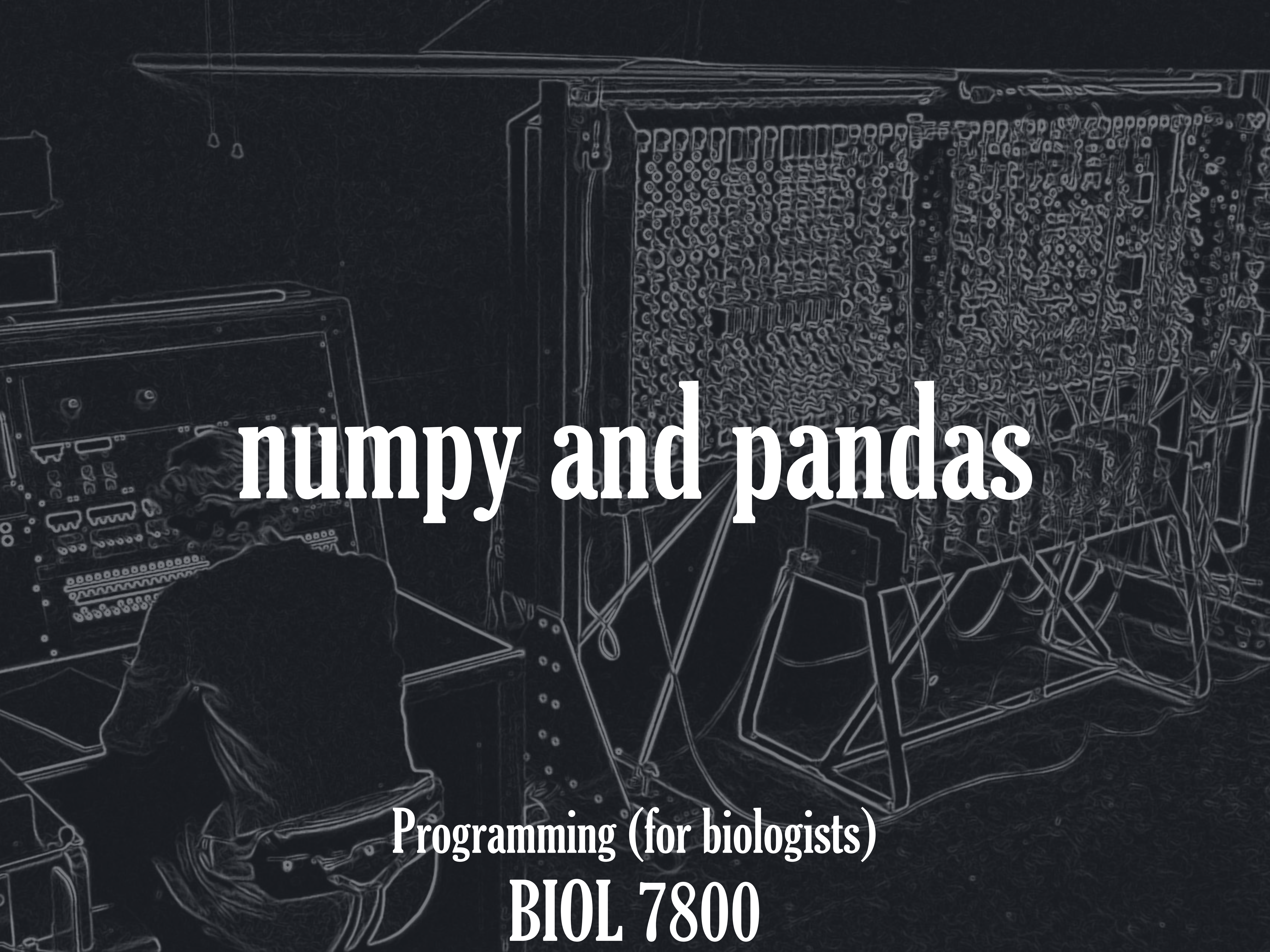


numpy and pandas

Programming (for biologists)

BIOL 7800

Python List

Enclosed in **square** brackets

```
my_list = ['lists', 'are', 'pretty', 'cool']
```

item of a list

item of a list

0 1 2 3

Indexes

You can access an element

```
my_list[3] = 'cool'
```

Python List

Enclosed in **square** brackets

['lists', 'are', 'pretty', 'cool']

item of a list

item of a list

0 1 2 3 4

You can slice a list

my_list[2:4] = ['pretty', 'cool']

Note: this returns another list

Python List

matrix

$$\begin{bmatrix} 1, & 2, & 3, & 4 \\ 5, & 6, & 7, & 8 \\ 9, & 10, & 11, & 12 \end{bmatrix}$$

But how do you represent a structure like this with Python `lists`?

