Data Processing and Visualizations using Python

Day 3 – Numpy and Pandas (Part 1)

SICSS 2022 – Haifa University

Amit Donner

Numpy – Numerical Python

- One of the most known and used modules in Python.
- Includes many helpful function and modules – saves programming time.
- Enables ‘vectorization’ – operations across lists and arrays, with no need of loops.
- Fast (very fast).
- Easy to use, especially with multidimensional arrays.
- Usually, abbreviated by *np*

```
import numpy as np
```

Numpy *ndarray* and lists

- Numpy works with *numpy.ndarray* objects, which are similar to lists.

```
x_li = [0, 1, 2]
x_np = np.array([0, 1, 2])
```

- Using the function *np.array* we can create a *numpy.ndarray.*

```
x_li
```

```
[0, 1, 2]
```

```
x_np
```

```
array([0, 1, 2])
```

```
print(type(x_np))
```

```
<class 'numpy.ndarray'>
```

- Or convert/coerce an existing list.

```
np.array(x_li)
```

```
array([0, 1, 2])
```

Element wise operations

- We can multiply or square (and many more options) each element in the array easily without loops.

```
2*x_np
```

```
array([0, 2, 4])
```

```
x_np**2
```

```
array([0, 1, 4], dtype=int32)
```

- With regular lists we will need to do the following:

```
[2*x for x in x_li]
```

```
[0, 2, 4]
```

```
[x**2 for x in x_li]
```

```
[0, 1, 4]
```

Mathematical functions

- Numpy includes many mathematical functions, with element-wise operations as well.

```
1 np.exp(x_np)
```

```
array([1.        , 2.71828183, 7.3890561 ])
```

```
1 np.sqrt(x_np)
```

```
array([0.        , 1.        , 1.41421356])
```

- The value of e^x and $\sqrt{x}$ for each value in *x_np*.
- Once again, with regular lists we must use loops.

Who needs the *range* function?

- Remember the function *range(start, stop, step)* that accepts only integer steps?

```
1 list(range(0, 10, 2))
```

```
[0, 2, 4, 6, 8]
```

```
1 list(range(0, 10, 1.5))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-11-af5b7860ad33> in <module>
----> 1 list(range(0, 10, 1.5))

TypeError: 'float' object cannot be interpreted as an integer
```

- Well, *numpy* has a solution for this as well. Using the function *np.arange* we can have non-integer step.

```
1 np.arange(1, 10, 0.5) # a sequence from 1 to 10 (not included) with step size 0.5.
```

```
array([1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5, 5. , 5.5, 6. , 6.5, 7. ,
       7.5, 8. , 8.5, 9. , 9.5])
```

```
1 np.arange(1, 10, 2) # a sequence from 1 to 10 (not included) with step size 2.
```

```
array([1, 3, 5, 7, 9])
```

```
1 np.arange(5, -5, -3) # a sequence from 5 to -5 (not included) with step size -3.
```

```
array([ 5,  2, -1, -4])
```

- *np.arange* is not alone. Using the function *np.linspace* we can create an array with fixed start and end points and pre-specified length.

```python
np.linspace(1, 3, 10) # a sequence from 1 to 3 of length 10
```

```
array([1.        , 1.22222222, 1.44444444, 1.66666667, 1.88888889,
       2.11111111, 2.33333333, 2.55555556, 2.77777778, 3.        ])
```

```python
len(np.linspace(1, 3, 10))
```

```
10
```

- The function automatically computes the required step size.

```python
np.linspace(-5, 0, 1000)
```

```
array([-5.        , -4.99499499, -4.98998999, -4.98498498, -4.97997998,
       -4.97497497, -4.96996997, -4.96496496, -4.95995996, -4.95495495,
       -4.94994995, -4.94494494, -4.93993994, -4.93493493, -4.92992993,
       -4.92492492, -4.91991992, -4.91491491, -4.90990991, -4.9049049 ,
       -4.8998999 , -4.89489489, -4.88988989, -4.88488488, -4.87987988,
       -4.87487487, -4.86986987, -4.86486486, -4.85985986, -4.85485485,
       -4.84984985, -4.84484484, -4.83983984, -4.83483483, -4.82982983,
       -4.82482482, -4.81981982, -4.81481481, -4.80980981, -4.8048048 ,
       -4.7997998 , -4.79479479, -4.78978979, -4.78478478, -4.77977978,
       -4.77477477, -4.76976977, -4.76476476, -4.75975976, -4.75475475,
       -4.74974975, -4.74474474, -4.73973974, -4.73473473, -4.72972973,
       -4.72472472, -4.71971972, -4.71471471, -4.70970971, -4.7047047 ,
       -4.6996997 , -4.69469469, -4.68968969, -4.68468468, -4.67967968,
       -4.67467467, -4.66966967, -4.66466466, -4.65965966, -4.65465465,
       -4.64964965, -4.64464464, -4.63963964, -4.63463463, -4.62962963,
       -4.62462462, -4.61961962, -4.61461461, -4.60960961, -4.6046046 ,
       -4.5995996 , -4.59459459, -4.58958959, -4.58458458, -4.57957958,
       -4.57457457, -4.56956957, -4.56456456, -4.55955956, -4.55455455,
       -4.54954955, -4.54454454, -4.53953954, -4.53453453, -4.52952953,
```

- Remember the list of grades from the first meeting?
- We used loops to find students who passed the course, and for counting the number of those who passed.
- Using *numpy*… well this task is much easier.

```
Grades = np.array([85, 67, 90, 100, 40, 26, 58, 32, 100, 21, 74, 30, 52, 49, 1, 84, 46])
print(Grades > 55)
```

```
[ True  True  True  True False False  True False  True False  True False
 False False False  True False]
```

```
print(np.sum(Grades > 55))
```

```
8
```

- What about finding students who passed the test? Just use *np.where*

```
np.where(Grades > 55)
```

```
(array([ 0,  1,  2,  3,  6,  8, 10, 15], dtype=int64),)
```

```
Pass = np.where(Grades > 55, 1, 0) # if True, 1. if False, 0.
Pass
```

```
array([1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0])
```

- Now, assume we want to give a ‘factor’ of 10 points only to those who passed the test. We can use the option of *indexing* in *numpy.ndarray*, using brackets [].

```
Grades[np.where(Grades > 55)] # Grades of students who passed
```

```
array([ 85,  67,  90, 100,  58, 100,  74,  84])
```

- Before we proceed, we will define a variable that includes the indices of those who passed the test.

```
passed = np.where(Grades > 55)
```

- Using the results from the previous slide we can add 10 points only to students who passed.

```
Grades[passed] + 10 # a bit problematic
```

```
array([ 95,  77, 100, 110,  68, 110,  84,  94])
```

- How can we fix the issue of grades above 100?
- Using the *np.minimum()* function. First, a small example:

```
np.minimum(np.array([1, 0, -5, 4]), np.array([10, -4, 0, 23]))
```

```
array([ 1, -4, -5,  4])
```

```
np.minimum(np.array([1, 0, -5, 4]), 0)
```

```
array([ 0,  0, -5,  0])
```

- The function returns the minimal value of each pair or compares each value in one array to a single given number.

- How can we use it in our case? We just give each student the minimal value between 100 and their own grade after adding 10 points.

```
np.minimum(Grades[passed] + 10, 100) # much better
```

```
array([ 95,  77, 100, 100,  68, 100,  84,  94])
```

- Now, we can change the values in the original array and obtain our final grades after the ‘factor’.

```
Grades[passed] = np.minimum(Grades[passed] + 10, 100)
```

```
Grades
```

```
array([ 95,  77, 100, 100,  40,  26,  68,  32, 100,  21,  84,  30,  52,
        49,   1,  94,  46])
```

- Recall the aggregated cars data from the first meeting.

```python
Category = ['Toyota', 'Mazda', 'BMW', 'KIA', 'Suzuki']
Freq = [125, 85, 10, 97, 102]
```

