Errata-Corrige from Lecture1:

To install packages in anaconda on ECB machines, on the anaconda prompt run

```
conda config --set ssl_verify false
```

then you can run `conda install your-package-name`

Solutions to the assignments:

1.

In [1]:
```
def Pascal(n):
    row = [1]
    T=[row]
    for _ in range(n):
        row=[l+r for l,r in zip(row+[0], [0]+row)]
        T.append(row)
    return T
Pascal(6)
```

Out[1]:
```
[[1],
 [1, 1],
 [1, 2, 1],
 [1, 3, 3, 1],
 [1, 4, 6, 4, 1],
 [1, 5, 10, 10, 5, 1],
 [1, 6, 15, 20, 15, 6, 1]]
```

In [2]:
```
def bin_exp(x,y,n):
    return sum([x**(n-k) * y**k *coeff for k,coeff in zip(range(n+1),Pascal(n)[-1
])])
def verify_binomial_theorem(x,y,n):
    return bin_exp(x,y,n) == (x+y)**n
verify_binomial_theorem(253,28,52)
```

Out[2]:
```
True
```

2.

In [3]:

```
#sum lines form the bottom to the top and maximise sums
def solution(A):
    A=[[int(d) for d in str(n)] for n in A] #list becomes list of lists
    while len(A) > 1:
        e1=A[-1]#last level
        e2=A[-2]#penultimate level
        s1=[e1[n]  +e2[n] for n in range(len(e2))]   #sum below
        s2=[e1[n+1]+e2[n] for n in range(len(e2))] #sum below right
        MS=[max(a,b) for a,b in zip(s1,s2)]        #max of possible sums
        A[-2]=MS #Replace penultimate line with MS
        A.pop()  #Remove last line
    return A[0][0]
```

In [4]:

```
####Generate long triangle
def gen_T(L):
    from random import randint, seed
    seed(100)
    T=[];
    for n in range(L):
        T.append(randint(10**(n) ,10**(n+1)-1))
    return T
```

In [5]:

```
solution([7,38,810,2744,45265])
```

Out[5]:

```
30
```

In [6]:

```
solution(gen_T(50))
```

Out[6]:

```
333
```

In [7]:

```python
#redefine solution to keep track of maxima
def solution(A):
    A=[[int(d) for d in str(n)] for n in A]
    global M
    M=[]
    for l in range(len(A)-1):
        e1=A[-1]#last level
        e2=A[-2]#penultimate level
        s1=[e1[n]+e2[n] for n in range(len(e2))]   #sum below
        s2=[e1[n+1]+e2[n] for n in range(len(e2))] #sum below right
        MS=[max(a,b) for a,b in zip(s1,s2)]        #max of possible sums
        A[-2]=MS #Replace penultimate line with MS
        A.pop()  #Remove last line
        M.append(MS)
    return A[0][0]
def find_path(A):
        global M
        S=solution(A)
        A=[[int(d) for d in str(n)] for n in A]
        ch_el=[M[-1][0]];path=[1]
        for n in range(2,len(M)+1):
            if M[-n][path[-1]-1]>M[-n][path[-1]]:
                path.append(path[-1])
            else:
                path.append(path[-1]+1)
        #add last element of path
        if A[-1][path[-1]-1]>A[-1][path[-1]]:
            path.append(path[-1])
        else:
            path.append(path[-1]+1)
        return path
#function to print triangle out of vector and path
def print_sol_tree(A, start = 0):
    S=solution(A)
    path=find_path(A)
    sm=0;
```

```
    for n in range(len(A)):
        D=[int(d) for d in str(A[n])];
        for d in range(len(D)):
            if d+1==path[n]:
                if n>=start:
                    print("\x1b[31m"+str(D[d])+"\x1b[0m",end="", flush=True)
                sm+=D[d]
            elif n>=start:
                print(D[d],end="", flush=True)
        if n>=start:
            print()
    if n>=start:
        print(" "*path[-2]+"\x1b[1;31m"+str(S)+"\x1b[0m")
    if sm==S and n>=start:
        print(sm==S)
    else:
        print(sm==S)
        print("Error!")
    return sm==S
cond=print_sol_tree(gen_T(50),35)
```

```
418744425182106254762324765268583738
8517027184494201168094853201776174995
19905495404807847318174851463000018942
211190092620649635838104821711183330845
8726513375633751610457942716768734238283
89399357577698226976873855548857615377619
368644597566770658191227891180199031930058
7641876277040658233068710319771596314829353
37027850249890306589849593995595782274793195
780384603136050422247697990613367538901646873
6083671798297876918069677488887949895817468912
89731873933709622126568196609975917250560892768
304579873520784682720034826644804968407591091808
1367712352974592097373652119073990828810352961386
59744921223361202330047087747395949536721275921715
                                333
True
```

A Python Lecture Series

Lecture 2

by Luca Mingarelli

Lecture 2

Content:

- I/O
- Modules
- NumPy and SciPy
- Pandas
- Matplotlib
- Importing data from the web

Input/Output

To write in a file:

In [8]:
```
f = open('a_work_file', 'w') # opens the file workfile
#more specifically it creates a file object
f.write('This is a test\n')
for n in range(5):
    f.write(str(n)+'\n')
f.close()
```

In [9]:
```
!ls #notice a new file!
```

```
ECB Python Lectures - Lecture 2 slides.pdf
ECB Python Lectures - Lecture 2.ipynb
ECB Python Lectures - Lecture 2.slides.html
a_work_file
img
res
```

To read from a file:

In [10]:

```
f = open('a_work_file', 'r')
s = f.read()
print(s)
f.close()
```

```
This is a test
0
1
2
3
4
```

Iterating over a file

For reading lines from a file, you can loop over the file object. This is memory efficient, fast, and leads to simple code.

In [11]:
```
f = open('a_work_file', 'r')
for line in f:
    print(line,end = '')
f.close()
```

```
This is a test
0
1
2
3
4
```

File modes

- Read-only: `r`
- Write-only: `w`
    - Note: Create a new file or overwrite existing file.
- Append to a file: `a`
- Read and Write: `r+`
- Binary mode: `b`

It is good practice to use the `with` keyword when dealing with file objects. The advantage is that the file is properly closed after its suite finishes, even if an exception is raised at some point (using `with` is also much shorter than writing equivalent try-finally blocks).

In [12]:
```
with open('A_new_test','w') as f:
    f.write('This is a NEW test\n\n')
    for n in range(6):
        f.write(f'{n} squared is {n**2}\n') # A formatted string - notice the 'f'
 at the beginning of the string
```

In [13]:
```
with open('A_new_test','r') as f:
    print(f.read())
```

```
This is a NEW test

0 squared is 0
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
5 squared is 25
```

Modules

i.e. how to write reusable code

A module is a file containing Python definitions and statements. The file name is the module name with the suffix `.py` appended.

In [14]:
```
%%writefile my_new_module.py

def a_complicated_function():
    print("Working... Done.")
```

```
Writing my_new_module.py
```

In [15]:
```
# import my_new_module as mnm
from my_new_module import a_complicated_function
a_complicated_function()
```

```
Working... Done.
```

In [16]:
```
!mkdir MODULES
```

In [17]:
```
%%writefile MODULES/module2.py
def function2():
    print("Working... Done.")
```

```
Writing MODULES/module2.py
```

We could now call this as `MODULES.module2.function2` . However, to make our life easier we can instead write the following `__init__.py` file (notice the `.` !):

In [18]:
```
%%writefile MODULES/__init__.py
from .module2 import function2
```

```
Writing MODULES/__init__.py
```

In [19]:
```
import MODULES as MD
MD.function2()
```

```
Working... Done.
```

Most of the useful operations needed for scientific computing are contained within some module (e.g. **NumPy**, **SciPy**, etc.).

This means that in order to access them we will need to import that module as

- `import module`,

or giving it an alias as

- `import module as md`.

Then we will be able to call the function as `module.specific_function()` or as `md.specific_function()`. Alternatively we can import the required tool/function as

- `from module import specific_funtion`.

NumPy and its arrays

NumPy provides an efficient extension package to Python for multidimensional arrays.

In [20]:
```
import numpy as np
x = np.array([1,2,3])
# convert list to numpy array object
x
```

Out[20]:
```
array([1, 2, 3])
```

In [21]:
```
###--- Notice that
x+x
###--- More on this later.
```

Out[21]:
```
array([2, 4, 6])
```

In [22]:
```
x = np.linspace(0,10,11) # as in Matlab!
x
```

Out[22]:
```
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.])
```

In [23]:
```
x = np.arange(1.5,10,2) # same as Matlab's [1.5:2:10]
x
```

Out[23]:
```
array([1.5, 3.5, 5.5, 7.5, 9.5])
```

NumPy's number types and associated risk (overflow)

In [24]:
```
x=np.array([0,1])
print("x =",x,"and has dtype", x.dtype)
```

```
x = [0 1] and has dtype int64
```

In [25]:
```
x=np.array([0,1], dtype = np.int8)
x[:] = 2**7-1
print("x =",x,"and has dtype",x.dtype)
```

```
x = [127 127] and has dtype int8
```

In [26]:
```
print("x + 1 =",x + 1,"\t (because dtype is"
      ,x.dtype,"!)")
```

```
x + 1 = [-128 -128]      (because dtype is int8 !)
```

In [27]:
```
x=np.array([2**63-1,2**63-1])
print("x[0] =",x[0],"and has dtype",x.dtype)
sum(x)
```

```
x[0] = 9223372036854775807 and has dtype int64

/anaconda/lib/python3.6/site-packages/ipykernel_launcher.py:3: RuntimeWarning:
overflow encountered in long_scalars
  This is separate from the ipykernel package so we can avoid doing imports un
til
```

Out[27]:
```
-2
```

Some of NumPy arrays' attributes and methods

In [28]:
```
x.ndim
```
Out[28]:
```
1
```

In [29]:
```
x.shape
```
Out[29]:
```
(2,)
```

In [30]:
```
len(x)
```
Out[30]:
```
2
```

In [31]:
```
x = np.array([[0, 1, 2], [3, 4, 5]])     # 2 x 3 array
```

In [32]:
```
x.shape
```
Out[32]:
```
(2, 3)
```

In [33]:
```
x.mean() ## an ojbject's method
```
Out[33]:
```
2.5
```

In [34]:
```
x.dtype ## data type
```
Out[34]:
```
dtype('int64')
```

In [35]:

```python
print("itemsize:", x.itemsize, "bytes")
print("nbytes:", x.nbytes, "bytes")
```

```
itemsize: 8 bytes
nbytes: 48 bytes
```

Higher dimensional arrays

In [36]:
```
np.zeros((2,3))
```

Out[36]:
```
array([[0., 0., 0.],
       [0., 0., 0.]])
```

In [37]:
```
np.ones((2,2))
```

Out[37]:
```
array([[1., 1.],
       [1., 1.]])
```

In [38]:
```
np.eye(3)
```

Out[38]:
```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

In [39]:
```
np.diag(range(1,5))
```

Out[39]:
```
array([[1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 0, 4]])
```

Indexing and Slicing

Recall: `x[start:stop:step]` ; when any is omitted the default values are `start=0` , `stop=size` , `step=1`

In [40]:
```
x
```

Out[40]:
```
array([[0, 1, 2],
       [3, 4, 5]])
```

In [41]:
```
print('x[0] = ', x[0])
print('x[1] = ', x[1])
```

```
x[0] =  [0 1 2]
x[1] =  [3 4 5]
```

In [42]:
```
x[0][-1]
```

Out[42]:
```
2
```

In [43]:
```
x[0,-1]
```

Out[43]:
```
2
```

In [44]:
```
x[:,::-1]
```

Out[44]:
```
array([[2, 1, 0],
       [5, 4, 3]])
```

In [45]:
```
x[::-1,:]
```

Out[45]:
```
array([[3, 4, 5],
       [0, 1, 2]])
```

In [46]:
```
## Notice the equivalence x[0] = x[0,:] for multidimensional arrays
print(x[0,:]) # first row of x
print(x[0])   # still first row of x
```

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

IMPORTANT 1: Be carefull about the datatype:

In [47]:
```
print('type: ',x.dtype)
x[:,:] = np.pi
x
```

```
type:  int64
```

Out[47]:
```
array([[3, 3, 3],
       [3, 3, 3]])
```

In [48]:
```
y = x.astype(bool)
# y = y.astype(float)
print('type: ',y.dtype)
y[:,:] = np.pi
y
```

```
type:  bool
```

Out[48]:
```
array([[ True,  True,  True],
       [ True,  True,  True]])
```

IMPORTANT 2: Slices return views, NOT copies!

In [49]:
```
x_slice = x[:2,:2]
x_slice
```

Out[49]:
```
array([[3, 3],
       [3, 3]])
```

In [50]:

```python
x_slice[:] = 2
x
```

Out[50]:

```
array([[2, 2, 3],
       [2, 2, 3]])
```

This behavior is quite useful: when working with large datasets, we can access and process pieces of these datasets without the need to copy the data.

Copying NumPy arrays

In [51]:
```
x[:] = 3
x_slice_copy = x[:2, :2].copy()
x_slice_copy
```

Out[51]:
```
array([[3, 3],
       [3, 3]])
```

In [52]:
```
x_slice_copy[:] = 2
x
```

Out[52]:
```
array([[3, 3, 3],
       [3, 3, 3]])
```

Reshaping

In [53]:
```
x = np.array([1, 2, 3])
# reshape to row vector
x.reshape((1, 3))
```

Out[53]:
```
array([[1, 2, 3]])
```

In [54]:
```
# reshape to row vector
x.reshape((3, 1))
```

Out[54]:
```
array([[1],
       [2],
       [3]])
```

Concatenation

In [55]:
```
x = np.array([1, 2, 3])
y = np.array([4, 5, 6])
Z = np.concatenate([x, y])
Z
```

Out[55]:
```
array([1, 2, 3, 4, 5, 6])
```

In [56]:
```
# for multidimensional arrays as well


np.concatenate([Z, Z])
```

Out[56]:
```
array([1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6])
```

Although `np.vstack` and `np.hstack` might be clearer:

In [57]:
```
x = np.array([1, 2, 3])
Z = np.array([[4, 5, 6],
              [7, 8, 9]])
# vertically stack the arrays
np.vstack([x, Z])
```

Out[57]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

In [58]:
```
# horizontally stack the arrays
y = np.array([[456],
              [789]])
np.hstack([Z, y])
```

Out[58]:
```
array([[  4,   5,   6, 456],
       [  7,   8,   9, 789]])
```

Use `np.dstack` to stack arrays along higher dimensional axis.

Splitting

In [59]:
```
Z1 = np.vstack([x, Z]).reshape((9,))
print(Z1)
x, y, z = np.split(Z1,[3,5])
print(x,y,z)
```

```
[1 2 3 4 5 6 7 8 9]
[1 2 3] [4 5] [6 7 8 9]
```

In [60]:
```
Z = np.arange(16).reshape((4, 4))

print(Z)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
```

In [61]:
```
upper, lower = np.vsplit(Z, [2])
print('Upper part:\n',upper)
print('-'*15)
print('Lower part: \n',lower)
```

```
Upper part:
 [[0 1 2 3]
 [4 5 6 7]]
---------------
Lower part: 
 [[ 8  9 10 11]
 [12 13 14 15]]
```

In [62]:
```python
left, right = np.hsplit(Z, [2])
print('Left part:\n',left)
print('-'*15)
print('Right part:\n',right)
```

```
Left part:
 [[ 0  1]
 [ 4  5]
 [ 8  9]
 [12 13]]
---------------
Right part:
 [[ 2  3]
 [ 6  7]
 [10 11]
 [14 15]]
```

Operations on NumPy arrays

Whenever possible, avoid looping: it's slow!

Instead it is advisable to make use of **NumPy**'s built in functions. These are highly optimised and are applied elementwise.

In [63]:
```
x = np.arange(-5,5)
np.abs(x)
```

Out[63]:
```
array([5, 4, 3, 2, 1, 0, 1, 2, 3, 4])
```

In [64]:
```
x = np.linspace(1,2,1000)
%timeit [1/x[n] for n in range(len(x))]
%timeit 1/x
```

```
420 µs ± 9.48 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
5.1 µs ± 160 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

The most common functions (trigonometric, exponentials, logarithms, etc.) can be found within **NumPy**. More specialised functions on the other hand, can be found in **SciPy**, within the sub-module `scipy.special`:

In [65]:
```
from scipy import special
x = np.array([0.5, 1.])
print("Γ(x)=", special.gamma(x))
print("B(x,2)=", special.beta(x, 2))
print("erf(x)=", special.erf(x))
```

```
Γ(x)= [1.77245385 1.        ]
B(x,2)= [1.33333333 0.5       ]
erf(x)= [0.52049988 0.84270079]
```

More *special* mathematical functions can be found here (https://docs.scipy.org/doc/scipy/reference/special.html#module-scipy.special).

