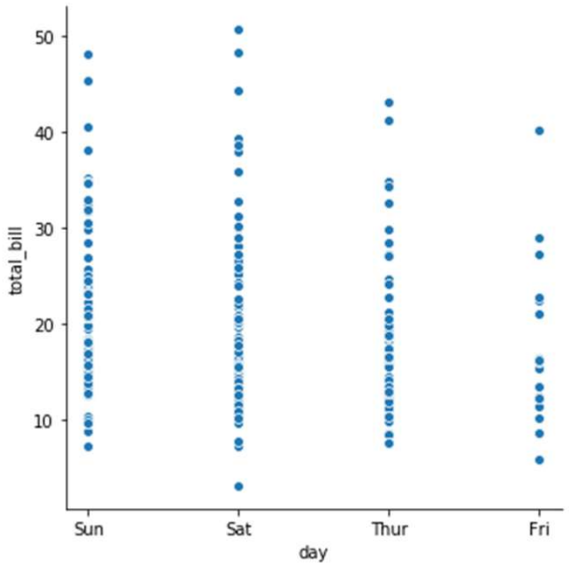
Plotting categorical variables

- *relplot()*, while is best to plot relationship between numeric variables, is able to handle categorical variables.
- *catplot()* is a function dedicated for categorical plotting.
- *sns.catplot(x="day", y="total_bill", data=tips)*



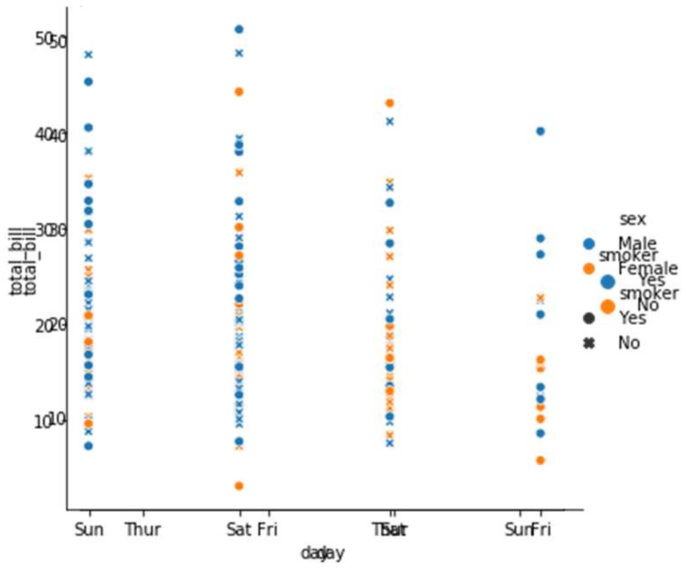
Spot three differences?





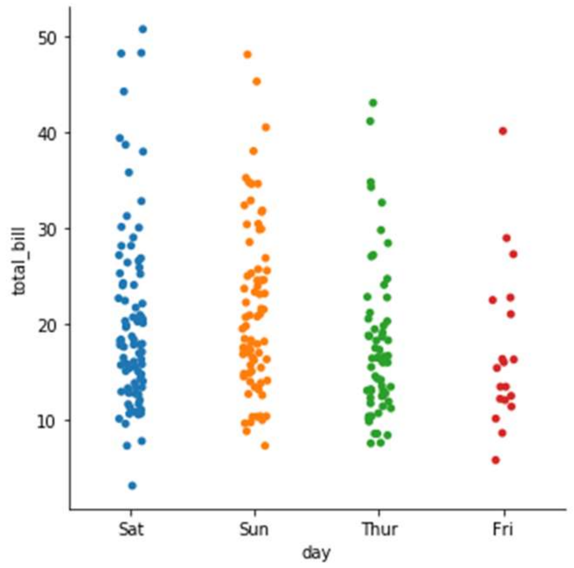
Adding dimension

- *catplot()* only support hue to add another dimension.
- If you need add more dimensions with different colours, styles or sizes, you can use *relplot()* instead.



Order the category

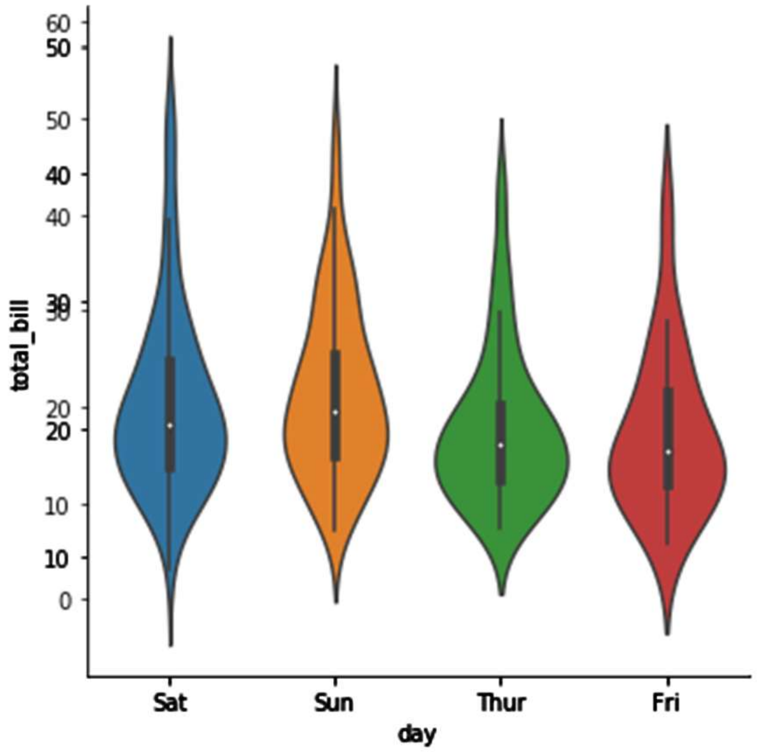
- By default, seaborn will try to infer the order of categories form your data, such as numerical order.
- However, it does not always work well. So we can manually set the order by passing a list of values to argument: order
- *sns.catplot(x="day", y="total_bill", order=['Sat', 'Sun', 'Thur', 'Fri'], data=tips)*



Visualizing the distribution

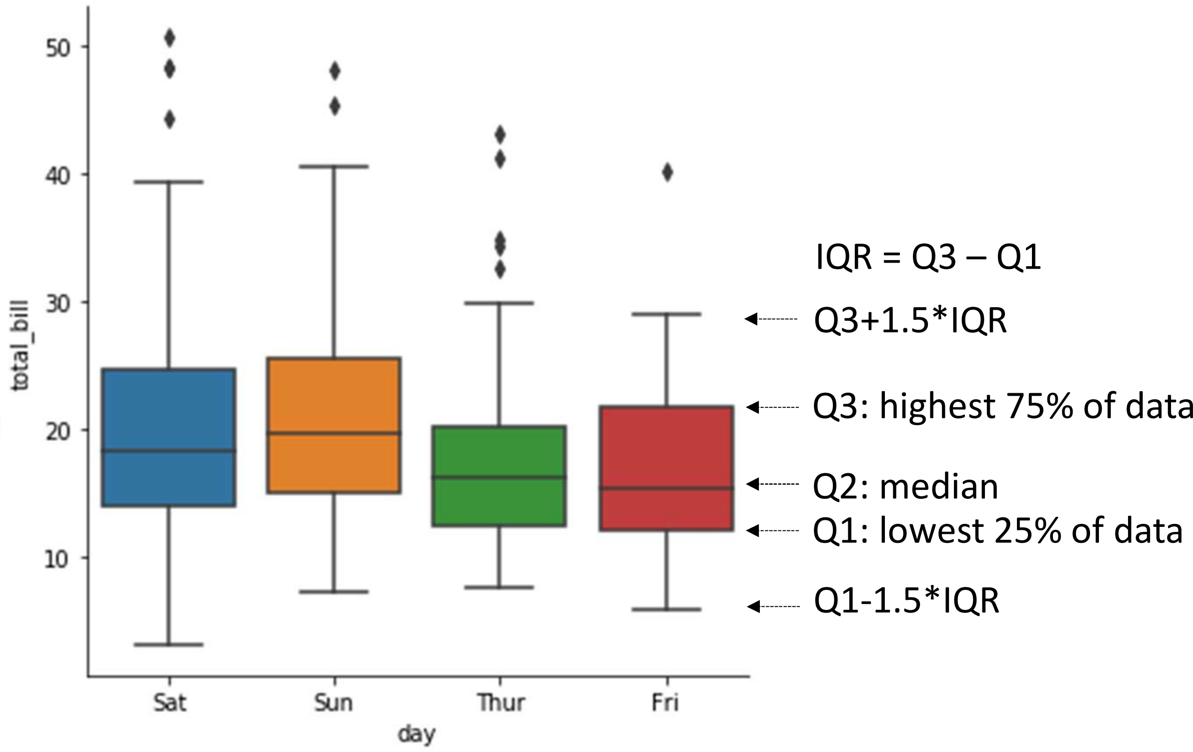
- With the default scatterplots, it gets harder to see the distribution of your data when the size gets bigger.
- We can use several other plotting approaches showing the distribution information, by passing argument kind with:

1. box and boxen
2. violin



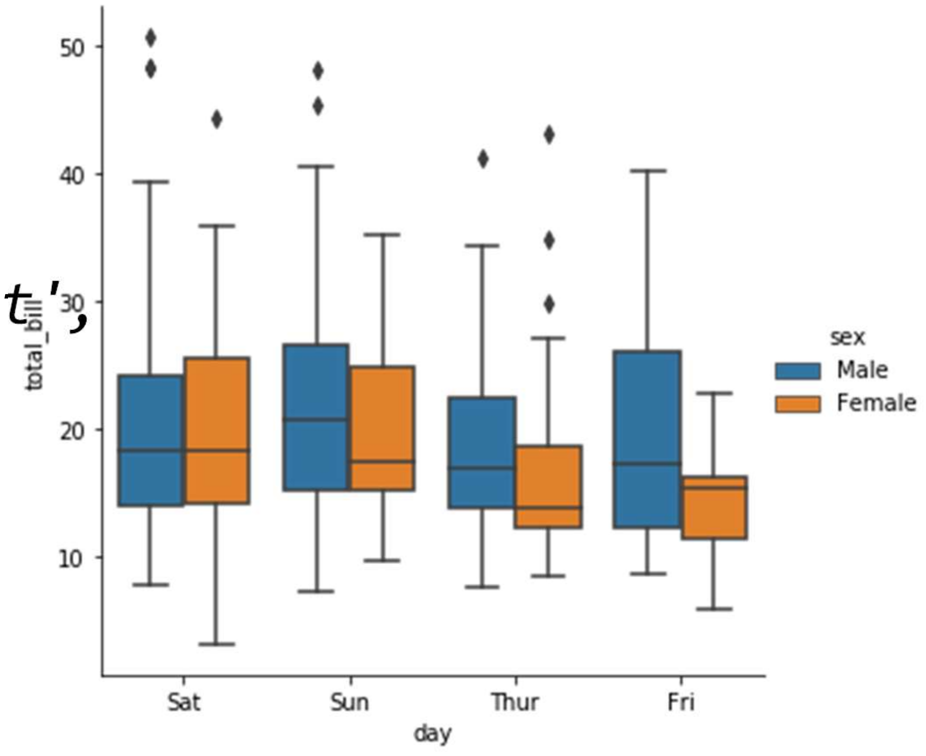
Box-plot explained

- Box plot is used for descriptive statistics. It is created based on quartiles, and also called as box-and-whisker plot.
- Box is bounded by 1st and 3rd quartiles.
- Whiskers extend to 1.5 IQRs of 1st and 3rd quartiles.



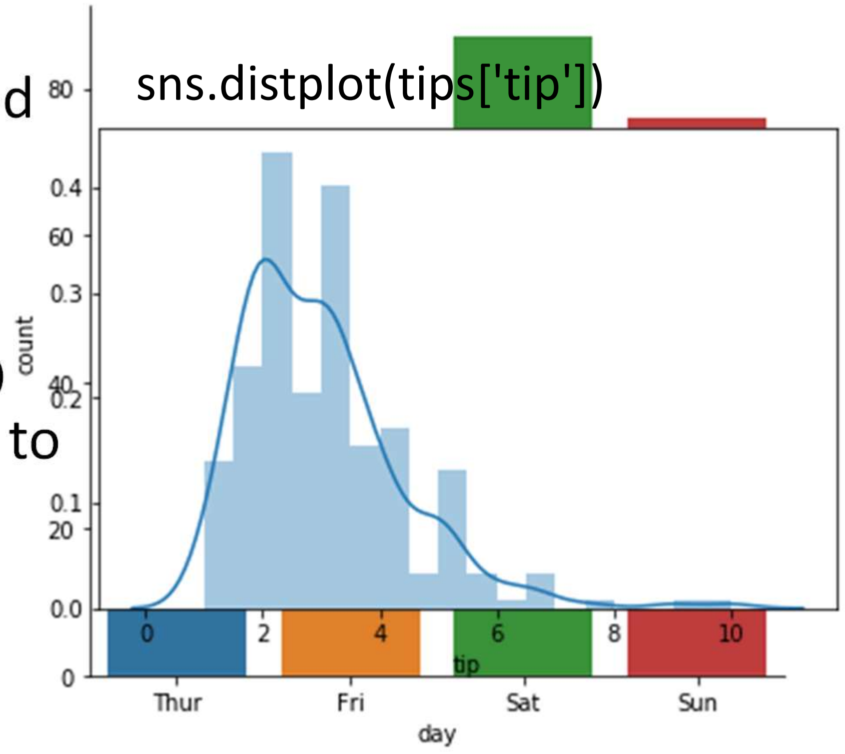
Adding dimensions to box plot

- Box plot also supports adding third dimensions with argument hue.
- *sns.catplot(x="day", y="total_bill", order=['Sat', 'Sun', 'Thur', 'Fri'], kind='box', hue= 'sex', data=tips)*



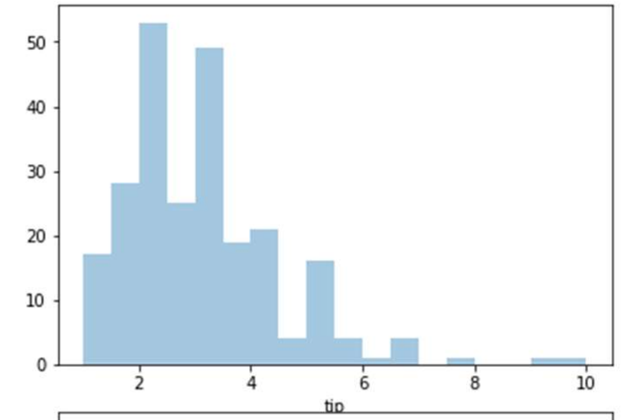
Descriptive statistics for individual variables

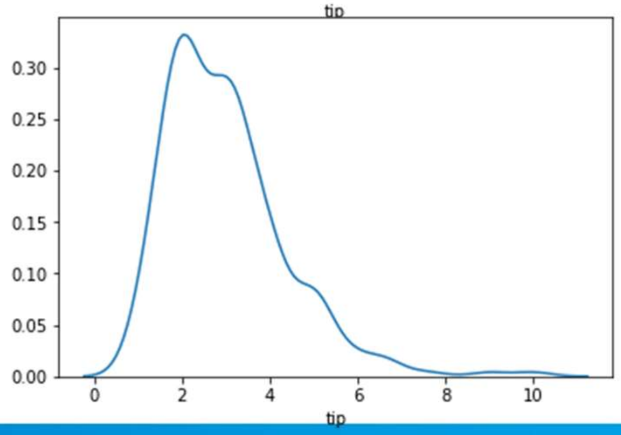
- So far, we focus on the relationships between two variables. We often need to have a better idea about individual variables.
- We can use histogram to help us.
- It can be "achieved" with *catplot()* for categorical variables with kind set to "count";
- and *distplot()* for numeric variables.



Histogram explained

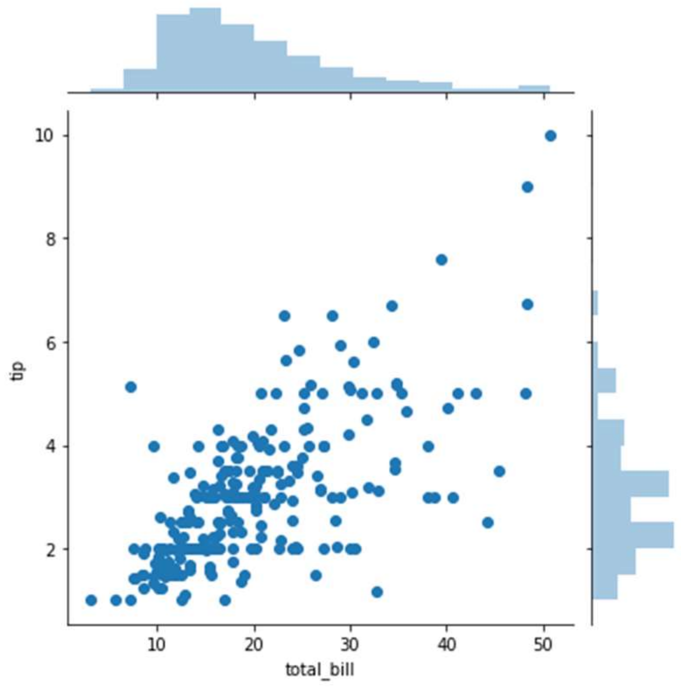
- Histogram is used to visualize univariate distributions.
- By default, it fits a kernel density estimate (KDE), which is basically a non-parametric approach to create a smoothed line based on discrete data.
- You may turn on and off histogram and KDE by setting arguments:
  - kde: false or True
  - hist: False or True





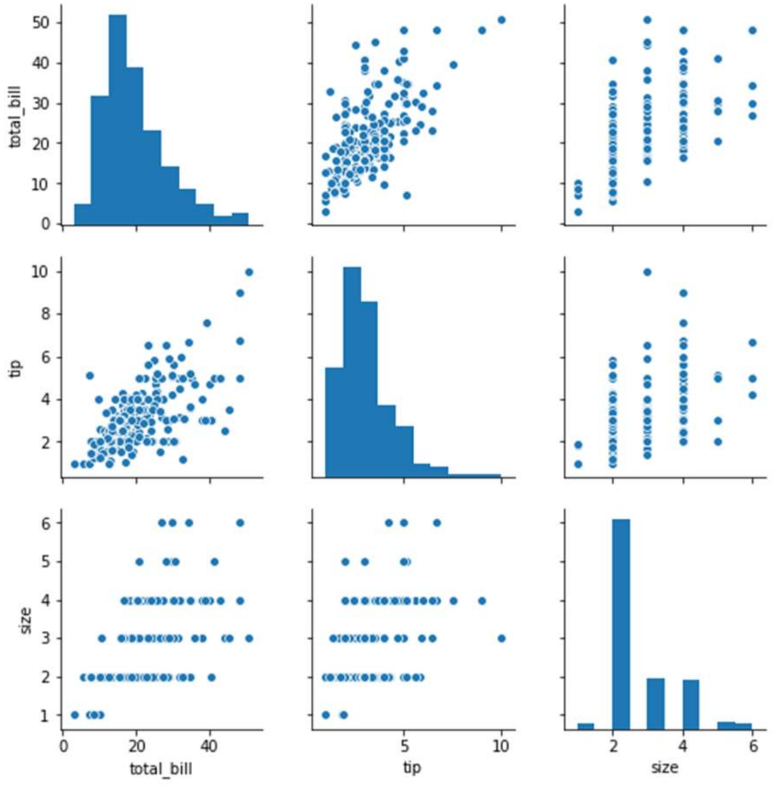
Joining histogram and scatter plots

- You can combine histogram (one variable) and scatter plots (two variables) together to give a comprehensive view.
- This can be done with *`jointplot()`*;
- *`sns.jointplot(x="total_bill", y="tip", data=tips)`*