- How can we create an array with 125 “Toyota” values, 85 “Mazda” values and so on?
- Using the *np.repeat* function which accepts an array of *values* and an additional array of *counts* and creates repetitions of each value in *values* according the corresponding count in *counts*.

```python
Cars = np.repeat(Category, Freq)
```

```python
Cars
```

```
array(['Toyota', 'Toyota', 'Toyota', 'Toyota', 'Toyota', 'Toyota',
       'Toyota', 'Toyota', 'Toyota', 'Toyota', 'Toyota', 'Toyota',
       'Toyota', 'Toyota', 'Toyota', 'Toyota', 'Toyota', 'Toyota',
       'Toyota', 'Toyota', 'Toyota', 'Toyota', 'Toyota', 'Toyota',
       'Toyota', 'Toyota', 'Toyota', 'Toyota', 'Toyota', 'Toyota',
       'Toyota', 'Toyota', 'Toyota', 'Toyota', 'Toyota', 'Toyota',
       'Toyota', 'Toyota', 'Toyota', 'Toyota', 'Toyota', 'Toyota',
       'Toyota', 'Toyota', 'Toyota', 'Toyota', 'Toyota', 'Toyota',
       'Toyota', 'Toyota', 'Toyota', 'Toyota', 'Toyota', 'Toyota',
       'Toyota', 'Toyota', 'Toyota', 'Toyota', 'Toyota', 'Toyota',
       'Toyota', 'Toyota', 'Toyota', 'Toyota', 'Toyota', 'Toyota',
       'Toyota', 'Toyota', 'Toyota', 'Toyota', 'Toyota', 'Toyota',
       'Toyota', 'Toyota', 'Toyota', 'Toyota', 'Toyota', 'Toyota',
       'Toyota', 'Toyota', 'Toyota', 'Toyota', 'Toyota', 'Toyota',
       'Toyota', 'Toyota', 'Toyota', 'Toyota', 'Toyota', 'Toyota',
       'Toyota', 'Toyota', 'Toyota', 'Toyota', 'Toyota', 'Toyota',
       'Toyota', 'Toyota', 'Toyota', 'Toyota', 'Toyota', 'Toyota',
       'Toyota', 'Toyota', 'Toyota', 'Toyota', 'Toyota', 'Toyota',
       'Toyota', 'Toyota', 'Toyota', 'Toyota', 'Toyota', 'Toyota',
```

```
'Mazda', 'Mazda', 'Mazda', 'Mazda', 'Mazda', 'Mazda', 'Mazda',
'Mazda', 'Mazda', 'Mazda', 'Mazda', 'Mazda', 'Mazda', 'Mazda',
'Mazda', 'Mazda', 'Mazda', 'Mazda', 'Mazda', 'Mazda', 'Mazda',
'Mazda', 'Mazda', 'Mazda', 'Mazda', 'Mazda', 'Mazda', 'Mazda',
'Mazda', 'Mazda', 'Mazda', 'Mazda', 'Mazda', 'Mazda', 'Mazda',
'Mazda', 'Mazda', 'Mazda', 'Mazda', 'Mazda', 'Mazda', 'Mazda',
'Mazda', 'Mazda', 'Mazda', 'Mazda', 'Mazda', 'Mazda', 'Mazda',
'Mazda', 'Mazda', 'Mazda', 'Mazda', 'Mazda', 'Mazda', 'Mazda',
'Mazda', 'Mazda', 'Mazda', 'Mazda', 'Mazda', 'Mazda', 'Mazda',
'Mazda', 'Mazda', 'Mazda', 'Mazda', 'Mazda', 'Mazda', 'Mazda',
'Mazda', 'Mazda', 'Mazda', 'Mazda', 'Mazda', 'Mazda', 'Mazda',
'Mazda', 'Mazda', 'Mazda', 'Mazda', 'Mazda', 'Mazda', 'BMW', 'BMW
'BMW', 'BMW', 'BMW', 'BMW', 'BMW', 'BMW', 'BMW', 'BMW', 'KIA',
'KIA', 'KIA', 'KIA', 'KIA', 'KIA', 'KIA', 'KIA', 'KIA', 'KIA',
'KIA', 'KIA', 'KIA', 'KIA', 'KIA', 'KIA', 'KIA', 'KIA', 'KIA',
'KIA', 'KIA', 'KIA', 'KIA', 'KIA', 'KIA', 'KIA', 'KIA', 'KIA',
```

```
'KIA', 'KIA', 'KIA', 'KIA', 'KIA', 'KIA', 'KIA', 'KIA', 'KIA',
'KIA', 'KIA', 'KIA', 'KIA', 'KIA', 'KIA', 'Suzuki', 'Suzuki',
'Suzuki', 'Suzuki', 'Suzuki', 'Suzuki', 'Suzuki', 'Suzuki',
'Suzuki', 'Suzuki', 'Suzuki', 'Suzuki', 'Suzuki', 'Suzuki',
'Suzuki', 'Suzuki', 'Suzuki', 'Suzuki', 'Suzuki', 'Suzuki',
'Suzuki', 'Suzuki', 'Suzuki', 'Suzuki', 'Suzuki', 'Suzuki',
'Suzuki', 'Suzuki', 'Suzuki', 'Suzuki', 'Suzuki', 'Suzuki',
'Suzuki', 'Suzuki', 'Suzuki', 'Suzuki', 'Suzuki', 'Suzuki',
'Suzuki', 'Suzuki', 'Suzuki', 'Suzuki', 'Suzuki', 'Suzuki',
'Suzuki', 'Suzuki', 'Suzuki', 'Suzuki', 'Suzuki', 'Suzuki',
'Suzuki', 'Suzuki', 'Suzuki', 'Suzuki', 'Suzuki', 'Suzuki',
'Suzuki', 'Suzuki', 'Suzuki', 'Suzuki', 'Suzuki', 'Suzuki',
'Suzuki', 'Suzuki', 'Suzuki', 'Suzuki', 'Suzuki', 'Suzuki',
'Suzuki', 'Suzuki', 'Suzuki', 'Suzuki', 'Suzuki', 'Suzuki',
'Suzuki', 'Suzuki', 'Suzuki', 'Suzuki', 'Suzuki', 'Suzuki',
'Suzuki', 'Suzuki', 'Suzuki', 'Suzuki', 'Suzuki', 'Suzuki',
'Suzuki', 'Suzuki', 'Suzuki', 'Suzuki', 'Suzuki', 'Suzuki',
```

- Usually, data is observed without aggregation, and we need to do it by our own. The *np.unique* function finds all the different values within an array.

```
1 np.unique(Cars)
```

```
array(['BMW', 'KIA', 'Mazda', 'Suzuki', 'Toyota'], dtype='<U6')
```

- If we use the *return_counts* argument and set it to *True* we also get the frequencies of each unique value.

```
1 Vals, Counts = np.unique(Cars, return_counts=True)
```

```
1 Vals
```

```
array(['BMW', 'KIA', 'Mazda', 'Suzuki', 'Toyota'], dtype='<U6')
```

```
1 Counts
```

```
array([ 10,  97,  85, 102, 125], dtype=int64)
```

- Before we continue to another example, we need another justification for using *numpy*. We already saw that it requires less code, very easy to use, but is it fast enough?
- We will repeat the running time comparison from the first meeting, where we measured how long it takes to count the number of digits in $10M$ numbers.
- But, with the addition of one more function that uses only *numpy* functions.

```
from math import log10, floor
def numdigit(x):
    return len(str(abs(x)))
def numdigit1(x):
    return floor(log10(abs(x))) + 1
def numdigit2(x):
    return np.floor(np.log10(np.abs(x))) + 1
```

- And the winner is….

```python
from random import randint
import time

x = [randint(a=1e2, b=1e7) for i in range(10000000)] # 10000000 random numbers between 100 to 10000000

start = time.time()
y = [numdigit(X) for X in x]
stop = time.time()
print(f'Running time of numdigit is {round(stop-start,3)} in seconds')

start = time.time()
y = [numdigit1(X) for X in x]
stop = time.time()
print(f'Running time of numdigit1 is {round(stop-start,3)} in seconds')

start = time.time()
y3 = numdigit2(x) # No loop needed
stop = time.time()
print(f'Running time of numdigit2 is {round(stop-start,3)} in seconds')
```

```
Running time of numdigit is 2.39 in seconds
Running time of numdigit1 is 2.753 in seconds
Running time of numdigit2 is 0.548 in seconds
```

- Numpy! (by far)

- Let's resume to our grades data and explore additional and important functions.

```
Grades = np.array([85, 67, 90, 100, 40, 26, 58, 32, 100, 21, 74, 30, 52, 49, 1, 84, 46])
np.max(Grades)
```

```
100
```

```
np.min(Grades)
```

```
1
```

```
np.argmax(Grades) # Recall that indices starts from 0
```

```
3
```

```
np.argmin(Grades) # Recall that indices starts from 0
```

```
14
```

```
np.sort(Grades)
```

```
array([  1,  21,  26,  30,  32,  40,  46,  49,  52,  58,  67,  74,  84,
        85,  90, 100, 100])
```

Multidimensional arrays

- Nothing but ordinary *numpy.ndarray* with multiple axes.

```
arr = np.array([[1, 5, 10, -3, 5, 6],
                [3, -12, 1.2, 1, 0, -3]])
arr
```

```
array([[  1. ,   5. ,  10. ,  -3. ,   5. ,   6. ],
       [  3. , -12. ,   1.2,   1. ,   0. ,  -3. ]])
```

- We can select specific values according their indices.

```
arr[1, 3] # The element on the second row and fourh coloumn
```

```
1.0
```

```
arr[0, 4] # The element on the first row and fifth coloumn
```

```
5.0
```

- We can even select multiple values using the ":" sign.

```
arr[0, 1:5] # First row and columns 2 up to 5
```

```
array([ 5., 10., -3.,  5.])
```

```
arr[:, 4] # fourth colomn and all the rows
```

```
array([5., 0.])
```

```
arr[0, :] # fourth colomn and all the rows
```

```
array([ 1.,  5., 10., -3.,  5.,  6.])
```

- The *shape* attribute shows the array's dimensions.

```
arr.shape # The array dimension
```

```
(2, 6)
```

- We can change the dimension using the *reshape* method.

```
arr.reshape(4, 3)
```

```
array([[  1. ,   5. ,  10. ],
       [ -3. ,   5. ,   6. ],
       [  3. , -12. ,   1.2],
       [  1. ,   0. ,  -3. ]])
```

```
arr.reshape(3, 4)
```

```
array([[  1. ,   5. ,  10. ,  -3. ],
       [  5. ,   6. ,   3. , -12. ],
       [  1.2,   1. ,   0. ,  -3. ]])
```

- We can only change an $n \times m$ arrays into $p \times k$ arrays such that

$$n \cdot m = p \cdot k$$

```
arr.reshape(3,3) # 3*3 not equal 2*6
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-53-1b551e41d4bf> in <module>
----> 1 arr.reshape(3,3) # 3*3 not equal 2*6

ValueError: cannot reshape array of size 12 into shape (3,3)
```

- Additional option is to 'transpose' an array, i.e. rows become columns and columns become rows.

```
arr
```

```
array([[  1. ,   5. ,  10. ,  -3. ,   5. ,   6. ],
       [  3. , -12. ,   1.2,   1. ,   0. ,  -3. ]])
```

```
arr.T # Transpose
```

```
array([[  1. ,   3. ],
       [  5. , -12. ],
       [ 10. ,   1.2],
       [ -3. ,   1. ],
       [  5. ,   0. ],
       [  6. ,  -3. ]])
```

- When dealing with multidimensional arrays we can apply functions such as *np.min, np.max* and others on the rows or columns separately.

```
np.min(arr, axis = 1) # the minimal value of each row
```

```
array([ -3., -12.])
```

```
np.min(arr, axis = 0) # the minimal value of each coloum
```

```
array([  1. , -12. ,   1.2,  -3. ,   0. ,  -3. ])
```

```
np.argmin(arr, axis = 1) # the index of the minimal value in each row
```

```
array([3, 1], dtype=int64)
```

```
np.argmin(arr, axis = 0) # the index of the minimal value in each col
```

```
array([0, 1, 1, 0, 1, 1], dtype=int64)
```

- The same with *np.max, np.argmax*

- Sorting can be made row-wise or column-wise as well.

```
arr
```

```
array([[  1. ,   5. ,  10. ,  -3. ,   5. ,   6. ],
       [  3. , -12. ,   1.2,   1. ,   0. ,  -3. ]])
```

```
np.sort(arr, axis = 0) # sorting each column separately
```

```
array([[  1. , -12. ,   1.2,  -3. ,   0. ,  -3. ],
       [  3. ,   5. ,  10. ,   1. ,   5. ,   6. ]])
```

```
np.sort(arr, axis = 1) # sorting each row separately
```

```
array([[ -3. ,   1. ,   5. ,   5. ,   6. ,  10. ],
       [-12. ,  -3. ,   0. ,   1. ,   1.2,   3. ]])
```

- Finally, we can flatten a multidimensional array into a 1-D array.

```
arr.ravel()
```

```
array([  1. ,   5. ,  10. ,  -3. ,   5. ,   6. ,   3. , -12. ,   1.2,
         1. ,   0. ,  -3. ])
```

```
arr.ravel().shape # converting the 1-D array
```

```
(12,)
```

The random module of numpy.