Even when computing aggregates: use NumPy's functions.

In [66]:
```
x = np.arange(1000)
%timeit sum(x)
%timeit x.sum()
%timeit np.sum(x) # the same as above!
```

```
133 µs ± 3.97 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
5.81 µs ± 551 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
7.81 µs ± 259 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

In [67]:
```
%timeit max(x)
%timeit x.max()
%timeit np.max(x) # the same as above!
```

```
93.4 µs ± 1.04 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
6.95 µs ± 249 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
8.4 µs ± 289 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

Same for `max` and `min`.

These operations can also be done along one axis only:

In [68]:
```python
print(Z)
print("\nSum columns:")
Z.sum(axis = 1)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]

Sum columns:
```

Out[68]:
```
array([ 6, 22, 38, 54])
```

In [69]:
```python
print("Max along columns:")
print(Z.max(axis = 0 ))
print("\nMax along rows:")
print(Z.max(axis = 1 ))
```

```
Max along columns:
[12 13 14 15]

Max along rows:
[ 3  7 11 15]
```

A summary of available aggregation functions

Function Name	NaN-safe Version	Description
`np.sum`	`np.nansum`	Compute sum of elements
`np.prod`	`np.nanprod`	Compute product of elements
`np.mean`	`np.nanmean`	Compute mean of elements
`np.std`	`np.nanstd`	Compute standard deviation
`np.var`	`np.nanvar`	Compute variance
`np.min`	`np.nanmin`	Find minimum value
`np.max`	`np.nanmax`	Find maximum value
`np.argmin`	`np.nanargmin`	Find index of minimum value
`np.argmax`	`np.nanargmax`	Find index of maximum value
`np.median`	`np.nanmedian`	Compute median of elements
`np.percentile`	`np.nanpercentile`	Compute rank-based statistics of elements
`np.any`	N/A	Evaluate whether any elements are true
`np.all`	N/A	Evaluate whether all elements are true

Boolean operations

In [70]:
```
x = np.array([1,2,3,4,5,6])
x>3
```

Out[70]:
```
array([False, False, False,  True,  True,  True])
```

Operator	Equivalent function
`==`	`np.equal`
`<`	`np.less`
`>`	`np.greater`
`!=`	`np.not_equal`
`<=`	`np.less_equal`
`>=`	`np.greater_equal`

Masks

A boolean array can be used to index which element to extract from a second array:

In [71]:

```
print(x)
print(x>3)
x[x>3]
```

```
[1 2 3 4 5 6]
[False False False  True  True  True]
```

Out[71]:

```
array([4, 5, 6])
```

Fancy indexing

In [72]:
```
l = np.array([1,2,3,4,5])
l[[0,2]]
```

Out[72]:
```
array([1, 3])
```

In [73]:
```
l[[-1,0,-2]]
```

Out[73]:
```
array([5, 1, 4])
```

Moreover:

In [74]:
```
ind = np.array([[3, 0],
                [4, 1]])
l[ind]
```

Out[74]:
```
array([[4, 1],
       [5, 2]])
```

When using fancy indexing, the output has the same shape as the index.

Broadcasting

Broadcasting is a feature allowing for binary operations to be performed on arrays with different shapes.

In [75]:
```
x = np.array([0,1,2])
print(x+3)
print(x+np.array([3,3,3]))
```

```
[3 4 5]
[3 4 5]
```

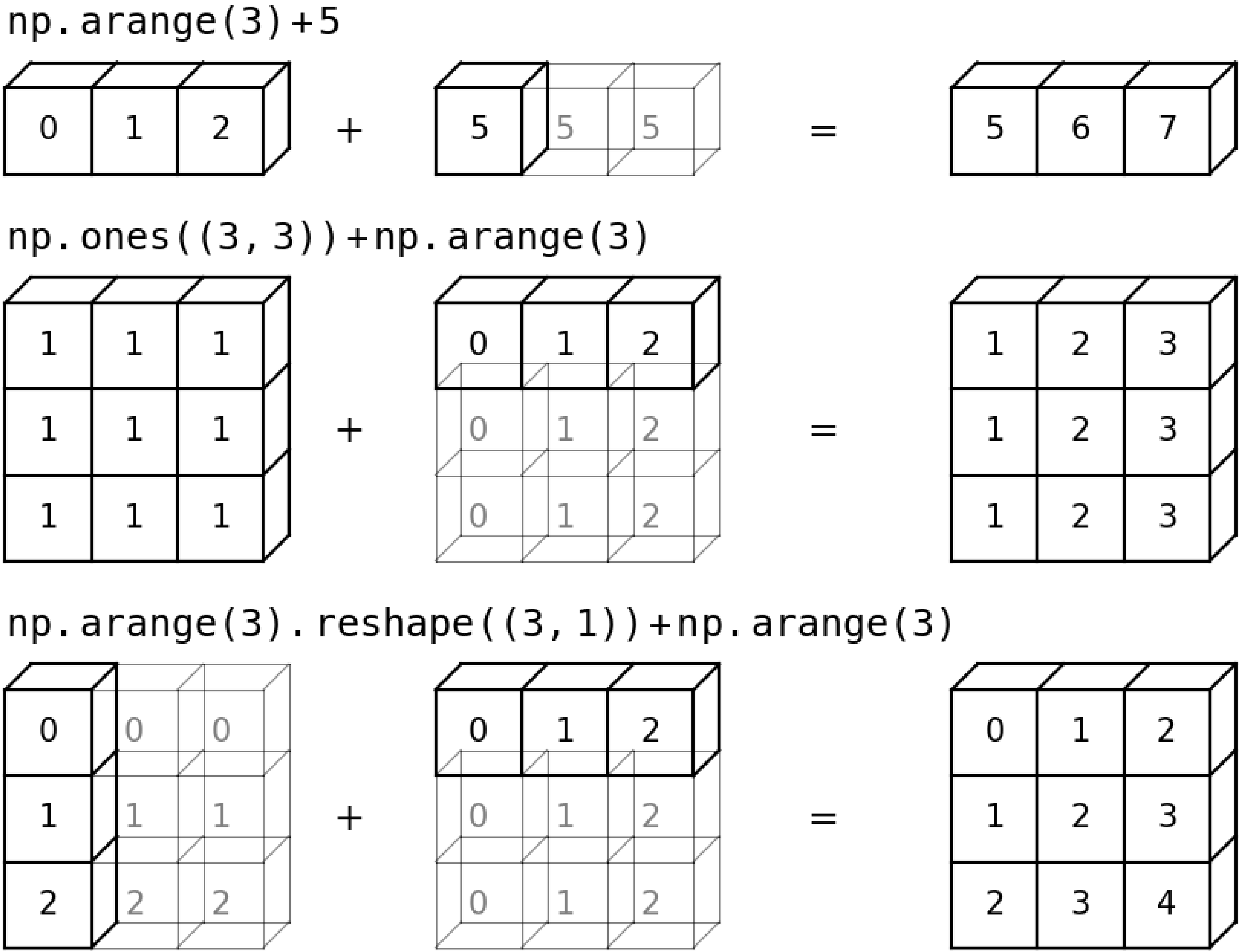
In [76]:
```
M = np.ones((3, 3))
M+x
```

Out[76]:
```
array([[1., 2., 3.],
       [1., 2., 3.],
       [1., 2., 3.]])
```

In [77]:
```
y = x.reshape((3,1))
print('y=\n',y)
print('-'*10)
print('x+y=\n',x+y)
```

```
y=
 [[0]
 [1]
 [2]]
----------
x+y=
 [[0 1 2]
 [1 2 3]
 [2 3 4]]
```

Rules of Broadcasting:



Copy NumPy arrays (Deep-copy)

In [79]:
```
A = np.arange(10)
B = A
B[0]= 100
A
```

Out[79]:
```
array([100,   1,   2,   3,   4,   5,   6,   7,   8,   9])
```

In [80]:
```
A = np.arange(10)
B = A.copy()
B[0]=100
A
```

Out[80]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Pandas

Main data structures in Pandas:

- Series
- DataFrames

Series

In [81]:
```
import pandas as pd
pd.Series([0.1, 0.2, 0.3, 0.4, 0.5])
```

Out[81]:
```
0    0.1
1    0.2
2    0.3
3    0.4
4    0.5
dtype: float64
```

In [82]:
```
s = pd.Series([0.1, 0.2, 0.3, 0.4, 0.5],
       index = ['a','b','c','d','e'])
s['a'] #s[0] s['a'] s[s>2] s[:2] s[[3,4]]
##i.e. can be treated as nparrays
```

Out[82]:
```
0.1
```

Be careful however about operations between different Series

In [83]:
```
s1 = pd.Series({'a': 0.1, 'b': 1.2, 'c': 2.3})
s2 = pd.Series({'a': 1.0, 'b': 2.0, 'c': 3.0})
s3 = pd.Series({'c': 0.1, 'd': 1.2, 'e': 2.3})
```

In [84]:
```
s1 + s2
```

Out[84]:
```
a    1.1
b    3.2
c    5.3
dtype: float64
```

In [85]:
```
s1 + s3
```

Out[85]:
```
a    NaN
b    NaN
c    2.4
d    NaN
e    NaN
dtype: float64
```

In [86]:
```
s1 = pd.Series([1,2,3],index=['a'] * 3)
s2 = pd.Series([4,5],index=['a'] * 2)
s1 + s2 #for non-unique indices: broadcasting to all common indices.
```

Out[86]:
```
a    5
a    6
a    6
a    7
a    7
a    8
dtype: int64
```

It is possible to access the underlying arrays through the attributes **`values`** and **`index`**

In [87]:
```
print(type(s3.values))
s3.values
```

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

Out[87]:
```
array([0.1, 1.2, 2.3])
```

In [88]:
```
s3.index = ['First', 'Second', 'Third']
print(s3)
s3.index[1]
```

```
First     0.1
Second    1.2
Third     2.3
dtype: float64
```

Out[88]:
```
'Second'
```

In [89]:
```
s = pd.Series([10,20,30],
              index=[13,2,89])
## Now indexing is ambiguous!
s[2]
# s[0]  # Error
```

Out[89]:
```
20
```

In [90]:
```
s.iloc[0:2] ## s.iloc[0:2] ##i.e. slicing works
```

Out[90]:
```
13    10
2     20
dtype: int64
```

In [91]:
```
s.loc[89] # s.loc[[13,89]]
##i.e. fancy indexing works
```

Out[91]:
```
30
```

Notable Methods of the `Series` data structure

Accessed as `my_series.method()`

Name	Description
`head()` and `tail()`	Display the first five and the last five rows respectively (first/last *n* rows if *n* is given as an argument)
`isnull()`	Returns a Series with same indices and boolean values indicating where the values are `NaNs` or `Nulls`
`notnull()`	Negation of `isnull()`
`iloc()`	Access integer location of a Series
`loc()`	Access location according to indexing of the Series
`describe()`	Returns summary and statistics of the Series
`unique()`	Returns the unique elements of a Series
`drop(index)`	Drops elements with the selected index
`dropna()`	Drops all `NaNs` and `Nulls` elements
`fillna(value)`	Fills all `NaNs` and `Nulls` with `value`
`append(series)`	Appends a Series to another Series

DataFrame

Dataframes are a collection of `Series`.

In [92]:

```
df = pd.DataFrame(np.array([[1,2],[3,4]]))
df
```

Out[92]:

	0	1
0	1	2
1	3	4

In [93]:

```
df.columns = ['col1','col2']
df.index = ['row1','row2']
df
```

Out[93]:

	col1	col2
row1	1	2
row2	3	4

In [94]:

```
pd.DataFrame(np.array([[1,2],[3,4]]),columns=['col1','col2'], index = ['row1','row
2'])
```

Out[94]:

	col1	col2
row1	1	2
row2	3	4

In [211]:

```python
s1 = pd.Series(np.arange(0,5))
s2 = pd.Series(np.arange(1,4))
s3 = pd.Series(np.arange(2,3))
pd.DataFrame({'col1': s1, 'col2': s2, 'col3': s3})
```

Out[211]:

	col1	col2	col3
0	0	1.0	2.0
1	1	2.0	NaN
2	2	3.0	NaN
3	3	NaN	NaN
4	4	NaN	NaN

In [212]:
```
df = pd.DataFrame({'col'+str(1+ i):pd.Series(np.arange(i,5.0-i)) for i in range(3
)})#np.random.randint(0,3,3)
```

In [213]:
```
df.describe()
```

Out[213]:

	col1	col2	col3
count	5.000000	3.0	1.0
mean	2.000000	2.0	2.0
std	1.581139	1.0	NaN
min	0.000000	1.0	2.0
25%	1.000000	1.5	2.0
50%	2.000000	2.0	2.0
75%	3.000000	2.5	2.0
max	4.000000	3.0	2.0

In [214]:
```
df.sum() ### NaN automatically diregarded!
```

Out[214]:
```
col1    10.0
col2     6.0
col3     2.0
dtype: float64
```

Selecting columns ...

In [215]:
```
print(df['col1'])
print(type(df['col1']))
```

```
0    0.0
1    1.0
2    2.0
3    3.0
4    4.0
Name: col1, dtype: float64
<class 'pandas.core.series.Series'>
```

In [216]:
```
print(df[['col1','col3']])
print(type(df[['col1','col3']]))
```

```
   col1  col3
0   0.0   2.0
1   1.0   NaN
2   2.0   NaN
3   3.0   NaN
4   4.0   NaN
<class 'pandas.core.frame.DataFrame'>
```

... selecting rows...

In [217]:
```
df[2:4]
```

Out[217]:

	col1	col2	col3
2	2.0	3.0	NaN
3	3.0	NaN	NaN

...and of course: selecting rows and columns...

In [218]:
```
df[2:4][['col2']]
```

Out[218]:

	col2
2	3.0
3	NaN

…deleting columns…

In [219]:
```
df2 = df.copy() #Recall the `issue` in numpy?
del df2['col2']
df2
```

Out[219]:

	col1	col3
0	0.0	2.0
1	1.0	NaN
2	2.0	NaN
3	3.0	NaN
4	4.0	NaN

In [220]:
```
df2.pop('col1')
```

Out[220]:
```
0    0.0
1    1.0
2    2.0
3    3.0
4    4.0
Name: col1, dtype: float64
```

In [221]:
```
df2
```

Out[221]:

	col3
0	2.0
1	NaN
2	NaN
3	NaN
4	NaN

In [222]:
```
df2 = df.drop(['col1','col3'],axis = 1)
df2
```

Out[222]:

	col2
0	1.0
1	2.0
2	3.0
3	NaN
4	NaN

In [223]:
```
df
```

Out[223]:

	col1	col2	col3
0	0.0	1.0	2.0
1	1.0	2.0	NaN
2	2.0	3.0	NaN
3	3.0	NaN	NaN
4	4.0	NaN	NaN

Data import with Pandas

CSV files (pandas.read_csv)

Comma-separated value files can be easily read using `pandas.read_csv`:

```
csv_data = pd.read_csv('file.csv')
```

Exel files (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_excel.html)

```
csv_data = pd.read_excel('file.xlsx')
```

`pandas.read_excel` requires two arguments: the name of the file and the name of the sheet.