Python List

But how do you represent a structure like this with Python lists?

```
my_matrix = [
               [1,  2,  3,  4],
               [5,  6,  7,  8],
               [9, 10, 11, 12]
                              ]

print(my_matrix)
[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
```

Python List

This kind of does what we want…

```
print(my_matrix)
[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]

print(my_matrix[0])
[1, 2, 3, 4]

print(my_matrix[1][1])
6
```

We can access "rows"

We can access "items"

Python List

But, there's a problem here… what if we want to perform operations on the matrix?

```
print(my_matrix * 2)
```

What do you expect this to produce?

Python List

But, there's a problem here… what if we want to perform operations on the matrix?

```
print(my_matrix * 2)
```

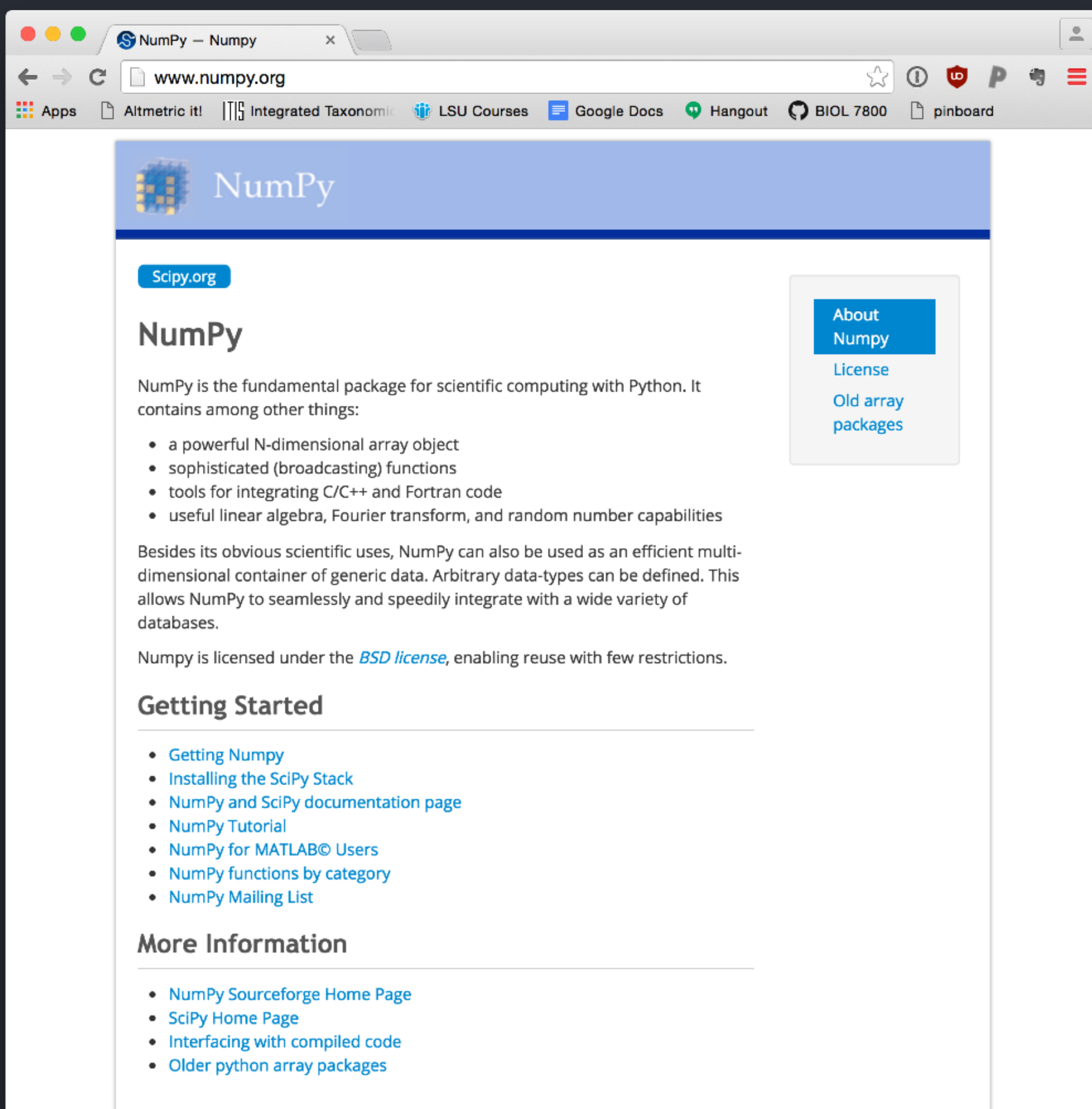
```
[[1, 2, 3, 4],
 [5, 6, 7, 8],
 [9, 10, 11, 12],
 [1, 2, 3, 4],
 [5, 6, 7, 8],
 [9, 10, 11, 12]]
```

