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 [3]:
```
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[3]:
```
[[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 [4]:
```
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[4]:
```
True
```

2.

In [5]:

```
#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 [6]:

```
####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 [7]:

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

Out[7]:

```
30
```

In [8]:

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

Out[8]:

```
333
```

In [9]:

```
#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),0)
```

```
3
68
565
3863
61515
867514
6867610
68185322
644242175
4436061708
25173696048
907798109191
5631933017735
38780348552268
358622896049262
8556180242414099
30295924558305991
```

```
5266526209268306762
47787511464095000022
438473641497075294559
6525925220336208052229
95090362386954153277866
666834875667511743535559
8840393159500945236560255
27471118670821144889823388
712031720090750496109990077
3384775734751265583865339255
12588267058902671932041113022
992596527692782155154451302011
1767343768961591346595647830999
73158555224500748888868932808255
649518909146166223686853284303311
8041278767793470441196915391695411
16433137838277973309741540769033644
598294349401770156377343721465714388
4187444251821062547623247652685837388
85170271844942011680948532017761749955
199054954048078473181748514630000189422
2111900926206496358381048217111833308455
87265133756337516104579427167687342382833
893993575776982269768738555488576153776199
3686445975667706581912278911801990319300588
76418762770406582330687103197715963148293533
370278502498903065898495939955957822747931955
7803846031360504222476979906133675389016468733
60836717982978769180696774888879498958174689122
897318739337096221265681966099759172505608927688
3045798735207846827200348266448049684075910918088
13677123529745920973736521190739908288103529613866
597449212233612023300470877473959495367212759217155
                                   333
True
```

A Python Lecture Series

Lecture 2

by Luca Mingarelli

Lecture 2: split into 2.1 and 2.2

Content:

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

Input/Output

To write in a file:

In [10]:
```
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 [11]:
```
!ls #notice a new file!
```

```
ECB Python Lectures - Lecture 2.1.ipynb a_work_file
```

To read from a file:

In [12]:

```
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 [13]:

```
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 [14]:

```
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 [15]:

```
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 [16]:
```
%%writefile my_new_module.py

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

```
Writing my_new_module.py
```

In [17]:
```
import my_new_module as mnm
mnm.a_complicated_function()
```

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

In [18]:

```
!mkdir MODULES
```

In [19]:

```
%%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 [20]:

```
%%writefile MODULES/__init__.py
from .module2 import function2
```

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

In [21]:

```
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 [22]:
```python
import numpy as np
x = np.array([1,2,3])
# convert list to numpy array object
x
```

Out[22]:
```
array([1, 2, 3])
```

In [23]:
```python
###--- Notice that
x+x
###--- More on this later.
```

Out[23]:
```
array([2, 4, 6])
```

In [24]:
```python
x = np.linspace(0,10,11) # as in Matlab!
x
```

Out[24]:
```
array([  0.,   1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.,  10.])
```

In [25]:
```python
x = np.arange(1.5,10,2) # same as Matlab's [1.5:2:10]
x
```

Out[25]:
```
array([1.5, 3.5, 5.5, 7.5, 9.5])
```

NumPy's number types and associated risk (overflow)

In [26]:
```
x=np.array([0,1])
print("x =",x,"and has dtype",x.dtype)
```

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

In [27]:
```
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 [28]:
```
print("x + 1 =",x + 1,"\t (because dtype is"
      ,x.dtype,"!)")
```

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

In [29]:
```
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[29]:
```
-2
```

Some of NumPy arrays' attributes and methods

In [30]:
```
x.ndim
```
Out[30]:
```
1
```

In [31]:
```
x.shape
```
Out[31]:
```
(2,)
```

In [32]:
```
len(x)
```
Out[32]:
```
2
```

In [33]:
```
x = np.array([[0, 1, 2], [3, 4, 5]])     # 2 x 3 array
```

In [34]:
```
x.shape
```
Out[34]:
```
(2, 3)
```

In [35]:
```
x.mean() ## an ojbject's method
```
Out[35]:
```
2.5
```

In [36]:
```
x.dtype ## data type
```
Out[36]:
```
dtype('int64')
```

In [37]:

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

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

Higher dimensional arrays

In [38]:
```
np.zeros((2,3))
```

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

In [39]:
```
np.ones((2,2))
```

Out[39]:
```
array([[1., 1.],
       [1., 1.]])
```

In [40]:
```
np.eye(3)
```

Out[40]:
```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

