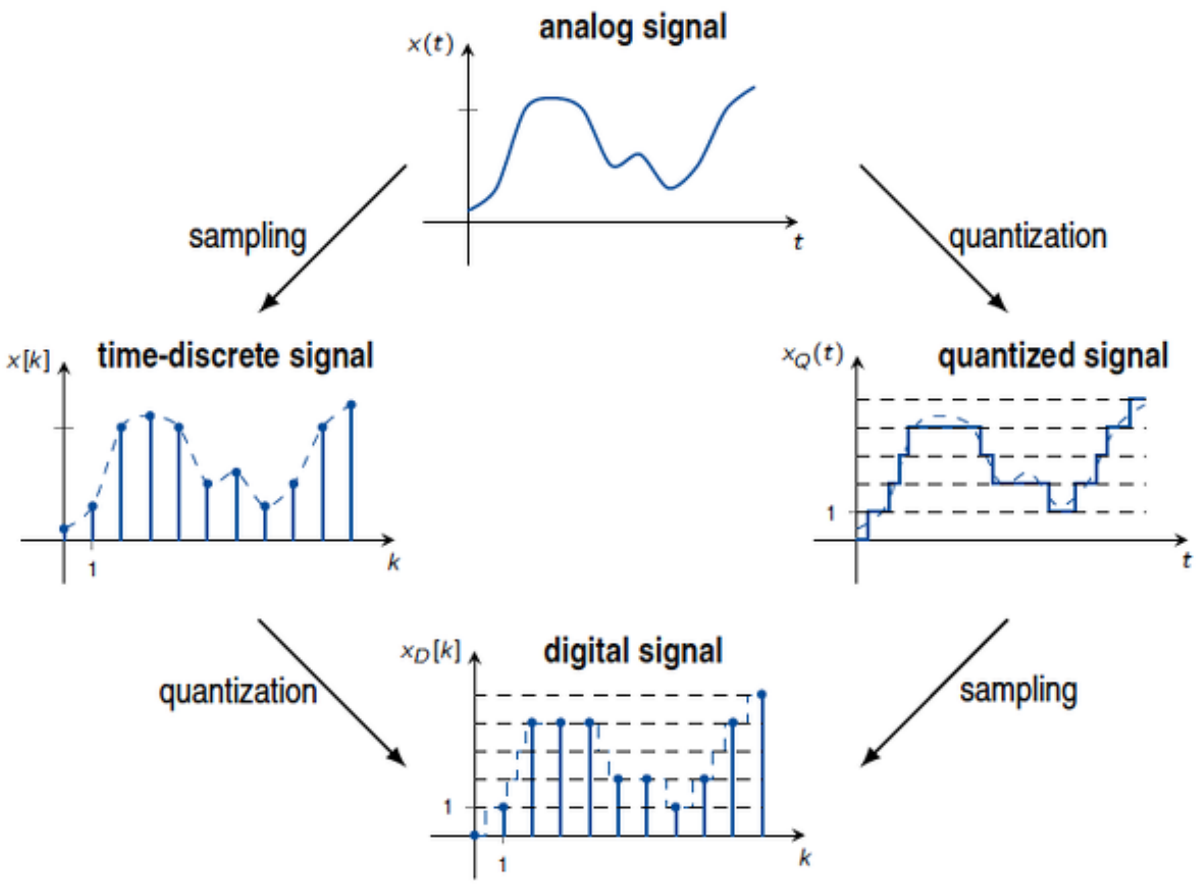
In [1]:
```
%%html
<link rel="stylesheet" type="text/css" href="rise.css" />
```

Multi-dimensional data arrays

Digital representation of data



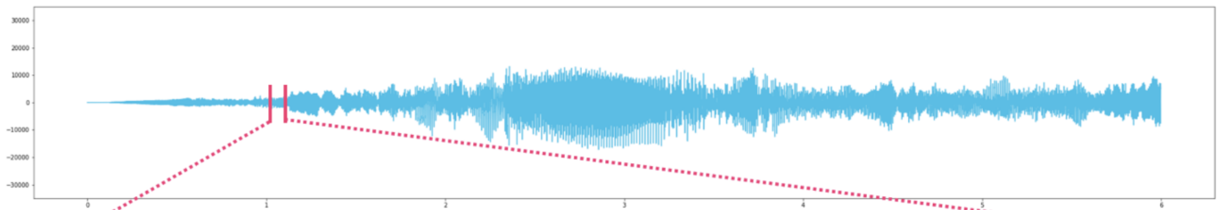
In [2]:
```
digital_signal = [0, 1, 4, 4, 4, 2, 2, 1, 2, 4, 5]
sample_interval = 1
```

Multi-dimensional data arrays

The vast majority of the data that you will work with will be stored as arrays of values in some number of dimensions.

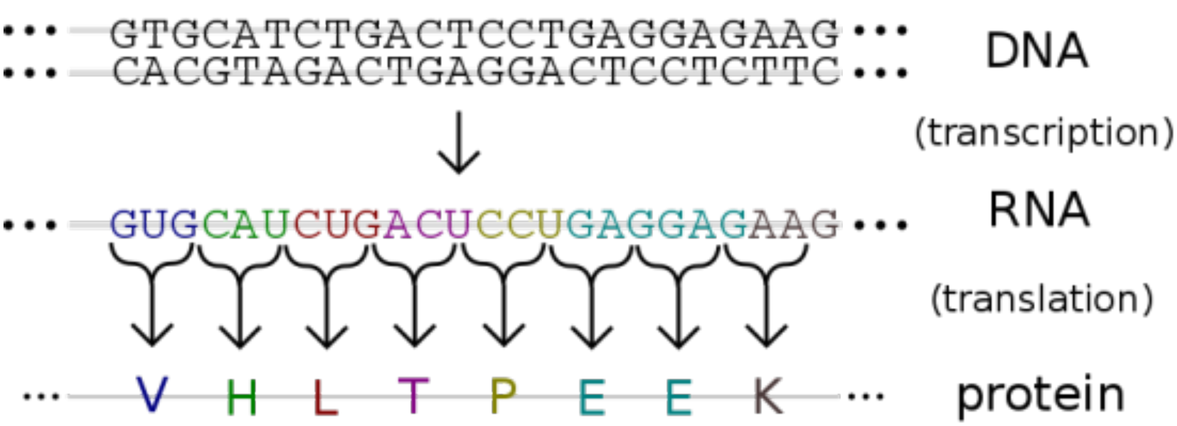
Let's consider some examples.

1-D: Time series at regular sample intervals

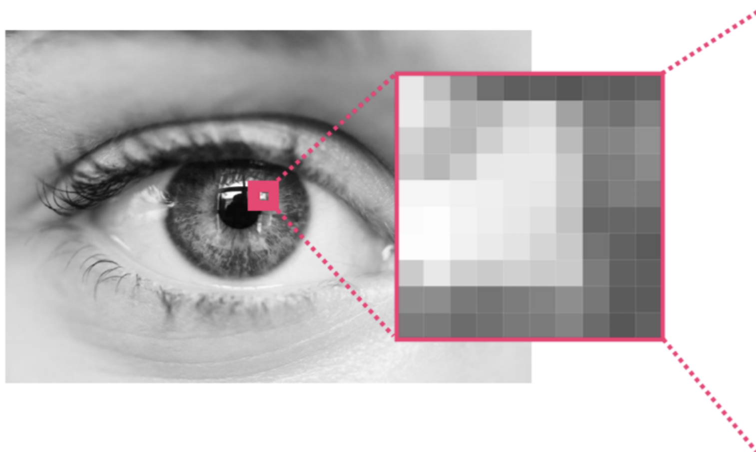


326	217	142	73	27	7	14	38	65	90	101	90	74	37	-12	-77	-154	-232	-304	-355	-433

1-D: DNA sequence



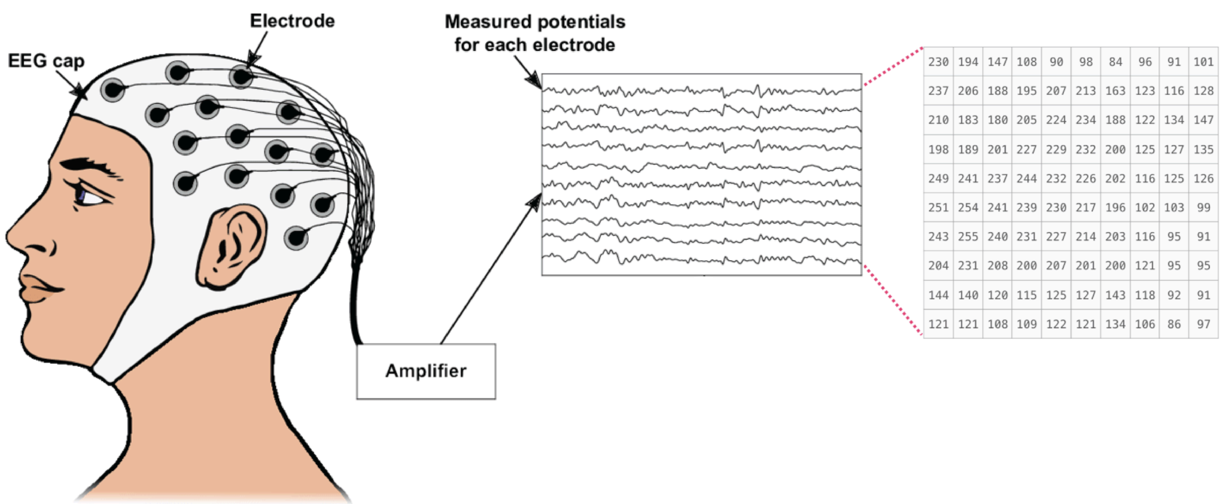
2-D: Grayscale image



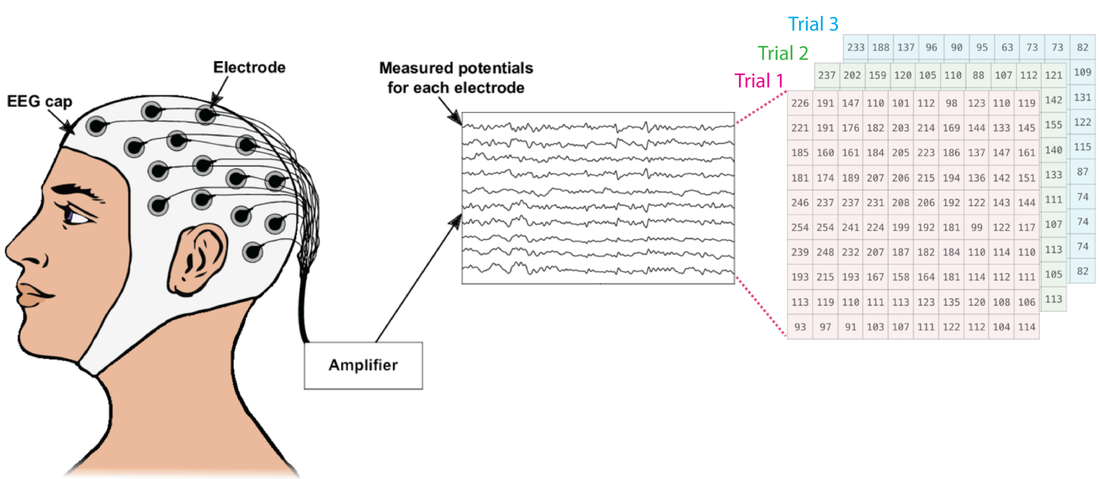
230	194	147	108	90	98	84	96	91	101
237	206	188	195	207	213	163	123	116	128
210	183	180	205	224	234	188	122	134	147
198	189	201	227	229	232	200	125	127	135
249	241	237	244	232	226	202	116	125	126
251	254	241	239	230	217	196	102	103	99
243	255	240	231	227	214	203	116	95	91
204	231	208	200	207	201	200	121	95	95
144	140	120	115	125	127	143	118	92	91
121	121	108	109	122	121	134	106	86	97