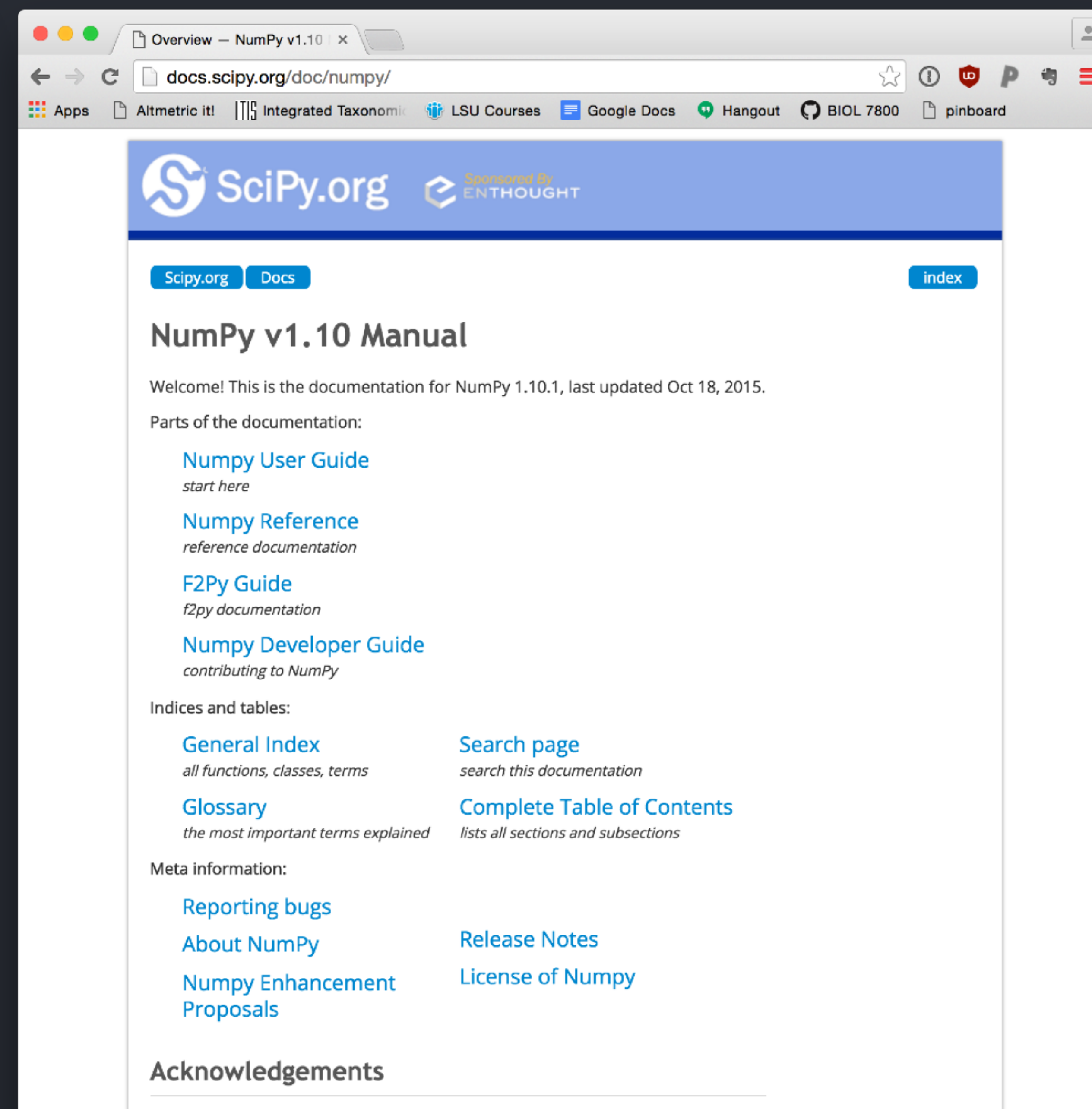
Is this what you wanted?

and that's where

numpy (numerical python)

becomes **useful**





What is numpy?

(Very) Fast N-dimensional array objects

you can create matrices/arrays of any dimension

arrays can have many types (integer, float, string, etc)

(Very) Fast methods for operating on N-dim arrays

you can transform, reshape, slice, alter, multiple, add, etc., etc.

(Very) Fast random number and linear algebra capabilities

you can generate hundreds, thousands, millions of random numbers, etc.