Moreover, more optional arguments can be parsed to these functions to specify where to start reading from, how many rows to read, etc.

Additionally, `pd.read_stata`, `pd.read_sql`, `pd.read_json`, and more (https://pandas.pydata.org/pandas-docs/stable/reference/io.html)

What to do with missing data?

- `None` Missing data inside of dataframe of type `object`
- `NaN` Missing numerical data

In [95]:
```
# None + 1
np.nan +1
```

Out[95]:
```
nan
```

In [226]:
```
pd.Series([1, np.nan, 2, None])
## Notice both the mapping None -> NaN
## as well as int -> float
```

Out[226]:
```
0    1.0
1    NaN
2    2.0
3    NaN
dtype: float64
```

Detection of missing data

In [227]:
```
df.count() #count non-missing elements
```

Out[227]:
```
col1     5
col2     3
col3     1
dtype: int64
```

In [228]:
```
df.notnull() # opposite: df.isnull()
```

Out[228]:

	col1	col2	col3
0	True	True	True
1	True	True	False
2	True	True	False
3	True	False	False
4	True	False	False

In [229]:
```
df['col2'][df['col2'].notnull()]
```

Out[229]:
```
0     1.0
1     2.0
2     3.0
Name: col2, dtype: float64
```

Dropping missing values

In [230]:

```
df.dropna()
## drops all rows
## with at least one missing value
```

Out[230]:

	col1	col2	col3
0	0.0	1.0	2.0

In [231]:

```
df.dropna(axis='columns')
```

Out[231]:

	col1
0	0.0
1	1.0
2	2.0
3	3.0
4	4.0

Filling missing values

In [235]:

```
df.fillna(0)
df
```

Out[235]:

	col1	col2	col3
0	0.0	1.0	2.0
1	1.0	2.0	NaN
2	2.0	3.0	NaN
3	3.0	NaN	NaN
4	4.0	NaN	NaN

In [233]:

```
# forward-fill
df.fillna(method='ffill') #bfill for back-fill
```

Out[233]:

	col1	col2	col3
0	0.0	1.0	2.0
1	1.0	2.0	2.0
2	2.0	3.0	2.0
3	3.0	3.0	2.0
4	4.0	3.0	2.0

In [234]:

```
# change axis
df.fillna(method='ffill',axis = 1)
```

Out[234]:

	col1	col2	col3
0	0.0	1.0	2.0
1	1.0	2.0	2.0
2	2.0	3.0	3.0
3	3.0	3.0	3.0
4	4.0	4.0	4.0

In [236]:

```
df = pd.DataFrame({"A":[12, 4, 5, None, 1],
                   "B":[None, 2, 54, 3, None],
                   "C":[20, 16, None, 3, 8],
                   "D":[14, 3, None, None, 6]})
df
```

Out[236]:

	A	B	C	D
0	12.0	NaN	20.0	14.0
1	4.0	2.0	16.0	3.0
2	5.0	54.0	NaN	NaN
3	NaN	3.0	3.0	NaN
4	1.0	NaN	8.0	6.0

In [237]:

```
# to interpolate the missing values
df. (method ='linear', limit_direction ='forward',axis = 1)
```

Out[237]:

	A	B	C	D
0	12.0	16.0	20.0	14.0
1	4.0	2.0	16.0	3.0
2	5.0	54.0	54.0	54.0
3	NaN	3.0	3.0	3.0
4	1.0	4.5	8.0	6.0

Alternatively:

- `linear`: Ignore the index and treat the values as equally spaced.
- `time`: Works on daily and higher resolution data to interpolate given length of interval.
- `index`, `values`: use the actual numerical values of the index.
- `pad`: Fill in `NaN`s using existing values.
- `nearest`, `zero`, `slinear`, `quadratic`, `cubic`, `spline`, `barycentric`, `polynomial`
- `krogh`, `piecewise_polynomial`, `spline`, `pchip`, `akima`

[More here (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.interpolate.html)](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.interpolate.html).

Probability and Statistics

Random generators

In [239]:
```
import random as rnd
rnd.random() ## Uniform in [0,1)
```

Out[239]:
```
0.2707940270988064
```

In [240]:
```
# uniform in range
rnd.uniform(1,10)
```

Out[240]:
```
6.628778659683503
```

In [241]:
```
#simulate die
rnd.randint(1,6)
```

Out[241]:
```
6
```

In [245]:
```
greetings = ['Hi', 'Hello', 'Welcome!', 'Hola']
rnd.choice(greetings)
```

Out[245]:
```
'Welcome!'
```

In [246]:
```
#Simulate wheel spins
colors = ['R', 'B', 'G'] # Red, Black and Green
rnd.choices(colors, weights=[18,18,2] ,k =10)
```

Out[246]:
```
['R', 'R', 'B', 'R', 'B', 'R', 'R', 'G', 'R', 'B']
```

In [247]:
```
# Shuffle cards
deck = list(range(1,53)) ## 52 cards
rnd.shuffle(deck)
print(deck)
```

```
[13, 32, 34, 15, 14, 24, 5, 46, 48, 18, 28, 17, 3, 44, 38, 26, 20, 1, 51, 33,
12, 8, 40, 29, 22, 4, 35, 30, 16, 52, 21, 42, 25, 23, 11, 39, 43, 9, 49, 50, 3
6, 47, 41, 19, 6, 45, 7, 31, 10, 37, 2, 27]
```

In [248]:
```
#Sample a hand from the deck
hand = rnd.sample(deck,k=5)
print(hand)## only unique values
```

```
[32, 49, 23, 37, 5]
```

NumPy random generators

In [249]:
```
import numpy.random as rnd
```

In [250]:
```
## UNIFORM
print(rnd.rand(3,4))
```

```
[[0.51846552 0.66827394 0.82240748 0.60246942]
 [0.52696974 0.90829626 0.37089569 0.84264402]
 [0.12407456 0.39154479 0.2680074  0.78966815]]
```

In [251]:
```
## STANDARD NORMAL
print(rnd.randn(3,4))
```

```
[[ 0.53121684  1.8907553   1.9218723   0.50584662]
 [-0.52137642  1.01327556  0.59665253  2.05572702]
 [-0.61345248  0.12035755 -0.91156546 -0.14825564]]
```

In [252]:
```
## UNIFORM INTEGERS
print(rnd.randint(0,100,(3,4)))
```

```
[[35 32 34 55]
 [83 84 55 98]
 [97 46 71 87]]
```

In [253]:
```
rnd.shuffle(deck)
print(deck)
```

```
[12, 8, 34, 11, 6, 21, 3, 19, 38, 25, 43, 31, 40, 52, 36, 24, 51, 10, 18, 49,
15, 1, 50, 39, 42, 47, 45, 23, 17, 33, 44, 48, 30, 13, 20, 7, 46, 32, 27, 28,
5, 16, 22, 2, 14, 35, 37, 9, 26, 4, 29, 41]
```

Function	Description
`uniform(a,b,k)`	Returns k draws from $U(a, b)$.
`normal(μ,σ,k)`	Returns k draws from $\mathcal{N}(\mu, \sigma)$.
`multivariate_normal(μ,Σ,k)`	Returns k draws from $\mathcal{N}(\vec{\mu}, \Sigma)$.
`lognormal(μ,σ,k)`	Returns k draws from $\mathrm{LogNormal}(\mu, \sigma)$.
`standard_t(ν,k)`	Returns k draws from $\mathrm{Student\text{-}t}(\nu)$.
`chisquare(nu,k)`	Returns k draws from χ^2_ν.
`poisson(λ,k)`	Returns k draws from $\mathrm{Poisson}(\lambda)$.
`binomial(n,p,k)`	Returns k draws from $B(n, p)$.
`binomial(1,p,k)`	Returns k draws from $\mathrm{Bernoulli}(p)$.
`multinomial(n,p,k)`	Returns k draws from $\mathrm{Multinomial}(n, \vec{p})$ (`n` trials, and a list of probabilities `p`).
`exponential(λ,k)`	Returns k draws from $\mathrm{Exponential}(\lambda)$.
`f(ν1,ν2,k)`	Returns k draws from F_{ν_1,ν_2}.
`gamma(α,θ,k)`	Returns k draws from $\Gamma(\alpha, \theta)$ (α and θ the shape and scale parameters).
and more...	...

Note 1: call as `rnd.function_name(...)`.

Note 2: the argument `k` is optional.

Note 3: replace `k` with `(k,l)` to obtain a $k \times l$ matrix instead.

More advanced statistical analysis packages

- statsmodels (http://www.statsmodels.org/stable/index.html): mainly to estimate statistical models, and perform statistical tests. Includes: Linear Regression, Generalized Linear Models, Generalized Estimating Equations, Robust Linear Models, Linear Mixed Effects Models, Regression with Discrete Dependent Variables, ANOVA, Time Series analysis, Models for Survival and Duration Analysis, Statistics (e.g. Multiple Tests, Sample Size Calculations etc.), Nonparametric Methods, Generalized Method of Moments, Empirical Likelihood, ...

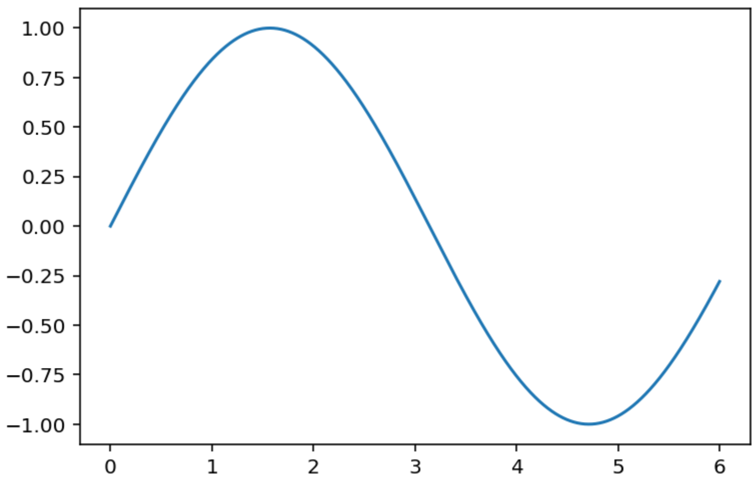
- PyMC (http://pymc-devs.github.io/pymc/): for Bayesian statistical models and fitting algorithms, including MCMC and Gaussian Processes.

- scikit-learn (https://scikit-learn.org/stable/): for machine learning, data mining, and data analysis, including supervised and unsupervised learning. Includes tools for: Classification , Regression , Clustering , Dimensionality reduction , Model selection.

Matplotlib - A brief tour

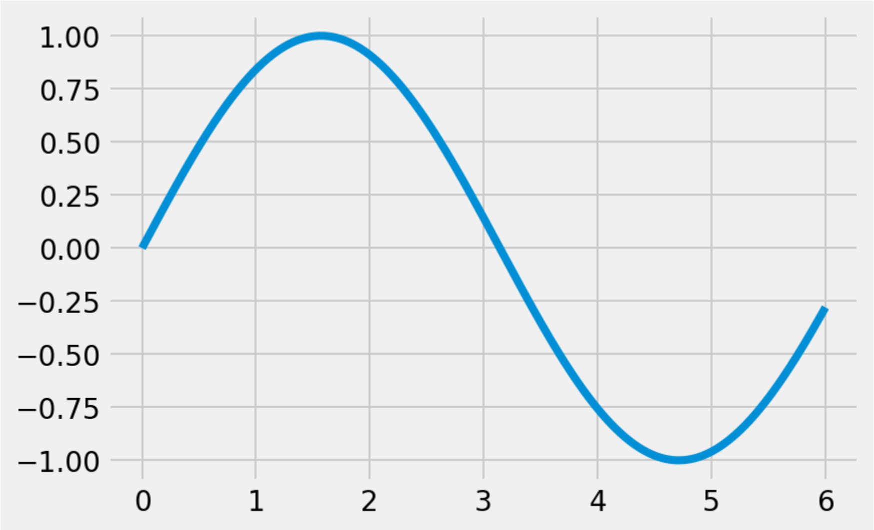
In [96]:
```
%matplotlib inline
import matplotlib.pyplot as plt
```

In [97]:
```
x = np.linspace(0,6,1000)
y = np.sin(x)
plt.plot(x,y)
plt.show()
## actually not necessary here,
## but needed in IPython
#or from command line
```



In [98]:

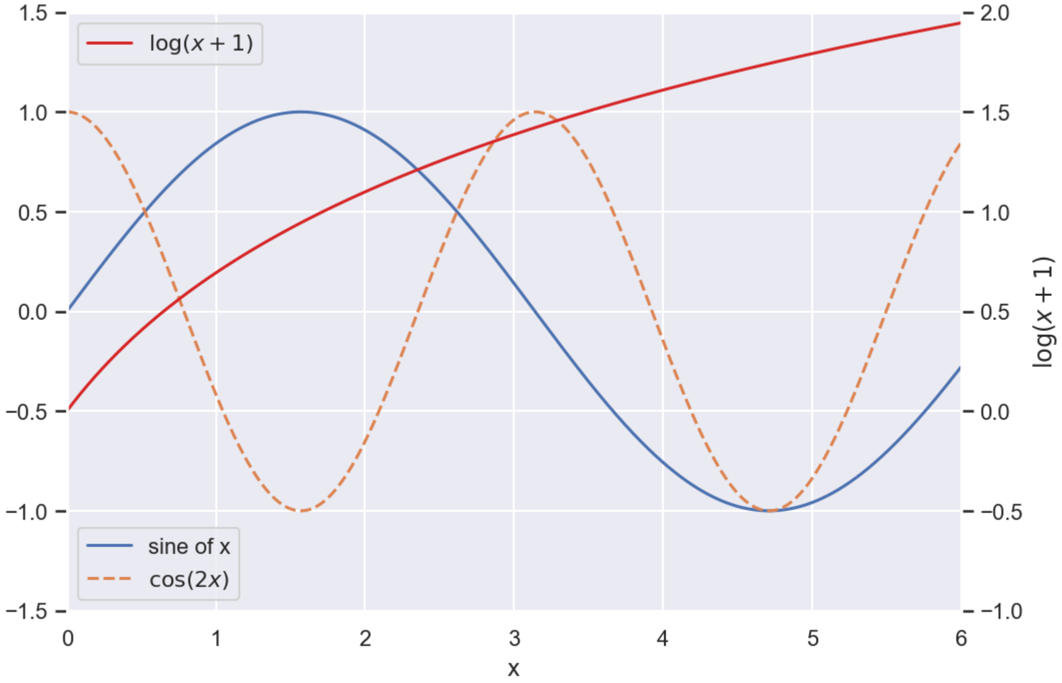
```
#It is possible to change the style as
with plt.style.context(
    'fivethirtyeight'):
    plt.plot(x,y);plt.ylim([-1.1,1.1])
```



`plt.style.use('style-name')` to change across all the notebook;`plt.style.available` to obtain all available styles. [More info here](https://jakevdp.github.io/PythonDataScienceHandbook/04.11-settings-and-stylesheets.html).