In [41]:
```
np.diag(range(1,5))
```

Out[41]:
```
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 [42]:
```
x
```

Out[42]:
```
array([[0, 1, 2],
       [3, 4, 5]])
```

In [43]:
```
print('x[0] = ', x[0])
print('x[1] = ', x[1])
```

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

In [44]:
```
x[0][-1]
```

Out[44]:
```
2
```

In [45]:
```
x[0,-1]
```

Out[45]:
```
2
```

In [46]:
```
x[:,::-1]
```

Out[46]:
```
array([[2, 1, 0],
       [5, 4, 3]])
```

In [47]:
```
x[::-1,:]
```
Out[47]:
```
array([[3, 4, 5],
       [0, 1, 2]])
```
In [48]:
```
## 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 [49]:
```
print('type: ',x.dtype)
x[:,:] = np.pi
x
```

```
type:  int64
```

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

In [50]:
```
y = x.astype(bool)
# y = y.astype(float)
print('type: ',y.dtype)
y[:,:] = np.pi
y
```

```
type:  bool
```

Out[50]:
```
array([[ True,  True,  True],
       [ True,  True,  True]])
```

IMPORTANT 2: Slices return views, NOT copies!

In [51]:
```
x_slice = x[:2,:2]
x_slice
```

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

In [52]:

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

Out[52]:

```
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 [53]:
```
x[:] = 3
x_slice_copy = x[:2, :2].copy()
x_slice_copy
```

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

In [54]:
```
x_slice_copy[:] = 2
x
```

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

Reshaping

In [55]:
```
x = np.array([1, 2, 3])
# reshape to row vector
x.reshape((1, 3))
```

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

In [56]:
```
# reshape to row vector
x.reshape((3, 1))
```

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

Concatenation

In [57]:
```
x = np.array([1, 2, 3])
y = np.array([4, 5, 6])
Z = np.concatenate([x, y])
Z
```

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

In [58]:
```
# for multidimensional arrays as well


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

Out[58]:
```
array([1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6])
```

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

In [59]:
```
x = np.array([1, 2, 3])
Z = np.array([[4, 5, 6],
                 [7, 8, 9]])
# vertically stack the arrays
np.vstack([x, Z])
```

Out[59]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

In [60]:
```
# horizontally stack the arrays
y = np.array([[456],
              [789]])
np.hstack([Z, y])
```

Out[60]:
```
array([[  4,   5,   6, 456],
       [  7,   8,   9, 789]])
```

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

Splitting

In [61]:

```
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 [62]:

```
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 [63]:

```
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 [64]:

```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 [65]:
```
x = np.arange(-5,5)
np.abs(x)
```

Out[65]:
```
array([5, 4, 3, 2, 1, 0, 1, 2, 3, 4])
```

In [66]:
```
x = np.linspace(1,2,1000)
%timeit [1/x[n] for n in range(len(x))]
%timeit 1/x
```

```
400 µs ± 16.8 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
4.8 µs ± 89.4 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 [67]:

```
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)](https://docs.scipy.org/doc/scipy/reference/special.html#module-scipy.special).

Even when computing aggregates: use NumPy's functions.