Anatomy of a numpy array

(it's a lot like a list… sort of…)

In: import numpy ← Import numpy

In: my_array = numpy.array([1,2,3,4]) ← Create array
In: my_array
Out: array([1, 2, 3, 4])

Wrap a list in numpy.array()

In: my_array[0] ← Access element
Out: 1
In: my_array[1:3] ← Slice
Out: array([2, 3])

Anatomy of a numpy array

(but, it's also **not** like a list at all…)

```
In:  my_array = numpy.array([1,2,3,4])
```
Create array

```
In:  my_array*2
Out: array([2, 4, 6, 8])
```
This operation multiplies every element by 2

```
In:  my_array_2 = numpy.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])
In:  my_array_2
Out: array([[ 1,  2,  3,  4],
            [ 5,  6,  7,  8],
            [ 9, 10, 11, 12]])
```
We can have 2d arrays

```
In:  my_array_2[0]
Out: array([1, 2, 3, 4])
```
We can get a row slice

Anatomy of a numpy array

(but, it's also **not** like a list at all…)

```
In:  my_array_2 = numpy.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])
In:  my_array_2
Out: array([[ 1,  2,  3,  4],
            [ 5,  6,  7,  8],
            [ 9, 10, 11, 12]])
```

We can have 2d arrays

we can do "fancy" slicing

all rows

0th column

```
In:  my_array_2[: , 0]
Out: array([1, 5, 9])
```

We can get a column slice

rows 2 & 3

1st column

```
In:  my_array_2[1:3, 1]
Out: array([6, 10])
```

We can get row + column slice

Anatomy of a numpy array

(but, it's also **not** like a list at all…)

we can do "fancy" indexing

```
In:   my_array = numpy.array([1,2,3,4])
In:   keep = numpy.array([True,False,True,False])

In:   my_array[keep]
Out:  array([1, 3])
```

We can slice/index an array **with another array!** (a boolean)

```
In:   keep = numpy.array([0,2,3])

In:   my_array[keep]
Out:  array([1, 3, 4])
```

We can slice/index an array **with another array!** (index integers)

Anatomy of a numpy array

(array attributes)

```
In:  my_array = numpy.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])
In:  my_array
Out: array([[ 1,  2,  3,  4],
            [ 5,  6,  7,  8],
            [ 9, 10, 11, 12]])
```

Create array (notice double [[ ]])

```
In:  my_array.ndim
Out: 2
```

Get array dimensions

```
In:  my_array.shape
Out: (3, 4)
```

Get array shape (rows,cols)

```
In:  my_array.size
Out: 12
```

Get array size (total elements)

```
In:  my_array.dtype
Out: dtype('int64')
```

Get array data type

Anatomy of a numpy array

(array methods - remember, these are all very fast)

```
In:   my_array = numpy.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])

In:   my_array.sum()
Out: 78

In:   my_array.min()
Out: 1

In:   my_array.max()
Out: 12

In:   my_array.mean()
Out: 6.5

In:   my_array.std()
Out: 3.452052529534663
```

Get array sum

Get array minimum

Get array maximum

Get array mean

Get array standard deviation

Anatomy of a numpy array

(array methods - remember, these are all very fast)

```
In:   my_array = numpy.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])

In:   my_array.reshape(2,6)
Out: array([[ 1,  2,  3,  4,  5,  6],
            [ 7,  8,  9, 10, 11, 12]])
```

reshape the array to (2 rows, 6 cols)

```
In:   my_array.reshape(6,2)
Out: array([[ 1,  2],
            [ 3,  4],
            [ 5,  6],
            [ 7,  8],
            [ 9, 10],
            [11, 12]])
```

reshape the array to (6 rows, 2 cols)

Anatomy of a numpy array

(array methods - remember, these are all very fast)

```
In:   my_array = numpy.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])

In:   new = my_array.reshape(3,2,2)
```

reshape the array to (3 dimensions, 2 rows, 2 cols)

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

            [[ 5,  6],
             [ 7,  8]],

            [[ 9, 10],
             [11, 12]]])

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

new[0] is now the first dimension

Anatomy of a numpy array

(array methods - remember, these are all very fast)

```
In:   my_array = numpy.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])
In:   my_array
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

```
In:   my_array.sum(axis=0)
Out: array([15, 18, 21, 24])
```

You can perform operations by column

```
In:   my_array.sum(axis=1)
Out: array([10, 26, 42])
```

You can perform operations by row

Anatomy of a numpy array

(array methods - remember, these are all very fast)

```
In:  my_array
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

transpose

```
In:  for row in my_array:
         print(row)
Out:
[1 2 3 4]
[5 6 7 8]
[ 9 10 11 12]
```

```
In:  for col in my_array.T:
         print(col)
Out:
[1 5 9]
[ 2  6 10]
[ 3  7 11]
[ 4  8 12]
```

You can iterate over rows

You can iterate over columns (by *transposing the array*)