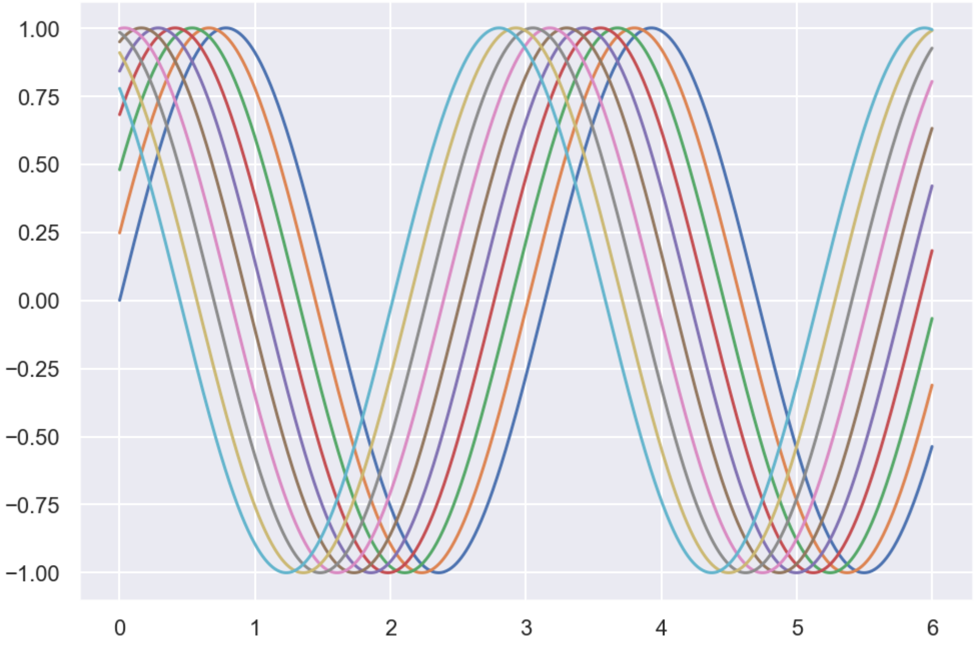
In [99]:

```python
## Plotting
import seaborn as sns
sns.set(rc={'figure.figsize':(8,5.5)})
plt.plot(x,y);
plt.plot(x,np.cos(2*x),'--');
plt.xlabel('x');plt.legend(['sine of x','$\cos(2x)$'],loc='lower left');
plt.ylim((-1.5,1.5));
plt.twinx(); ## creates new y-axis
plt.plot(x,np.log(x+1),color = 'tab:red');
plt.grid(None)
plt.ylabel('$\log(x+1)$');plt.legend(['$\log(x+1)$']);
plt.ylim((-1,2));
plt.xlim((0,6));
# ## Export (Right click and download!)
plt.savefig('img/my_plot.png',dpi=500,transparent=True);
# ## dpi = dots-per-inch; transparent sets alpha-channel to 0
```



log(x + 1)
sine of x
cos(2x)
x
log(x + 1)

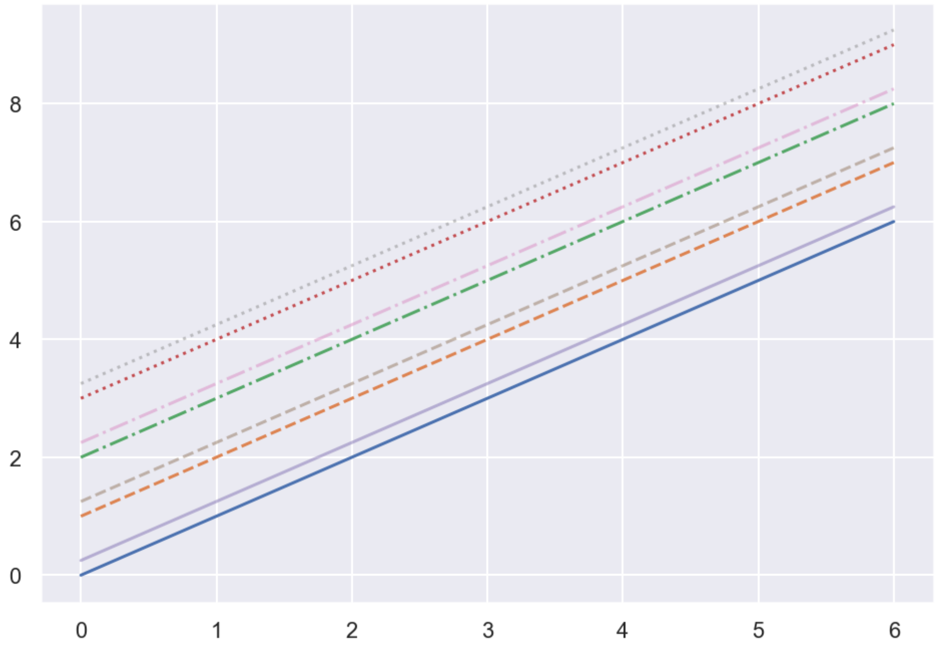
In [100]:
```
for k in range(10):
    plt.plot(x,np.sin(2*x+k/4));
```



Line Styles

In [101]:

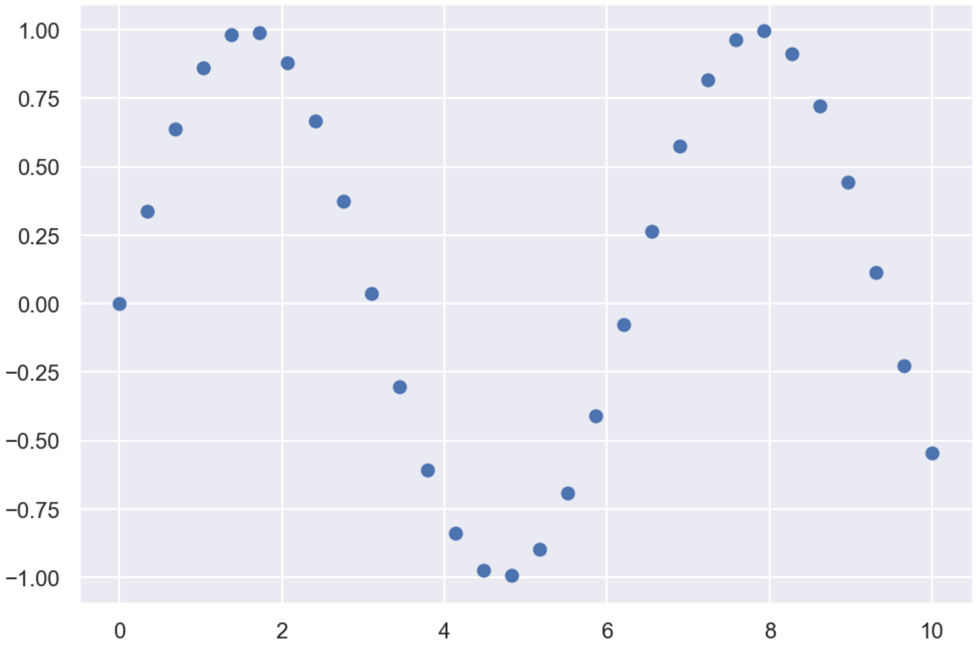
```
plt.plot(x, x + 0, linestyle='solid')
plt.plot(x, x + 1, linestyle='dashed')
plt.plot(x, x + 2, linestyle='dashdot')
plt.plot(x, x + 3, linestyle='dotted');
# For short, you can use the following codes:
plt.plot(x, x + 0.25, linestyle='-',alpha= 0.5)  # solid
plt.plot(x, x + 1.25, linestyle='--',alpha= 0.5) # dashed
plt.plot(x, x + 2.25, linestyle='-.',alpha= 0.5) # dashdot
plt.plot(x, x + 3.25, linestyle=':',alpha= 0.5);  # dotted
```



Scatter Plots

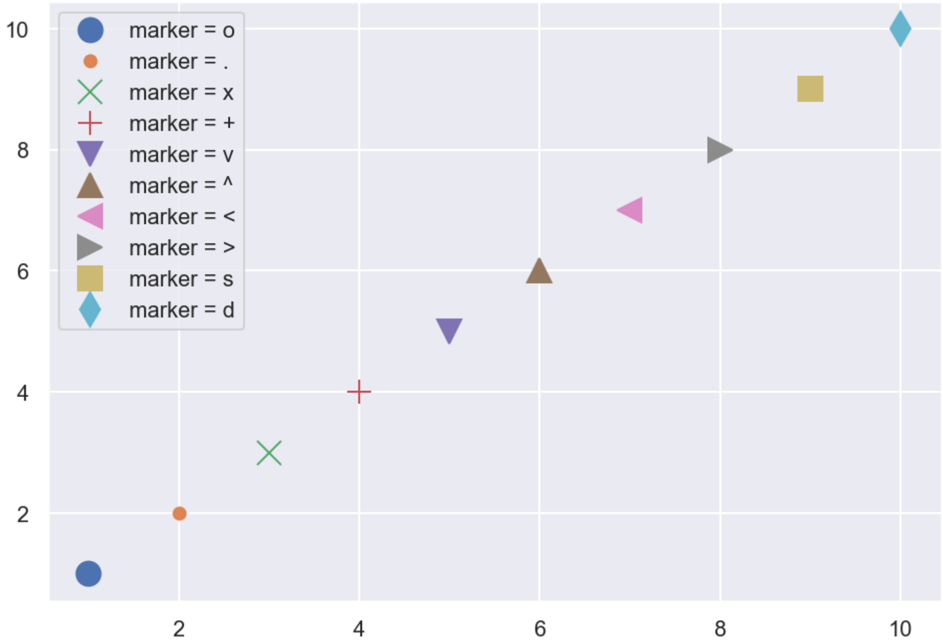
In [102]:

```
x = np.linspace(0, 10, 30)
y = np.sin(x)
plt.plot(x, y, 'o');# '-o'
```



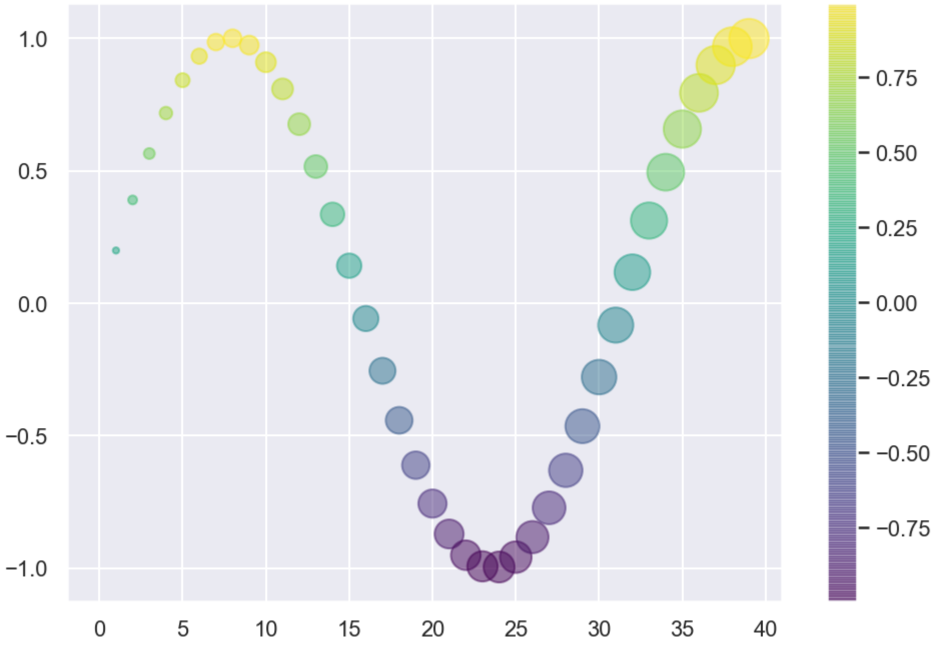
Markers

In [103]:
```
x=0
for marker in ['o', '.', 'x', '+', 'v', '^', '<', '>', 's', 'd']:
    x = x + 1
    plt.plot(x, x, marker,markersize=12,label="marker = " + marker)
plt.legend();
```



Alternatively, Scatter Plot with `plt.scatter`

In [104]:
```
x = np.linspace(0,39,40)
y = np.sin(x/5)
colors = y
sizes = 10*x
plt.scatter(x,y,c=colors,cmap='viridis',s=sizes, alpha = 0.5)
plt.colorbar();  # show color scale
```

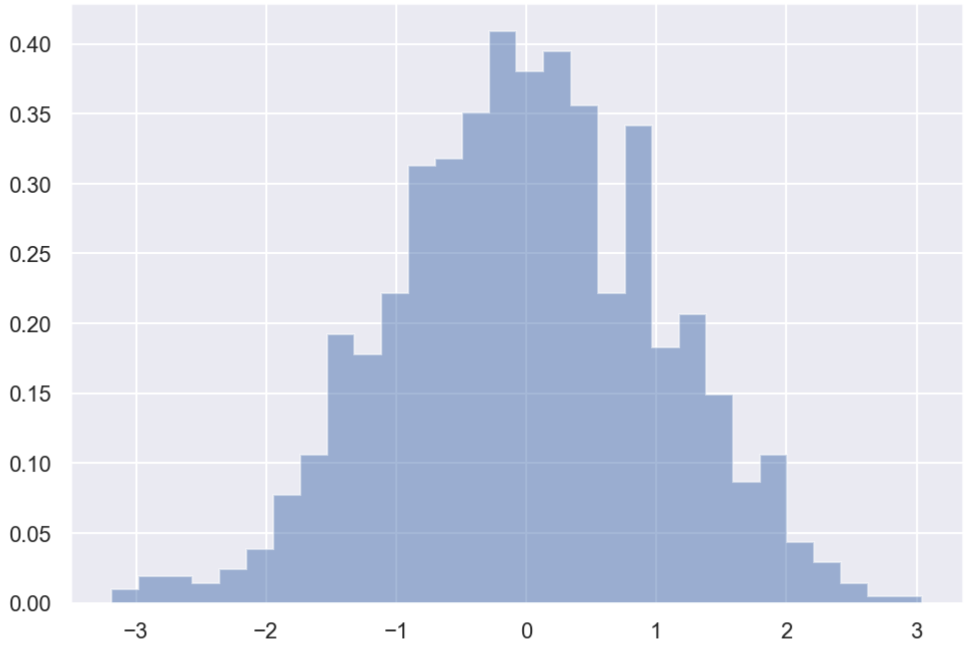


[More colormaps here (https://matplotlib.org/examples/color/colormaps_reference.html)](https://matplotlib.org/examples/color/colormaps_reference.html).