- The numpy module itself includes many other modules, one of them is *random* which gives similar options as the one we encountered on the first meeting but with the support of vectorization.
- One of the functions included in the *numpy.random* module is *choice* which selects randomly (with equal probabilities) k elements out of a given array.
- The sample can be with or without replacement and with non-equal probabilities.

```
np.random.choice(arr.ravel(), 20, replace = True)
```

```
array([  5.,  -3.,   1.,   5.,   1.,  -3., -12., -12.,   5.,  10.,   1.,
         1., -12.,  -3.,  -3.,   5.,   6.,   5.,   5.,   1.])
```

- If we want to sample without replacement, we need to set *replace=False*.
- For sampling with different probabilities, we should pass to the argument *p*: an additional array with the same shape as the array of interest with positive elements that sums up to 1.
- For example, let's sample 10000 numbers from the array [1,2,3] with replacement and probabilities of [0.3, 0.3, 0.4].

```
sample = np.random.choice(np.array([1,2,3]), 10000, replace = True, p = [0.3, 0.3, 0.4])
```

```
_, Counts = np.unique(sample, return_counts=True)
```

```
Counts/10000
```

```
array([0.2968, 0.2961, 0.4071])
```

“Controlling” the randomness

- Since we sample randomly from a *numpy.array* the results won’t be the same if we will run the code many times.

```
np.random.choice(arr.ravel(), 20, replace = True)
```

```
array([  5.,  -3.,   1.,   5.,   1.,  -3., -12., -12.,   5.,  10.,   1.,
         1., -12.,  -3.,  -3.,   5.,   6.,   5.,   5.,   1.])
```

```
np.random.choice(arr.ravel(), 20, replace = True)
```

```
array([-12. ,   6. ,   1. ,  -3. ,   5. ,   5. ,   3. ,   5. ,  10. ,
        10. ,   6. ,  10. ,   0. ,  10. ,   5. ,   5. ,   1.2,   5. ,
         5. ,   6. ])
```

```
np.random.choice(arr.ravel(), 20, replace = True)
```

```
array([ 1. ,  5. , -3. ,  1.2,  5. ,  1.2,  0. , -3. ,  1. , 10. ,  5. ,
        5. ,  0. , -3. ,  6. ,  1.2,  1. ,  0. ,  5. ,  1. ])
```

- In order to deal with that we can use the function *np.random.seed()* which makes sure that each time we will get the same results.

```
np.random.seed(1)
np.random.choice(arr.ravel(), 20, replace = True)
```

```
array([  6. ,  -3. ,   1.2,   1. ,  -3. ,   6. ,   1. ,   1. ,   5. ,
       -12. ,   3. ,   1. ,  10. ,   5. ,   6. ,  10. ,   5. ,  -3. ,
         0. ,  10. ])
```

```
np.random.seed(1)
np.random.choice(arr.ravel(), 20, replace = True)
```

```
array([  6. ,  -3. ,   1.2,   1. ,  -3. ,   6. ,   1. ,   1. ,   5. ,
       -12. ,   3. ,   1. ,  10. ,   5. ,   6. ,  10. ,   5. ,  -3. ,
         0. ,  10. ])
```

```
np.random.seed(1)
np.random.choice(arr.ravel(), 20, replace = True)
```

```
array([  6. ,  -3. ,   1.2,   1. ,  -3. ,   6. ,   1. ,   1. ,   5. ,
       -12. ,   3. ,   1. ,  10. ,   5. ,   6. ,  10. ,   5. ,  -3. ,
         0. ,  10. ])
```

- Wait, sampling with replacement… why does it sound familiar?
- That’s exactly what the *Bootstrap* procedure does!
- Reminder, given a sample of n observations where we are interested in estimating some parameter θ, we can sample with replacement n observations out of the original sample, compute the estimator of θ for the current bootstrap sample and by repeating this steps B times we can approximate the distribution of the sample estimator.

- For example, recall the data from last meeting.

```
x = [3.1, 2.7, 3.3, 3.3, 5.2, 10.9, 6.1, 2.4, 5.5, 15.1, 10.1, 6.2, 10.7,
     13.9, 10.6, 13.2, 8.5, 12.5, 11.1, 6.4, 11.6, 11, 12.2, 9.5, 6.7,
     10.3, 7.1, 2.3, 4.5, 11.1, 2.3, 9.3, 2.9, 1.4, 7.4, 1.7, 3,
     11.3, 7.8, 10.3, 6.8, 6.2, 5.3, 5, 11.5, 9.7, 4.2, 4.5, 4.3, 6.9]
```

- Now assume we want to calculate a 95% CI for the **standard deviation** of X.
- First, we will create B bootstrap samples

```
B = 5000
n = len(x)
```

```
np.random.seed(10)
Boot = np.random.choice(x, n*B, replace = True).reshape(n, B)
```

```
Boot
```

```
array([[15.1,  3. , 13.2, ...,  2.7, 13.2,  6.9],
       [ 3.3,  1.4,  5.2, ..., 10.3, 11.3,  5.5],
       [ 6.4,  3.3,  3.3, ...,  2.3,  2.3,  7.8],
       ...,
       [11.1, 10.6, 11.1, ...,  2.4,  4.2,  2.3],
       [ 9.5,  4.2,  4.3, ...,  5.2,  3.1,  7.4],
       [ 1.7,  9.7, 13.9, ..., 10.6,  3. ,  6.8]])
```

```
Boot.shape # 5000 bootstrap samples
```

```
(50, 5000)
```

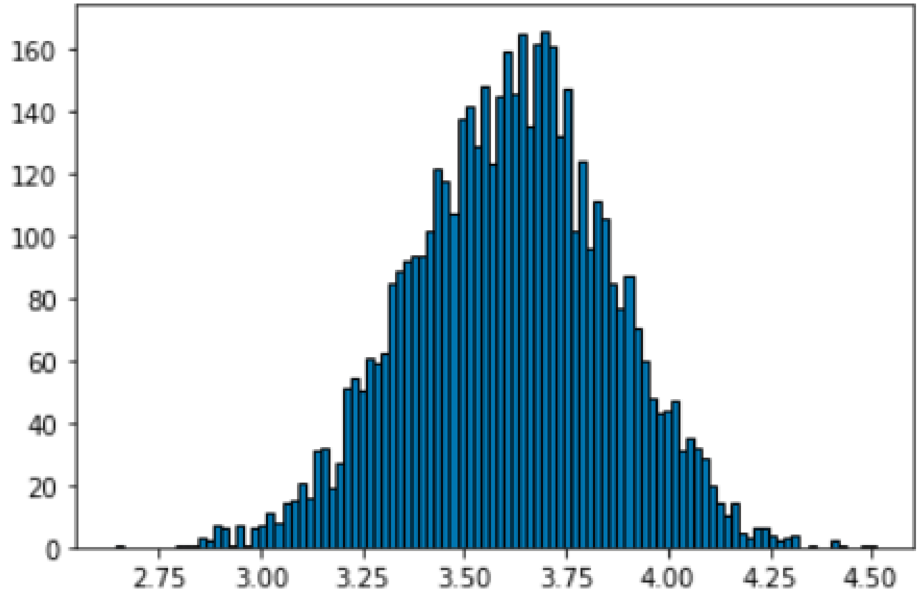
- Then, we can compute the standard deviation of each column.

```
boot_values = np.std(Boot, axis = 0)
boot_values.shape
```

```
(5000,)
```

- The approximated distribution of the sample estimator is

```
import matplotlib.pyplot as plt
plt.hist(boot_values, bins = 100, edgecolor = "black")
plt.show()
```



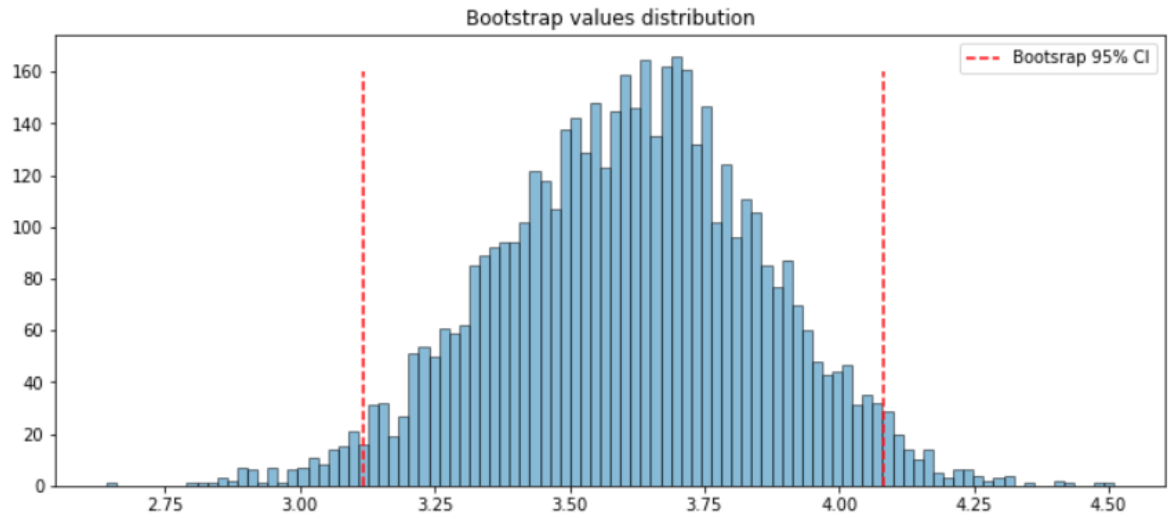
- And the approximated CI can be obtained by computing the 0.025 and 0.975 quantiles of the approximated distribution.

```
np.quantile(boot_values, [0.025, 0.975])
```

```
array([3.11855909, 4.08052814])
```