Anatomy of a numpy array

You can quickly create arrays of different types

```
In:   my_array = numpy.zeros([3,3])
In:   my_array
Out: array([[ 0.,  0.,  0.],
            [ 0.,  0.,  0.],
            [ 0.,  0.,  0.]])

In:   my_array = numpy.ones([3,3])
In:   my_array
Out: array([[ 1.,  1.,  1.],
            [ 1.,  1.,  1.],
            [ 1.,  1.,  1.]])

In:   my_array = numpy.identity(3)
In:   my_array
Out: array([[ 1.,  0.,  0.],
            [ 0.,  1.,  0.],
            [ 0.,  0.,  1.]])
```

Anatomy of a numpy array

You can quickly create arrays of different types

```
In:   my_array_1 = numpy.ones([3,3])
In:   my_array_1
Out: array([[ 1.,  1.,  1.],
            [ 1.,  1.,  1.],
            [ 1.,  1.,  1.]])

In:   my_array_2 = numpy.identity(3)
In:   my_array_2
Out: array([[ 1.,  0.,  0.],
            [ 0.,  1.,  0.],
            [ 0.,  0.,  1.]])

In:   my_array_1 + my_array_2
Out: array([[ 2.,  1.,  1.],
            [ 1.,  2.,  1.],
            [ 1.,  1.,  2.]])
```

You can add arrays
(it's elementwise)

Anatomy of a numpy array

You can quickly create arrays of different types

```
In:   my_array_1 = numpy.array(list('dog'))
In:   my_array_1
Out: array(['d', 'o', 'g'],
           dtype='<U1')

In:   my_array_2 = numpy.array(list('cat'))
In:   my_array_2
Out: array(['c', 'a', 't'],
           dtype='<U1')

In:   numpy.vstack([my_array_1, my_array_2])
Out: array([['d', 'o', 'g'],
            ['c', 'a', 't']],
           dtype='<U1')
```

You can stack arrays

numpy

many, many, many, many <u>other</u> **methods & functions**

Now, one thing you may be thinking is that numpy seems ideal for tabular data

So this

height	width
1	2
3	4
5	6
7	8
9	10
11	12

Can be represented as this

```
In:   my_array
Out: array([[ 1,  2],
            [ 3,  4],
            [ 5,  6],
            [ 7,  8],
            [ 9, 10],
            [11, 12]])
```

numpy

Now, one thing you may be thinking is that numpy seems ideal for tabular data

So this

height	width
1	2
3	4
5	6
7	8
9	10
11	12

Can be represented as this

```
In:  my_array
Out: array([[ 1,  2],
            [ 3,  4],
            [ 5,  6],
            [ 7,  8],
            [ 9, 10],
            [11, 12]])
```

But, what's **one** problem here?