3-D: Color image

2-D: EEG *time series* for multiple *locations*

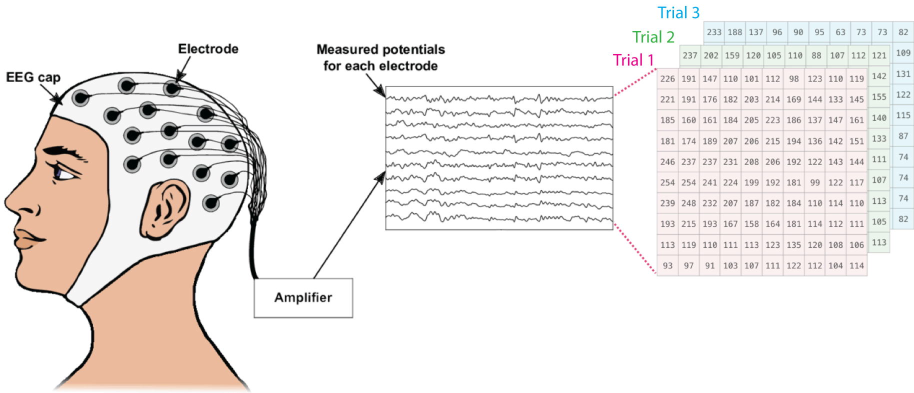


3-D: EEG *time series* for multiple *locations, trials*



N-D: EEG *time series* for multiple *locations, trials, subjects, conditions, ...*

Hard to visualize, but easy to use mathematically and in code.



NumPy

Without NumPy we would ***NOT USE PYTHON*** for scientific computing or data analysis.

Here's a quick comparison of NumPy with Python lists to give you an idea of why it is so essential.

Python lists	NumPy arrays
Very inefficient and slow for large arrays.	Highly optimized C code behind the scenes.
Nested lists are a nightmare for multi-dimensional data arrays.	N-dimensional array syntax is simple and easy.
Great for small arrays of arbitrary types of objects.	Only allows arrays of the same type of object.

Install NumPy

In a cmd shell or terminal run:

```
conda activate neu365
pip install numpy
```

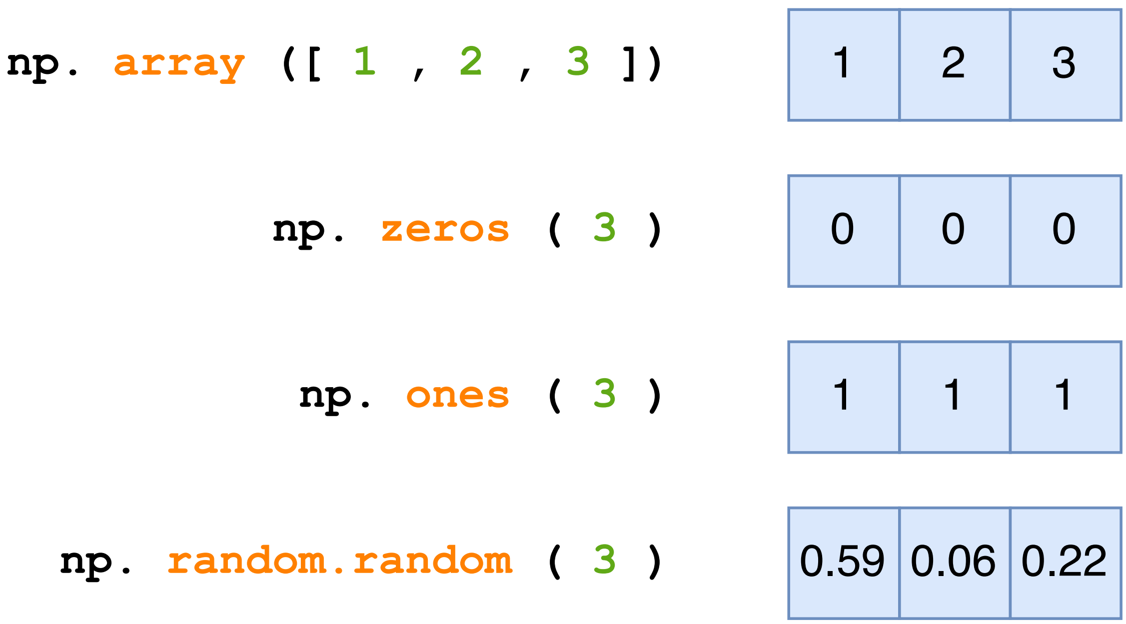
NumPy 1-D arrays

Learning goals

- You will be able to initialize arrays.
- You will be able to index/slice into arrays.
- You will be able to index/slice with logical masks.
- You will be able to compute array statistics.
- You will be able to do array math.

NumPy 1-D array initialization

NumPy arrays generally *cannot change size*, so you must *create the array for the size you need*.



In [3]:
```
import numpy

numpy.array([1, 2, 3])
```

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

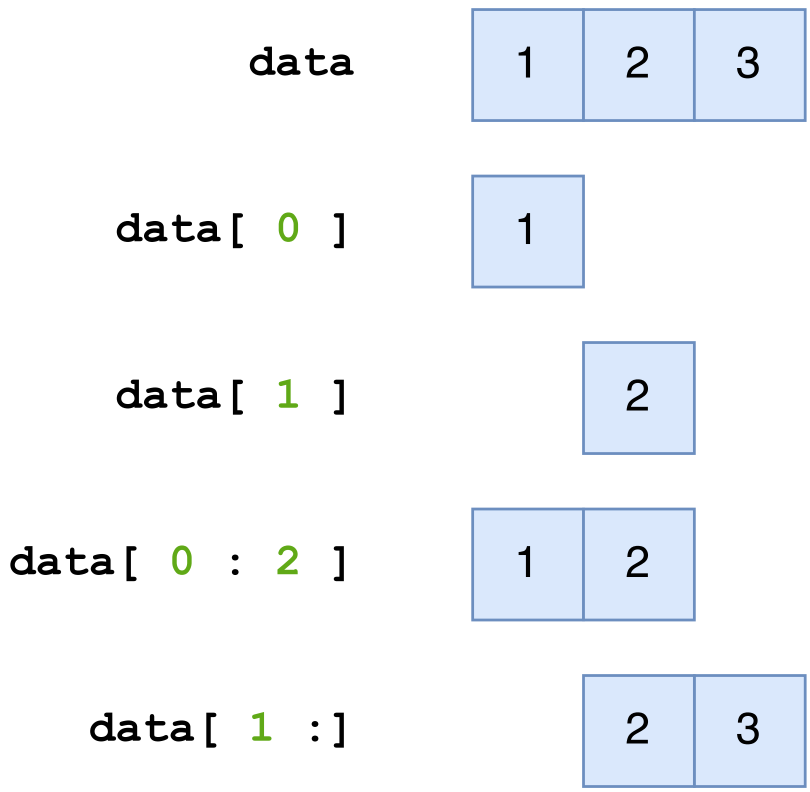
In [4]:
```
import numpy as np

np.array([1, 2, 3])
```

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

In [5]:
```
np.ones(3), np.zeros(3), np.random.random(3)
```

Out[5]:
```
(array([1., 1., 1.]),
 array([0., 0., 0.]),
 array([0.326398  , 0.7878623 , 0.90420253]))
```

Index and slice just like a Python list



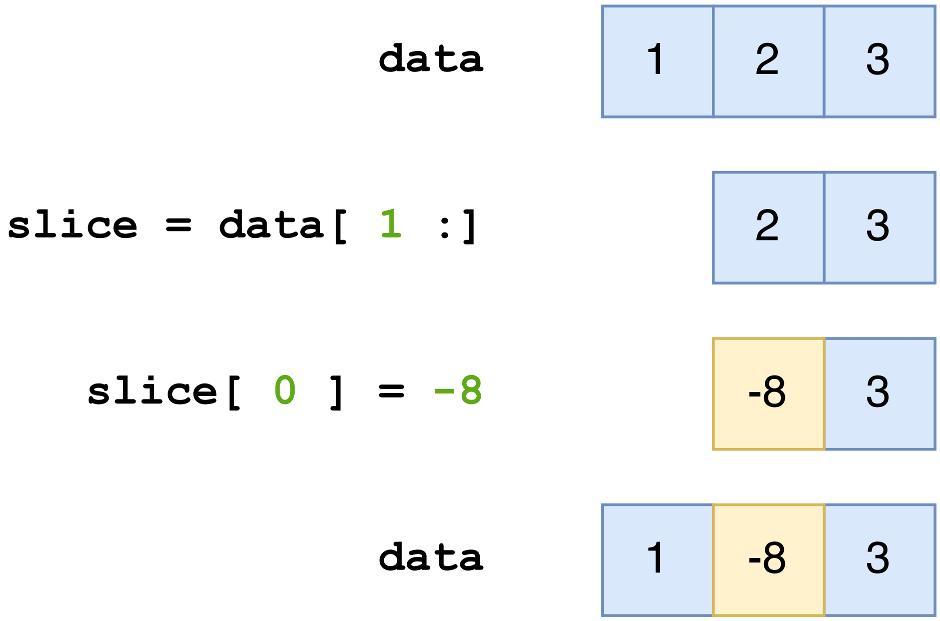
In [6]:
```
data = np.array([1, 2, 3])

data
```

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

In [7]:
```
data[0], data[1], data[0:2], data[1:]
```

Out[7]:
```
(1, 2, array([1, 2]), array([2, 3]))
```

!!! A slice into a NumPy array can be assigned to a variable and it will still **reference the original array data** (not a copy)



In [8]:
```
myarray = np.array([1, 2, 3])
array_slice = myarray[1:]

# reference to data in myarray
array_slice
```