- Later we can have ourselves a nice plot.

```
quant = np.quantile(boot_values, [0.025, 0.975])
plt.figure(figsize=(12,5))
plt.hist(boot_values, bins = 100, edgecolor = "black", alpha = 0.5)
plt.vlines(quant, 0, 160, color = "red", linestyle = "dashed")
plt.legend(loc = 'best', labels = ['Bootsrap 95% CI'])
plt.title('Bootstrap values distribution')
plt.show()
```



- Just to make sure you got it; this is **all** the code we needed for computing a 95% bootstrap CI for the standard deviation of an unknown distribution:

```
B = 5000
n = len(x)
np.random.seed(10)
Boot = np.random.choice(x, n*B, replace = True).reshape(n, B)
boot_values = np.std(Boot, axis = 0)
np.quantile(boot_values, [0.025, 0.975])
```

- Now, try to do it without *numpy*…
- Later, we will demonstrate why bootstrap works.

Pandas (Part 1) – Let's get this data started

- THE Python module for dealing and analyzing datasets.
- Enables many aggregation and grouping options.
- Integrates perfectly with *numpy, matplotlib.pyplot*,
  and *seaborn* (next meeting).
- Usually abbreviated by *pd*

```
import pandas as pd
```

Excel Spreadsheets

Pandas

Creating a pandas DataFrame

- Recall the *numpy.ndarray* from before

```
arr
```

```
array([[  1. ,   5. ,  10. ,  -3. ,   5. ,   6. ],
       [  3. , -12. ,   1.2,   1. ,   0. ,  -3. ]])
```

- Using the pd.DataFrame() function we can convert it to a data frame.

```
data = pd.DataFrame(arr) # Converting numpy array to pandas data frame
data
```

	0	1	2	3	4	5
0	1.0	5.0	10.0	-3.0	5.0	6.0
1	3.0	-12.0	1.2	1.0	0.0	-3.0

```
print(type(data))
```

```
<class 'pandas.core.frame.DataFrame'>
```

- Each pandas data frame has the attributes *columns* and *index*

```
data.columns
```

```
RangeIndex(start=0, stop=6, step=1)
```

```
data.index
```

```
RangeIndex(start=0, stop=2, step=1)
```

- If we don’t specify specific values, pandas just assign consecutive numbers.
- We can change it in the following way

```
data.columns = ['Var 1', 'Var 2', 'Var 3', 'Var 4', 'Var 5', 'Var 6']
data.index = ["Obs 1", 'Obs 1']
data
```

	Var 1	Var 2	Var 3	Var 4	Var 5	Var 6
Obs 1	1.0	5.0	10.0	-3.0	5.0	6.0
Obs 1	3.0	-12.0	1.2	1.0	0.0	-3.0

- We can also specify values using the *columns* and *index* arguments of the *pd.DataFrame* function.

```
pd.DataFrame(arr,
             columns=['Var 1', 'Var 2', 'Var 3', 'Var 4', 'Var 5', 'Var 6'],
             index = ["Obs 1", 'Obs 1'])
```

	Var 1	Var 2	Var 3	Var 4	Var 5	Var 6
Obs 1	1.0	5.0	10.0	-3.0	5.0	6.0
Obs 1	3.0	-12.0	1.2	1.0	0.0	-3.0

Data frames can be made from scratch using dictionaries.

```
pd.DataFrame({"Var 1": [1, 3],
              "Var 2": [5, -12]},
             index = ['Obs 1', 'Obs 2'])
```

	Var 1	Var 2
Obs 1	1	5
Obs 2	3	-12

- Usually we import/read data sets from an excel file or other sources.
- Using the *read.csv* function we can do so.
- We will demonstrate it using the IMDB data set.

```
dat = pd.read_csv('IMDB.csv')
```

- Using the *head()* method we can see only the top rows.

```
dat.head(10)
```

	Title	Genre	Year	Runtime (Minutes)	Rating	Votes	Revenue (Millions)
0	Guardians of the Galaxy	Action,Adventure,Sci-Fi	2014	121	8.1	757074	333.13
1	Prometheus	Adventure,Mystery,Sci-Fi	2012	124	7.0	485820	126.46
2	Split	Horror,Thriller	2016	117	7.3	157606	138.12
3	Sing	Animation,Comedy,Family	2016	108	7.2	60545	270.32
4	Suicide Squad	Action,Adventure,Fantasy	2016	123	6.2	393727	325.02
5	The Great Wall	Action,Adventure,Fantasy	2016	103	6.1	56036	45.13
6	La La Land	Comedy,Drama,Music	2016	128	8.3	258682	151.06
7	Mindhorn	Comedy	2016	89	6.4	2490	NaN
8	The Lost City of Z	Action,Adventure,Biography	2016	141	7.1	7188	8.01
9	Passengers	Adventure,Drama,Romance	2016	116	7.0	192177	100.01

- Using the *shape* attribute, we can see the number of rows and columns.

```
dat.shape # 1000 rows on 7 columns
```

```
(1000, 7)
```

- In addition, we can get the column names using the *columns* attribute.

```
dat.columns
```

```
Index(['Title', 'Genre', 'Year', 'Runtime (Minutes)', 'Rating', 'Votes',
       'Revenue (Millions)'],
      dtype='object')
```

- The *describe()* method gives a summary of descriptive statistics for all the numerical variables.

```
1 dat.describe() # Only numerical
```

	Year	Runtime (Minutes)	Rating	Votes	Revenue (Millions)
count	1000.000000	1000.000000	1000.000000	1.000000e+03	872.000000
mean	2012.783000	113.172000	6.723200	1.698083e+05	82.956376
std	3.205962	18.810908	0.945429	1.887626e+05	103.253540
min	2006.000000	66.000000	1.900000	6.100000e+01	0.000000
25%	2010.000000	100.000000	6.200000	3.630900e+04	13.270000
50%	2014.000000	111.000000	6.800000	1.107990e+05	47.985000
75%	2016.000000	123.000000	7.400000	2.399098e+05	113.715000
max	2016.000000	191.000000	9.000000	1.791916e+06	936.630000

- We can also get a summary of the categorical variables, using the argument *include = "object"*

```
1 dat.describe(include = 'object')
```

	Title	Genre
count	1000	1000
unique	999	207
top	The Host	Action,Adventure,Sci-Fi
freq	2	50

- We can see the *dtypes* of each column using the *dtype* attribute.

```
dat.dtypes
```

```
Title        object
Genre        object
Year          int64
Runtime       int64
Rating      float64
Votes         int64
Revenue     float64
dtype: object
```

- Using the knowledge of the different dtypes we can use the *select_dtype* method and select only columns with the required dtypes.

```
dat.select_dtypes(include=['float64', 'int64'])
```

	Year	Runtime (Minutes)	Rating	Votes	Revenue (Millions)
0	2014	121	8.1	757074	333.13
1	2012	124	7.0	485820	126.46
2	2016	117	7.3	157606	138.12
3	2016	108	7.2	60545	270.32
4	2016	123	6.2	393727	325.02
...	...	...	...	...	...
995	2015	111	6.2	27585	NaN
996	2007	94	5.5	73152	17.54
997	2008	98	6.2	70699	58.01
998	2014	93	5.6	4881	NaN
999	2016	87	5.3	12435	19.64

1000 rows × 5 columns

- We saw before that the year variable was considered numerical and not categorical.
- We can easily change that

```
dat[['Year']] = dat[['Year']].astype('object')
```

```
dat.dtypes
```

```
Title                  object
Genre                  object
Year                   object
Runtime (Minutes)       int64
Rating                float64
Votes                   int64
Revenue (Millions)    float64
dtype: object
```

```
dat.describe(include=['object'])
```

	Title	Genre	Year
count	1000	1000	1000
unique	999	207	11
top	The Host	Action,Adventure,Sci-Fi	2016
freq	2	50	297

- Apparently, we have 2 movies with the same name? or is it a mistake?