Histograms

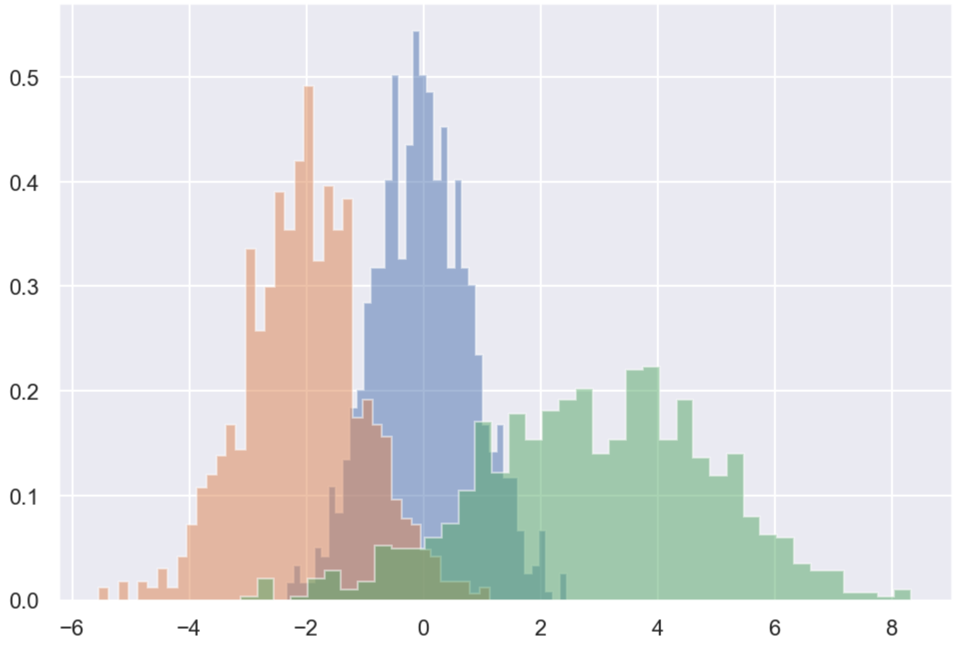
In [105]:

```
data = np.random.randn(1000)
plt.hist(data, bins=30, density=True, alpha=0.5,histtype='stepfilled');
```



In [106]:

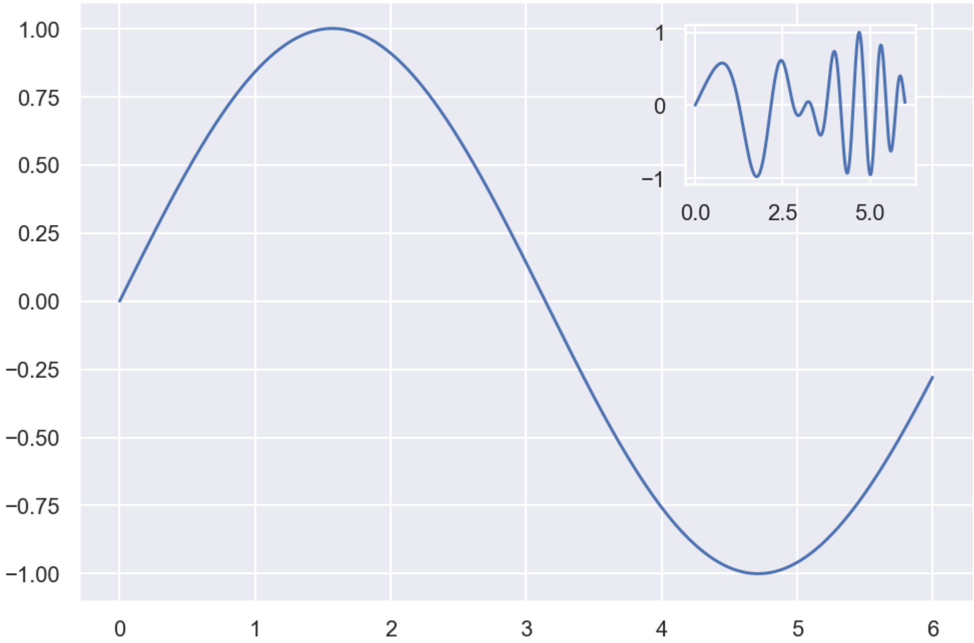
```
x1 = np.random.normal(0, 0.8, 1000)
x2 = np.random.normal(-2, 1, 1000)
x3 = np.random.normal(3, 2, 1000)

## by the way, we can pass the same options to multiple plots!
kwargs = dict(histtype='stepfilled', alpha=0.5, density=True, bins=40)
plt.hist(x1, **kwargs)
plt.hist(x2, **kwargs)
plt.hist(x3, **kwargs);
```



Subplots - by hand

In [107]:

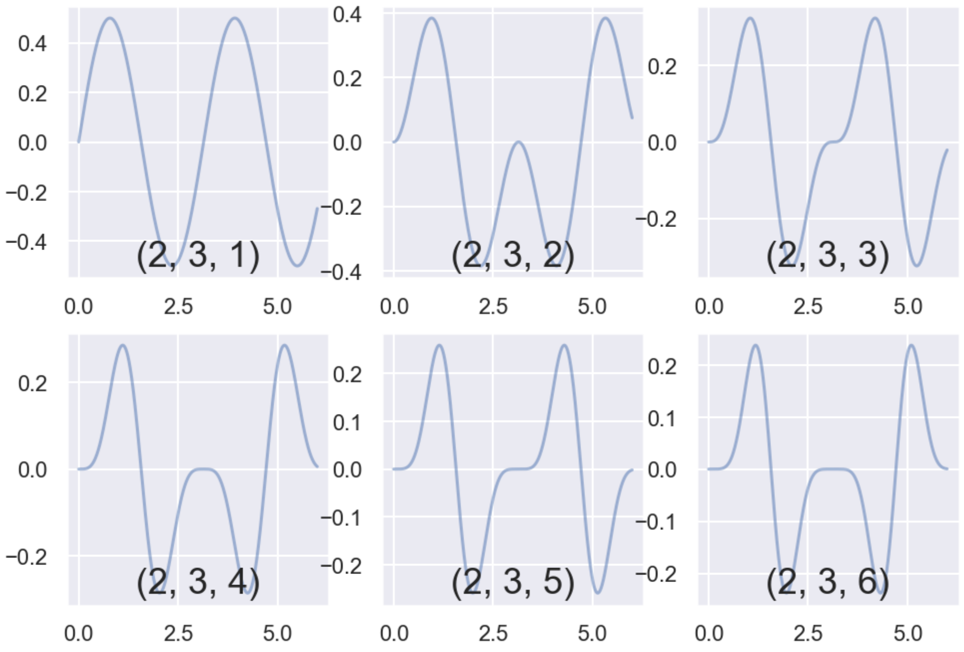
```
x = np.linspace(0, 6,1000)
ax1 = plt.axes()  # standard axes
ax2 = plt.axes([0.65, 0.65, 0.2, 0.2])
ax1.plot(x,np.sin(x));
ax2.plot(x,np.sin(x)*np.cos(x**2));
```



Subplots - with `plt.subplot`

In [108]:

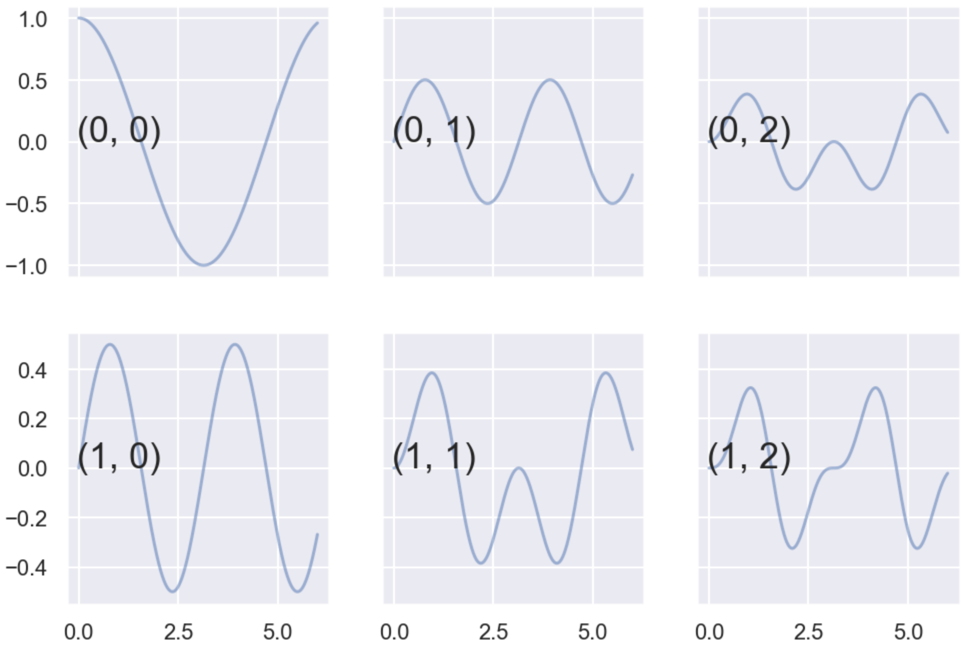
```
for i in range(1, 7):
    plt.subplot(2, 3, i)
    y=np.sin(x)**i*np.cos(x)
    plt.plot(x,y,alpha=0.5)
    plt.text(np.mean(x), min(y), str((2, 3, i)),
             fontsize=18, ha='center')
```



Subplots - or with `plt.subplots`

to share axis

In [109]:
```
fig, ax = plt.subplots(2, 3, sharex='col', sharey='row')
for i in range(2):
    for j in range(3):
        y=np.sin(x)**(i+j)*np.cos(x)
        ax[i, j].plot(x,y,alpha=0.5)
        ax[i, j].text(1, 0, str((i, j)),
                      fontsize=18, ha='center')
```

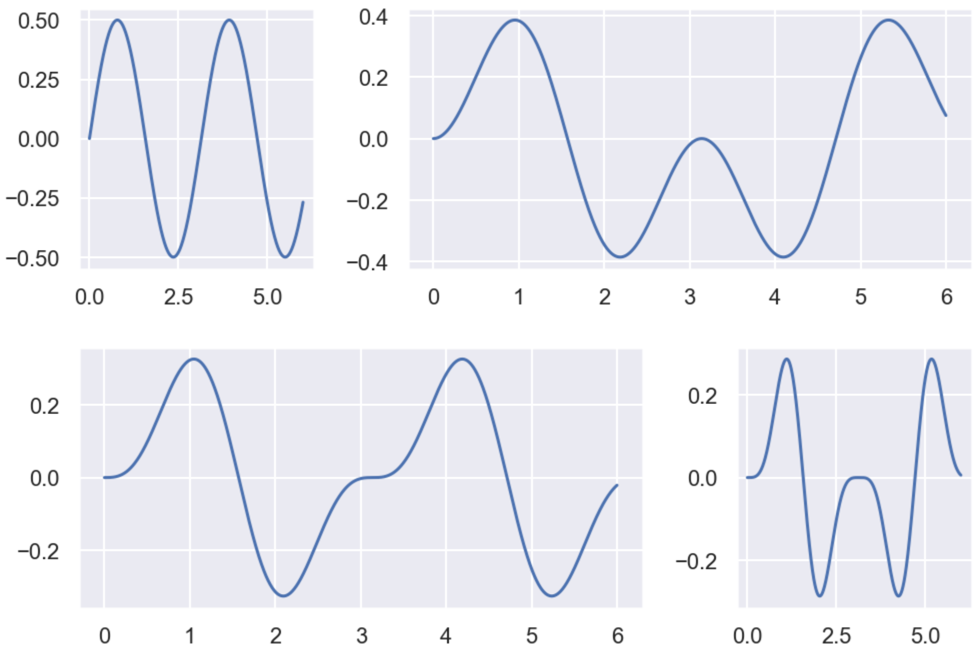


Subplots - or with `plt.GridSpec`

for more complicated arrangements

In [110]:

```
grid = plt.GridSpec(2, 3, wspace=0.4, hspace=0.3)
plt.subplot(grid[0, 0])
plt.plot(x,np.sin(x)*np.cos(x))
plt.subplot(grid[0, 1:])
plt.plot(x,np.sin(x)**(2)*np.cos(x))
plt.subplot(grid[1, :2])
plt.plot(x,np.sin(x)**(3)*np.cos(x))
plt.subplot(grid[1, 2]);
plt.plot(x,np.sin(x)**(4)*np.cos(x));
```



A few more complicated plots

In [111]:

```
# Double donut
# Make data: consider 3 groups and 7 subgroups
group_names=['groupA', 'groupB', 'groupC']
group_size=[12,11,30]
subgroup_names=['A.1', 'A.2', 'A.3', 'B.1', 'B.2', 'C.1', 'C.2', 'C.3', 'C.4', 'C.
5']
subgroup_size=[4,3,5,6,5,10,5,5,4,6]
# Create colors
a, b, c=[plt.cm.Blues, plt.cm.Reds, plt.cm.Greens]
# First Ring (outside)
fig, ax = plt.subplots()
mypie, _ = ax.pie(group_size, radius=1.3, labels=group_names,
                  colors=[a(0.6), b(0.6), c(0.6)] )
plt.setp( mypie, width=0.3, edgecolor='white')
# Second Ring (Inside)
mypie2, _ = ax.pie(subgroup_size, radius=1.3-0.3, labels=subgroup_names,
                   labeldistance=0.7,
                   colors=[a(0.5), a(0.4), a(0.3), b(0.5), b(0.4),
                           c(0.6), c(0.5), c(0.4), c(0.3), c(0.2)])
plt.setp( mypie2, width=0.4, edgecolor='white')
plt.margins(0,0)
```

In [ ]:

```
%%capture
## prevents cell to print output
from mpl_toolkits.mplot3d import Axes3D
import pandas as pd
data = pd.read_csv('res/vulcano.csv')

# Transform data to a long format
df=data.unstack().reset_index()
df.columns=["X","Y","Z"]
# And transform the old column name in something numeric
df['X']=pd.Categorical(df['X'])
df['X']=df['X'].cat.codes

for angle in range(0,360,1):
    # Make the plot
    fig = plt.figure()
    ax = fig.gca(projection='3d')
    ax.plot_trisurf(df['Y'], df['X'], df['Z'], cmap=plt.cm.viridis, linewidth=0.2)

    # Set the angle of the camera
    ax.view_init(30,angle)

    # Save it
    filename='./img/PNG/ANIMATION/Vulcano_step'+str(angle)+'.png'
    plt.savefig(filename, dpi=180)
```

SegmentLocal

A very brief tour of Seaborn

In [112]:

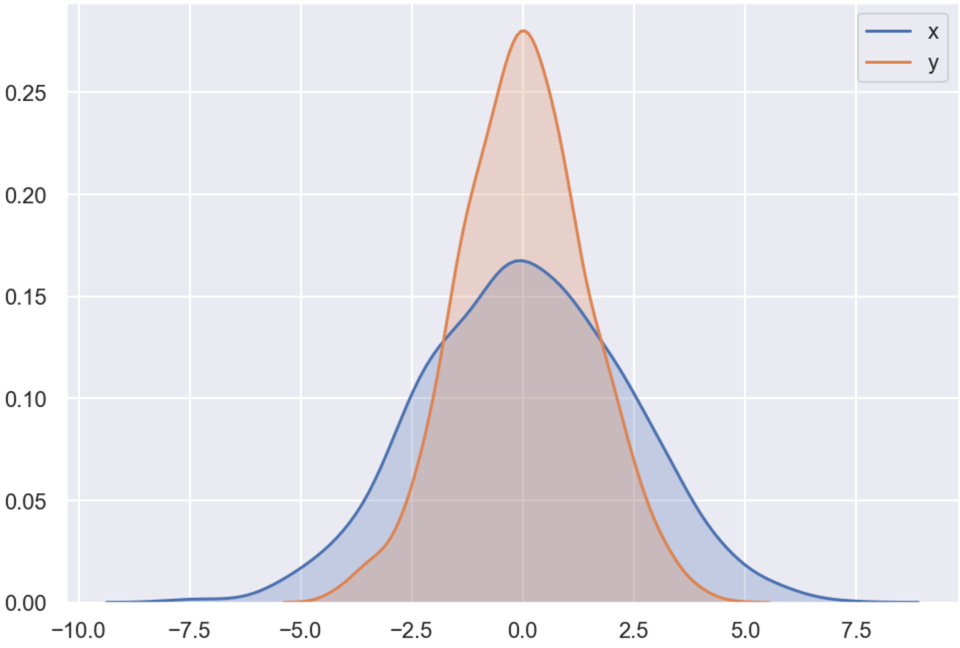
```
import pandas as pd
data = np.random.multivariate_normal(
    [0, 0], [[5, 2], [2, 2]],
    size=2000)
data = pd.DataFrame(
    data, columns=['x', 'y'])
data.head(5)
```

Out[112]:

	x	y
0	2.900026	1.941117
1	-1.362563	-0.649883
2	-0.278481	-1.442579
3	-2.836998	-1.331469
4	1.841642	0.246689