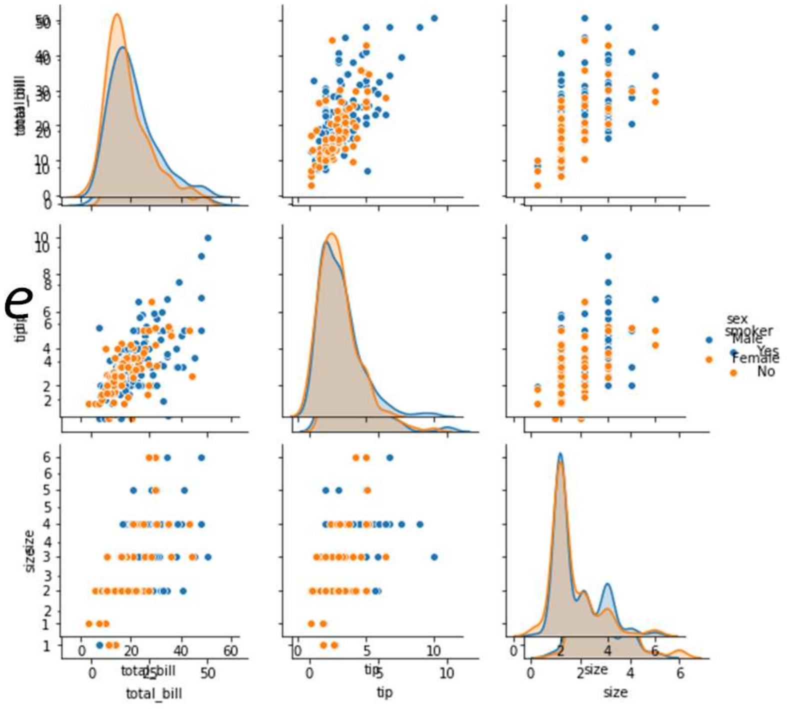


Visualizing pairwise correlations

- A very handy function *`pairplot()`* can be used to visualize the pairwise correlations for a quick overview.
- *`sns.pairplot(tips)`*
- When you have too many dimensions in you dataset, you may specify the dimensions in a list and pass to argument vars:



- *pairplot()* only plots continuous (numeric) variables. You may pass categorical variable with argument hue to group the data.
- *sns.pairplot(tips,hue='smoker')*



Exercise

- Iris flower data set is a famous data set originally used by biologist Ronald Fisher in 1936. It is about three different species of Iris flowers. It becomes a classic dataset for practicing data science techniques and is provided as built-in dataset in many Python libraries.
- Import the Iris dataset and explore factors related to the speciation.

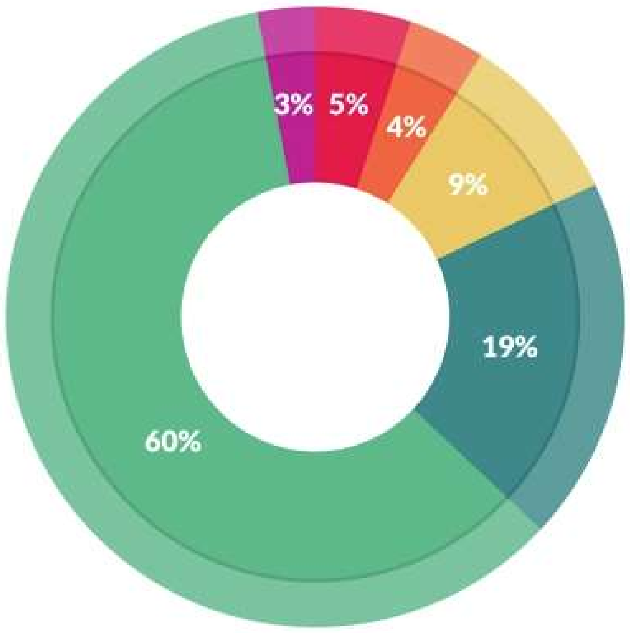
```
iris = sns.load_dataset("iris")
```

Data Wrangling

Data cleaning and processing

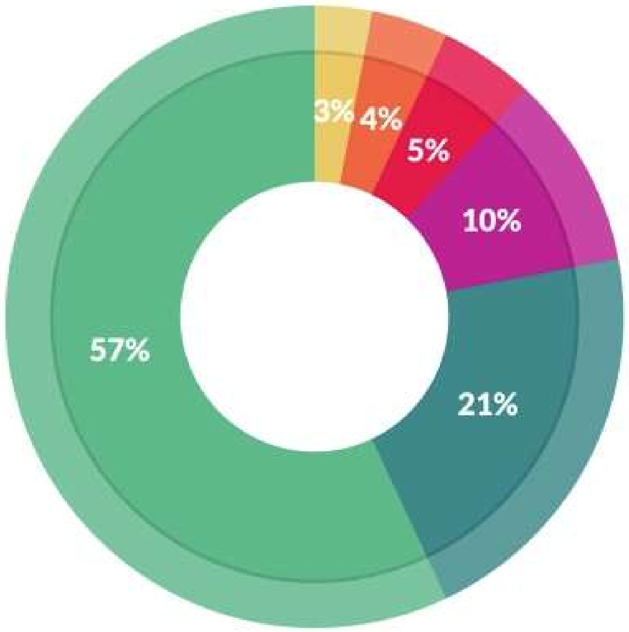
- Data wrangling is an extremely important yet time consuming step that to process and clean your data in order to transform it from raw data into analysable data.
- It may involve following operations:
    1. Importing, splitting and/or merging the datasets.
    2. Exploring your data through visualization.
    3. Generating new data through calculating or converting.
    4. Dealing with missing data.
    5. Removing unrelated data.

What data scientists spend the most time doing



What data scientists spend the most time doing

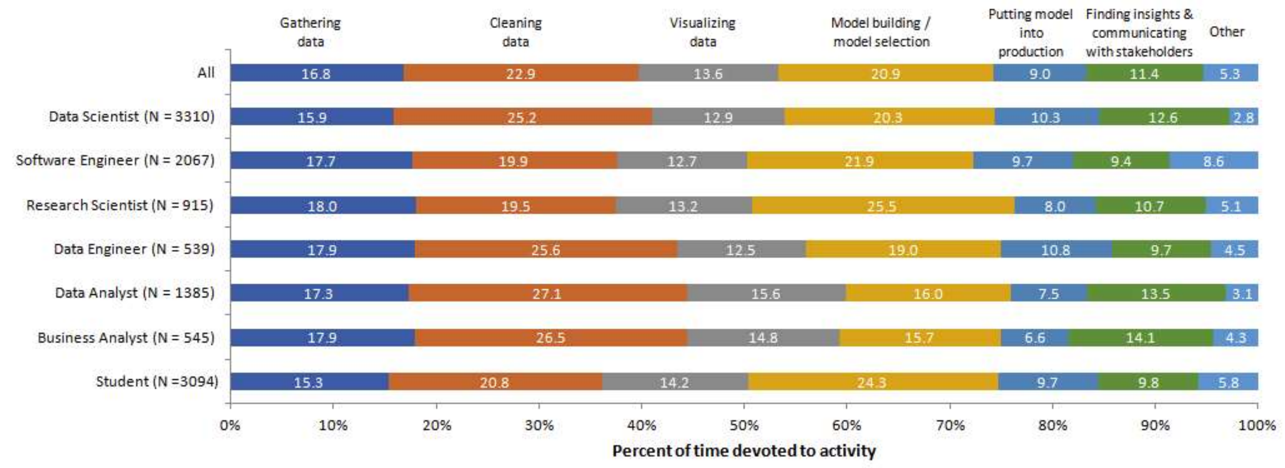
- *Building training sets: 3%*
- *Cleaning and organizing data: 60%*
- *Collecting data sets; 19%*
- *Mining data for patterns: 9%*
- *Refining algorithms: 4%*
- *Other: 5%*



What's the least enjoyable part of data science?

- *Building training sets: 10%*
- *Cleaning and organizing data: 57%*
- *Collecting data sets: 21%*
- *Mining data for patterns: 3%*
- *Refining algorithms: 4%*
- *Other: 5%*

During a typical data science project at work or school, approximately what proportion of your time is devoted to the following?

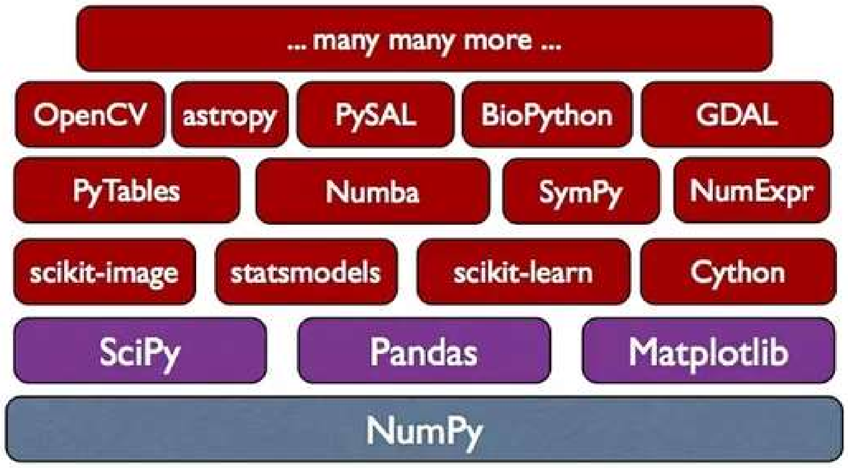


Note: Data are from the 2018 Kaggle ML and Data Science Survey. You can learn more about the study here: http://www.kaggle.com/kaggle/kaggle-survey-2018.
A total of 23859 respondents completed the survey; the percentages in the graph are based on a total of 15937 respondents who provided an answer to this question. Only selected job titles are presented.

BUSINESS BROADWAY
DATA SCIENCE | CUSTOMER ANALYTICS | MACHINE LEARNING

NumPy

- NumPy stands for numerical Python.
- Foundation library for scientific computing in Python



https://www.quora.com/What-is-the-relationship-among-NumPy-SciPy-Pandas-and-Scikit-learn-and-when-should-I-use-each-one-of-them

Advantages over built-in functions

- Ndarray: A **multidimensional array** much faster and more efficient than those provided by the basic package of Python.
- Element-wise computation: A set of functions for performing this type of **calculation with arrays** and mathematical operations between arrays.
- Reading-writing datasets: A set of tools for reading and writing data stored in the hard disk.
- Integration with other languages such as C, C++, and FORTRAN: A set of tools to integrate code developed with these programming languages.