- Using the *loc()* method we can select only rows with *Title == 'The Host'*.

```
dat.loc[dat.Title == 'The Host']
```

	Title	Genre	Year	Runtime (Minutes)	Rating	Votes	Revenue (Millions)
239	The Host	Action,Adventure,Romance	2013	125	5.9	96852	26.62
632	The Host	Comedy,Drama,Horror	2006	120	7.0	73491	2.20

- What about all the movies from 2008?

```
dat.loc[dat['Year']==2008].head(10)
```

	Title	Genre	Year	Runtime (Minutes)	Rating	Votes	Revenue (Millions)
54	The Dark Knight	Action,Crime,Drama	2008	152	9.0	1791916	533.32
128	Mamma Mia!	Comedy,Family,Musical	2008	108	6.4	153481	143.70
165	Twilight	Drama,Fantasy,Romance	2008	122	5.2	361449	191.45
177	Tropic Thunder	Action,Comedy	2008	107	7.0	321442	110.42
203	Iron Man	Action,Adventure,Sci-Fi	2008	126	7.9	737719	318.30
303	The House Bunny	Comedy,Romance	2008	97	5.5	67033	48.24
322	RocknRolla	Action,Crime,Thriller	2008	114	7.3	203096	5.69
360	Step Brothers	Comedy	2008	98	6.9	223065	100.47
365	Slumdog Millionaire	Drama	2008	120	8.0	677044	141.32
373	Wanted	Action,Crime,Fantasy	2008	110	6.7	312495	134.57

- We can also select specific columns.

```
dat.loc[dat['Year']==2008, ['Title', 'Runtime (Minutes)']].head()
```

	Title	Runtime (Minutes)
54	The Dark Knight	152
128	Mamma Mia!	108
165	Twilight	122
177	Tropic Thunder	107
203	Iron Man	126

- What about more complicated conditions?

```
dat.loc[(dat['Year']==2008) & (dat['Runtime (Minutes)']>140) , ['Title', 'Runtime (Minutes)']]
```

	Title	Runtime (Minutes)
54	The Dark Knight	152
425	The Curious Case of Benjamin Button	166
703	Australia	165
859	Changeling	141
893	Sex and the City	145

- The *iloc* method enables selecting rows and columns using indexing.

```
dat.iloc[2:4, 1:6] # rows 2 and 3, columns 1 to 5
```

	Genre	Year	Runtime (Minutes)	Rating	Votes
2	Horror,Thriller	2016	117	7.3	157606
3	Animation,Comedy,Family	2016	108	7.2	60545

- Another example,

```
dat.iloc[2:4, :] # rows 2 and 3, and all the columns
```

	Title	Genre	Year	Runtime (Minutes)	Rating	Votes	Revenue (Millions)
2	Split	Horror,Thriller	2016	117	7.3	157606	138.12
3	Sing	Animation,Comedy,Family	2016	108	7.2	60545	270.32

DataFrame Vs. Series

- The two main objects in pandas are *DataFrame* and *Series*. The main difference is that the first is a 2-dimensional array and the second is a 1-dimensional.

```python
dat[['Title']] # DataFrame
```

	Title
0	Guardians of the Galaxy
1	Prometheus
2	Split
3	Sing
4	Suicide Squad
...	...
995	Secret in Their Eyes
996	Hostel: Part II
997	Step Up 2: The Streets
998	Search Party
999	Nine Lives

1000 rows × 1 columns

```python
dat['Title'] # Series
```

```
0        Guardians of the Galaxy
1                     Prometheus
2                          Split
3                           Sing
4                  Suicide Squad
                  ...
995         Secret in Their Eyes
996              Hostel: Part II
997       Step Up 2: The Streets
998                 Search Party
999                   Nine Lives
Name: Title, Length: 1000, dtype: object
```

```python
type(dat[['Title']])
```

```
pandas.core.frame.DataFrame
```

```python
type(dat['Title'])
```

```
pandas.core.series.Series
```

```python
dat[['Title']].shape
```

```
(1000, 1)
```

```python
dat['Title'].shape
```

```
(1000,)
```

- Sometimes the column names are too long, complicated or uninformative, so we can change them using the *rename* method.

```
dat=dat.rename(columns={'Runtime (Minutes)': 'Runtime',
                        'Revenue (Millions)': 'Revenue'})
dat.columns
```

```
Index(['Title', 'Genre', 'Year', 'Runtime', 'Rating', 'Votes', 'Revenue'], dtype='object')
```

- We can also change the indices names and using the *replace* method we can change values. For example, we can change the *Horror,Thriller* genre to just *Horror*

```
dat.replace({"Genre": {"Horror,Thriller": "Horror"}}).head(5)
```

	Title	Genre	Year	Runtime	Rating	Votes	Revenue
0	Guardians of the Galaxy	Action,Adventure,Sci-Fi	2014	121	8.1	757074	333.13
1	Prometheus	Adventure,Mystery,Sci-Fi	2012	124	7.0	485820	126.46
2	Split	Horror	2016	117	7.3	157606	138.12
3	Sing	Animation,Comedy,Family	2016	108	7.2	60545	270.32
4	Suicide Squad	Action,Adventure,Fantasy	2016	123	6.2	393727	325.02

- Another important method is *unique.* For example, we can find all the different years in the data.

```
dat['Year'].unique()
```

```
array([2014, 2012, 2016, 2015, 2007, 2011, 2008, 2006, 2009, 2010, 2013],
      dtype=object)
```

- We can also sort it using the *np.sort* function.

```
np.sort(dat['Year'].unique())
```

```
array([2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016],
      dtype=object)
```

Discretization

- The pd.cut function is very helpful when we want to divide a numerical variable into categories. For example, split the *Year* variable to 3 groups.

```
1 pd.cut(dat['Year'], bins = 3) # a bit problematic
```

```
0        (2012.667, 2016.0]
1      (2009.333, 2012.667]
2        (2012.667, 2016.0]
3        (2012.667, 2016.0]
4        (2012.667, 2016.0]
                ...
995      (2012.667, 2016.0]
996     (2005.99, 2009.333]
997     (2005.99, 2009.333]
998      (2012.667, 2016.0]
999      (2012.667, 2016.0]
Name: Year, Length: 1000, dtype: category
Categories (3, interval[float64]): [(2005.99, 2009.333] < (2009.333, 2012.667] < (2012.667, 2016.0]]
```

- We see that by default *pandas* creates 3 equally length categories, and the bounds are not necessarily integers.
- In addition, observe that the categories are ordered.

- We can fix it with specifying our own limits.

```
1  pd.cut(dat['Year'], bins=[-np.inf, 2009, 2013, 2016])
```

```
0      (2013.0, 2016.0]
1      (2009.0, 2013.0]
2      (2013.0, 2016.0]
3      (2013.0, 2016.0]
4      (2013.0, 2016.0]
             ...
995    (2013.0, 2016.0]
996      (-inf, 2009.0]
997      (-inf, 2009.0]
998    (2013.0, 2016.0]
999    (2013.0, 2016.0]
Name: Year, Length: 1000, dtype: category
Categories (3, interval[float64]): [(-inf, 2009.0] < (2009.0, 2013.0] < (2013.0, 2016.0]]
```

- We can even set the labels of each category.

```
1  pd.cut(dat['Year'], bins=[-np.inf, 2009, 2013, 2016], labels = ['<=2009', '(2009,2013]', '>2013'])
```

```
0             >2013
1       (2009,2013]
2             >2013
3             >2013
4             >2013
           ...
995           >2013
996          <=2009
997          <=2009
998           >2013
999           >2013
Name: Year, Length: 1000, dtype: category
Categories (3, object): ['<=2009' < '(2009,2013]' < '>2013']
```

- Finally, we can add the aggregated variable to our data.

```python
dat[['Year Category']] = pd.cut(dat['Year'], bins=[-np.inf, 2009, 2013, 2016],
                                labels = ['<=2009', '(2009,2013]', '>2013'])
dat.head(10)
```

	Title	Genre	Year	Runtime	Rating	Votes	Revenue	Year Category
0	Guardians of the Galaxy	Action,Adventure,Sci-Fi	2014	121	8.1	757074	333.13	>2013
1	Prometheus	Adventure,Mystery,Sci-Fi	2012	124	7.0	485820	126.46	(2009,2013]
2	Split	Horror,Thriller	2016	117	7.3	157606	138.12	>2013
3	Sing	Animation,Comedy,Family	2016	108	7.2	60545	270.32	>2013
4	Suicide Squad	Action,Adventure,Fantasy	2016	123	6.2	393727	325.02	>2013
5	The Great Wall	Action,Adventure,Fantasy	2016	103	6.1	56036	45.13	>2013
6	La La Land	Comedy,Drama,Music	2016	128	8.3	258682	151.06	>2013
7	Mindhorn	Comedy	2016	89	6.4	2490	NaN	>2013
8	The Lost City of Z	Action,Adventure,Biography	2016	141	7.1	7188	8.01	>2013
9	Passengers	Adventure,Drama,Romance	2016	116	7.0	192177	100.01	>2013

- After creating the new variable, we would like to count the number of observations in each category. Using the *value_counts* method we get:

```
1 dat['Year Category'].value_counts()
```

```
>2013          522
(2009,2013]    278
<=2009         200
Name: Year Category, dtype: int64
```

- The default is to present the categories in descending order according to the frequencies. We can override it by setting *sort = False*.

```
1 dat['Year Category'].value_counts(sort = False) # Don't sort by frequencies
```

```
<=2009         200
(2009,2013]    278
>2013          522
Name: Year Category, dtype: int64
```

- Another option is to return proportions instead of counts, using *normalize = True.*

```
1 dat['Year Category'].value_counts(sort = False, normalize=True)
```

```
<=2009         0.200
(2009,2013]    0.278
>2013          0.522
Name: Year Category, dtype: float64
```

- Before we introduce the next function, Let’s split the *Runtime* variable into categories as well.

```python
dat[['Runtime Category']] = pd.cut(dat['Runtime'], bins = [-np.inf, 90, 120, np.inf],
                                   labels = ['Short Movies', 'Regular', 'Long Movies'])
dat['Runtime Category'].value_counts()
```

```
Regular         630
Long Movies     289
Short Movies     81
Name: Runtime Category, dtype: int64
```

Cross tabulation

- The *pd.crosstab* function is a generalization of *value_counts* and it enables us to create 2-dimensional frequencies table.

```
pd.crosstab(dat['Runtime Category'], dat['Year Category'])
```

Year Category / Runtime Category	<=2009	(2009,2013]	>2013
Short Movies	10	9	62
Regular	119	180	331
Long Movies	71	89	129

- We can add the marginal counts using *margins = True.*

```
pd.crosstab(dat['Runtime Category'], dat['Year Category'], margins=True)
```

Year Category / Runtime Category	<=2009	(2009,2013]	>2013	All
Short Movies	10	9	62	81
Regular	119	180	331	630
Long Movies	71	89	129	289
All	200	278	522	1000

- Normalization is allowed as well, with respect to the total sample size.

```
pd.crosstab(dat['Runtime Category'], dat['Year Category'],
            normalize='all', margins=True)
```

Year Category	<=2009	(2009,2013]	>2013	All
Runtime Category				
Short Movies	0.010	0.009	0.062	0.081
Regular	0.119	0.180	0.331	0.630
Long Movies	0.071	0.089	0.129	0.289
All	0.200	0.278	0.522	1.000