numpy

So this

height	width
1	2
3	4
5	6
7	8
9	10
11	12

Can be represented as this

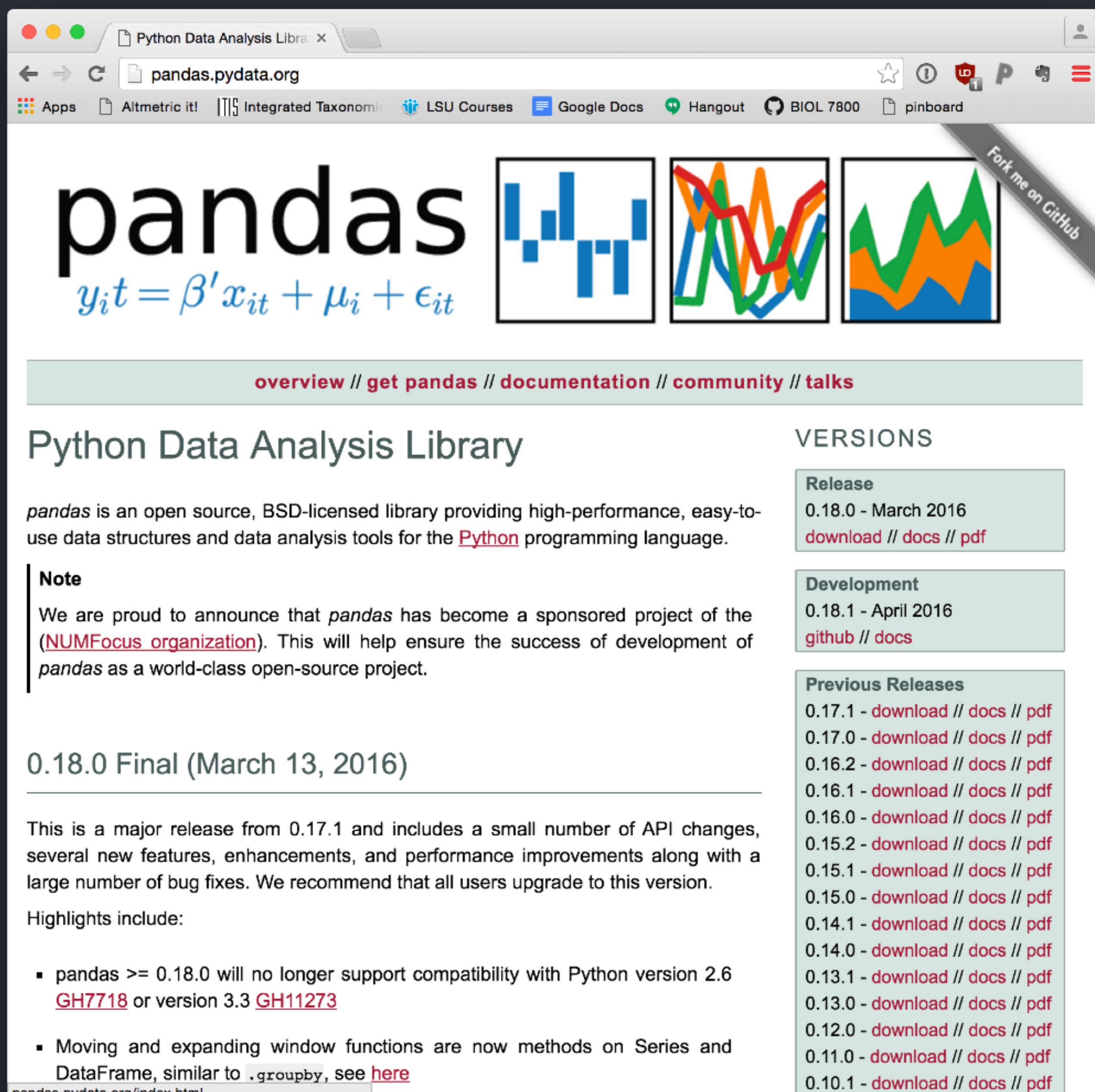
```
In:   my_array
Out: array([[ 1,  2],
            [ 3,  4],
            [ 5,  6],
            [ 7,  8],
            [ 9, 10],
            [11, 12]])
```

But, what's **one** problem here?

It's hard to keep track of columns and rows

`my_array[ : , 0]`   `my_array[ : , 1]`

and that's where pandas comes into play



pandas is a data structure built on the numpy framework that gives us access to data as *"frames"*

(also includes **many more** functions)

pandas

has two major types of data objects

Series

one dimensional labelled array

Data Frame

two-dimensional labelled array

pandas

Series - one dimensional, labelled array

```
In:   import pandas as pd

In:   my_series = pd.Series([1,2,3,4], index=['a','b','c','d'])
In:   my_series
Out:
a   1
b   2
c   3
d   4
dtype: int64
```

← Create series

index holds data labels

```
In:   my_series['a']
Out: 1
```

← We can slice by label

```
In:   my_series['b':'c']
Out:
b   2
c   3
dtype: int64
```

← We can slice by labels

pandas

Series - one dimensional, labelled array

```
In:   my_dict = {'a':1, 'b':2, 'c':3, 'd':4}
In:   my_series = pd.Series(my_dict)
In:   my_series
Out:
a   1
b   2
c   3
d   4
dtype: int64
```

We can also create a series from a dict

```
In:   my_series[my_series > my_series.median()]
Out:
c   3
d   4
dtype: int64
```

We can filter with fancy indexing

```
In:   my_series + my_series
Out:
a   2
b   4
c   6
d   8
dtype: int64
```

We can perform elementwise operations

pandas

Series - one dimensional, labelled array

```
In:   my_dict = {'a':1, 'b':2, 'c':3, 'd':4}
In:   my_series = pd.Series(my_dict)
In:   my_series[1:]
Out:
b   2
c   3
d   4
dtype: int64
```

Slice rows b, c, d

```
In:   my_series[:-1]
Out:
a   1
b   2
c   3
dtype: int64
```

Slice rows a, b, c

```
In:   my_series[1:] + my_series[-1]
Out:
a  NaN
b    4
c    6
d  NaN
dtype: float64
```

Add two slices together

pandas operates on data based on the label (index). So, if there are not corresponding labels in a series, you get no result

pandas

Series - one dimensional, labelled array

```
In:   my_series_1 = pd.Series({'a':1, 'b':2, 'c':3, 'd':4})
In:   my_series_2 = pd.Series({'c':10, 'd':20, 'e':30, 'f':40})
In:   my_series_1 + my_series_2
Out:
a  NaN
b  NaN
c   13
d   24
e  NaN
f  NaN
dtype: float64
```