Installation

- NumPy should be installed as part of the Anaconda installation. If not or in a new environment:

```
conda install numpy
Pip install numpy
```

- It is pre-installed in Colab.

Quick recall of Python data types

- Number
- String
- List
- Tuple
- Dictionary

NumPy array

- The NumPy library is based on one main object: ndarray (which stands for N-dimensional array).
- A ndarray is created by the *array()* function.

```
>>> a = np.array([1, 2, 3]) # list as argument
>>> a
array([1, 2, 3])
>>> type(a)
<type 'numpy.ndarray'>
```

- ndarray is homogeneous: consisted of data in the same date type.

NumPy data type dtype

bool_	Boolean (True or False) stored as a byte
int_	Default integer type (same as C long; normally either int64 or int32)
intc	Identical to C int (normally int32 or int64)
intp	Integer used for indexing (same as C ssize_t; normally either int32 or int64)
int8	Byte (-128 to 127)
int16	Integer (-32768 to 32767)
int32	Integer (-2147483648 to 2147483647)
int64	Integer (-9223372036854775808 to 9223372036854775807)
uint8	Unsigned integer (0 to 255)
uint16	Unsigned integer (0 to 65535)
uint32	Unsigned integer (0 to 4294967295)
uint64	Unsigned integer (0 to 18446744073709551615)
float_	Shorthand for float64.
float16	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
float32	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
float64	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
complex_	Shorthand for complex128.
complex64	Complex number, represented by two 32-bit floats
complex128	Complex number, represented by two 64-bit floats

Python	Numpy
int	int_
bool	bool_
float	float_
complex	cfloat
bytes	bytes_
str	unicode_

```
>>> a.dtype
dtype('int64')
```

```
>>> a.dtype
dtype('int64’)
>>>b = np.array([‘a’, 2, 3])
dtype('<U1')
```

If you mix some strings with the numbers then all of the elements will get converted into a string type and we won’t be able to perform most of the numpy operations on that array.

Creating a Numpy Array

- A numpy array can be created in three ways:

1. From Python list or tuple:

```
>>>larray = np.array([1,2,3])
>>>tarray = np.array((1,2,3))
>>>larray
array([1, 2, 3])
>>>tarray
array([1, 2, 3])
```

```
>>>marray = np.array([[1,2,3],
                      [2,3,4],
                      [3,4,5]])
>>>marray
array([[1, 2, 3],
       [2, 3, 4],
       [3, 4, 5]])
```

2. Using numpy functions:

❖*arange(start, stop, step)*, similar to *range()*.

```
>>>narray = np.arange(1,10,2)
>>>narray
array([1, 3, 5, 7, 9])
```

❖other functions to create special forms of array.

3. Reading data files.

   import data in text (csv) file into numpy array with *getfromtxt()* and *loadtxt()*.

   - *getfromtxt()* gives additional options if you have missing data.
   - *loadtxt()* is equivalent to getfromtxt() if there is no missing *data.*

```
>>>farray = np.getfromtxt(‘data.csv’)
```

Advantages of numpy array

- python list/tuple is dynamic typed, it can be mixed with different types of data and the type is not pre-defined.
- numpy array is statically typed and pre-defined. It can leverage complied language like C to improve the computing efficiency.
- numpy array supports complex mathematical calculations, such as matrix and dot multiplication, which can improve efficiency significantly.

Case: multiply numbers from two lists

Problem: multiply each element in a list with the corresponding element in another list of the same length.

Python loop statements

```
c =[]
for i in range(len(a)):
    c.append(a[i]*b[i])
```

Numpy operation

```
c = a * b
```

Shape of NumPy arrary

- The **shape** property is used to get the current shape of an array. It returns **tuple** of array dimensions

```
>>>larray.shape
```

(3,) #a has only one dimension and three elements in the first dimension.

```
>>>marray.shape
```

(3, 3) # c has two dimensions and two elements in the first dimension, 3 elements in the second dimension. In other words, it has three one-dimensional elements, each with three elements.

Exercise

```
>>>c = np.array([[[ 1,  2,  3,  4],
                  [ 5,  6,  7,  8],
                  [ 9, 10, 11, 12]],
                 [[13, 14, 15, 16],
                  [17, 18, 19, 20],
                  [21, 22, 23, 24]]])
>>>c.shape
```

(2,3,4)

Reshape your array

- Numpy array can be reshaped.

```
>>>c.reshape(3,2,4)
array([[[ 1,  2,  3,  4], [ 5,  6,  7,  8]],
       [[ 9, 10, 11, 12], [13, 14, 15, 16]],
       [[17, 18, 19, 20], [21, 22, 23, 24]]])
```

```
>>>c.reshape(4,3,2)
array([[[ 1,  2], [ 3,  4], [ 5,  6]],
       [[ 7,  8], [ 9, 10], [11, 12]],
       [[13, 14], [15, 16], [17, 18]],
       [[19, 20], [21, 22], [23, 24]]])
```

reshape does not change the original array. It only returns the result of reshaped array.

Special reshape

- You can easily transpose a 2-D array with T or *transpose()*.

```
c = np.arange(1,25).shape(4,6)
c = c.T
C = c.transpose()
```

- You can also flatten multi-dimensional array into one dimension array with *ravel()*.

```
c = c.ravel()
```

- You can directly change the shape of array by *resize()*.

```
c.resize(2,3,4)
```

Array stacking

- Multiple arrays can be stacked, vertically and horizontally.

```
a1 = np.array([[1,2],[3,4]])
a2 = np.array([[5,6],[7,8])
a3 = np.vstack((a1,a2))
array([[1, 2],
       [3, 4],
       [5, 6],
       [7, 8]])
a4 = np.hstack((a1,a2))
array([[1, 2, 5, 6],
       [3, 4, 7, 8]])
```

More attributes of numpy array

- ndim returns the total number of dimensions of the array.

```
>>>c.ndim
3
```

- size returns the total number of elements in the array.

```
>>>c.size
24
```

More functions to create numpy array

- The function *zeros()* creates an array full of zeros,
- The function *ones()* creates an array full of ones,
- The function *empty()* creates an array whose initial content is random and depends on the state of the memory.
- You can provide shape and data type, dtype, as arguments to these functions.

Example

- Create a 3 by 2 matrix with all zeros with data type int16.

```
>>>zarray = np.zeros((3,2), dtype=np.int16)
```

Create a 2 by 3 by 4 array with all ones with data type float64.

```
>>>oarray = np.ones((2,3,4), dtype=np.float64)
```

Create an empty 3 by 3 array with data type

```
>>>earray = np.empty((3,3))
```

Index your array

- Similar to Python nested list, you can index and slice your array.
- For multi-dimensional array, use comma to separate index for each dimension.

```
>>>a = np.arange(1,25).reshape(2,3,4)
array([[[ 1,  2,  3,  4],
        [ 5,  6,  7,  8],
        [ 9, 10, 11, 12]],
       [[13, 14, 15, 16],
        [17, 18, 19, 20],
        [21, 22, 23, 24]]])
>>>a[1,2,3]
24
```

- If index has been specified for all dimensions, a data value will be returned, otherwise a NumPy array will be returned.

```
>>>a[1,2]
array([21, 22, 23, 24])
>>>a[1]
array([[13, 14, 15, 16],
       [17, 18, 19, 20],
       [21, 22, 23, 24]])
```

Slice the array

- Slice 1-D array with [start:end:step]

```
>>>a = np.array([1,2,3,4,5,6,7,8])
>>>a[1:5]
array([2, 3, 4, 5])
>>>a[1:6:2]
array([2, 4, 6])
```

Slice multi-dimensional array

- Use comma separating slicing on each dimension. When fewer indices are provided than the number of dimensions, the missing indices are considered complete slices

```
>>>a[0:2,0:2,0:2]
array([[[ 1,  2], [ 5,  6]],
       [[13, 14], [17, 18]]])
>>>a[0:2,0:2]
array([[[ 1,  2,  3,  4], [ 5,  6,  7,  8]],
       [[13, 14, 15, 16], [17, 18, 19, 20]]])
>>>a[0:2]
array([[[ 1,  2,  3,  4], [ 5,  6,  7,  8], [ 9, 10, 11, 12]],
       [[13, 14, 15, 16], [17, 18, 19, 20], [21, 22, 23, 24]]])
```

Index and slice your array

```
>>>a[1,1:3,3]
array([20, 24])
```

With high dimensional array, it may get harder to index. We can use three dots ... to indicate complete slices. For example, x is a 5-D array.

- x[1,2,...] is equivalent to x[1,2,:,:,:],
- x[...,3] to x[:,:,:,:,3] and
- x[4,...,5,:] to x[4,:,:,5,:].

Index tricks

- Using array as index.

```
>>> a = np.arange(10)
>>> i = np.array([ 1,1,4,7,5 ])
>>> a[i]  # the elements of a at the positions i
array([ 1,  1,  4, 7, 5])
>>> j = np.array( [ [ 3, 4], [ 9, 7 ] ] )
>>> a[j]
array([[ 9, 16], [81, 49]])  # return an array with the same
shape as j
```

Basic operations

- Arithmetic operators on arrays apply elementwise.

```
>>>a = np.array([1,2,3])
>>>b = np.array([4,5,6])
>>>a + b # array([5, 7, 9])
>>>a – b # array([-3, -3, -3])
>>>a * b # array([ 4, 10, 18])
>>>a / b # array([0.25, 0.4 , 0.5 ])
>>>a ** b # array([  1,  32, 729], dtype=int32)
>>>a < 2 # array([ True, False, False])
```

* is not for matrix product in NumPy.