- Or with respect to rows/columns.

```
pd.crosstab(dat['Runtime Category'], dat['Year Category'],
            normalize='index')
```

Year Category	<=2009	(2009,2013]	>2013
Runtime Category			
Short Movies	0.123457	0.111111	0.765432
Regular	0.188889	0.285714	0.525397
Long Movies	0.245675	0.307958	0.446367

```
pd.crosstab(dat['Runtime Category'], dat['Year Category'],
            normalize='columns')
```

Year Category	<=2009	(2009,2013]	>2013
Runtime Category			
Short Movies	0.050	0.032374	0.118774
Regular	0.595	0.647482	0.634100
Long Movies	0.355	0.320144	0.247126

- Instead of using *describe* we can compute only specific statistics.
- For example, correlation between all numerical variables.

```
dat.corr().round(3)
```

	Runtime	Rating	Votes	Revenue
Runtime	1.000	0.392	0.407	0.268
Rating	0.392	1.000	0.512	0.218
Votes	0.407	0.512	1.000	0.640
Revenue	0.268	0.218	0.640	1.000

- Or means and standard deviations.

```
dat.mean().round(3)
```

```
Year          2012.783
Runtime        113.172
Rating           6.723
Votes       169808.255
Revenue         82.956
dtype: float64
```

```
dat.std().round(3)
```

```
Year             3.206
Runtime         18.811
Rating           0.945
Votes       188762.648
Revenue        103.254
dtype: float64
```

- Although Python excludes *Year* in the correlation matrix, it was included in the mean and standard deviation computations.
- We can fix it using the *drop* method.

```
dat.mean().round(3).drop("Year")
```

```
Runtime        113.172
Rating           6.723
Votes       169808.255
Revenue         82.956
dtype: float64
```

```
dat.std().round(3).drop("Year")
```

```
Runtime         18.811
Rating           0.945
Votes       188762.648
Revenue        103.254
dtype: float64
```

Group by

- On many occasions we would like to have a separate computation for each category in our data. The *groupby* method does exactly that.
- For example, the means and standard deviations for the numerical variables in each category of *Year Category.*

```
dat.groupby(dat['Year Category']).mean().round(3)
```

Year Category	Runtime	Rating	Votes	Revenue
<=2009	117.240	6.997	260846.980	96.699
(2009,2013]	115.371	6.847	246491.784	95.936
>2013	110.443	6.552	94088.397	68.114

```
dat.groupby(dat['Year Category']).std().round(3)
```

Year Category	Runtime	Rating	Votes	Revenue
<=2009	21.163	0.956	214734.004	105.256
(2009,2013]	18.059	0.823	201371.094	100.845
>2013	17.822	0.969	130714.878	102.110

- And what about correlation in each level?

```
dat.groupby(dat['Year Category']).corr().round(3)
```

Year Category		Runtime	Rating	Votes	Revenue
<=2009	Runtime	1.000	0.340	0.299	0.280
	Rating	0.340	1.000	0.504	0.124
	Votes	0.299	0.504	1.000	0.650
	Revenue	0.280	0.124	0.650	1.000
(2009,2013]	Runtime	1.000	0.363	0.439	0.334
	Rating	0.363	1.000	0.600	0.244
	Votes	0.439	0.600	1.000	0.655
	Revenue	0.334	0.244	0.655	1.000
>2013	Runtime	1.000	0.396	0.414	0.200
	Rating	0.396	1.000	0.435	0.223
	Votes	0.414	0.435	1.000	0.653
	Revenue	0.200	0.223	0.653	1.000

- Grouped data frames have multilevel columns, with the categorical variable’s levels as indices. It is sometimes useful to cancel it, using the *reset_index* method.

```
dat.groupby(dat['Year Category']).mean().round(3)
```

	Runtime	Rating	Votes	Revenue
Year Category				
<=2009	117.240	6.997	260846.980	96.699
(2009,2013]	115.371	6.847	246491.784	95.936
>2013	110.443	6.552	94088.397	68.114

```
dat.groupby(dat['Year Category']).mean().reset_index()
```

	Year Category	Runtime	Rating	Votes	Revenue
0	<=2009	117.240000	6.997000	260846.980000	96.699312
1	(2009,2013]	115.370504	6.847122	246491.784173	95.935867
2	>2013	110.442529	6.552299	94088.396552	68.114490

- Another example, grouping by the different years:

```
dat.groupby(dat['Year']).mean().round(3)
```

Year	Runtime	Rating	Votes	Revenue
2006	120.841	7.125	269289.955	86.297
2007	121.623	7.134	244331.038	87.882
2008	110.827	6.785	275505.385	99.083
2009	116.118	6.961	255780.647	112.601
2010	111.133	6.827	252782.317	105.082
2011	114.603	6.838	240790.302	87.612
2012	119.109	6.925	285226.094	107.973
2013	116.066	6.812	219049.648	87.122
2014	114.490	6.838	203930.224	85.079
2015	114.496	6.602	115726.220	78.355
2016	107.374	6.437	48591.754	54.691

- We can also select only a specific column.

```
dat.groupby(dat['Year']).mean().round(3)[['Revenue']].reset_index()
```

	Year	Revenue
0	2006	86.297
1	2007	87.882
2	2008	99.083
3	2009	112.601
4	2010	105.082
5	2011	87.612
6	2012	107.973
7	2013	87.122
8	2014	85.079
9	2015	78.355
10	2016	54.691

- More manipulations can be made by *sort_values.*

```
dat.groupby(dat['Year']).mean().round(3)\
.reset_index().sort_values('Revenue')
```

	Year	Runtime	Rating	Votes	Revenue
10	2016	107.374	6.437	48591.754	54.691
9	2015	114.496	6.602	115726.220	78.355
8	2014	114.490	6.838	203930.224	85.079
0	2006	120.841	7.125	269289.955	86.297
7	2013	116.066	6.812	219049.648	87.122
5	2011	114.603	6.838	240790.302	87.612
1	2007	121.623	7.134	244331.038	87.882
2	2008	110.827	6.785	275505.385	99.083
4	2010	111.133	6.827	252782.317	105.082
6	2012	119.109	6.925	285226.094	107.973
3	2009	116.118	6.961	255780.647	112.601

```
dat.groupby(dat['Year']).mean().round(3)\
.reset_index().sort_values('Revenue', ascending=False)
```

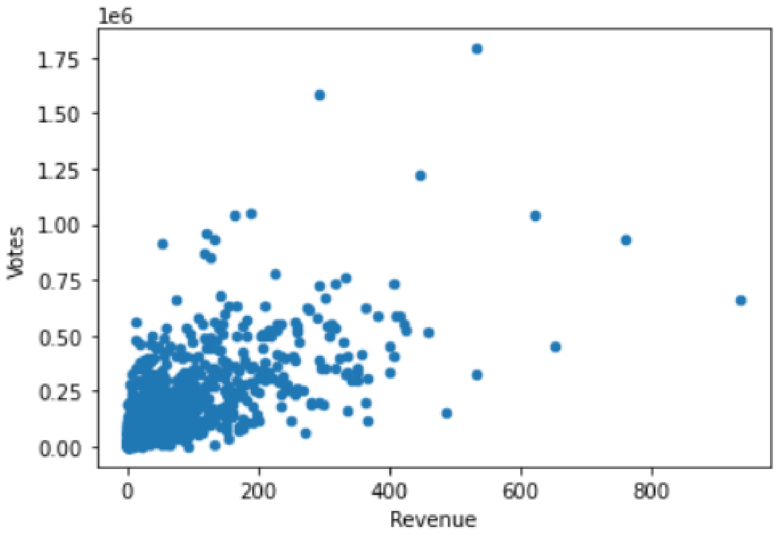
	Year	Runtime	Rating	Votes	Revenue
3	2009	116.118	6.961	255780.647	112.601
6	2012	119.109	6.925	285226.094	107.973
4	2010	111.133	6.827	252782.317	105.082
2	2008	110.827	6.785	275505.385	99.083
1	2007	121.623	7.134	244331.038	87.882
5	2011	114.603	6.838	240790.302	87.612
7	2013	116.066	6.812	219049.648	87.122
0	2006	120.841	7.125	269289.955	86.297
8	2014	114.490	6.838	203930.224	85.079
9	2015	114.496	6.602	115726.220	78.355
10	2016	107.374	6.437	48591.754	54.691

- *The "\" sign enables line breaks in the code.*

Plotting with pandas

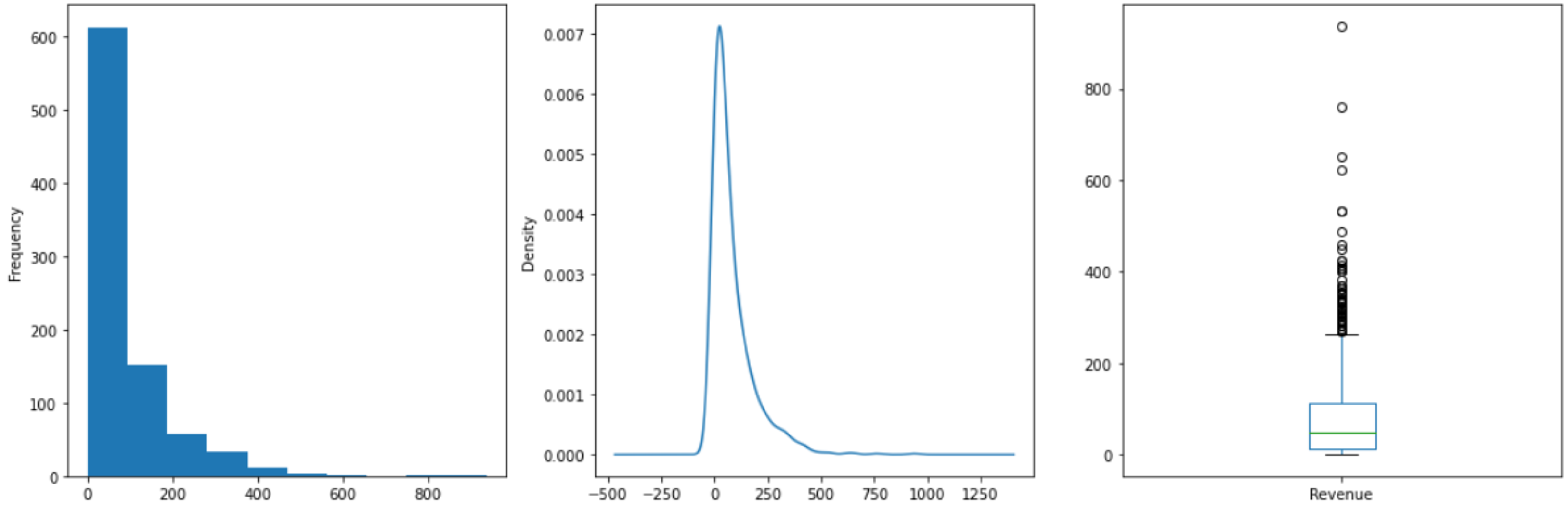
- Data frames in pandas include the *plot* method that integrates with matplotlib.pyplot perfectly and enables instant plotting.

```
dat.plot(kind = "scatter", x = 'Revenue', y = 'Votes')
plt.show() # For this line - make sure the matplotlib.pyplot is loaded
```