Out[8]:
```
array([2, 3])
```

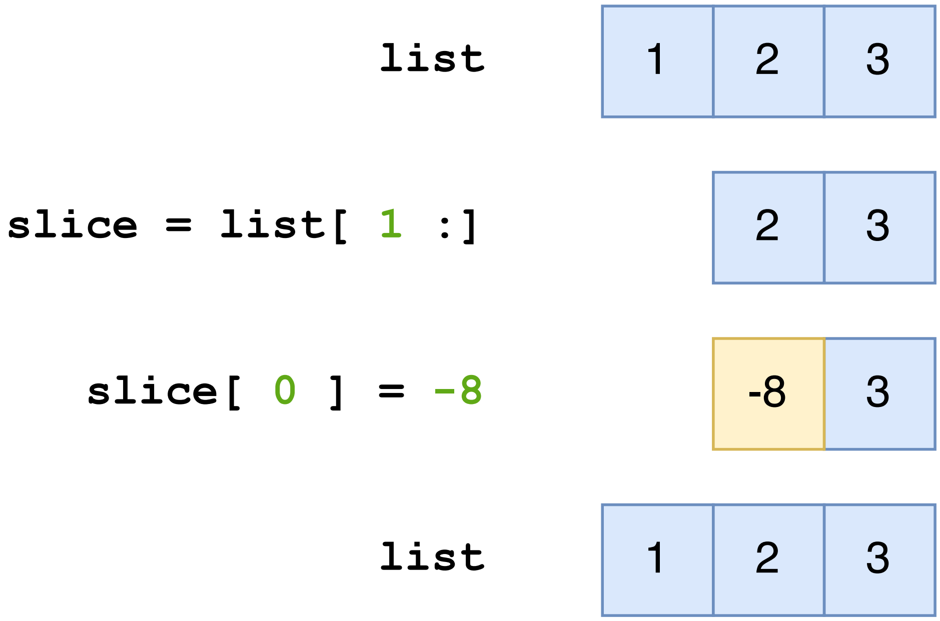
In [9]:
```
array_slice[0] = -8
array_slice
```

Out[9]:
```
array([-8,  3])
```

In [10]:
```
myarray
```

Out[10]:
```
array([ 1, -8,  3])
```

In contrast, assigning a slice of a Python list to a variable **creates a copy**



In [11]:
```
mylist = [1, 2, 3]
list_slice = mylist[1:]

# copy of data from mylist
list_slice
```

Out[11]:
```
[2, 3]
```

In [12]:
```
list_slice[0] = -8
list_slice
```

Out[12]:
```
[-8, 3]
```

In [13]:
```
mylist
```

Out[13]:
```
[1, 2, 3]
```

This means you can pass a slice to a function and still mutate the original array!

In [14]:
```
myarray = np.array([1, 2, 3])

myslice = myarray[1:]

def change_array(arr):
    arr[0] = 100

change_array(myslice)

myarray
```

Out[14]:
```
array([  1, 100,   3])
```

What if you want a copy of a slice into a NumPy array?

In [15]:
```
myarray = np.array([1, 2, 3])

ref_slice = myarray[1:]

copy_slice = myarray[1:].copy()

print(' ref = ', ref_slice)
print('copy = ', copy_slice)
```

```
 ref =  [2 3]
copy =  [2 3]
```

In [16]:
```
ref_slice[0] = -50
copy_slice[0] = 100

print(' ref = ', ref_slice)
print('copy = ', copy_slice)
```

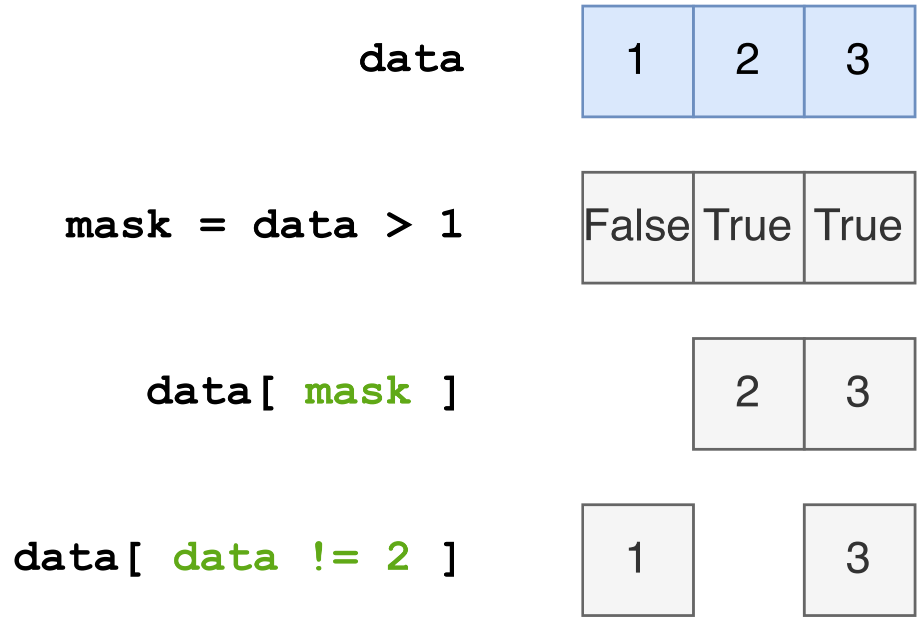
```
 ref =  [-50   3]
copy =  [100   3]
```

What happened to `myarray`?

In [17]:
```
myarray
```

Out[17]:
```
array([  1, -50,   3])
```

NumPy also let's you slice with logical arrays!



In [18]:
```
data = np.array([1, 2, 3])
mask = data > 1

mask
```

Out[18]:
```
array([False,  True,  True])
```

In [19]:
```
data[mask]
```

Out[19]:
```
array([2, 3])
```

In [20]:
```
data[data != 2]
```

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

Array statistics are a breeze with NumPy!

In [21]:
```
data = np.array([1, 2, 3])

data.mean()
```

Out[21]:
```
2.0
```

In [22]:
```
data.min(), data.max(), data.sum(), data.prod()
```

Out[22]:
```
(1, 3, 6, 6)
```

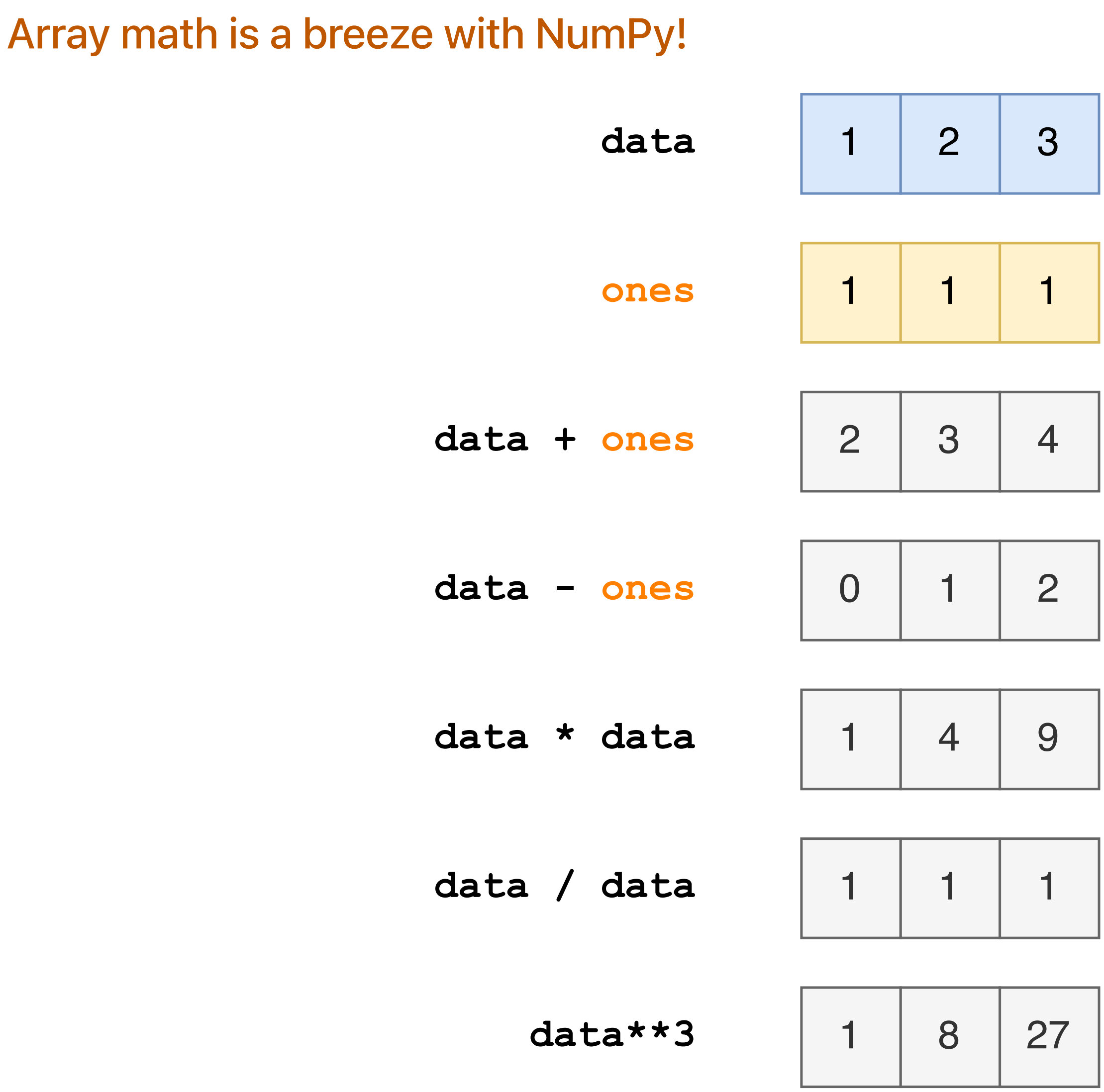
In [23]:
```
# variance and standard deviation
```

```
data.var(), data.std()
```

Out[23]: `(0.6666666666666666, 0.816496580927726)`

In [24]:
```
np.mean(data)
```

Out[24]: `2.0`



In [25]:
```
data = np.array([1, 2, 3])
ones = np.ones(3)
```

In [26]:
```
print(data + ones)
print(data - ones)
print(data * data)
print(data / data)
print(data**3)
```

```
[2. 3. 4.]
[0. 1. 2.]
[1 4 9]
[1. 1. 1.]
[ 1  8 27]
```

In [27]:
```
2 * (data + 1)
```

Out[27]:
```
array([4, 6, 8])
```

Some useful array functions: arange, linspace, logspace

In [28]:
```
# start, stop, step

np.arange(2, 12, 2)
```

Out[28]:
```
array([ 2,  4,  6,  8, 10])
```

In [29]:
```
# 5 values from 0 to 1
# evenly spaced on a linear scale

np.linspace(0, 1, 5)
```