Matrix product

$$\begin{pmatrix} A1 & A2 \\ A3 & A4 \end{pmatrix} \times \begin{pmatrix} B1 & B2 \\ B3 & B4 \end{pmatrix} = \begin{pmatrix} A1*B1 + A2*B3 & A1*B2 + A2*B4 \\ A3*B1 + A4*B3 & A3*B2 + A4*B4 \end{pmatrix}$$

Matrix product can be performed with @ or dot().

```
>>>a = np.array([[1,2],[3,4]])
>>>b = np.array ([[5,6],[7,8]])
>>>a @ b  # array([[19, 22], [43, 50]])
>>>a.dot(b) # array([[19, 22], [43, 50]])
```

Matrix product

- Make sure you have same number of elements in matching row and column.

```
>>>a = np.array([[1,2,3],[3,4,5]])
>>>b = np.array ([[5,6],[7,8]])
>>>a @ b # ValueError: shapes (2,3) and (2,2) not aligned: 3 (dim 1) != 2 (dim 0)
```

- The number of columns (2nd dimension) in the first matrix should equal to the number of rows (1st dimension) in the second matrix

```
>>>c = np.array ([[5,6],[7,8],[9,10]]) #shape(3,2)
>>>a @ b # array([[46,52], [88,100]])
```

Computation comparison

```
rng = np.random.RandomState(42)
x = rng.rand(100000) #(100000 ,1)
y = rng.rand(100000)
%timeit x + y
```

2.43 ms ± 52.3 µs per loop

```
%timeit results = [xi + yi for
xi, yi in zip(x, y)]
result = []
for xi, yi in zip(x,y):
    results.append(xi+yi)
```

181 ms ± 2.43 ms per loop

For more information

- https://docs.scipy.org/doc/numpy/user/quickstart.html

Pandas

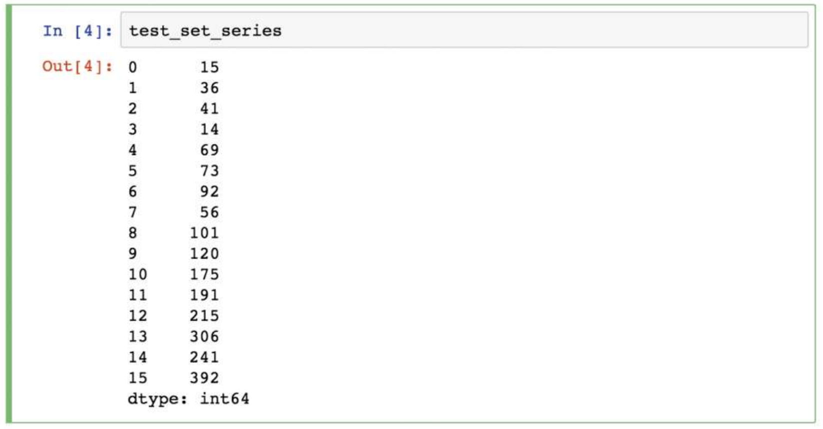
- Pandas is a nalysis. The name Pand y ("Panel Data" from Wiki),
- It depends d Matplotlib.
- To some ex y library mostly dealing wit ation functionalit
- It is probab rary for Python.

Data structure

- There are two types of data structures in pandas: Series and DataFrames.
- **Series**: one dimensional data structure (“a one dimensional ndarray”), and for every value it holds a unique **index**.
- **DataFrame:** a two dimensional data structure – basically a table with rows and columns. The columns have names and the rows have **indexes**.

1. A series can be created using pandas function Series with python **list** or numpy **1-D array** as the argument. By default, each item will receive an numeric index label starting from 0.

```
>>>s1 = pd.Series([1,2,3])
>>>s2 = pd.Series(np.array([1,2,3,4,5]))
```



```
In [4]: test_set_series
Out[4]: 0      15
        1      36
        2      41
        3      14
        4      69
        5      73
        6      92
        7      56
        8     101
        9     120
        10    175
        11    191
        12    215
        13    306
        14    241
        15    392
        dtype: int64
```

Manually creating a Series data

1. An explicit index can also be specified when creating the series by providing the index with a **list** as the second argument. This is often called label.

```
>>>s3 = pd.Series([1,2,3,'a','b','c'],
                  index=['A','B','C','D','E','F'])
```

2. When a dictionary is provided as the argument, the key will be used as the index.

```
>>> s4 = pd.Series({'A':1,'B':2,'C':3})
```

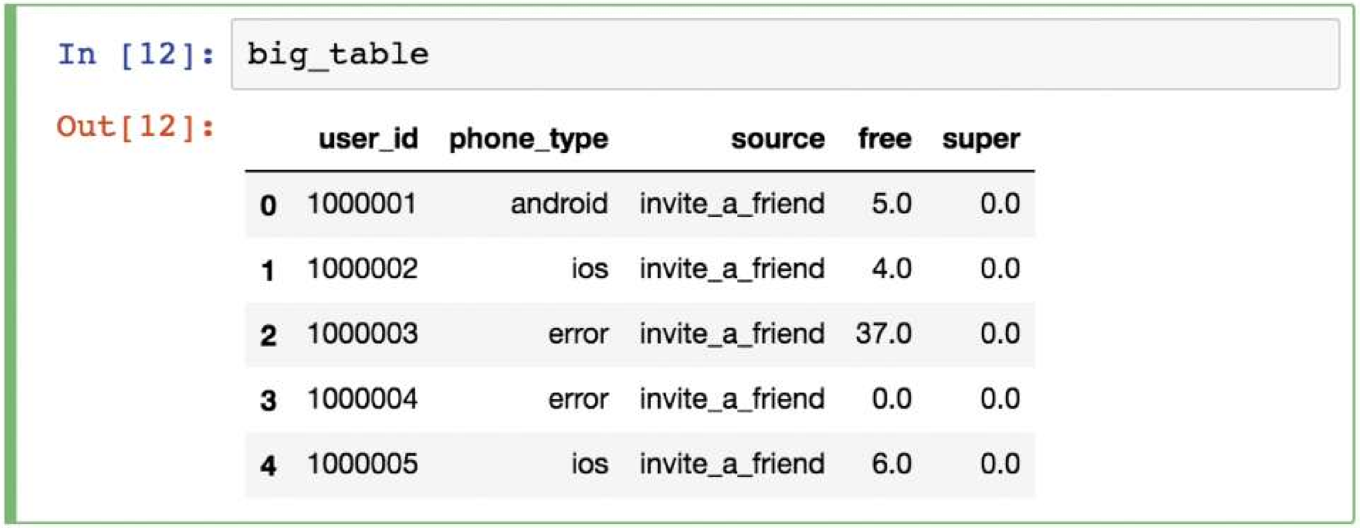
3. Each index label needs to be unique?

Indexing and slicing Series Data

1. Data in the series can be accessed similar to that in a Python list when having the default numeric index.
   `s2[2]`
   `s2[:2]` # return a series data.
2. Data in the series can be accessed similar to that in a Python dictionary when having specified index label.
   `s3['A']`
3. You can retrieve multiple data by providing a **list** of "keys"/labels.
   `s3[['A','B','C']]`

Dataframe

- A dataframe has labeled axes (rows and columns) and can be created by pandas function: *DataFrame()*



```
In [12]: big_table
```

Out[12]:

	user_id	phone_type	source	free	super
0	1000001	android	invite_a_friend	5.0	0.0
1	1000002	ios	invite_a_friend	4.0	0.0
2	1000003	error	invite_a_friend	37.0	0.0
3	1000004	error	invite_a_friend	0.0	0.0
4	1000005	ios	invite_a_friend	6.0	0.0

Create DataFrame from a dictionary

- You can create a DataFrame from **dictionary** of narrays/lists/series. Keys will be used as the column labels by default. Values become the columns corresponding to the key.

```
>>>d1 = pd.DataFrame({'A':[1,2,3],'B':[2,3,4]})
```

- You may also specify index label for the rows.

```
>>>d2 = pd.DataFrame({'A':[1,2,3],'B':[2,3,4]},index=['X','Y','Z'])
>>>d3 = pd.DataFrame({'A':np.array([1,2,3]),
                     'B':[2,3,4]}, index=['X','Y','Z'])
>>> d4 = pd.DataFrame({'A':[1,2,3],'B':s1})#s1 is a ndarray
```

Create DataFrame from a dictionary

- Note: Items in the dictionary must have the same length unless they are all series.

```
>>>d5 = pd.DataFrame({'A' : [1,2,3], 'B' :[2,3,4,5]}) #error
>>>d6 = pd.DataFrame({'A' : s1, 'B' :[1,2]}) #error
```

- When series have different length, Python will try to match their index to create the dataframe and NaN (Not a Number) is appended in missing areas.

```
>>>d7 = pd.DataFrame({'A' : s1, 'B' :s2}) #using default numeric index
>>>d8 = pd.DataFrame({'A' : pd.Series([1, 2, 3], index=['a', 'b', 'c']), 'B' : pd.Series([1, 2, 3, 4], index=['b', 'c', 'd', 'e'])})#using specified index
```

Create DataFrame from a list

- A DataFrame can be created using a single list or a list of lists.

```
>>> d9 = pd.DataFrame([1,2,3,'a','b','c']) #compare with s1.
>>> d10 = pd.DataFrame([[1,2,3],[2,3,4],[3,4,5]])
```

- Numeric labels will be created for row and column by default. You can also specify the labels for columns and index (row).

```
>>> d11 = pd.DataFrame([[1,2,3],[2,3,4],[3,4,5]],columns=['A','B','C'])
>>> d12=pd.DataFrame([[1,2,3],[2,3,4],[3,4,5]], columns=['A','B','C'],
                      index=['X','Y','Z'])
```

Create DataFrame from a list

- You can create a DataFrame from a list of dictionaries. Keys will be used as the column labels by default.

```
>>>d13 = pd.DataFrame([{'a': 1, 'b': 2},{'a': 5, 'b': 10}])
```

```
>>>d14 = pd.DataFrame([{'a': 1, 'b': 2},{'a': 5, 'b': 
10}],index=['A','B'])
```

- Each item in the list is like a row in a table. Items in the list can have different length.