In [68]:

```
x = np.arange(1000)
%timeit sum(x)
%timeit x.sum()
%timeit np.sum(x) # the same as above!
```

```
122 µs ± 1.51 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
5.38 µs ± 98 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
7.42 µs ± 153 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

In [69]:

```
%timeit max(x)
%timeit x.max()
%timeit np.max(x) # the same as above!
```

```
88.3 µs ± 1.69 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
6.15 µs ± 122 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
7.68 µs ± 78.2 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 [70]:
```
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[70]:
```
array([ 6, 22, 38, 54])
```

In [71]:
```
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 [72]:
```
x = np.array([1,2,3,4,5,6])
x>3
```

Out[72]:
```
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 [73]:
```
print(x)
print(x>3)
x[x>3]
```

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

Out[73]:
```
array([4, 5, 6])
```

Fancy indexing

In [74]:
```
l = np.array([1,2,3,4,5])
l[[0,2]]
```

Out[74]:
```
array([1, 3])
```

In [75]:
```
l[[-1,0,-2]]
```

Out[75]:
```
array([5, 1, 4])
```

Moreover:

In [76]:
```
ind = np.array([[3, 0],
                [4, 1]])
l[ind]
```

Out[76]:
```
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 [77]:
```
x = np.array([0,1,2])
print(x+3)
print(x+np.array([3,3,3]))
```

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

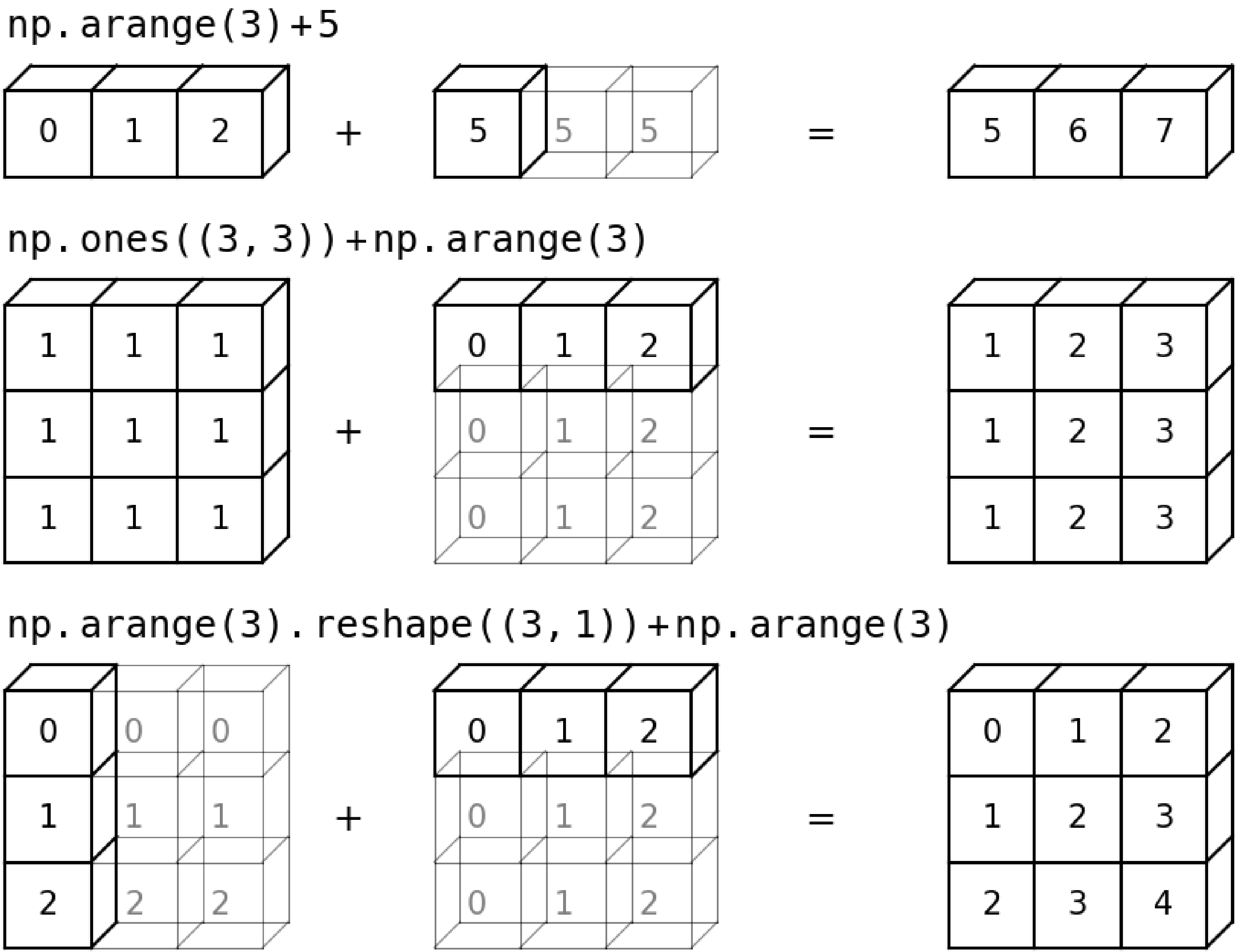
In [78]:
```
M = np.ones((3, 3))
M+x
```