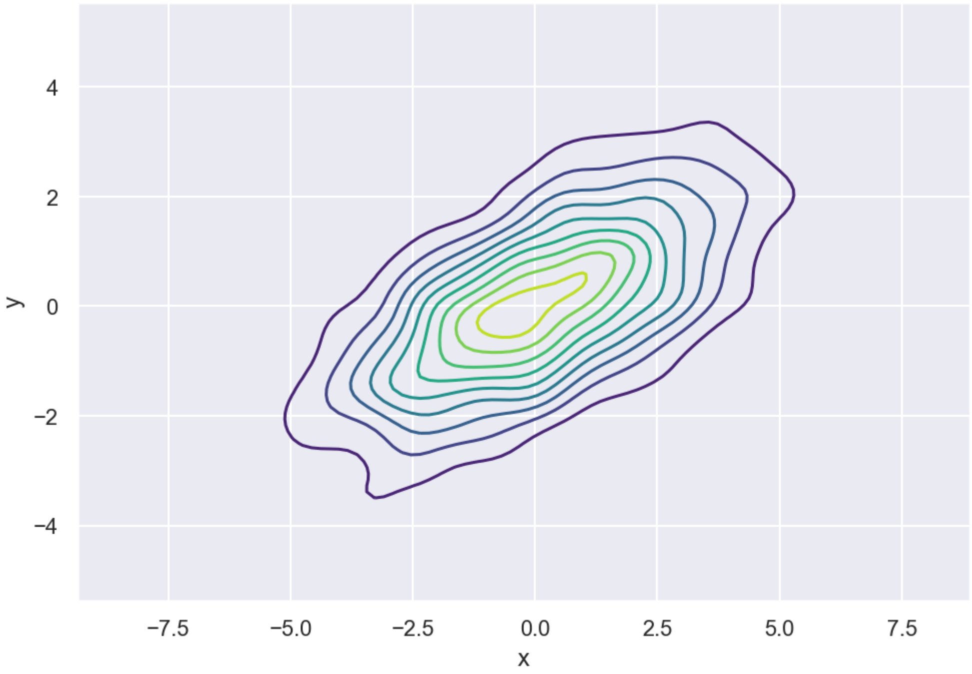
In [113]:

```
for col in 'xy':
    sns.kdeplot(data[col], shade=True)
```



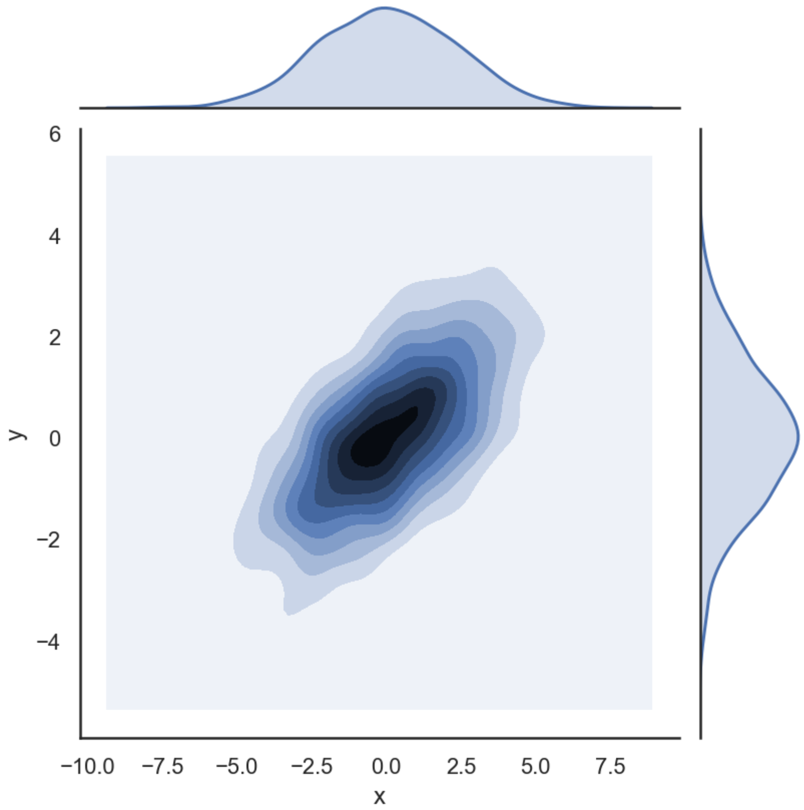
Visualise the joint distribution

In [114]:
```
sns.kdeplot(data['x'],data['y'],
            cmap='viridis');
```



In [115]:

```
with sns.axes_style('white'):
    sns.jointplot("x", "y", data,
                  kind='kde');
# 'hex','kde'
```



In [116]:

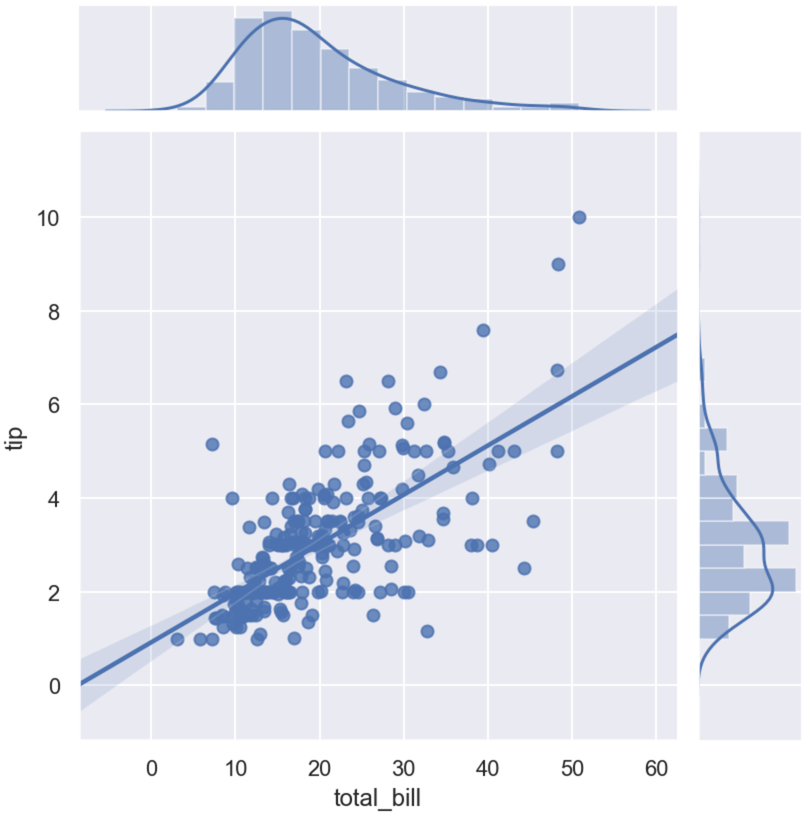
```python
tips = sns.load_dataset("tips")
tips.head() ##tips to restaurant staff
```

Out[116]:

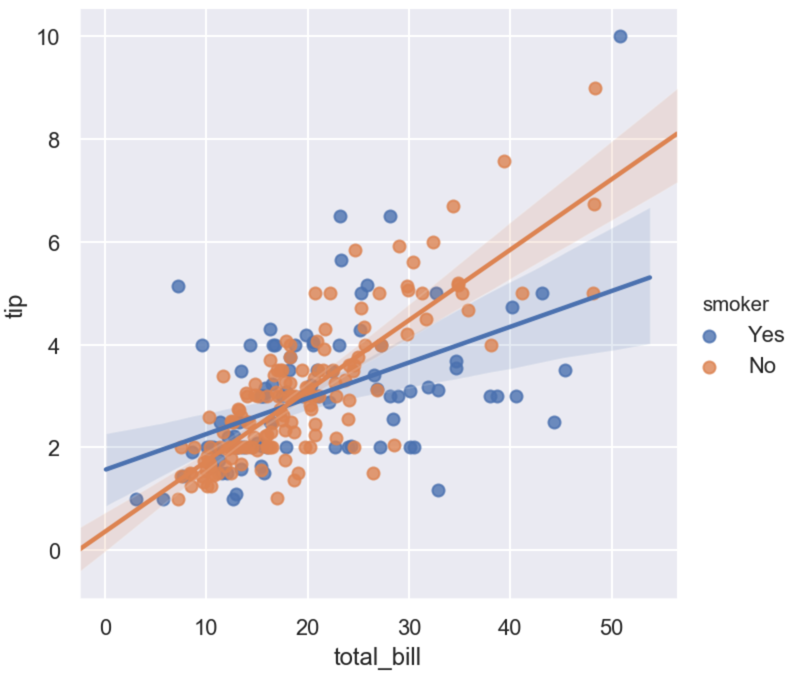
	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

Column	Description
total_bill	Total bill including tax [USD]
tip	Tip [USD]
sex	Sex of person paying
smoker	Smoker in party?
day	Day of the week
time	Time of the day
size	Size of the party

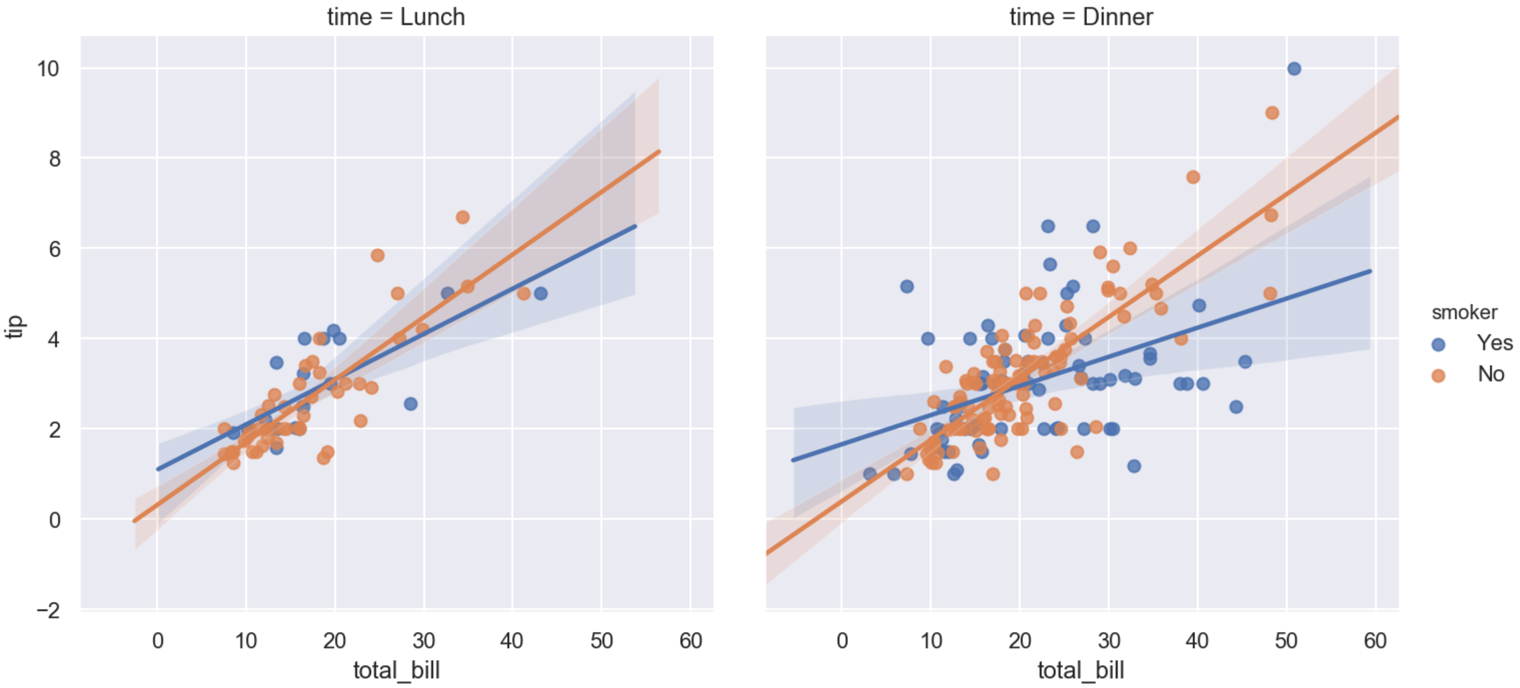
In [117]:
```
sns.jointplot(x="total_bill", y="tip",
              data=tips, kind="reg");
```



In [118]:
```
sns.lmplot(x="total_bill", y="tip", hue="smoker", data=tips);
```

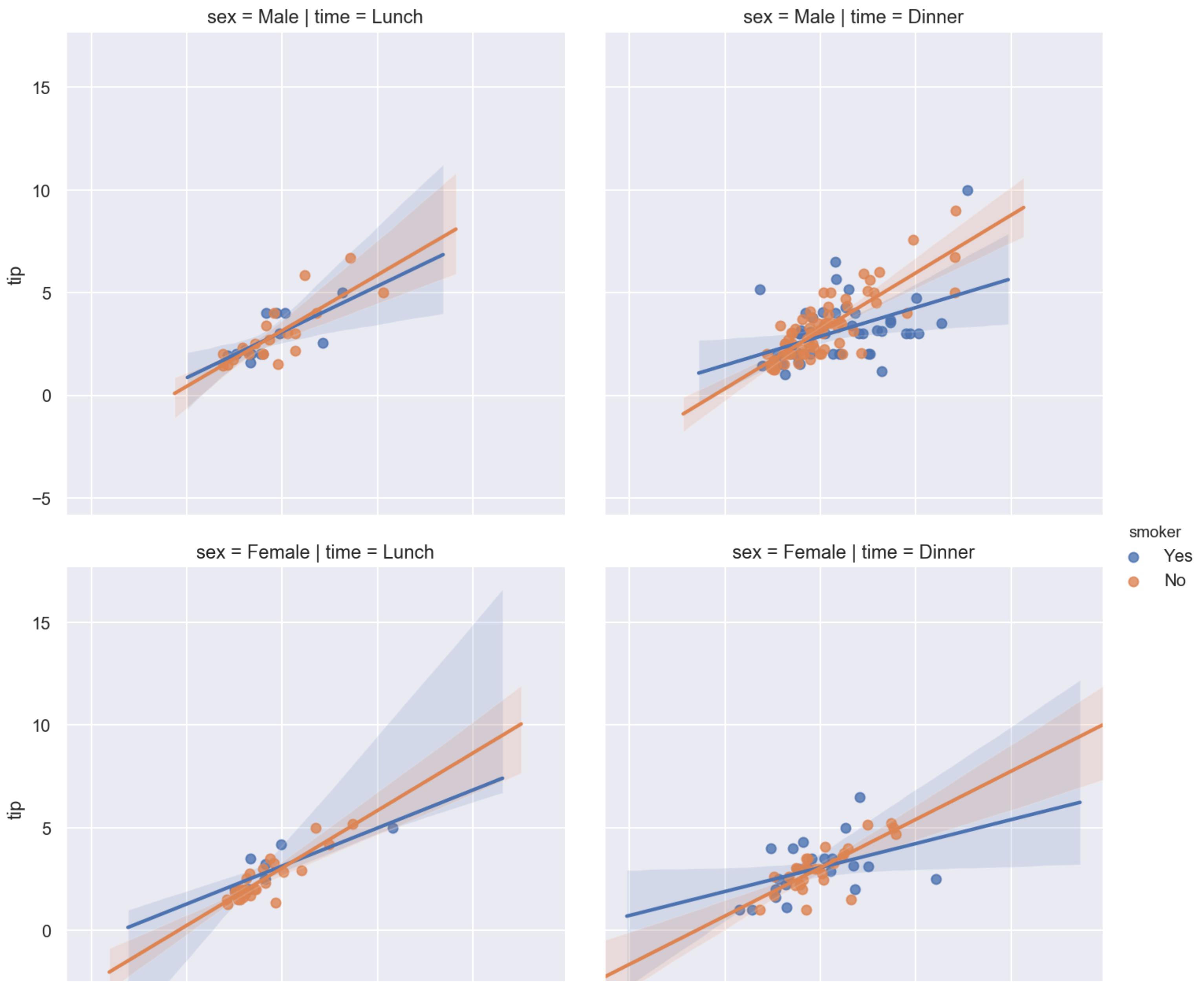


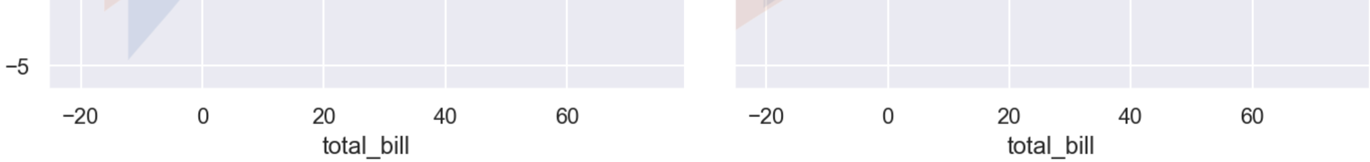
In [119]:
```
sns.lmplot(x="total_bill", y="tip", hue="smoker", col="time", data=tips);
```