Exercise

- Create a DataFrame with shape(1,3)

List of elements with different lengths

- When no specific column label is provided, Python will match the default labels (number index or keys) to create the dataframe and NaN is appended in missing areas.

```
>>> d15 = pd.DataFrame([[1,2],[2,3],[3,4,5]])
```

```
>>> d16= pd.DataFrame([{'a': 1, 'b': 2},{'b': 5, 'c':
10,'d':15}])
```

- When column labels are specified, Python will create DataFrame based on the column labels and try to match keys with the labels. Values with non-match keys will be ignored.

```
>>>d17 = pd.DataFrame([{'a': 1, 'b': 2},{'b': 5,
'c':10,'d':15}], columns=['b','d','e'])
```

Create DataFrame from files

- Pandas can read data directly from a wide range of file formats, such as csv, Excel, JSON, SQL database, Stata, SAS, etc. We will focus on csv files in this class.
- Use *read_csv()* function. Filename is the only required argument.

```
df_titan = pd.read_csv('titanic_train.csv')
```

  - ❖ Many optional arguments can be passed when importing data.
  - ❖ https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html

Key parameters of read_csv()

- **delimiter**: comma by default, set when necessary.
- **header**: row to be used as the column headers, and the start of data. 0 by default. Set to None if no header.

```
df_titan = pd.read_csv('titanic_train.csv',header=None)
```

- **names**: a list of column names to be used instead header.

```
df_titan = pd.read_csv('titanic_train.csv',header=None,names=[1,2])
```

- **index_col**: specify a column to be used for row index.

```
df_titan = pd.read_csv('titanic_train.csv',index_col=0)
```

Row index can also be specified with column name

```
df_titan = pd.read_csv('titanic_train.csv',index_col='PassengerId')
```

Key parameters of read_csv()

- **usecols**: import selected columns by passing a list of column **index or names**.

```
columns = [2,3,4] #columns = ['Pclass', 'Name', 'Sex']
df_titan = pd.read_csv('titanic_train.csv',usecols=columns)
```

- **skiprows**: specify the number of first n rows to skip.
- **nrows**: specify the total number of rows to read. Useful when reading large files.

```
df_titan = pd.read_csv('titanic_train.csv',skiprows=3, nrows=10)
```

Key parameters of read_csv()

- na_values: list of **strings** to be treated as NaN. Most common ones can be detected automatically, such as #N/A, n/a, null, etc.

```
missing = ['not available', 'missing']
df_titan = pd.read_csv('titanic_train.csv',na_values=missing)
```

Column selection and deletion

- Column selection using the column label:

`>>>print(df_titan['Age'])` #column label as the key, return a series

`>>>print(df_titan[['Age']])` #return a dataframe

- Add new column with label, similar as adding new item to a dictionary:

`>>> df_titan['f'] = pd.Series([10,10])` #series/list/narray

- New column can be added by calculating existing columns:

`>>> df_titan['g'] = df_titan['b'] + df_titan['f']` #NaN if one cell is Nan.

- del to delete a column:

`>>>del df_titan['g']`

Row Selection

- Row selection by passing row **labels** to loc[] method. The row will be returned as a series or a dataframe:

`>>> df_titan.loc[1]` #column labels will be used as row index.

- Multiple rows can be selected:

`>>> df_titan.loc[['a','c','e']]` #a **list** of indexes/labels, returns a dataframe

`>>> df_titan.loc['a':'c']` #slice, both start and end included.

- Column labels can be provided to filter the results:

`>>> df_titan.loc[['a','c','e'],'A']`

- Select rows with Boolean list indicating whether to be selected:

`>>> df_titan.loc[[True,False,False,True,False]]`#same length as row.

- Select rows with Boolean expression:

```
>>> df_titan.loc[df8['B'] > 2]
```

`>>> df_titan.loc[df8['B'] > 2,'A']`# only display column A

Row Selection

- Select rows by passing **integer position** (index) to method iloc[].

```
>>> df_titan.iloc[1]
```
#implicit numeric index starting from 0.

- Note: `df8.loc[1]` differs from `df8.iloc[1]`

```
>>> df_titan.iloc[0:2]
```

Add rows

- Add new rows with append( ) method at the end of a dataframe.

```
>>> df_titan.append([99,99]) #existing dataframe not updated.
>>> df_titan.append(pd.DataFrame([99,99]))
```

- Python will align matching columns.

```
>>> df_titan.append(pd.DataFrame([99,99],columns=['A','B']))
>>> df_titan.append(pd.DataFrame([[99,99]],columns=['A','B']))
```

- You can reset row labels by setting ignore_index=True (default is False).
- df_titan.append(pd.DataFrame([[99,99]],columns=['A','B']), ignore_index=True)

Row deletion

- You can remove rows by drop() method with a list of labels.

`df_titan.drop([4])` #same as df8.drop(4)

`df_titan.drop([0:2])` #error [0:2] is not a list. Alternatively, df8 = df8[2:].

`df_titan.drop([0,1,2])`

`df_titan.drop(range(3))` #existing dataframe not updated

`df_titan = df8.drop([4])`

`df_titan.drop([5],inplace=True)` # existing dataframe updated.

Important dataframe attributes

- .index returns a "list" of row **indexes** (numeric positions rather than labels).
- `df_titan.index`
- `df_titan.loc[df_titan['B'] > 2].index`
- Very handy to remove rows based on condition.
- `df_titan.drop(df_titan.loc[df_titan['B'] > 2].index)`

Scalar operation

- Basic arithmetic and Boolean operations with scalar data are element-wise.
- `df_titan *2` #broadcasting
- `df_titan.add(2)`
- `df_titan >2`
- `df_titan['B']*2`

Python Operator	Pandas Method(s)
+	add()
-	sub(), subtract()
*	mul(), multiply()
/	truediv(), div(), divide()
//	floordiv()
%	mod()
**	pow()

More operations

- Arithmetic and Boolean operations with another list or Series will be performed based on matching labels (columns).
- `d10 = pd.DataFrame([[1,2,3],[3,4,5],[5,6,7]])`
- `d10 - [1,2,3]` #default to compare by column.
- `d10 > [3,3,3]`
- `d10 - [1,2]`#error, different length
- `d10 - pd.Series([1,2])` # NaN for no-match column.
- `d8 - pd.Series([1,2])`

- For operations by row, you need to specify axis to be "index".
- `d10.sub(pd.Series([1,2,3]),axis='index')`
- `d10.sub(pd.Series([1,2,3]),axis='index')` # NaN for no-match rows.
- `d10.mul(pd.Series([1,2,3],index=[1,2,3]),axis='index')` # NaN for no-match rows.

Data processing

Explorer Dataset

- Dimensions of the dataset.

`df_titan.shape` # quick view of the dimensions.

`df_titan.size` # total number of measurements in your data.

`df_titan.info()` # count and datatype for each column

- Preview your data with head(n) and tail(n), n is 5 by default.

`df8.head(3)` # first 3 rows; `tail(3)` for last 3 rows.

Descriptive statistics

- The **`describe()`** method computes a summary (NaN will be excluded) of statistics pertaining to the DataFrame columns. Summary is also a DataFrame.

```
df_titan.describe()
```

- You may pass proper value to argument include:
  - **number** – default, only summarizes Numeric columns.
  - **object** – only summarizes String columns.
  - **all** – Summarizes all columns together (Should not pass it as a list value).

```
df_titan.describe(include='all')
```

Descriptive statistics

- You can get descriptive measurements using methods like: sum(),min(),max(),mean(),count(),median(),std(), and more. All measurements are returned as Series.

```
df_titan.mean()
```

- By default, these statistics are for each column. You can get row-based statistics by specifying axis=1.

```
df_titan.mean(axis=1)# default axis = 0
```

- To get the unique values and the corresponding counts for a column:

```
df_titan['Pclass'].value_counts() #only works for individual column.
```

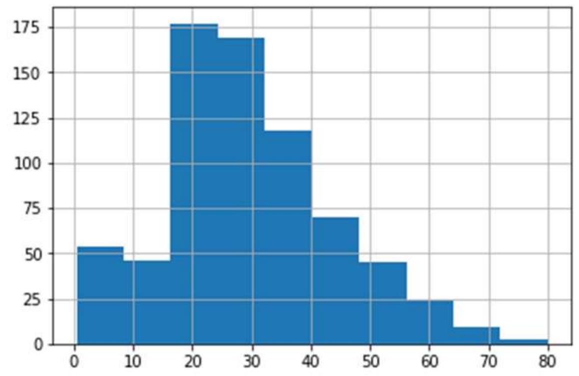
Exploring your data with simple visualization

- Pandas offer some functions for basic plotting.
- Histogram

```
df_titan.hist()
```

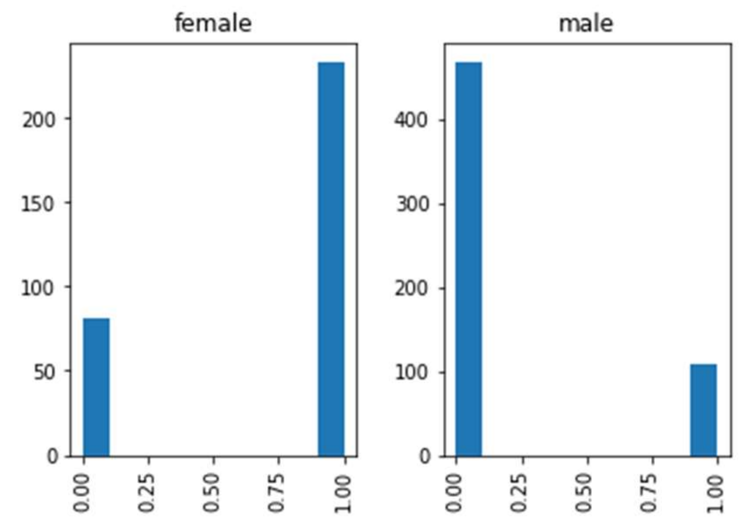
```
df_titan.hist('Age')
```

- Line
- Pie
- https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.plot.html



- Instead of showing a single distribution, you can split based on another column.