Add two slices together

pandas operates on data based on the label (index). So, if there are not corresponding labels in a series, you get no result

```
In:   import numpy
In:   numpy.exp(my_series_1 + my_series_2)
Out:
a           NaN
b           NaN
c  4.424134e+05
d  2.648912e+10
e           NaN
f           NaN
dtype: float64
```

most numpy functions also work with series

(here, get exponential of **all** elements)

pandas

Data Frame - two-dimensional, labelled array

Lots of ways to create a **DataFrame**

```
In:   my_series_1 = pd.Series({'a':1, 'b':2, 'c':3, 'd':4})
In:   my_series_2 = pd.Series({'c':10, 'd':20, 'e':30, 'f':40})

In:   my_df = pd.DataFrame({'col1': my_series_1, 'col2': my_series_1})
In:   my_df
Out:
   col1  col2
a     1   NaN
b     2   NaN
c     3    10
d     4    20
e   NaN    30
f   NaN    40
```

pandas

Data Frame - two-dimensional, labelled array

Lots of ways to create a DataFrame

```
In:   my_dict = {'one' : [1., 2., 3., 4.], 'two' : [4., 3., 2., 1.]}
In:   my_df = pd.DataFrame(my_dict, index=['a','b','c','d'])
In:   my_df
Out:
   one  two
a    1    4
b    2    3
c    3    2
d    4    1

In:   my_df['one']['a':'c']
Out:
a  1
b  2
c  3
Name: one, dtype: float64

In:   my_df['one']['a':'c'].sum()
Out: 6.0
```

pandas

Lots of ways to create a DataFrame

Read from a CSV file

In: `print(open('table.csv').read())`

Out:

```
height,width
1,2
3,4
5,6
7,8
9,10
11,12
```

In: `my_df = pd.read_csv('table.csv')`

In: `my_df`

Out:

```
   height  width
0       1      2
1       3      4
2       5      6
3       7      8
4       9     10
5      11     12
```

As 'table.csv'
in filesystem

height	width
1	2
3	4
5	6
7	8
9	10
11	12

pandas

Lots of ways to manipulate a DataFrame

```
In:   my_df = pd.read_csv('table.csv')
In:   my_df.T
Out:
        0  1  2  3   4   5
height  1  3  5  7   9  11
width   2  4  6  8  10  12

In:   my_df_2 = pd.read_csv('table2.csv')
In:   pd.concat([my_df, my_df_2]
Out:
  girth  height  length  width
0   NaN       1     NaN      2
1   NaN       3     NaN      4
2   NaN       5     NaN      6
3   NaN       7     NaN      8
4   NaN       9     NaN     10
5   NaN      11     NaN     12
0   100     NaN      10    NaN
1   200     NaN      20    NaN
2   300     NaN      30    NaN
3   400     NaN      40    NaN
4   500     NaN      50    NaN
5   600     NaN      60    NaN
```

As 'table.csv'

height	width
1	2
3	4
5	6
7	8
9	10
11	12

As 'table2.csv'

length	girth
10	100
20	200
30	300
40	400
50	500
60	600

How would we merge side-by-side (based on index)?

pandas

Lots of ways to manipulate a DataFrame

How would we merge side-by-side (based on index)?

```
In:   my_df_2 = pd.read_csv('table2.csv')
In:   new_df = pd.concat([my_df, my_df_2], axis = 1)
In:   new_df
Out:
   height  width  length  girth
0       1      2      10    100
1       3      4      20    200
2       5      6      30    300
3       7      8      40    400
4       9     10      50    500
5      11     12      60    600

In:   new_df['height'] > 5
0   False
1   False
2   False
3    True
4    True
5    True
Name: height, dtype: bool
```

As 'table.csv'

height	width
1	2
3	4
5	6
7	8
9	10
11	12

As 'table2.csv'

length	girth
10	100
20	200
30	300
40	400
50	500
60	600

pandas

Lots of ways to manipulate a DataFrame

How would we merge side-by-side (based on index)?

In: new_df['height'] > 5

Out:

```
0    False
1    False
2    False
3     True
4     True
5     True
Name: height, dtype: bool
```

In: new_df[new_df['height'] > 5]

Out:

```
   height  width  length  girth
3       7      8      40    400
4       9     10      50    500
5      11     12      60    600
```

In: new_df[new_df[(new_df['height'] > 5) & (new_df['width'] >= 10)]]

Out:

```
   height  width  length  girth
4       9     10      50    500
5      11     12      60    600
```