In [120]:

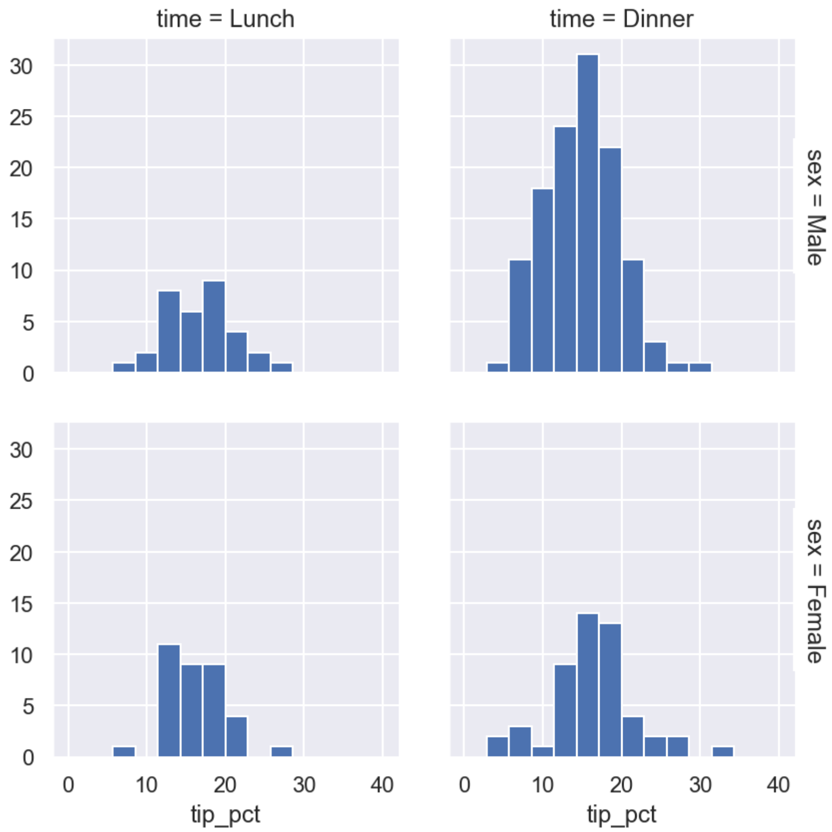
```
sns.lmplot(x="total_bill", y="tip", hue="smoker",col="time", row="sex", data=tips
);
```




−5
−20
0
20
40
60
total_bill
−20
0
20
40
60
total_bill

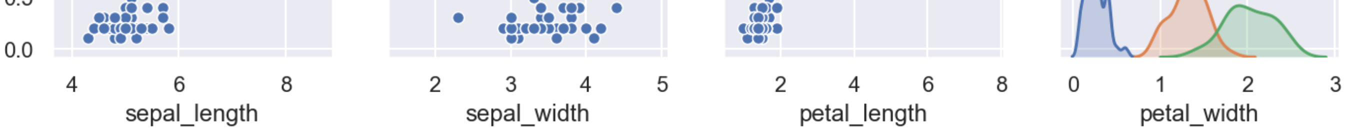
In [121]:

```python
tips['tip_pct'] = 100 * tips['tip'] / tips['total_bill']
grid = sns.FacetGrid(tips, row="sex", col="time", margin_titles=True)
grid.map(plt.hist, "tip_pct", bins=np.linspace(0, 40, 15));
```



In [122]:

```
iris = sns.load_dataset("iris")
sns.pairplot(iris, hue='species', height=2.5);
```


0.0
4
6
8
sepal_length
2
3
4
5
sepal_width
2
4
6
8
petal_length
0
1
2
3
petal_width

Importing data from the web

i.e. Pandas' DataReader (https://pandas-datareader.readthedocs.io/en/latest/index.html)

Remote Data Access to:

- FRED
- World Bank
- OECD
- Eurostat
- Yahoo Finance
- ...

and more (https://pandas-datareader.readthedocs.io/en/latest/remote_data.html).

Suppose we want recent data on economic growth for the EU founder countries.

To download data from, say, the WorldBank, we must know the exact indicator of the data we want to read.

In [123]:
```
from pandas_datareader import wb
# wb.search('gdp')
wb.search('gdp.*capita.*const').iloc[:,:2]
### `.*` indicates that any text in that position is a match
```

Out[123]:

	id	name
646	6.0.GDPpc_constant	GDP per capita, PPP (constant 2011 internation...
9116	NY.GDP.PCAP.KD	GDP per capita (constant 2010 US$)
9118	NY.GDP.PCAP.KN	GDP per capita (constant LCU)
9120	NY.GDP.PCAP.PP.KD	GDP per capita, PPP (constant 2011 internation...
9121	NY.GDP.PCAP.PP.KD.87	GDP per capita, PPP (constant 1987 internation...

In [124]:
```
gdp_pc_conts_idx = wb.search('gdp.*capita.*const').iloc[1,0]
gdp_pc_conts_idx
```

Out[124]:
```
'NY.GDP.PCAP.KD'
```

We create a list of country indicators:

In [125]:
```
countries = ['DE', 'FR', 'IT', 'NL', 'BE','LU']
```

In [126]:
```
data = wb.download(indicator='NY.GDP.PCAP.KD',country=countries,start=1991, end=2018)
## rearrange data
GDP = data.reset_index().pivot('year','country')
GDP.head(4)
```

Out[126]:

	NY.GDP.PCAP.KD					
country	Belgium	France	Germany	Italy	Luxembourg	Netherlands
year						
1991	33586.958310	32855.995990	33742.219217	31292.053095	70667.238370	36063.470851
1992	33962.986386	33216.008996	34130.852398	31531.690020	71003.669122	36402.472894
1993	33505.165503	32869.778973	33583.006036	31243.679023	72999.737640	36604.233482
1994	34479.937552	33515.713512	34289.124749	31909.236711	74763.871165	37461.566161

At this point, we can easily compute each country's growth as

In [127]:
```
GROWTH = 100 * GDP.pct_change()
GROWTH.head(5)
```

Out[127]:

	NY.GDP.PCAP.KD					
country	**Belgium**	**France**	**Germany**	**Italy**	**Luxembourg**	**Netherlands**
year						
1991	NaN	NaN	NaN	NaN	NaN	NaN
1992	1.119566	1.095730	1.151771	0.765808	0.476077	0.940015
1993	-1.347999	-1.042359	-1.605135	-0.913402	2.811219	0.554250
1994	2.909319	1.965132	2.102607	2.130215	2.416630	2.342168
1995	2.170550	1.716968	1.439068	2.885202	0.017300	2.607973

(It is also possible to automatically generate a $\LaTeX$ table as `GROWTH.tail(6).round(2).to_latex('my_table.tex')`. This creates a file called `my_table.tex` in the current directory.)

Finally, we plot the results:

In [128]:
```
GROWTH.columns = countries
GROWTH.plot();
```

In [129]:

```
GROWTH.plot(subplots=True,
        layout=[3,2], sharey=True);
```

Suppose we want to regress change in consumption (as Personal Consumption Expenditures) on the change in gdp:

$$\Delta \ln(c_t) = \alpha + \beta\Delta \ln(y_t) + \epsilon_t$$

In [130]:

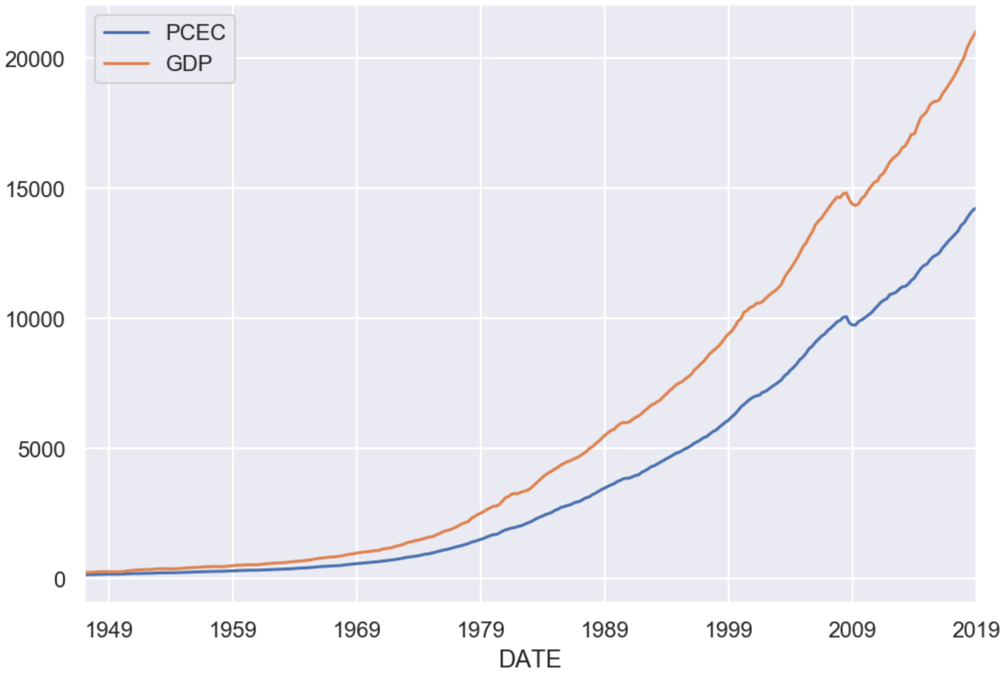
```
import pandas_datareader.data as web
usdata=web.DataReader(['PCEC','GDP'],
                      'fred', 1947, 2019)
usdata.head(8)
```

Out[130]:

	PCEC	GDP
DATE		
1947-01-01	156.161	243.164
1947-04-01	160.031	245.968
1947-07-01	163.543	249.585
1947-10-01	167.672	259.745
1948-01-01	170.372	265.742
1948-04-01	174.142	272.567
1948-07-01	177.072	279.196
1948-10-01	177.928	280.366

In [131]:

```
usdata.plot();
```



In [132]:

```python
import statsmodels.formula.api as smf
smf.ols('PCEC ~ GDP', np.log(usdata).diff()).fit().summary()
```

Out[132]:

OLS Regression Results

Dep. Variable:	PCEC	**R-squared:**	0.491
Model:	OLS	**Adj. R-squared:**	0.490
Method:	Least Squares	**F-statistic:**	276.2
Date:	Fri, 10 May 2019	**Prob (F-statistic):**	7.07e-44
Time:	19:00:28	**Log-Likelihood:**	1027.8
No. Observations:	288	**AIC:**	-2052.
Df Residuals:	286	**BIC:**	-2044.
Df Model:	1		
Covariance Type:	nonrobust		

	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975]**
Intercept	0.0061	0.001	8.730	0.000	0.005	0.008
GDP	0.6162	0.037	16.620	0.000	0.543	0.689

Omnibus:	103.655	**Durbin-Watson:**	2.537
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	1131.340
Skew:	-1.117	**Prob(JB):**	2.15e-246
Kurtosis:	12.449	**Cond. No.**	91.9

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

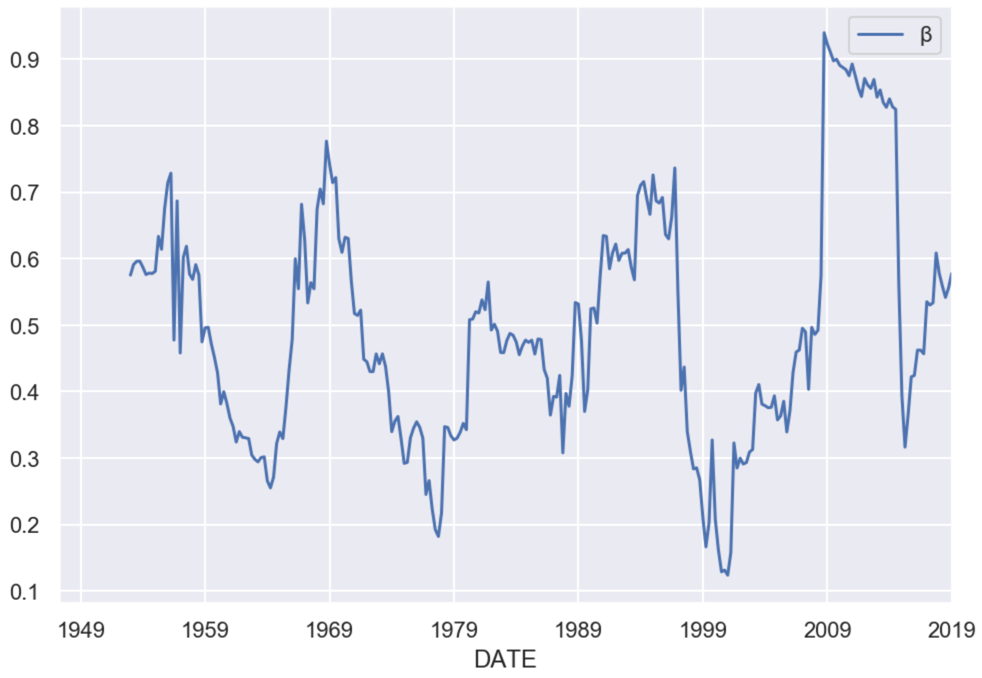
The sample covers a long period (~70y of quarterly observation), thus it is reasonable to wonder whether the parameters are constant.

Let us estimate with a rolling sample. In particular, consider 24 quarterly observations rolling window.