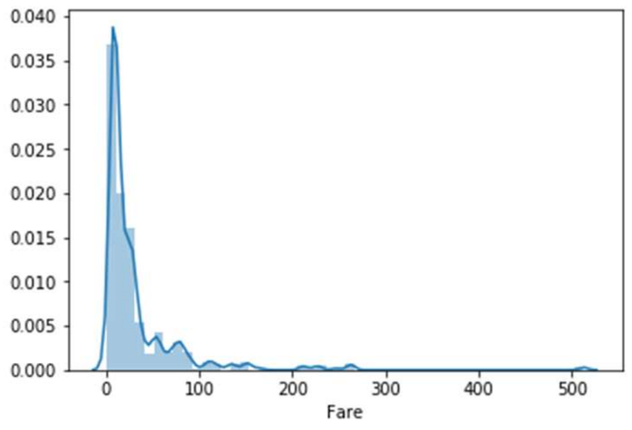
```
df_titan.hist("Survived", by="Sex")
```

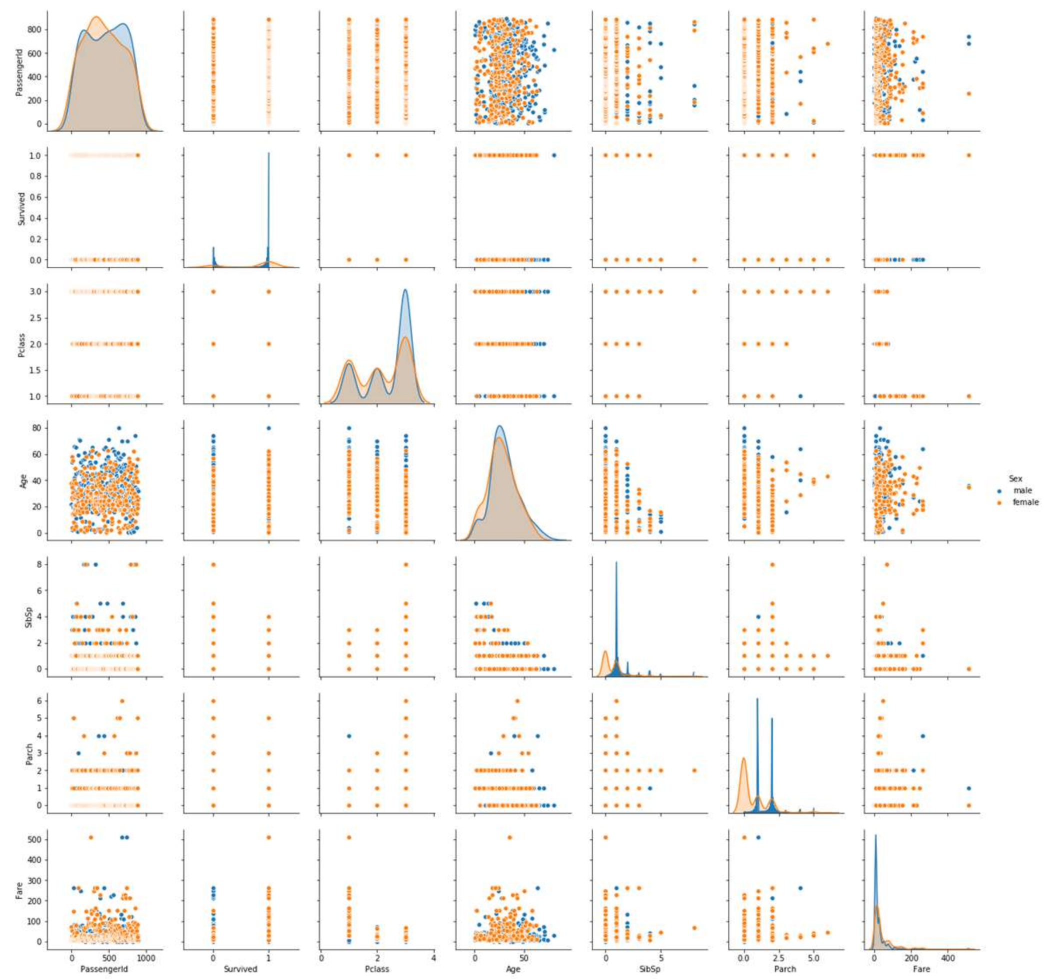


Visualization with Seaborn

- Pandas dataframe can be used directly by Seaborn as dataset, if there is no NaN value.

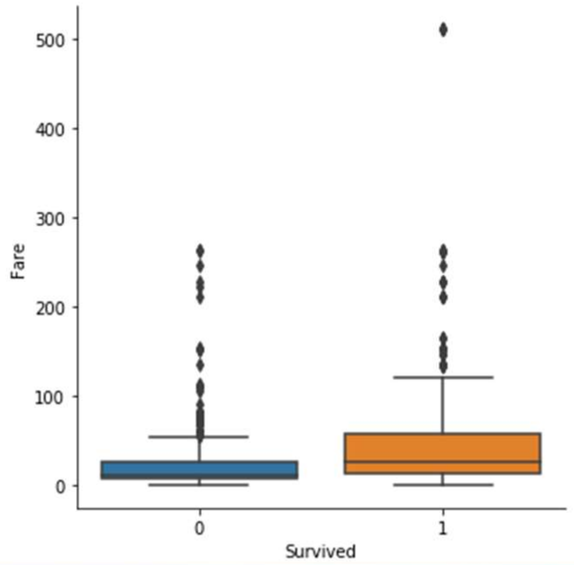
```
import seaborn as sns
sns.distplot(df_titan['Age'])
sns.distplot(df_titan['Fare'])
sns.distplot(df_titan['Fare'],
hue='Sex')
```

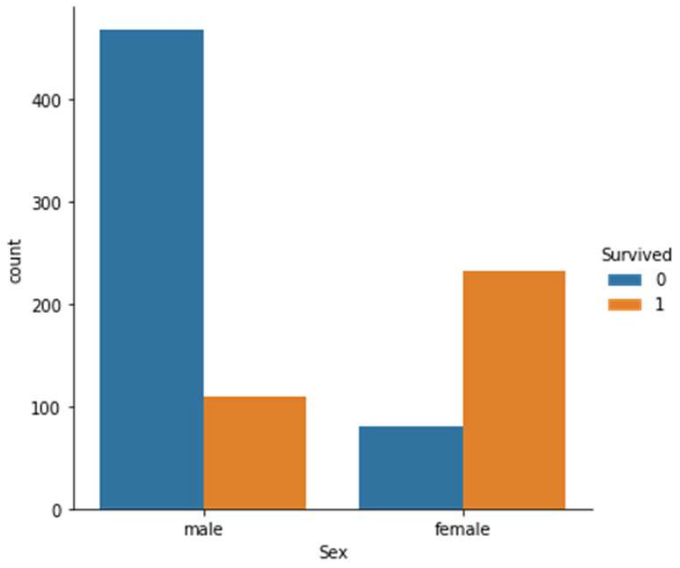



PassengerId
Survived
Pclass
Age
SibSp
Parch
Fare
Sex
male
female

Exercise

- Can you do a quick check if people paying higher fare was more likely to survive? Was female passenger more likely to survive?





Detect missing data

In most cases, you will have missing data issue in your dataset.

Check if there is any missing data

- `df_titan.describe(include='all')` #missing data in Age and Cabin.
- `df_titan.isnull().sum()` # another trick to find out missing data.

Reasons for missing data

1. Missing Completely at Random(MCAR): The missingness has nothing to do with any other factors.
   - ❖ Age is missing due to neglect.
2. Missing at Random(MCR): The missingness is caused by other items been measured but has nothing to do with the the its own measurement.
   - ❖ Age is missing because female don't like to report their ages.
3. Missing Not at Random (MNAR): the missingness is caused by its own measurement.
   - ❖ Age is missing because elder people don't like to report their ages.

Strategies to deal with missing data

- Delete data with missing value, with **CAUTION**.
- Removing rows with missing data by using dropna().

`df_titan.dropna()` # old dataframe will not be replaced automatically.

- Removing columns with missing data by specifying axis=1

`df_titan.dropna(axis=1)` # age and cabin dropped.

- Set threshhold to remove.

`df_titan.dropna(thresh=11)` # keep rows with at least 11 non-missing data. i.e. rows missing both age and cabin get dropped.

Strategies to deal with missing data

- Impute missing data. Very tricky and a lot of consideration. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3668100/

1. Impute with a constant with `fillna()`.

`df_titan.fillna(1)` #fill all NaN with 1.

`df_titan.fillna({'Age':20,'Cabin':'B96'})` # different value for two columns.

2. Impute with mean/median/mode.

`df_titan.fillna({'Age':df_titan['Age'].mean(),'Cabin':'B96'})` #replace with mean.

3. Impute with statistic estimation (will discuss later).

Exercise

- Replace all missing ages with mean and drop the Cabin column.

```
df_titan = df_titan.fillna({'Age':df_titan['Age'].mean()}).dropna(axis=1)
```

Data Cleaning

- Remove irrelevant columns (variables) using drop(columns = [column names]).

df_titan.drop(columns = [Ticket']) # remove Ticket column from data.

- Sometimes, you may also want to remove outliers from you data.
  - ➢For example, remove values outside 2 std of mean for normally distributed variables.

- top = df_titan['Age'].mean()+2*df_titan['Age'].std()
- bot = df_titan['Age'].mean()-2*df_titan['Age'].std()
- df_titan[df_titan['Age']>top].info()
- df_titan.drop[df_titan[df_titan['Age']>top].index]

Split multi-value columns

- Sometimes, you may want to split one column into multiple ones.
  - ➢Datetime -> Year, Month, Date, Hour, Mins, Secs.
  - ➢Name -> Title, First Name, Last Name.
  - ➢Email -> Username, Domain.
- Regular expression can often do the trick.
  - Series.str can be used to access the values of the series as strings and apply several methods to it.
  - extract() is a Series.str method to capture groups in the regex pat as columns in a DataFrame.

Example

- Extract title and last name from column Name as new columns.

```
df_titan.Name.str.extract('\s(\w+)\.')
df_titan['Title'] = df_titan.Name.str.extract('\s(\w+)\.')
df_titan.Name.str.extract('^(\w+),')# NaN
df_titan.Name.str.extract('^([\w\s]+),') #NaN
df_titan.Name.str.extract('^([\w\s\']+),')
df_titan['LastName'] = df_titan.Name.str.extract('^(\D+),')
```

Derive and transform columns

- Sometimes, you may want to create new columns derived from existing ones or transform existing ones.