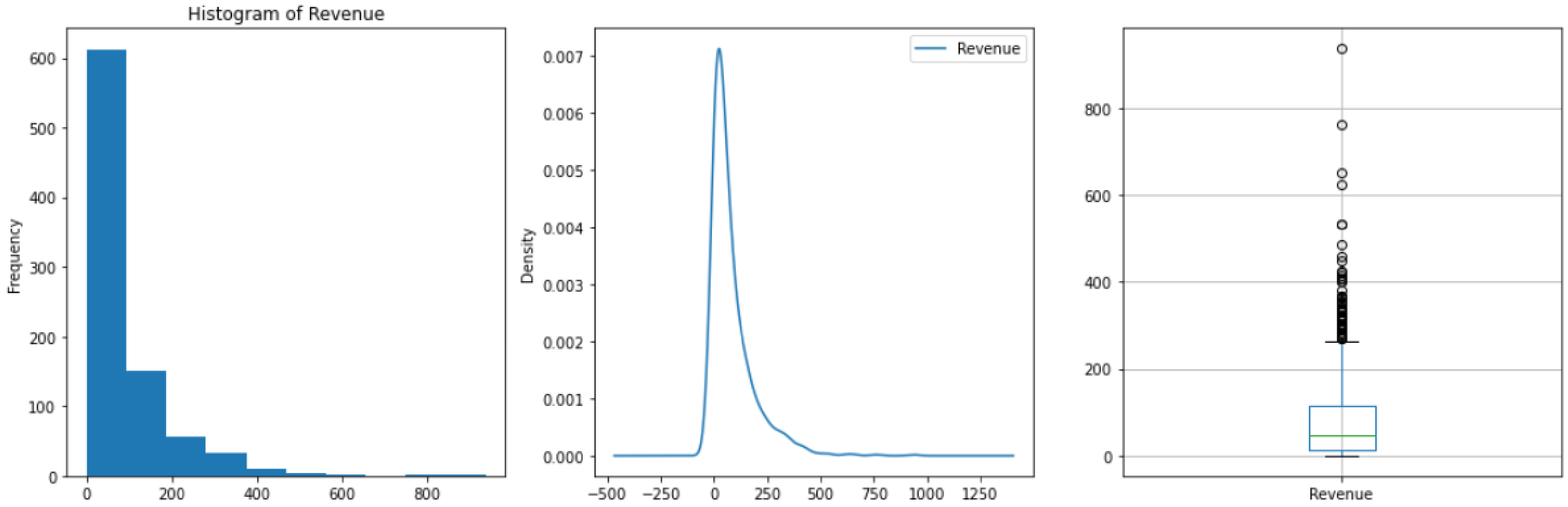
- We can also have histograms, density plots, boxplots and many more.

```python
plt.figure(figsize=(15,5))
plt.subplot(1,3,1)
dat['Revenue'].plot(kind = "hist")
plt.subplot(1,3,2)
dat['Revenue'].plot(kind = "kde")
plt.subplot(1,3,3)
dat['Revenue'].plot(kind = "box")
plt.tight_layout()
plt.show()
```



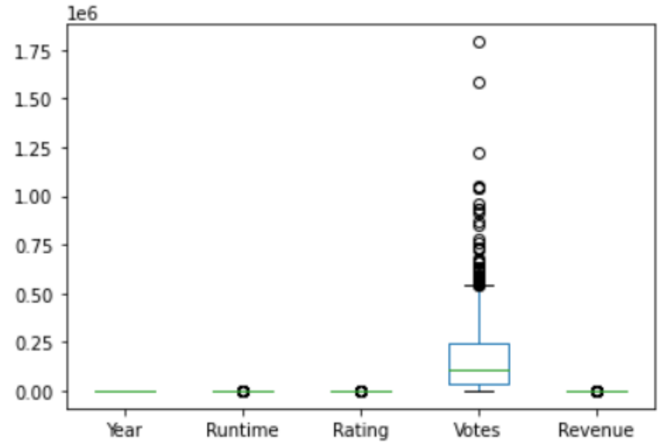
- Additional arguments can be passed to the *plot* method, such as *title, grid, legend, etc.*

```
plt.figure(figsize=(15,5))
plt.subplot(1,3,1)
dat['Revenue'].plot(kind = "hist", title = "Histogram of Revenue")
plt.subplot(1,3,2)
dat['Revenue'].plot(kind = "kde", legend = True)
plt.subplot(1,3,3)
dat['Revenue'].plot(kind = "box", grid = True)
plt.tight_layout()
plt.show()
```



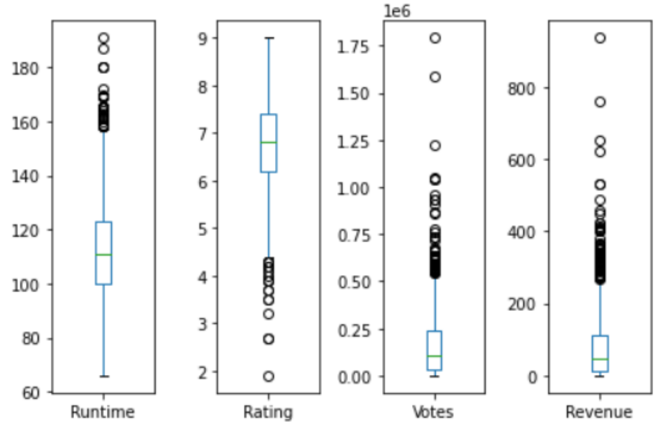
- We can even have multiple boxplots

```
dat.plot(kind = "box")
plt.show()
```



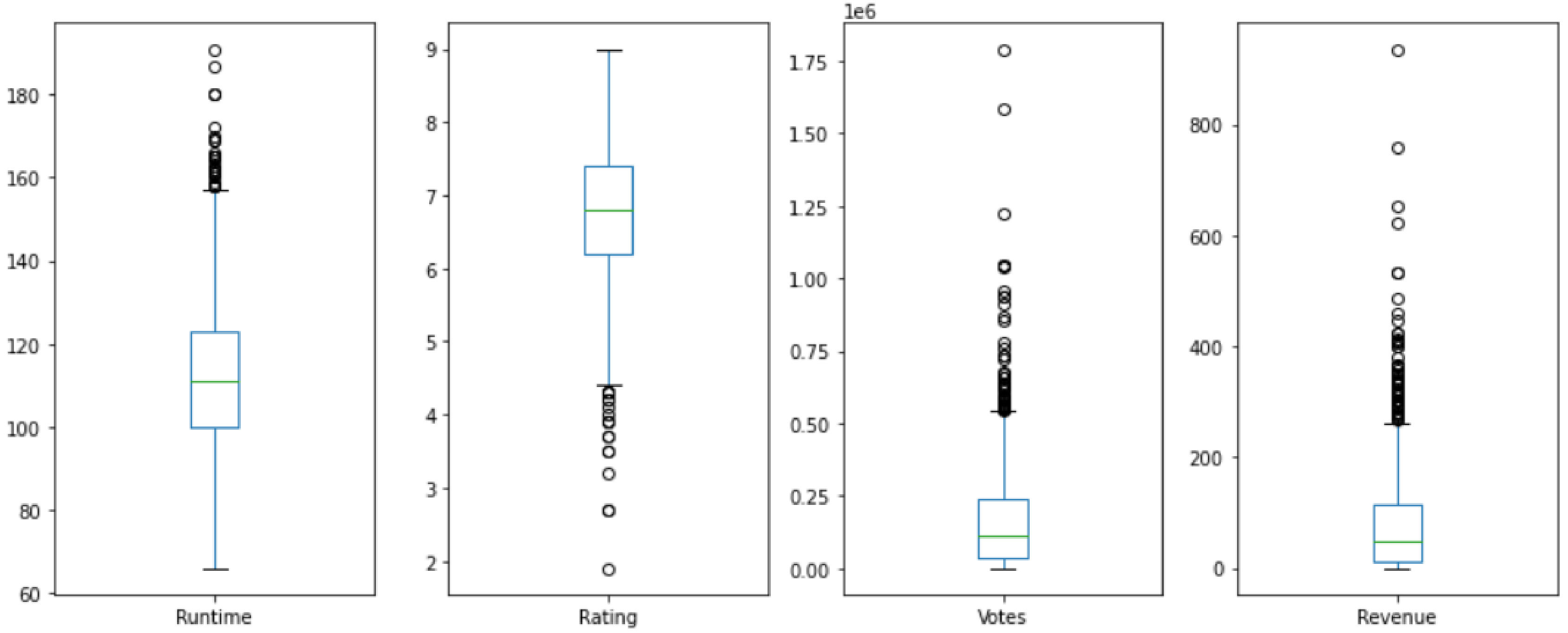
- But once again, *pandas* includes the *Year* variable, and the scales of the variables are completely different. So first we will exclude the *Year* variable with *select_dtypes* and then set *subplots = True* inside the *plot* method.

```
dat.select_dtypes(exclude='object').plot(kind = "box",
                                          subplots = True)
plt.tight_layout()
plt.show()
```