In [133]:
```
growth=(100*np.log(usdata).diff())[1:]
T, _ = growth.shape ###---- T = # of observations
h = 24
```

In [134]:
```
def window_β(k): return smf.ols('PCEC~GDP',growth[k-h:k]).fit().params['GDP']
```

In [135]:
```
growth.loc[h-1:,'β'] = [window_β(k) for k in range(h,T+1)]
growth[['β']].plot();
```



Another example

In [136]:

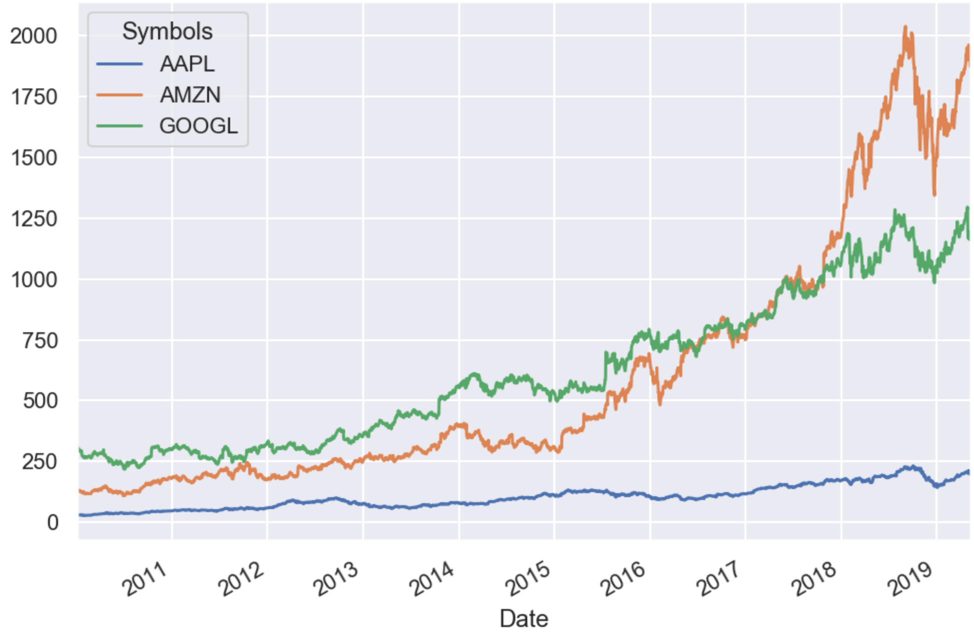
```
# AAPL AMZN and GOOGL stocks
from pandas_datareader import data
tickers = ['AAPL', 'AMZN', 'GOOGL']
start_date, end_date = '2010-01-01', '2019-05-10'
df = data.get_data_yahoo(tickers, start_date, end_date)
df.head()
```

Out[136]:

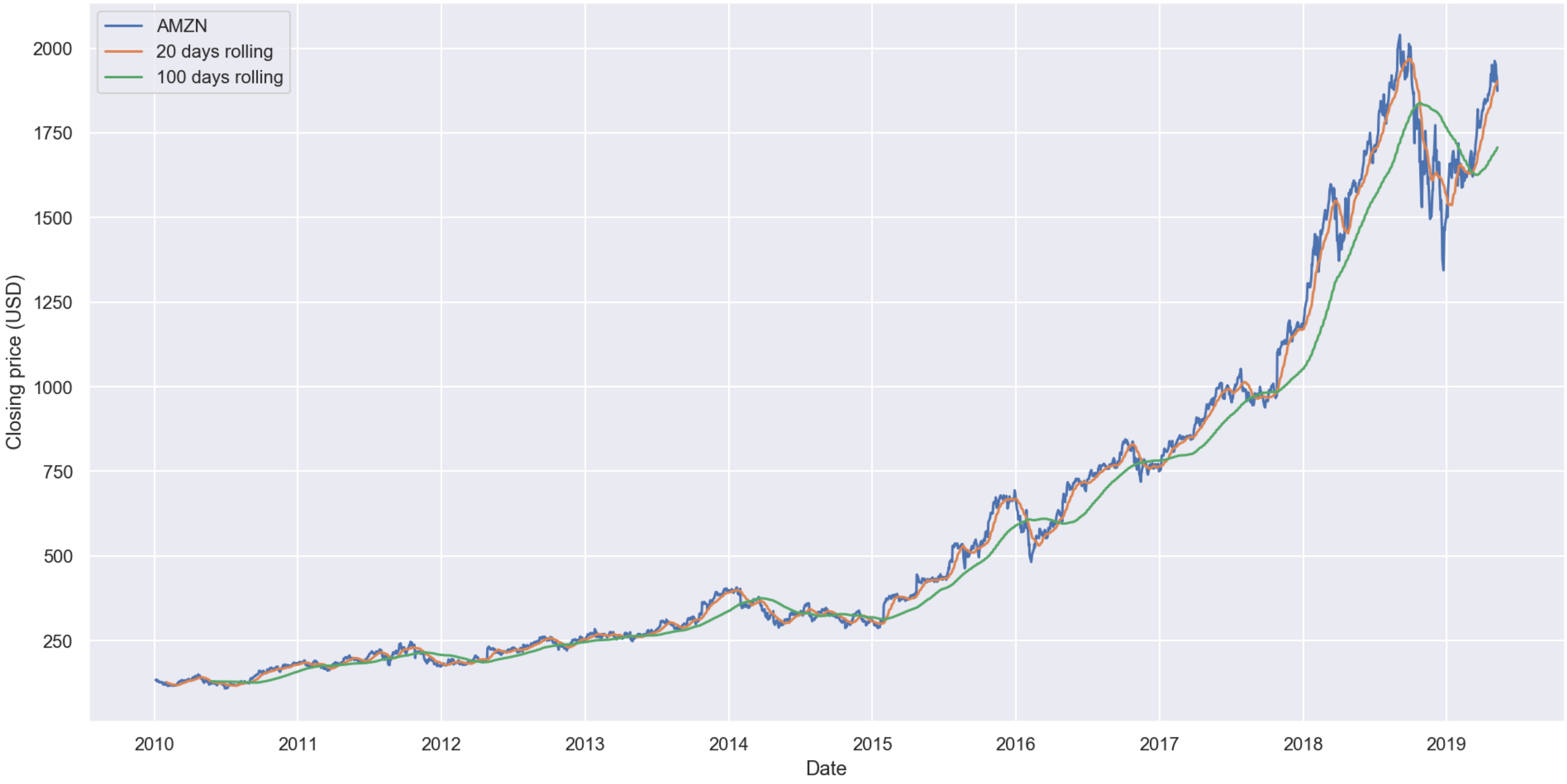
Attributes	High			Low			Open			Close
Symbols	AAPL	AMZN	GOOGL	AAPL	AMZN	GOOGL	AAPL	AMZN	GOOGL	AAPL
Date										
2010-01-04	30.642857	136.610001	315.070068	30.340000	133.139999	312.432434	30.490000	136.250000	313.788788	30.572857
2010-01-05	30.798571	135.479996	314.234222	30.464285	131.809998	311.081085	30.657143	133.429993	313.903900	30.625713
2010-01-06	30.747143	134.729996	313.243256	30.107143	131.649994	303.483490	30.625713	134.600006	313.243256	30.138571
2010-01-07	30.285715	132.320007	305.305298	29.864286	128.800003	296.621613	30.250000	132.009995	305.005005	30.082857
2010-01-08	30.285715	133.679993	301.926941	29.865715	129.029999	294.849854	30.042856	130.559998	296.296295	30.282858

In [137]:

```
df['Close'].plot();
```



In [138]:
```
##Plotting
amzn = df['Close']["AMZN"]
# Calculate moving averages of the closing prices rolling at 20 and 100 days
roll1_amzn = amzn.rolling(window=20).mean()
roll2_amzn = amzn.rolling(window=100).mean()
fig, ax = plt.subplots(figsize=(16,8))
ax.plot(amzn, label='AMZN')
ax.plot(roll1_amzn, label='20 days rolling')
ax.plot(roll2_amzn, label='100 days rolling')
ax.set_xlabel('Date')
ax.set_ylabel('Closing price (USD)')
ax.legend();
```



Choropleth Maps

In [139]:

```
import pycountry
EU_countries = ["Austria","Belgium","Bulgaria","Croatia","Cyprus","Czechia","Denmark",
                "Estonia","Finland","France","Germany","Greece","Hungary","Ireland","Italy","Latv
ia",
                "Lithuania","Luxembourg","Malta","Netherlands","Poland","Portugal","Romania","Slo
vakia",
                "Slovenia","Spain","Sweden","United Kingdom"]
print(len(EU_countries) == 28)
countries = [pycountry.countries.get(name= country).alpha_2 for country in EU_countries]
```

```
True
```

In [140]:

```
from pandas_datareader import wb
wb.search('gdp.*capita.*current').iloc[:,:2]
### `.*` indicates that any text in that position is
```

Out[140]:

	id	name
9114	NY.GDP.PCAP.CD	GDP per capita (current US$)
9115	NY.GDP.PCAP.CN	GDP per capita (current LCU)
9119	NY.GDP.PCAP.PP.CD	GDP per capita, PPP (current international $)

In [141]:

```
GDP = wb.download(indicator='NY.GDP.PCAP.CD',country=countries,start=2017, end=2017)
GDP = GDP.reset_index().drop(columns=['year'])## rearrange data
GDP = GDP.rename(columns = {'NY.GDP.PCAP.CD':'GDP'})
GDP.transpose()
```

Out[141]:

	0	1	2	3	4	5	6	7	8	9	...	18	19	
country	Austria	Belgium	Bulgaria	Cyprus	Czech Republic	Germany	Denmark	Spain	Estonia	Finland	...	Luxembourg	Latvia	Ma
GDP	47380.8	43467.4	8228.01	25658.8	20379.9	44665.5	57218.9	28208.3	20200.4	45804.7	...	104499	15684.6	26

2 rows × 28 columns

In [142]:

```
from datetime import date
import currency_converter as CC### Data from ECB
c = CC.CurrencyConverter()
usd_eur = c.convert(1,'EUR', 'USD', date=date(2017,3,21))
GDP['GDP'] /= usd_eur
GDP.transpose()
```

Out[142]:

	0	1	2	3	4	5	6	7	8	9	...	18	19	
country	Austria	Belgium	Bulgaria	Cyprus	Czech Republic	Germany	Denmark	Spain	Estonia	Finland	...	Luxembourg	Latvia	Malta
GDP	43863	40240.2	7617.12	23753.7	18866.8	41349.3	52970.6	26114	18700.6	42403.9	...	96740.2	14520.1	2476

2 rows × 28 columns

In [143]:

```
cty_a_2 = []
for country in GDP['country']:
    try:
        cty_a_2.append(
            pycountry.countries.get(
                name= country).alpha_3)
    except:
        cty_a_2.append(
            pycountry.countries.get(
                official_name= country).alpha_3)
GDP['country'] = cty_a_2
GDP.transpose()
```

Out[143]:

	0	1	2	3	4	5	6	7	8	9	...	18	19	20	
country	AUT	BEL	BGR	CYP	CZE	DEU	DNK	ESP	EST	FIN	...	LUX	LVA	MLT	N
GDP	43863	40240.2	7617.12	23753.7	18866.8	41349.3	52970.6	26114	18700.6	42403.9	...	96740.2	14520.1	24762.3	4

2 rows × 28 columns

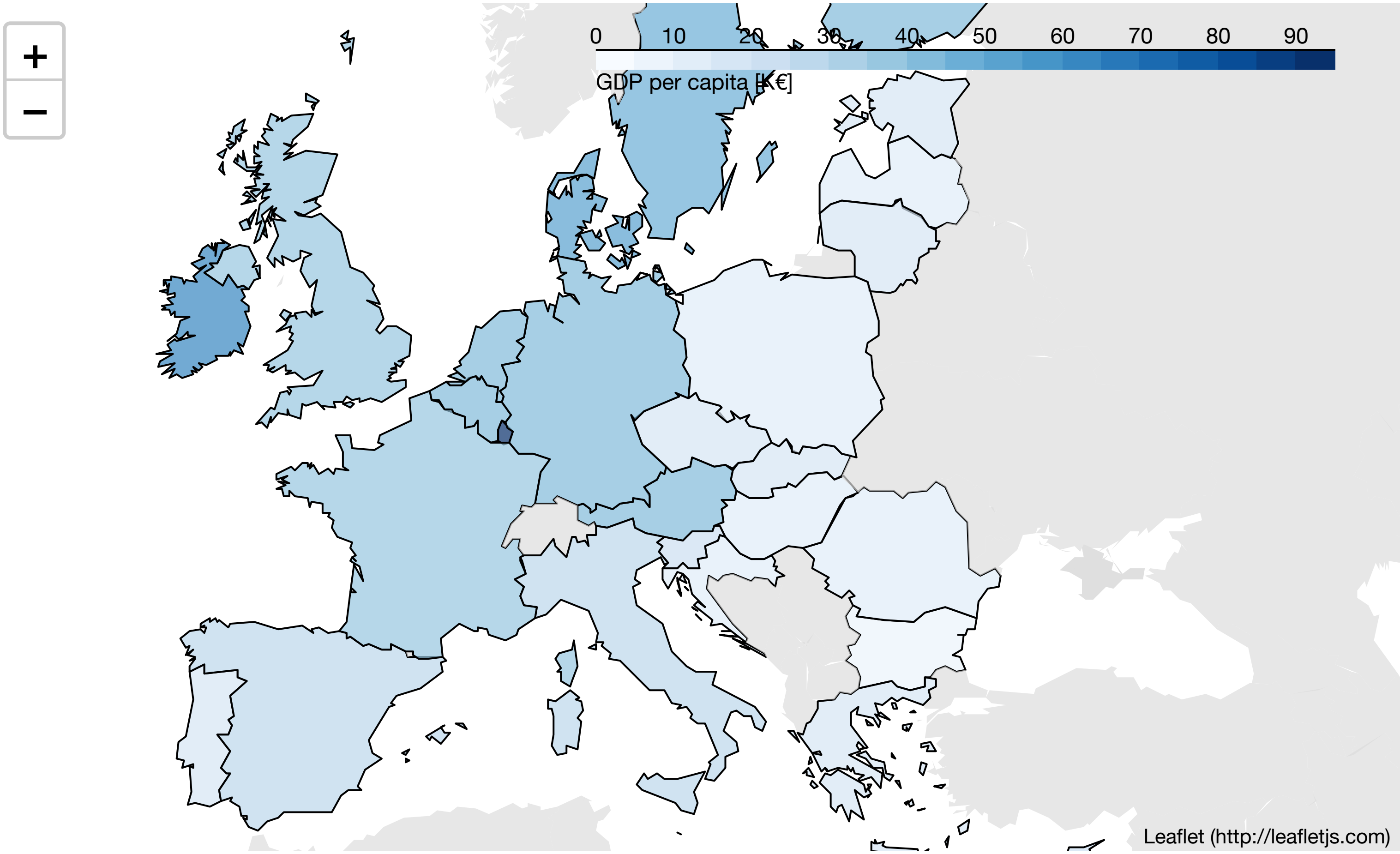
In [144]:

```
## Choropleth
import folium
from branca import colormap

map_data = pd.DataFrame({
    'A3': list(GDP['country']),
    'value': list(GDP['GDP']/1000)
})

map_dict = map_data.set_index('A3')['value'].to_dict()
vmin = min(map_dict.values())
vmax = max(map_dict.values())
color_scale = colormap.linear.Blues_09.scale(vmin, vmax )
###### try dir(colormap.linear) for more colormaps
# color_scale = colormap.LinearColormap(['azure','darkblue'], vmin = vmin, vmax = vmax)
# color_scale = colormap.LinearColormap(['yellow','red'], vmin = vmin, vmax = vmax)
color_scale = color_scale.to_step(index=range(0,100,5))# 0,70,10
color_scale.caption = 'GDP per capita [K€]'

def get_color(feature,border = False):
    value = map_dict.get(feature['properties']['A3'])
    if not border:### SET FILLING COLOR
        if value is None:
            return '#DDDDDD' # MISSING -> gray
#             return 'white'  # MISSING -> white
        else:
            return color_scale(value)
    else:           ### SET BORDER COLOR
        if value is None:
            return None # MISSING -> no color
        else:
            return 'black'

m = folium.Map(
    tiles=None, #Stamen Terrain, OpenStreetMap, Stamen Toner, Mapbox Bright, and Mapbox Control R
oom
    location = [50, 15],
    zoom_start = 4
)
folium.GeoJson(
    data = './res/world_geo.json_files/coastline_cty_10km.geo.json',
```

In [145]: m

```
m
```

Out[145]:



In [ ]:

```
m.save('map.html')
```

End of Lecture 2