Out[29]:
```
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

In [30]:
```
# 5 values from 10^-1 to 10^1
# evenly spaced on a log scale

np.logspace(-1, 1, 5)
```

Out[30]:
```
array([ 0.1       ,  0.31622777,  1.        ,  3.16227766, 10.        ])
```

NumPy 2-D arrays

Learning goals

- You will be able to initialize arrays.
- You will be able to index/slice into arrays.
- You will be able to index/slice with logical masks.
- You will be able to compute array statistics.
- You will be able to do array math.
- You will understand broadcasting.

NumPy 2-D array initialization

```
np. array ([[ 1 , 2 ],[ 3 , 4 ]])
```

1	2
3	4

```
np. zeros ([ 2 , 3 ])
```

0	0	0
0	0	0

```
np. ones ([ 2 , 3 ])
```

1	1	1
1	1	1

```
np. random.random ([ 2 , 3 ])
```

0.59	0.06	0.22
0.37	0.75	0.63

In [31]:
```
np.zeros([2, 3])
```

Out[31]:
```
array([[0., 0., 0.],
       [0., 0., 0.]])
```

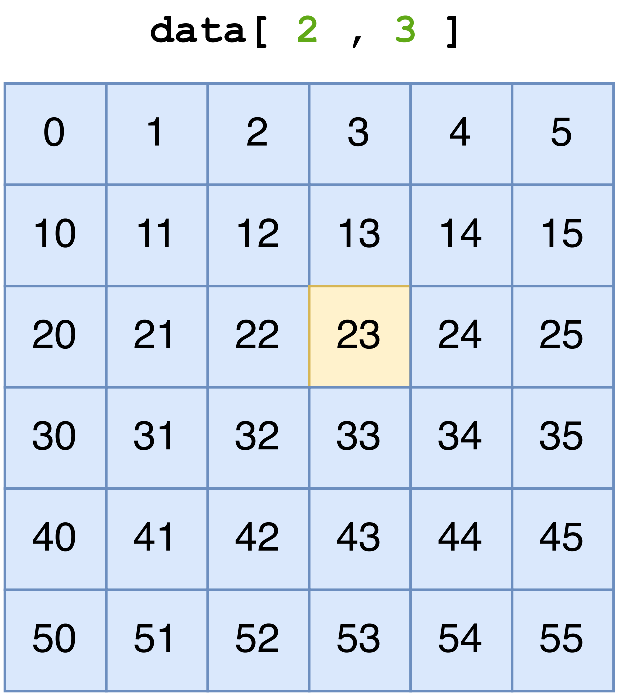
In [32]:
```
np.ones([2,3])
```

Out[32]:
```
array([[1., 1., 1.],
       [1., 1., 1.]])
```

In [33]:
```
np.random.random([2,3])
```

Out[33]:
```
array([[0.63464258, 0.80357956, 0.02969486],
       [0.94341755, 0.77258912, 0.04859342]])
```

NumPy 2-D array indexing/slicing



In [34]:
```
# to understand this line
# see broadcasting below
data = np.arange(6).reshape([1,6]) \
     + np.arange(0, 60, 10).reshape([6,1])
data
```

Out[34]:
```
array([[ 0,  1,  2,  3,  4,  5],
       [10, 11, 12, 13, 14, 15],
       [20, 21, 22, 23, 24, 25],
       [30, 31, 32, 33, 34, 35],
       [40, 41, 42, 43, 44, 45],
       [50, 51, 52, 53, 54, 55]])
```

In [35]:
```
data[2,3]
```

Out[35]:
```
23
```

NumPy 2-D array indexing/slicing

data[0 , 3 : 5]

0	1	2	**3**	**4**	5
10	11	12	13	14	15
20	21	22	23	24	25
30	31	32	33	34	35
40	41	42	43	44	45
50	51	52	53	54	55

In [36]:
```
data[0,3:5]
```

Out[36]:
```
array([3, 4])
```

NumPy 2-D array indexing/slicing

data[4 :, 4 :]

0	1	2	3	4	5
10	11	12	13	14	15
20	21	22	23	24	25
30	31	32	33	34	35
40	41	42	43	**44**	**45**
50	51	52	53	**54**	**55**

In [37]:
```
data[4:,4:]
```

Out[37]:
```
array([[44, 45],
       [54, 55]])
```

NumPy 2-D array indexing/slicing

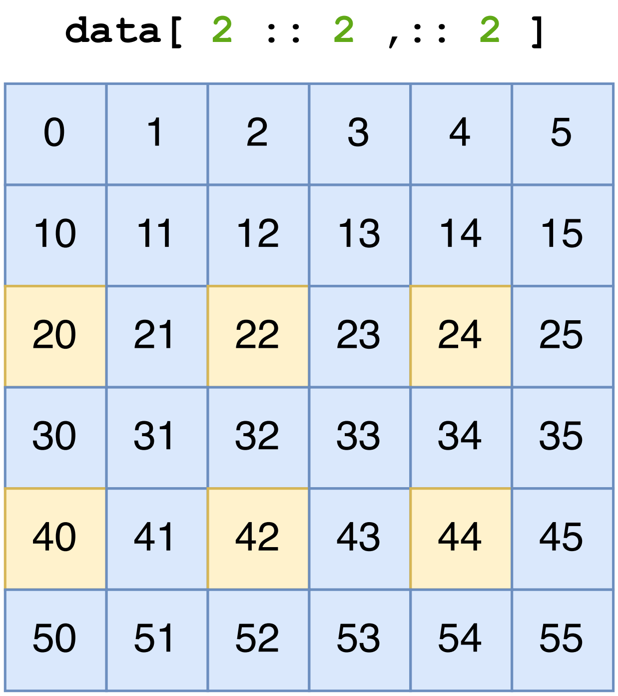
data[:, 2]

0	1	2	3	4	5
10	11	12	13	14	15
20	21	22	23	24	25
30	31	32	33	34	35
40	41	42	43	44	45
50	51	52	53	54	55

In [38]:
```
data[:,2]
```

Out[38]:
```
array([ 2, 12, 22, 32, 42, 52])
```

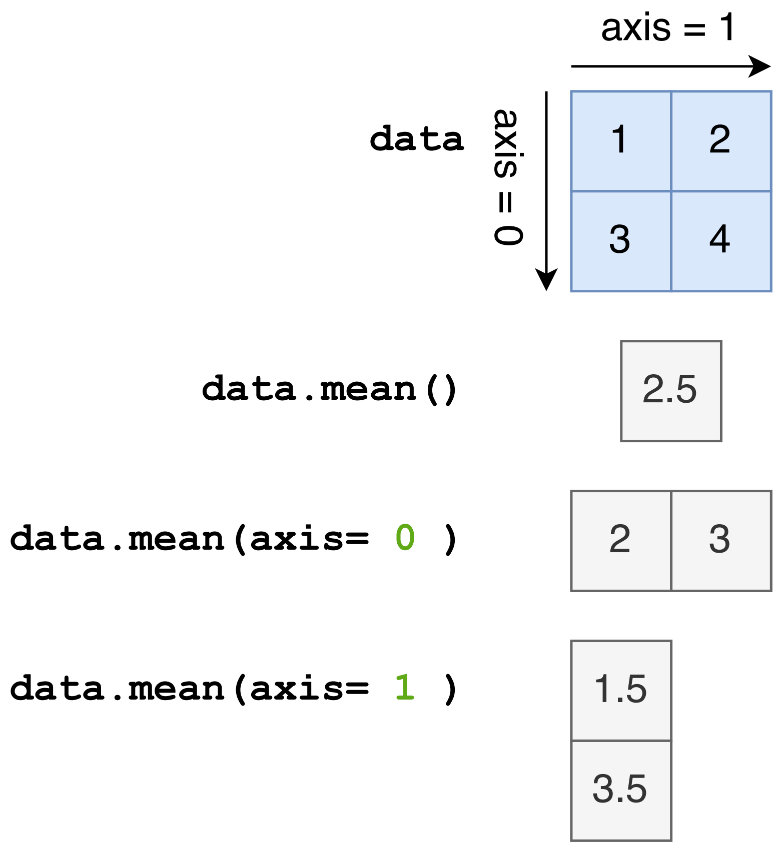
NumPy 2-D array indexing/slicing



In [39]:
```
data[2::2,::2]
```

Out[39]:
```
array([[20, 22, 24],
       [40, 42, 44]])
```

NumPy 2-D array statistics



In [40]:
```python
data = np.array([[1,2],[3,4]])
data
```

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

In [41]:
```python
data.max()
```

Out[41]:
```
4
```

In [42]:
```python
data.max(axis=0)
```

Out[42]:
```
array([3, 4])
```

In [43]:
```python
data.max(axis=1)
```

Out[43]:
```
array([2, 4])
```

NumPy 2-D array math (basically the same as 1-D)

data $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$

ones $\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$

data + ones $\begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix}$

data2** $\begin{bmatrix} 1 & 4 \\ 9 & 16 \end{bmatrix}$

In [44]:
```
data = np.array([[1,2],[3,4]])
ones = np.ones([2,2])

data, ones
```

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

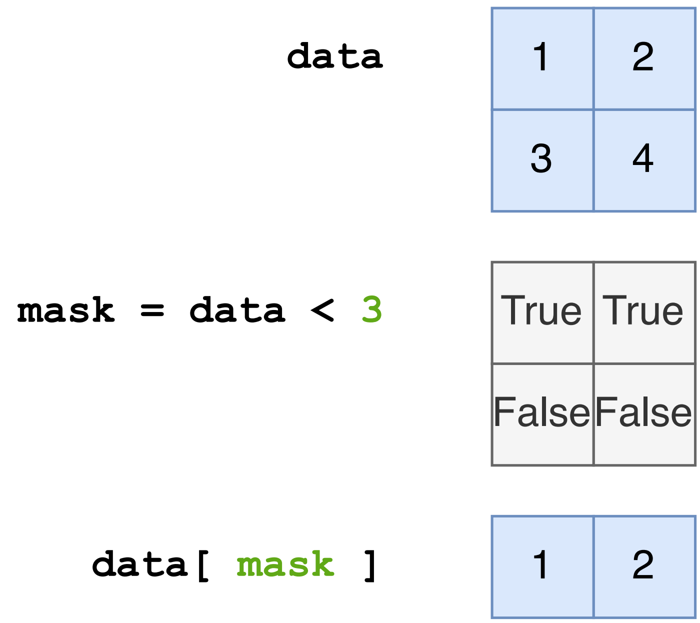
In [45]:
```
data + ones
```

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

In [46]:
```
data**2
```

Out[46]:
```
array([[ 1,  4],
       [ 9, 16]])
```

NumPy 2-D logical indexing **(basically the same as 1-D)**



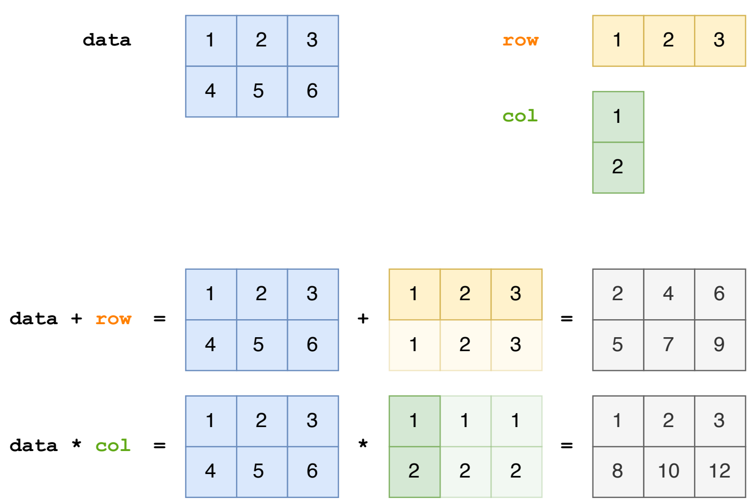
In [47]:
```
data = np.array([[1,2],[3,4]])
mask = data < 3