As 'table.csv'

height	width
1	2
3	4
5	6
7	8
9	10
11	12

As 'table2.csv'

length	girth
10	100
20	200
30	300
40	400
50	500
60	600

pandas

Lots of ways to manipulate a `DataFrame`

Lots of ways to do many things

many, many, many, many things

`Too many` things, in fact, to *show you all of them*…

(so, one small example)

pandas

Lots of ways to create a DataFrame

In: amniote = pd.read_csv('Amniote_Database_Aug_2015.csv')

In: amniote.info() ← Get info on data

Out:

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 21322 entries, 0 to 21321
Data columns (total 36 columns):
class                                   21322 non-null object
order                                   21322 non-null object
family                                  21322 non-null object
genus                                   21322 non-null object
species                                 21322 non-null object
subspecies                              21322 non-null int64
common_name                             21322 non-null object
female_maturity_d                       21322 non-null float64
(…truncated…)
egg_width_mm                            21322 non-null float64
egg_length_mm                           21322 non-null float64
fledging_mass_g                         21322 non-null float64
adult_svl_cm                            21322 non-null float64
male_svl_cm                             21322 non-null float64
female_svl_cm                           21322 non-null float64
birth_or_hatching_svl_cm                21322 non-null float64
female_svl_at_maturity_cm               21322 non-null float64
female_body_mass_at_maturity_g          21322 non-null int64
no_sex_svl_cm                           21322 non-null float64
no_sex_maturity_d                       21322 non-null float64
dtypes: float64(28), int64(2), object(6)
memory usage: 6.0+ MB
```

pandas

Lots of ways to create a DataFrame

In: amniote.head()

Out:

```
  class            order         family      genus      species  subspecies  \
0  Aves  Accipitriformes  Accipitridae  Accipiter  albogularis        -999
1  Aves  Accipitriformes  Accipitridae  Accipiter       badius        -999
2  Aves  Accipitriformes  Accipitridae  Accipiter      bicolor        -999
3  Aves  Accipitriformes  Accipitridae  Accipiter    brachyurus        -999
4  Aves  Accipitriformes  Accipitridae  Accipiter     brevipes        -999

              common_name  female_maturity_d  litter_or_clutch_size_n  \
0            Pied Goshawk           -999.000                  -999.00
1                  Shikra            363.468                     3.25
2           Bicolored Hawk           -999.000                     2.70
3  New Britain Sparrowhawk           -999.000                  -999.00
4        Levant Sparrowhawk            363.468                     4.00
(…truncated…)
```

Look at top rows of data

Missing entries seem to be coded -999

In: amniote.tail(4)

Out:

```
          class       order        family       genus        species  \
21318  Reptilia  Testudines  Trionychidae       Palea  steindachneri
21319  Reptilia  Testudines  Trionychidae  Pelochelys        bibroni
21320  Reptilia  Testudines  Trionychidae  Pelodiscus       sinensis
21321  Reptilia  Testudines  Trionychidae     Trionyx      triunguis

       subspecies                                 common_name  \
21318        -999           Wattle-necked Soft-shelled Turtle
21319        -999  Asian or Bibron's Giant soft-shelled Turtle
21320        -999                 Chinese Soft-shelled Turtle
21321        -999                    Nile Soft-shelled Turtle
```

Look at bottom (4) rows of data

pandas

DataFrame methods

```
In:  amniote = amniote.replace(-999, numpy.nan)
In:  amniote.head(2)
Out:
  class            order         family      genus      species  subspecies  \
0  Aves  Accipitriformes  Accipitridae  Accipiter  albogularis         NaN
1  Aves  Accipitriformes  Accipitridae  Accipiter       badius         NaN

    common_name  female_maturity_d  litter_or_clutch_size_n  \
0  Pied Goshawk                NaN                      NaN
1        Shikra            363.468                     3.25
(…truncated…)
```

Replace all -999 with numpy.nan

Missing entries now coded NaN

subset data by reptiles

get litter/clutch size

use describe method on col

```
In:  amniote[amniote['class']=='Reptilia']['litter_or_clutch_size_n'].describe()
Out:
count    2675.000000
mean        7.233857
std        10.129951
min         1.000000
25%         2.100000
50%         4.600000
75%         8.800000
max       156.000000
```

Get descriptive statistics for all reptilian clutch sizes

pandas

DataFrame methods

get litter/clutch
size

Get max value

In: amniote[amniote['class']=='Aves']['litter_or_clutch_size_n'].max()
Out: 35.649999999999999

chained conditionals

subset data by birds & filter on litter/clutch size

In: amniote[(amniote['class'] == 'Aves') & (amniote['litter_or_clutch_size_n'] > 35)]
Out:

```
     class          order       family     genus  species  subspecies  \
3757  Aves  Passeriformes  Estrildidae  Neochmia  phaeton         NaN

        common_name  female_maturity_d  litter_or_clutch_size_n  \
3757  Crimson Finch                NaN                    35.65
```