➢Normalization and standardization.

➢Log transformation.

➢Continuous to categorial.

➢Dummy variables.

One-hot encoding

- get_dummies(column) is a Pandas function used to create dummy variables columns based on unique values in current column. This process is also called one-hot encoding. This function returns a DataFrame.
- pd.get_dummies(df_titan['Sex'])
- df_titan[['Female','Male']] = pd.get_dummies(df_titan['Sex'])

- Normalization
- df_titan['FareNor'] =(df_titan['Fare']-df_titan['Fare'].mean())/df_titan['Fare'].std()
- Log transformation
- df_titan['FareLog'] = np.log(df_titan['Fare']) # zero division

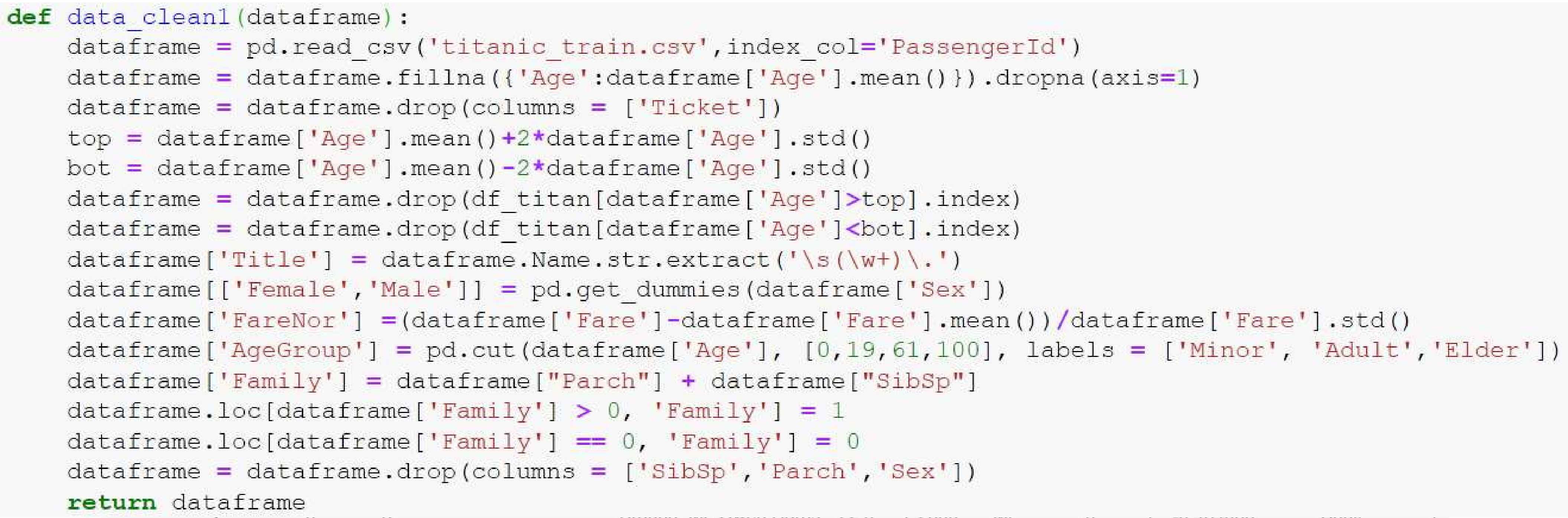
Continuous to categorical

- We can group a range of continuous values into a category by using cut(**column**,**catergory**,labels) function. For category, you can pass three types of values:
  1. An integer: defines the number of equal-width categories.
  2. sequence of scalars : Defines the category boundaries allowing for non-uniform width.
  3. IntervalIndex : Defines the exact categories to be used.
- pd.cut(df_titan['Age'],3) # 3 groups.
- pd.cut(df_titan['Age'],[0,19,61,100]) # 3 groups with boundaries.
- df_titan['AgeGroup'] = pd.cut(df_titan['Age'], [0,19,61,100], labels = ['Minor', 'Adult','Elder'])

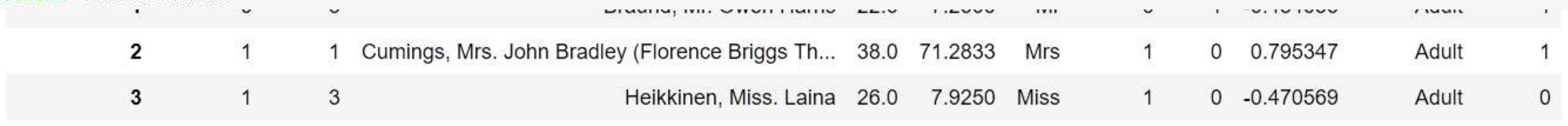
Derived columns

- Instead of differentiating parent/children and sibling/spouse, we are only interested in family relationships.
- df_titan['Family'] = df_titan["Parch"] + df_titan["SibSp"]
- df_titan.loc[df_titan['Family'] > 0, 'Family'] = 1
- df_titan.loc[df_titan['Family'] == 0, 'Family'] = 0

Final results



```python
def data_clean1(dataframe):
    dataframe = pd.read_csv('titanic_train.csv',index_col='PassengerId')
    dataframe = dataframe.fillna({'Age':dataframe['Age'].mean()}).dropna(axis=1)
    dataframe = dataframe.drop(columns = ['Ticket'])
    top = dataframe['Age'].mean()+2*dataframe['Age'].std()
    bot = dataframe['Age'].mean()-2*dataframe['Age'].std()
    dataframe = dataframe.drop(df_titan[dataframe['Age']>top].index)
    dataframe = dataframe.drop(df_titan[dataframe['Age']<bot].index)
    dataframe['Title'] = dataframe.Name.str.extract('\s(\w+)\.')
    dataframe[['Female','Male']] = pd.get_dummies(dataframe['Sex'])
    dataframe['FareNor'] =(dataframe['Fare']-dataframe['Fare'].mean())/dataframe['Fare'].std()
    dataframe['AgeGroup'] = pd.cut(dataframe['Age'], [0,19,61,100], labels = ['Minor', 'Adult','Elder'])
    dataframe['Family'] = dataframe["Parch"] + dataframe["SibSp"]
    dataframe.loc[dataframe['Family'] > 0, 'Family'] = 1
    dataframe.loc[dataframe['Family'] == 0, 'Family'] = 0
    dataframe = dataframe.drop(columns = ['SibSp','Parch','Sex'])
    return dataframe
```



1	0	3	Braund, Mr. Owen Harris	22.0	7.2500	Mr	0	1	[illegible]	Adult	1
2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	38.0	71.2833	Mrs	1	0	0.795347	Adult	1
3	1	3	Heikkinen, Miss. Laina	26.0	7.9250	Miss	1	0	-0.470569	Adult	0

Regular Expression

- A Regular Expression, or RegEx, is a sequence of characters that specifies a pattern to be searched.
- RegEx is like a mini "programming language" that embedded in Python, as well as other languages (more or less).
- For example, `\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}\b` is a regular expression to match valid email addresses.
- Very useful for data collection, extraction and cleaning.
- But requires practice and "trial and error".

Sets []

[] is used to match a single character specified in the brackets..

- [abcd]: Matches either a, b, c or d. It does not match "abcd".
- [a-d]: Matches any one alphabet from a to d.
- [a-] and [-a] | Matches a or -, because - is not being used to indicate a series of characters.
- [a-z0-9] | Matches any character from a to z and also from 0 to 9.

[^] is used to match a single character not specified in the brackets

- [^abc] matches any character that is not a, b and c.

RegEx in Python

- Python has build-in module re for regular expression operation.

re.findall(A, B) will matches all instances of a string or an expression A in a string B and returns them in a list.

```
print(re.findall("o","I love python")) # ['o','o']
```

- Add r before string A to indicate a regular expression.

```
print(re.findall(r"[a-p]","I love python")) # ['l', 'o', 'e', 'p', 'h', 'o', 'n']
print(re.findall(r"[lop]","I love python")) # ['l', 'o', 'p', 'o']
print(re.findall(r"[o-t][v-z]","I love python")) # ['ov', 'py']
```

Special Sequences

\w Matches alphanumeric characters, which means a-z, A-Z, and 0-9. It also matches the ideogram and underscore, _.

print(re.findall(r"\w","**I love**爱 **python3**")) # ['I', 'l', 'o', 'v', 'e', '爱', 'p', 'y', 't', 'h', 'o', 'n', '3']

\W matches any character not included in \w.

\d Matches digits, which means 0-9.

print(re.findall(r"\d","I love python**3**")) # ['3']

print(re.findall(r"\w\d","I love pytho**n3**")) #['n3']

\D Matches any non-digits.

print(re.findall(r"\D","I love python3")) # ['I', ' ', 'l', 'o', 'v', 'e', ' ', 'p', 'y', 't', 'h', 'o', 'n']

Example

- Find all WBS student id in a text, such as u1888888.

```
re.findall(r'u1\d\d\d\d\d\d')
```

\s | Matches whitespace characters, which include the \t (tab space), \n (new line), \r (return), and space characters.

\S | Matches non-whitespace characters.

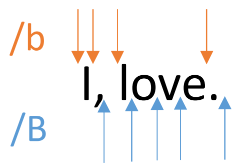
print(re.findall(r"\S","I love python3.")) # ['I', 'l', 'o', 'v', 'e', 'p', 'y', 't', 'h', 'o', 'n', '3', '.']

\b | matches the empty string (zero-width character, not blank space) at the beginning or end, i.e. boundary of a word (\w), in other words, between \w and \W.

\B | matches the any position that is not a word boundary \b.

print(re.findall(r"\w\b","I, love.")) #['I', 'e']

print(re.findall(r"\w\B","I, love.")) # ['l', 'o', 'v']



Special Characters

^ | matches the starting position of the string.

```
print(re.findall(r'^\w','I, love, python')) # ['I']
```

$ | matches the ending position of the string.

```
print(re.findall(r'\w$','I, love, python'))  # ['n']
```

- . | matches any character except line terminators like \n.
- \ | Escapes special characters or denotes character classes.
- A|B | Matches expression A or B.

Exercise