data, mask
```

Out[47]:
```
(array([[1, 2],
        [3, 4]]),
 array([[ True,  True],
        [False, False]]))
```

In [48]:
```
data[mask]
```

Out[48]:
```
array([1, 2])
```

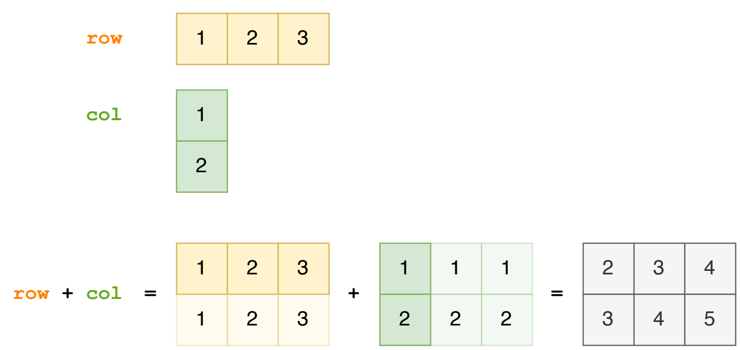
Broadcasting



In [49]:
```
data = np.array([[1,2,3],[4,5,6]])
row = np.array([[1,2,3]])
col = np.array([[1],[2]])

data + row, data * col
```

Out[49]:
```
(array([[2, 4, 6],
        [5, 7, 9]]),
 array([[ 1,  2,  3],
        [ 8, 10, 12]]))
```

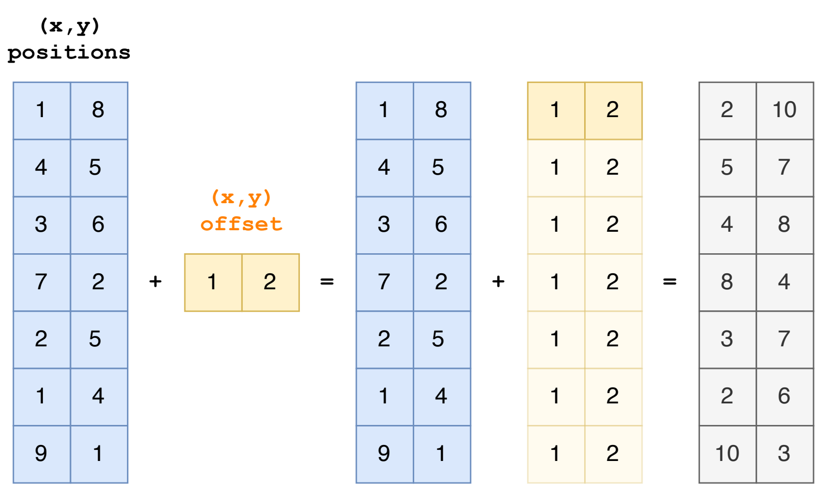
Broadcasting



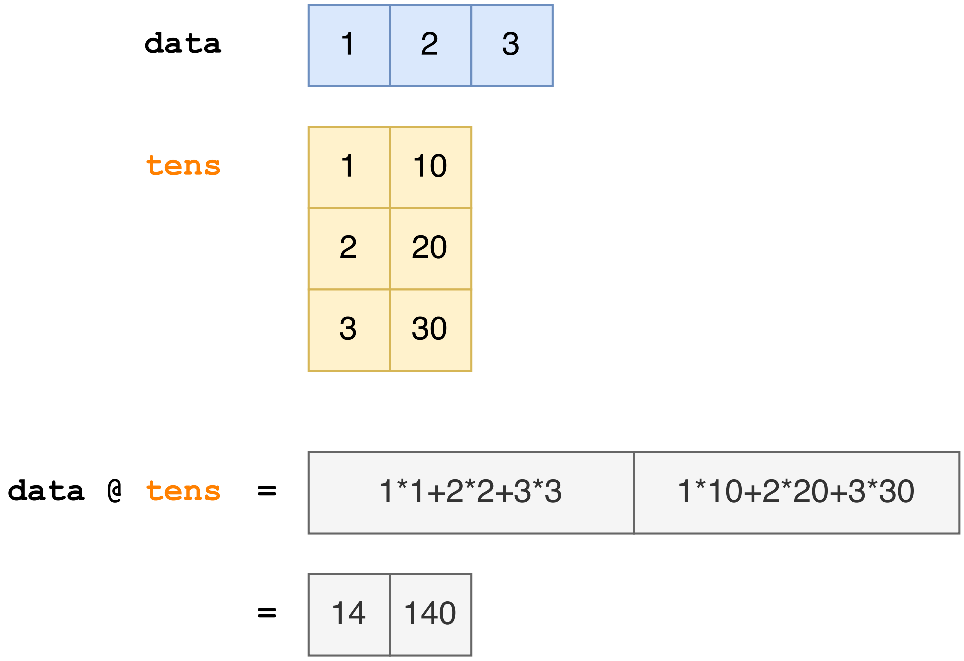
In [50]:
```
row = np.array([[1,2,3]])
col = np.array([[1],[2]])

row + col
```

Out[50]:
```
array([[2, 3, 4],
       [3, 4, 5]])
```

Broadcasting



Matrix multiplication



In [51]:
```
data = np.array([1,2,3])
tens = np.array([[1,10],[2,20],[3,30]])

data, tens
```

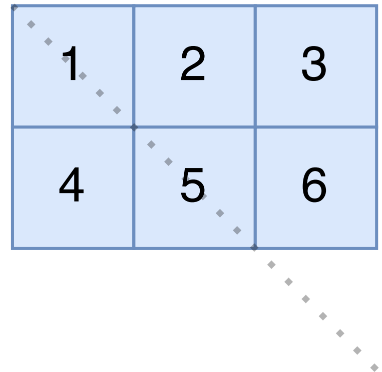
Out[51]:
```
(array([1, 2, 3]),
 array([[ 1, 10],
        [ 2, 20],
        [ 3, 30]]))
```

In [52]:
```
data @ tens
```

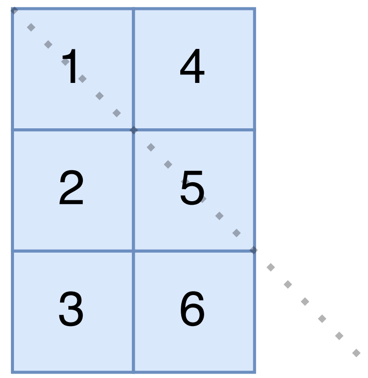
Out[52]:
```
array([ 14, 140])
```

Transpose

data



data.T



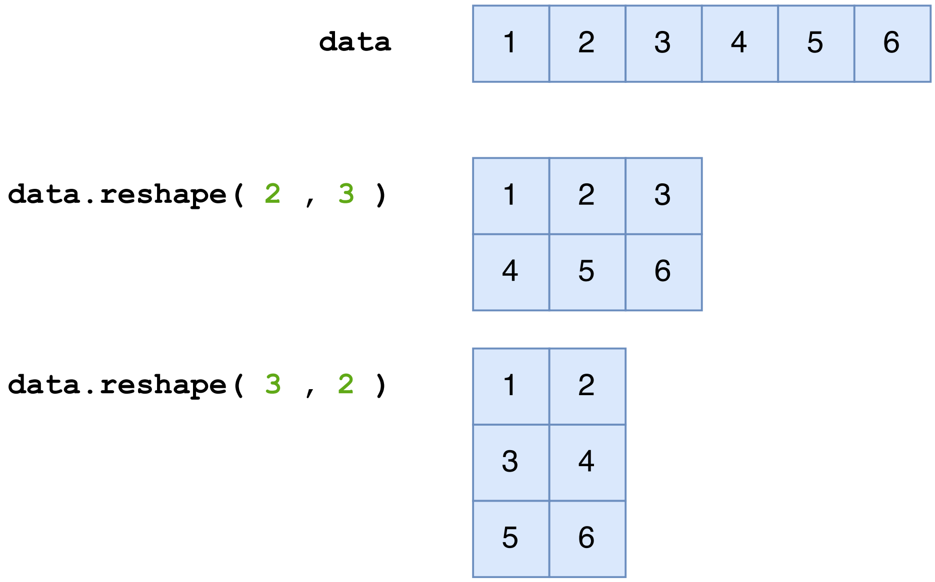
In [53]:
```
data = np.array([[1,2,3],[4,5,6]])
data
```

Out[53]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [54]:
```
data.T
```

Out[54]:
```
array([[1, 4],
       [2, 5],
       [3, 6]])
```

Reshape



In [55]:
```
data = np.arange(1,7)
data
```

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

In [56]:
```
data.reshape(2,3)
```

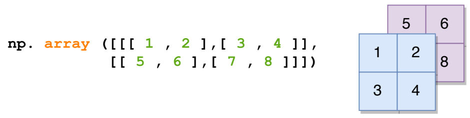
Out[56]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

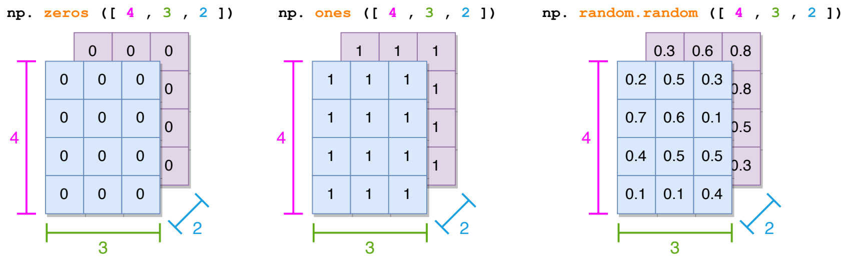
NumPy N-D arrays

Learning goals

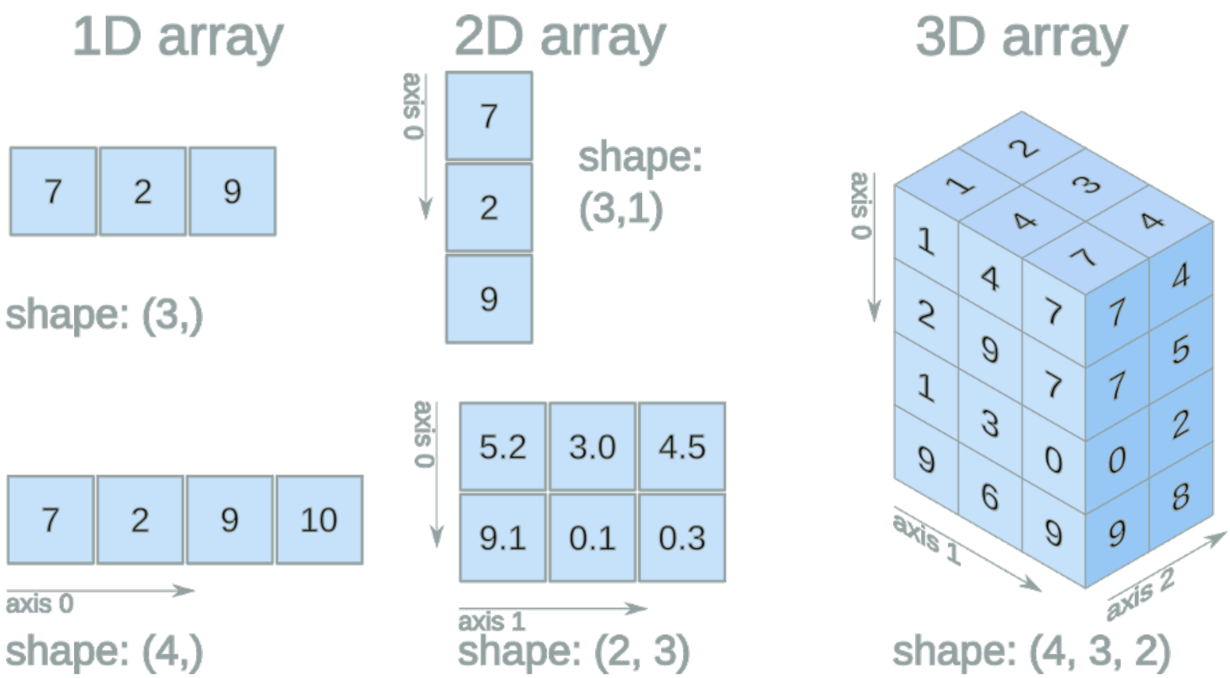
- You will be able to work with arrays of any number of dimensions.
- You will be able to index/slice into arrays of any number of dimensions.
- You will appreciate the usefulness of N-D arrays for real data.
- You will understand that each array can only contain a single type of data.
- You will appreciate that NumPy is fast.

NumPy 3-D arrays

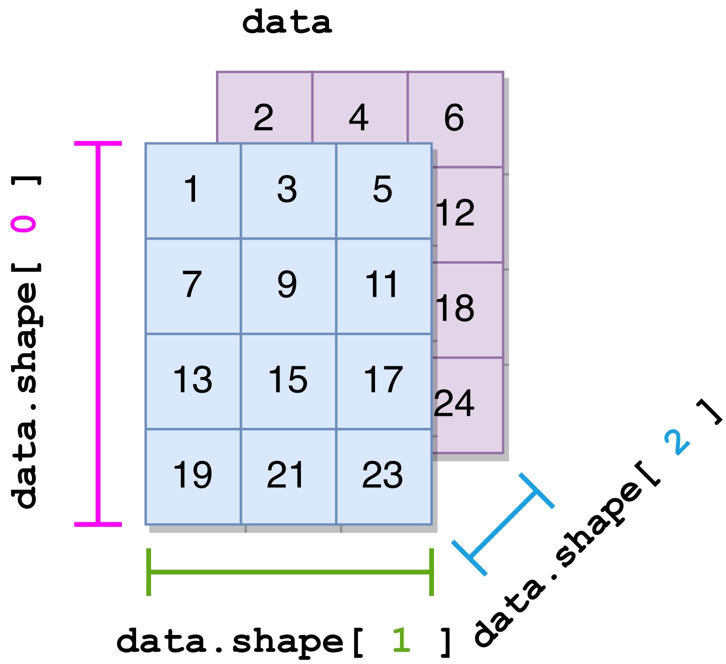




Array shape



Array shape



In [57]:
```
# data is represented in previous image
data = np.arange(1,25).reshape(4,3,2)