Out[78]:
```
array([[1., 2., 3.],
       [1., 2., 3.],
       [1., 2., 3.]])
```

In [79]:
```python
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 [80]:
```
A = np.arange(10)
B = A
B[0]= 100
A
```

Out[80]:
```
array([100,   1,   2,   3,   4,   5,   6,   7,   8,   9])
```

In [81]:
```
A = np.arange(10)
B = A.copy()
B[0]=100
A
```

Out[81]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Pandas

Main data structures in Pandas:

- Series
- DataFrames

Series

In [82]:

```
import pandas as pd
#from pandas import Series
pd.Series([0.1, 0.2, 0.3, 0.4, 0.5])
```

Out[82]:

```
0    0.1
1    0.2
2    0.3
3    0.4
4    0.5
dtype: float64
```

In [83]:

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

Out[83]:

```
a    0.1
b    0.2
c    0.3
d    0.4
e    0.5
dtype: float64
```

Be careful however about operations between different Series

In [84]:
```
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 [85]:
```
s1 + s2
```

Out[85]:
```
a    1.1
b    3.2
c    5.3
dtype: float64
```

In [86]:
```
s1 + s3
```

Out[86]:
```
a    NaN
b    NaN
c    2.4
d    NaN
e    NaN
dtype: float64
```

In [87]:
```python
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[87]:
```
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 [88]:
```
print(type(s3.values))
s3.values
```

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

Out[88]:
```
array([0.1, 1.2, 2.3])
```

In [89]:
```
s3.index = ['First', 'Second', 'Third']
print(s3)
s3.index[1]
```

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

Out[89]:
```
'Second'
```

In [90]:
```
s = pd.Series([10,20,30],
              index=[13,2,89])
## Now indexing is ambiguous!
s[2]
# s[0]  # Error
```

Out[90]:
```
20
```

In [91]:

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

Out[91]:

```
13    10
2     20
dtype: int64
```

In [92]:

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

Out[92]:

```
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 [93]:
```
df = pd.DataFrame(np.array([[1,2],[3,4]]))
df
```

Out[93]:

	0	1
0	1	2
1	3	4

In [94]:
```
df.columns = ['col1','col2']
df.index = ['row1','row2']
df
```

Out[94]:

	col1	col2
row1	1	2
row2	3	4

In [95]:
```
pd.DataFrame(np.array([[1,2],[3,4]]),columns=['col1','col2'], index = ['row1','row
2'])
```

Out[95]:

	col1	col2
row1	1	2
row2	3	4

In [96]:

```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[96]:

	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 [97]:
```
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 [98]:
```
df.describe()
```

Out[98]:

	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 [99]:
```
df.sum() ### NaN automatically diregarded!
```

Out[99]:
```
col1    10.0
col2     6.0
col3     2.0
dtype: float64
```

Selecting columns ...

In [100]:
```
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 [101]:
```
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 [102]:
```
df[2:4]
```

Out[102]:

	col1	col2	col3
2	2.0	3.0	NaN
3	3.0	NaN	NaN

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

In [103]:
```
df[2:4][['col2']]
```

Out[103]:

	col2
2	3.0
3	NaN

...deleting columns...

In [104]:
```
df2 = df.copy() #Recall the `issue` in numpy?
del df2['col2']
df2
```

Out[104]:

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

In [105]:
```
df2.pop('col1')
```

Out[105]:
```
0    0.0
1    1.0
2    2.0
3    3.0
4    4.0
Name: col1, dtype: float64
```

In [106]:
```
df2
```

Out[106]:

	col3
0	2.0
1	NaN
2	NaN
3	NaN
4	NaN

In [107]:

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

Out[107]:

	col2
0	1.0
1	2.0
2	3.0
3	NaN
4	NaN

In [108]:

```
df
```

Out[108]:

	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 [109]:
```
# None + 1
np.nan +1
```

Out[109]:
```
nan
```

In [110]:
```
pd.Series([1, np.nan, 2, None])
## Notice both the mapping None -> NaN
## as well as int -> float
```

Out[110]:
```
0    1.0
1    NaN
2    2.0
3    NaN
dtype: float64
```

Detection of missing data

In [111]:
```
df.count() #count non-missing elements
```

Out[111]:
```
col1     5
col2     3
col3     1
dtype: int64
```

In [112]:
```
df.notnull() # opposite: df.isnull()
```

Out[112]:

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

In [113]:
```
df['col2'][df['col2'].notnull()]
```

Out[113]:
```
0    1.0
1    2.0
2    3.0
Name: col2, dtype: float64
```

Dropping missing values

In [114]:

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

Out[114]:

	col1	col2	col3
0	0.0	1.0	2.0

In [115]:

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

Out[115]:

	col1
0	0.0
1	1.0
2	2.0
3	3.0
4	4.0

Filling missing values

In [116]:
```
df.fillna(0)
```

Out[116]:

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

In [117]:
```
# forward-fill
df.fillna(method='ffill') #bfill for back-fill
```

Out[117]:

	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 [118]:
```
# change axis
df.fillna(method='ffill',axis = 1)
```

Out[118]:

	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 [119]:

```
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[119]:

	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 [120]:

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

Out[120]:

	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 [121]:
```
import random as rnd
rnd.random() ## Uniform in [0,1)
```

Out[121]:
```
0.42632677014265585
```

In [122]:
```
# uniform in range
rnd.uniform(1,10)
```

Out[122]:
```
5.054553636111223
```

In [123]:
```
#simulate die
rnd.randint(1,6)
```

Out[123]:
```
6
```

In [124]:
```
greetings = ['Hi', 'Hello', 'Welcome!', 'Hola']
rnd.choice(greetings)
```

Out[124]:
```
'Hola'
```

In [125]:
```
#Simulate wheel spins
colors = ['R', 'B', 'G'] # Red, Black and Green
rnd.choices(colors, weights=[18,18,2] ,k =10)
```

Out[125]:
```
['B', 'B', 'B', 'R', 'B', 'B', 'R', 'G', 'R', 'G']
```

In [126]:

```
# Shuffle cards
deck = list(range(1,53)) ## 52 cards
rnd.shuffle(deck)
print(deck)
```

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

In [127]:

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

```
[9, 49, 16, 36, 48]
```

NumPy random generators

In [128]:
```
import numpy.random as rnd
```

In [129]:
```
## UNIFORM
print(rnd.rand(3,4))
```

```
[[0.93695749 0.274056   0.53011925 0.97679767]
 [0.7436092  0.03851414 0.27102461 0.41150143]
 [0.27627046 0.35089043 0.99386763 0.03871757]]
```

In [130]:
```
## STANDARD NORMAL
print(rnd.randn(3,4))
```

```
[[-1.52295668 -0.13276398 -0.5029103   1.83349    ]
 [-1.37253978  0.41098531  0.92069655 -0.33936542]
 [-0.04344644  2.43514737  1.31225644  1.47672998]]
```

In [131]:
```
## UNIFORM INTEGERS
print(rnd.randint(0,100,(3,4)))
```

```
[[16 17 36 22]
 [61 66 21 13]
 [ 9 91  2 71]]
```

In [132]:
```
rnd.shuffle(deck)
print(deck)
```

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

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.

End of Lecture 2.1