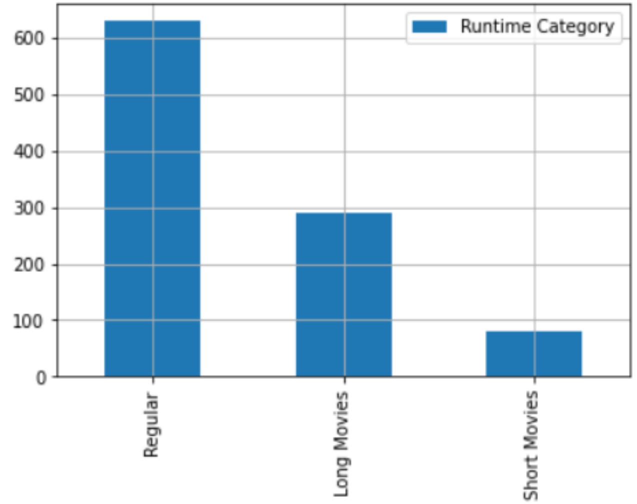
- Another improvement will be changing the figure size, using the *figsize* argument.

```python
dat.select_dtypes(exclude='object').plot(kind = "box",
                                         subplots = True,
                                         figsize = (12, 5))
plt.tight_layout()
plt.show()
```

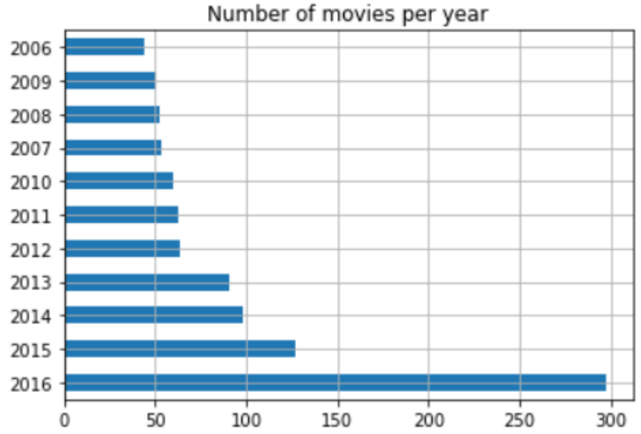


- Barplots are available as well!

```
dat['Runtime Category'].value_counts().plot(kind = "bar", grid = True)
plt.show()
```

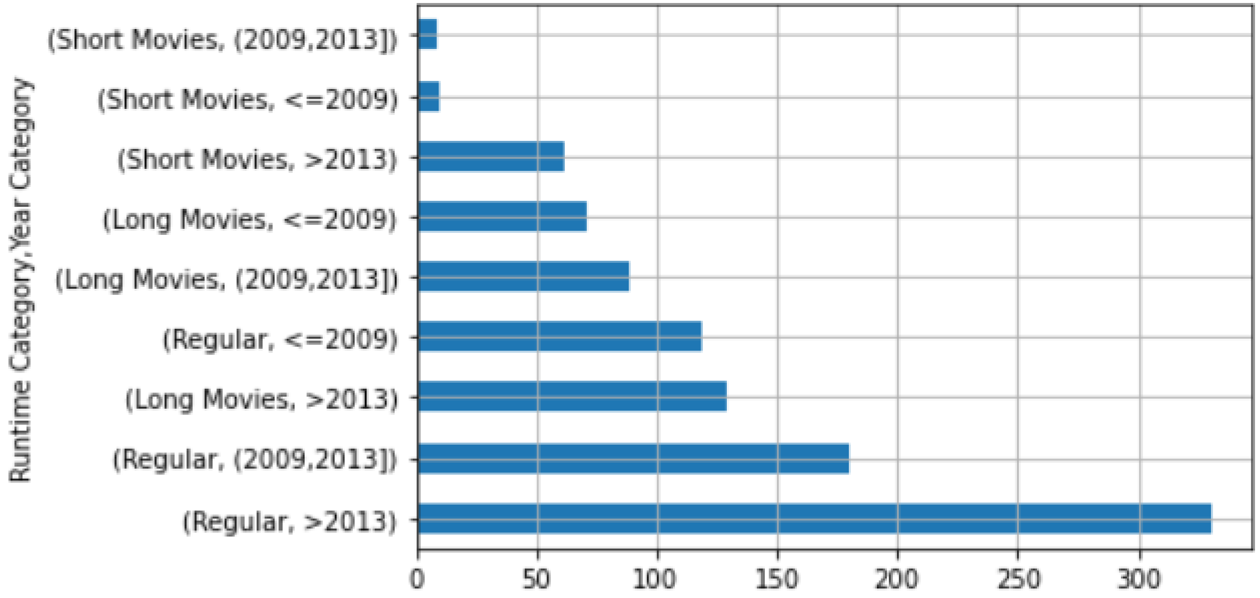



```
dat['Year'].value_counts().plot(kind = 'barh', grid = True,
                                title = 'Number of movies per year')
plt.show()
```



- And even more complicated aggregations.

```
dat[['Runtime Category', 'Year Category']].value_counts().plot(kind = "barh", grid = True)
plt.show()
```

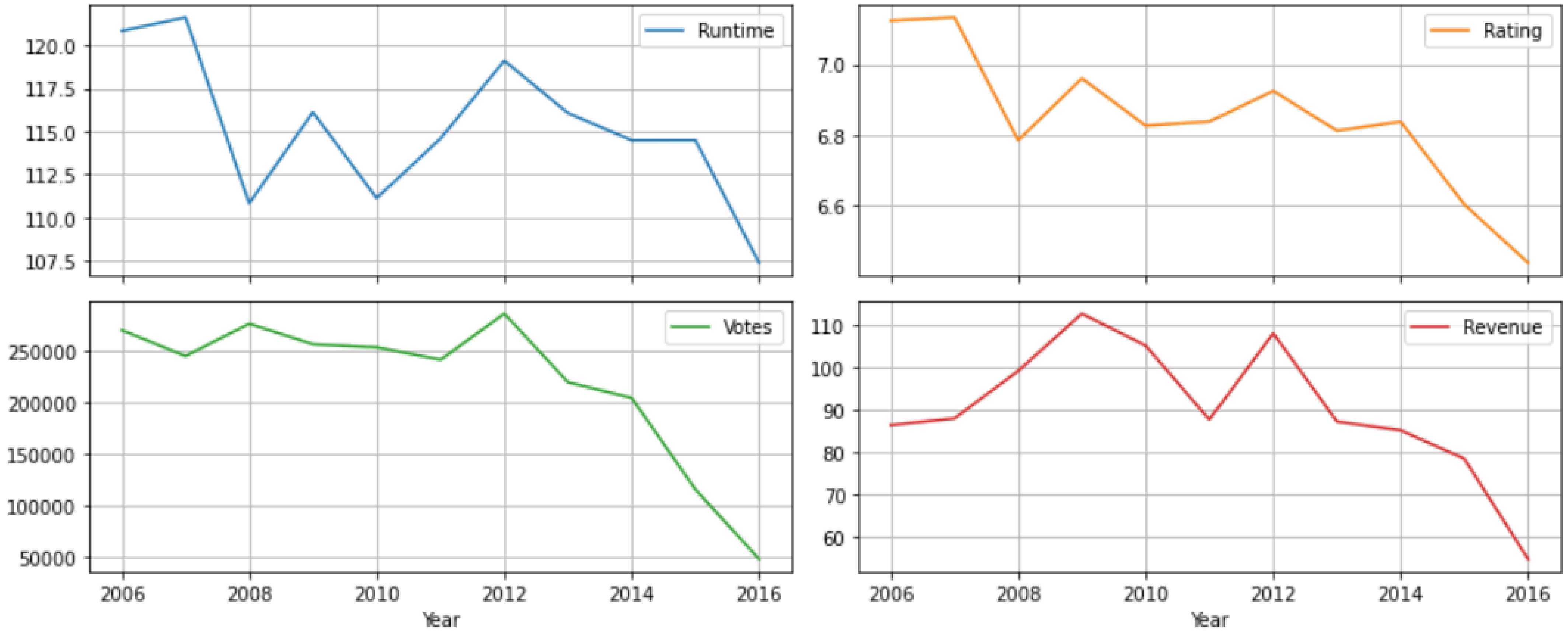


- The *plot* method is even more powerful when combined with *groupby*.
- For example, take the following grouped data frame and observe that the data frame indices are the years.
- Combined with *plot* we will get the following result (next slide)

```
1 dat.groupby(dat['Year']).mean()
```

Year	Runtime	Rating	Votes	Revenue
2006	120.840909	7.125000	269289.954545	86.296667
2007	121.622642	7.133962	244331.037736	87.882245
2008	110.826923	6.784615	275505.384615	99.082745
2009	116.117647	6.960784	255780.647059	112.601277
2010	111.133333	6.826667	252782.316667	105.081579
2011	114.603175	6.838095	240790.301587	87.612258
2012	119.109375	6.925000	285226.093750	107.973281
2013	116.065934	6.812088	219049.648352	87.121818
2014	114.489796	6.837755	203930.224490	85.078723
2015	114.496063	6.602362	115726.220472	78.355044
2016	107.373737	6.436700	48591.754209	54.690976

```python
dat.groupby(dat['Year']).mean().plot(subplots = True,
                                     layout = (2,2),
                                     figsize = (12,5),
                                     grid = True)
plt.tight_layout()
plt.show()
```



- Just a little bit of fine-tuning yields:

```
dat.groupby(dat['Year']).mean().plot(subplots = True,
                                     layout = (2,2),
                                     figsize = (12,5),
                                     grid = True,
                                     rot = 45,
                                     xticks = sorted(dat.Year.unique()))
plt.tight_layout()
plt.show()
```