data.shape
```

Out[57]: (4, 3, 2)

In [58]:
```
rows = data.shape[0]
cols = data.shape[1]
depth = data.shape[2]

rows, cols, depth
```

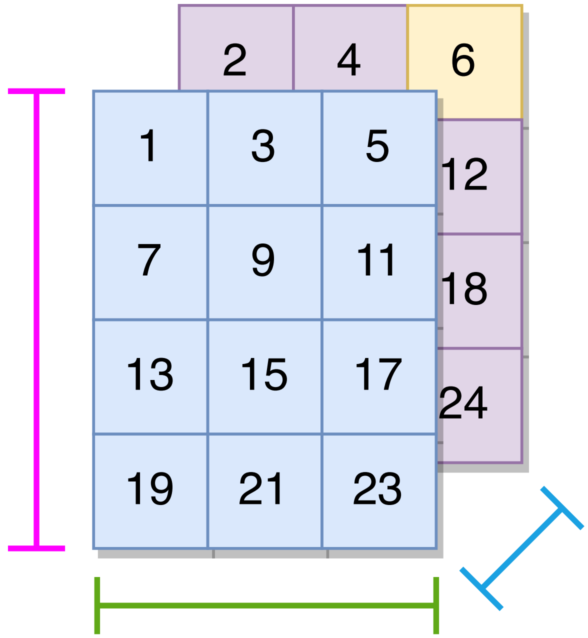
Out[58]: (4, 3, 2)

In [59]:
```
rows, cols, depth = data.shape

rows, cols, depth
```

Out[59]: (4, 3, 2)

NumPy 3-D array indexing/slicing

data[0 , 2 , 1]

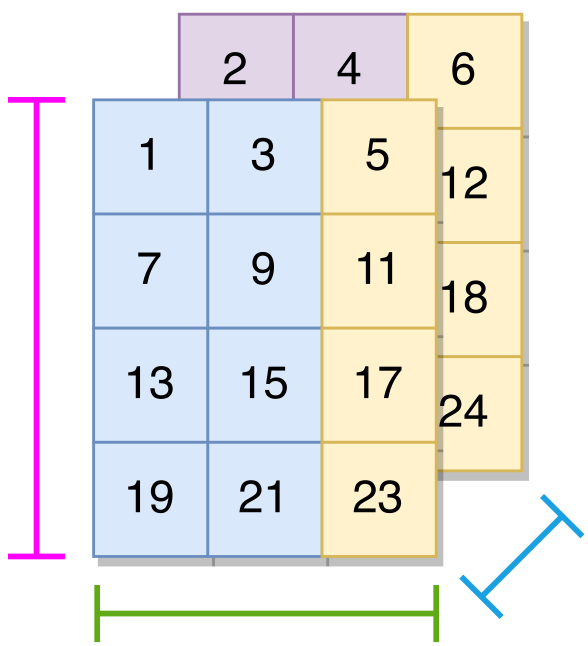


In [60]:
```python
data[0,2,1]
```

Out[60]: 6

NumPy 3-D array indexing/slicing

data[: , -1 , :]



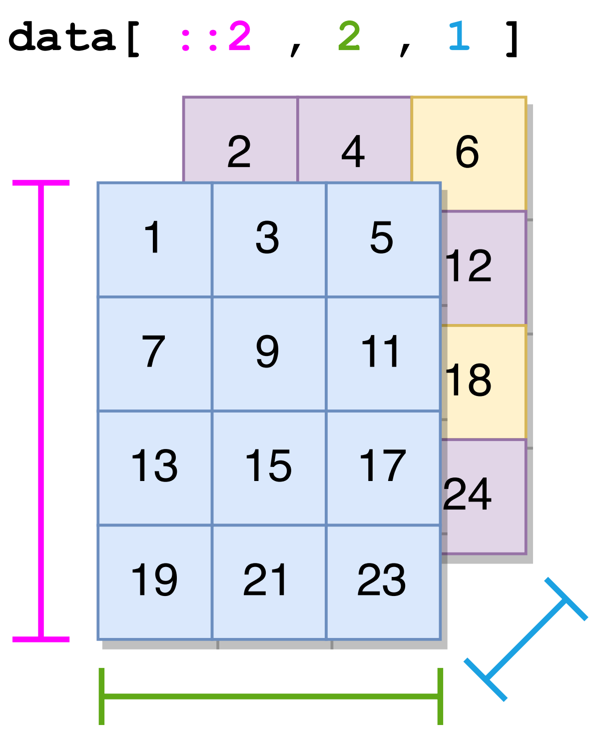
In [61]:
```
data[:,-1,:]
```

Out[61]:
```
array([[ 5,  6],
       [11, 12],
       [17, 18],
       [23, 24]])
```

In [62]:
```
data[:,-1]
```

Out[62]:
```
array([[ 5,  6],
       [11, 12],
       [17, 18],
       [23, 24]])
```

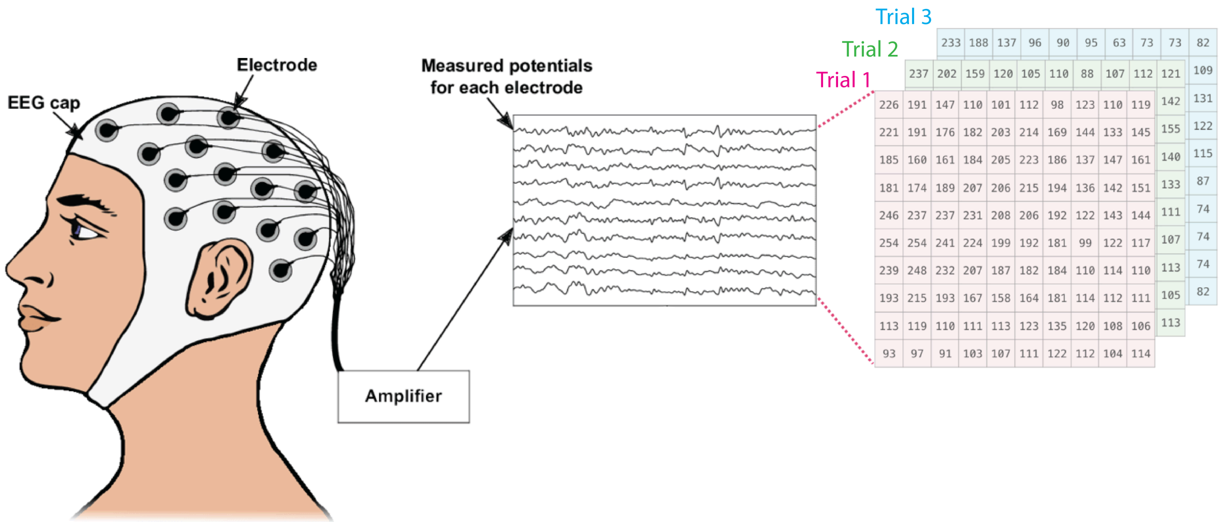
NumPy 3-D array indexing/slicing

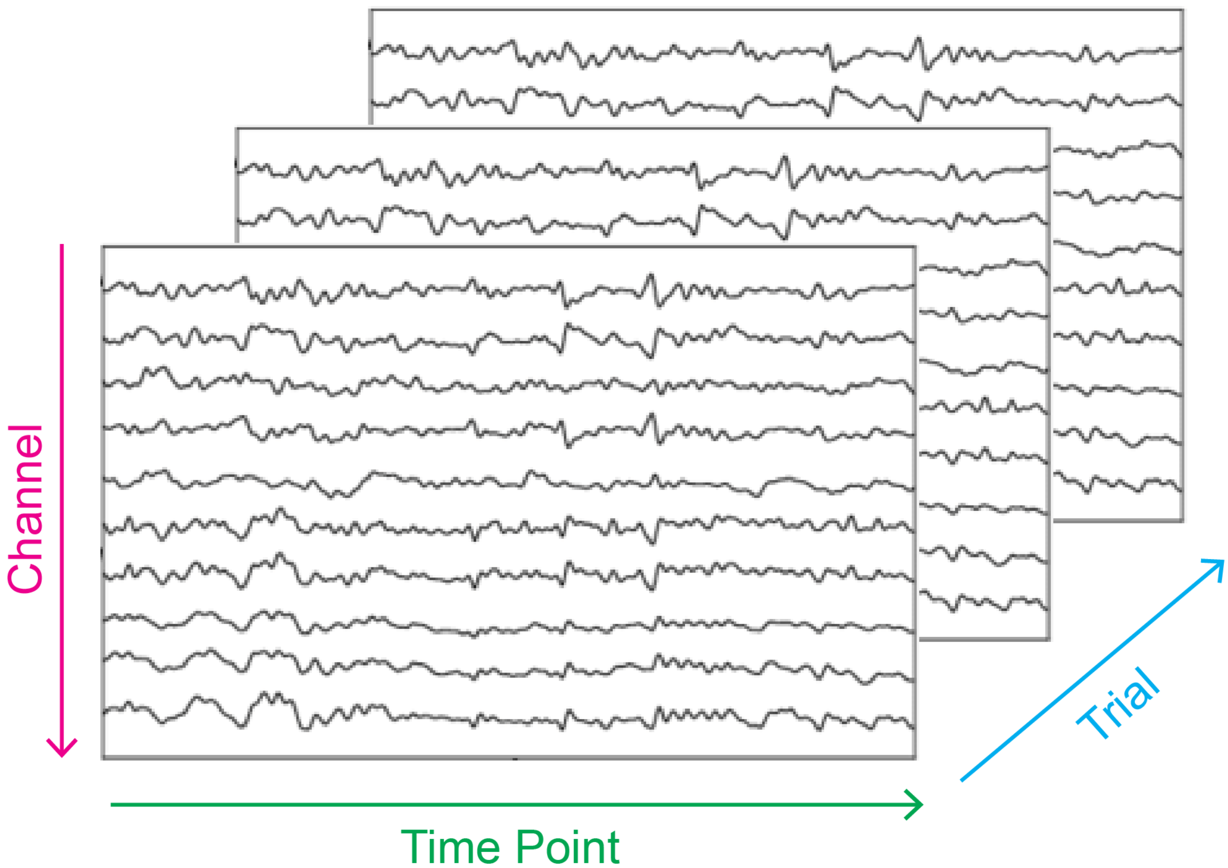


In [63]:
```
data[::2,2,1]
```

Out[63]:
```
array([ 6, 18])
```

3-D: EEG *time series* for multiple *channels, trials*



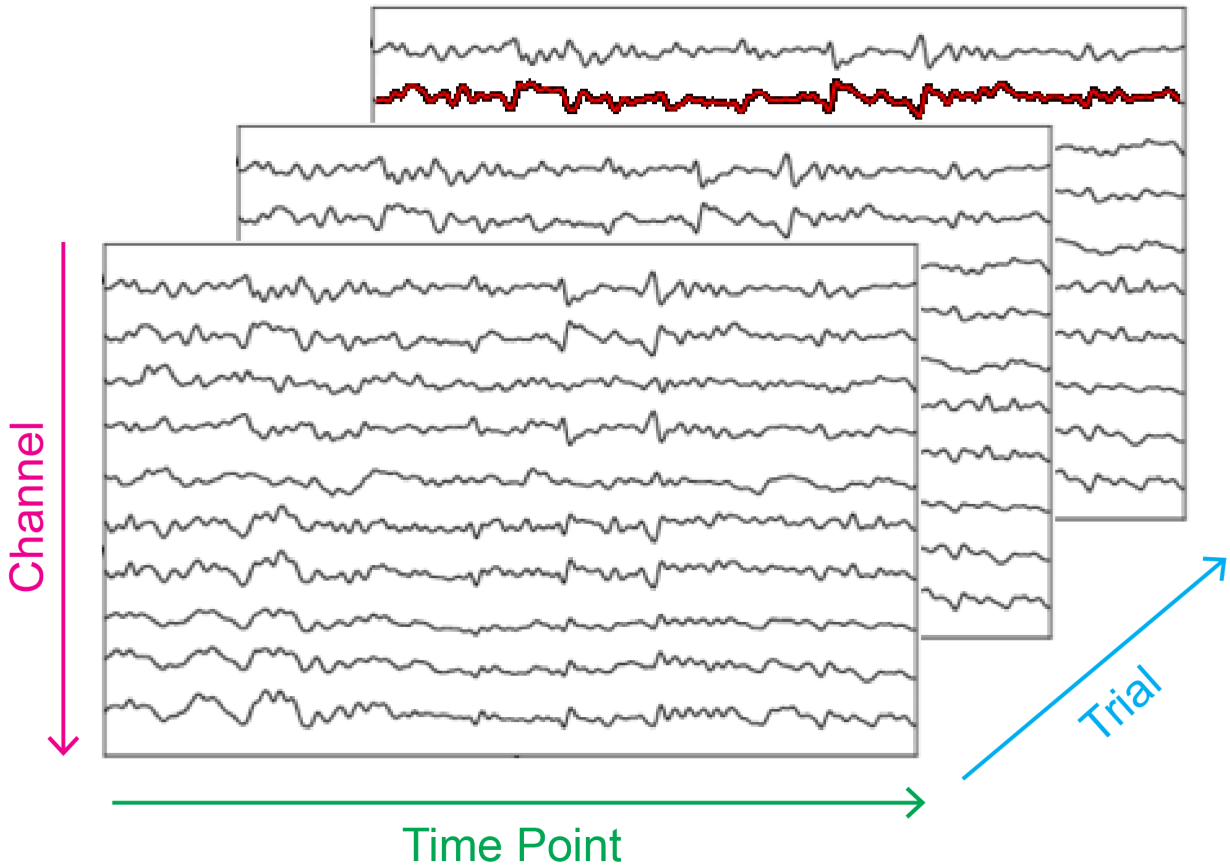


In [64]:
```
n_channels = 10
n_time_pts = 500
n_trials = 3

# fake EEGs
EEGs = np.random.random(
    [n_channels, n_time_pts, n_trials]
)
```

```
EEGs.shape
```

Out[64]: (10, 500, 3)

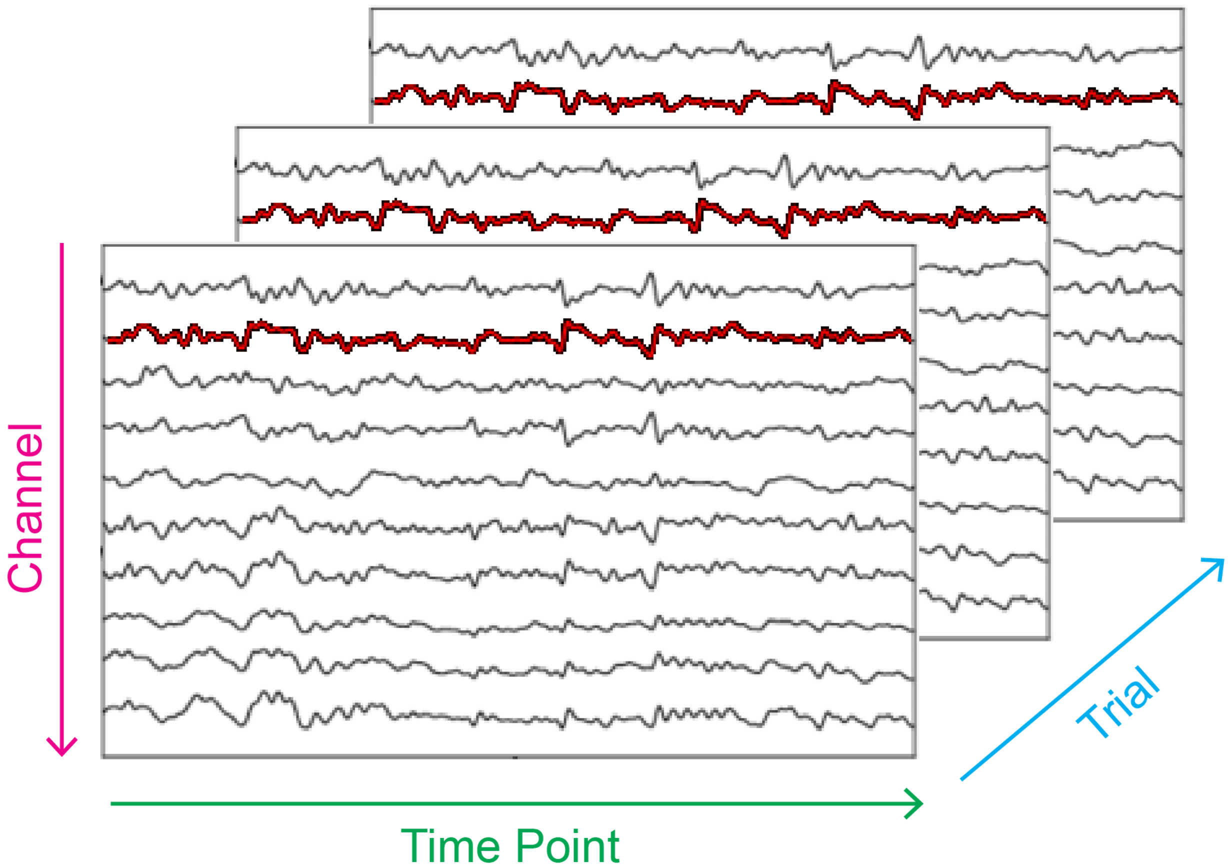
In [ ]:



In [65]:
```
# data = channel 1, trial 2

data = EEGs[1,:,2]

data.shape
```

Out[65]: (500,)

In [ ]:



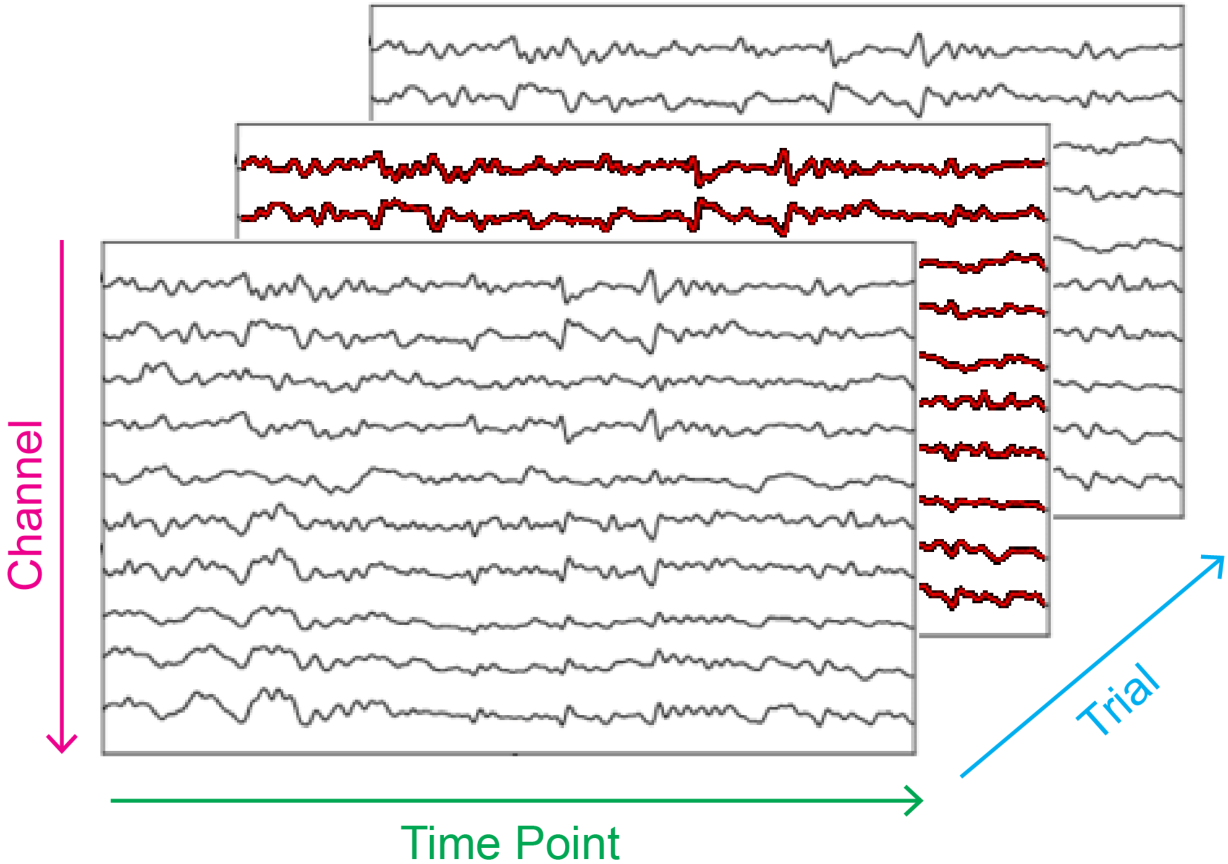
In [66]:
```
# data = channel 1, all trials

data = EEGs[1,:,:]

data.shape
```

Out[66]: (500, 3)

In [67]:
```
# channel 1 trial average

chan1_avg = data.mean(axis=1)

# chan1_avg = EEGs[1,:,:].mean(axis=1)

chan1_avg.shape
```

Out[67]: (500,)

In [ ]:



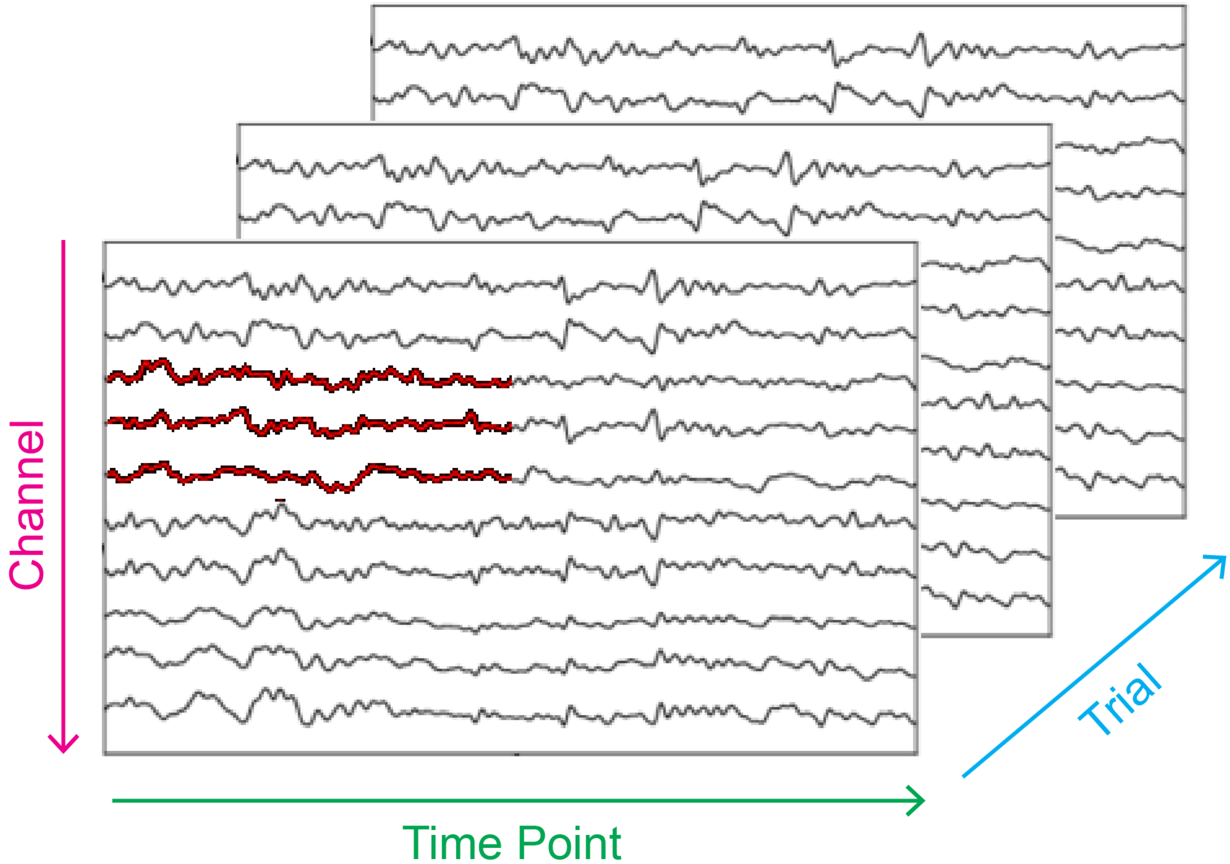
In [68]:
```
# data = all channels, trial 1

data = EEGs[:,:,1]

data.shape
```

Out[68]: (10, 500)

In [69]:
```
# average EEG across channels for trial 1

trial1_avg = data.mean(axis=0)

# trial1_avg = EEGs[:,:,1].mean(axis=0)

trial1_avg.shape
```

Out[69]: (500,)

In [ ]:



In [70]:
```
# data = channels 2-4, trial 0,
#        first 250 time pts

data = EEGs[2:5,:250,0]

data.shape
```

Out[70]: (3, 250)

4-D: EEG *time series* for multiple *subjects, channels, trials*

In [71]:
```
n_subjects = 15
n_channels = 10
n_time_pts = 500
n_trials = 3

# fake EEGs
EEGs = np.random.random(
    [n_subjects, n_channels, n_time_pts, n_trials]
)

EEGs.shape
```

Out[71]: (15, 10, 500, 3)

In [72]:
```
# data = everything for subject 7
data = EEGs[7,:,:,:]
data.shape
```

Out[72]: (10, 500, 3)

In [73]:
```
# data = everything for subject 7
data = EEGs[7]
data.shape
```

Out[73]: (10, 500, 3)

In [74]:
```
# data = all subjects, channel 3, trial 2
data = EEGs[:,3,:,2]
data.shape
```

Out[74]: (15, 500)

As long as you know how your data array is structured, it is simple to get whatever portions of data you want.

```
EEGs[subject, channel, time, trial, condition, sex, age, ...]
```

Just input the desired index or indexes along each dimension. Yes, it's that easy!

Array data type

NumPy arrays must contain data of **only a single type** *(e.g., float, int, bool, etc.)*

You **cannot mix different types of data** in a single array like you can in a Python list.

In [75]:
```
data = np.random.random([2,3])
data
```

Out[75]:
```
array([[0.15187631, 0.13323444, 0.27032076],
       [0.79030714, 0.96409636, 0.47234137]])
```

In [76]:
```
data.dtype
```

Out[76]: dtype('float64')

In [77]:
```
data = np.arange(5)
```

```
data
```

Out[77]:
```
array([0, 1, 2, 3, 4])
```

In [78]:
```
data.dtype
```

Out[78]:
```
dtype('int64')
```

In [79]:
```
data = np.arange(5).astype(float)
data
```

Out[79]:
```
array([0., 1., 2., 3., 4.])
```

In [80]:
```
data.dtype
```

Out[80]:
```
dtype('float64')
```

In [81]:
```
floats = np.random.random([2,3]) * 10
ints = floats.astype(int)

floats, ints
```

Out[81]:
```
(array([[0.33768618, 0.63979362, 1.78346574],
        [4.93966793, 4.45008176, 0.81904804]]),
 array([[0, 0, 1],
        [4, 4, 0]]))
```

In [82]:
```
np.zeros(3)
```

Out[82]:
```
array([0., 0., 0.])
```

In [83]:
```
np.zeros(3, dtype=float)
```

Out[83]:
```
array([0., 0., 0.])
```

In [84]:
```
np.zeros(3, dtype=int)
```

Out[84]:
```
array([0, 0, 0])
```

In [85]:
```
np.zeros(3, dtype=bool)
```

Out[85]:
```
array([False, False, False])
```

NumPy is much faster than basic Python

In [86]:
```
%%timeit
# time this entire cell
```

```
tot = 0
for i in range(50000):
    tot += i
```

917 µs ± 9.53 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)

In [87]:
```
# time a single line
%timeit np.arange(50000).sum()
```

20.1 µs ± 95.2 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)