You can access these slides on the course Github:
https://github.com/natrask/ENM1050

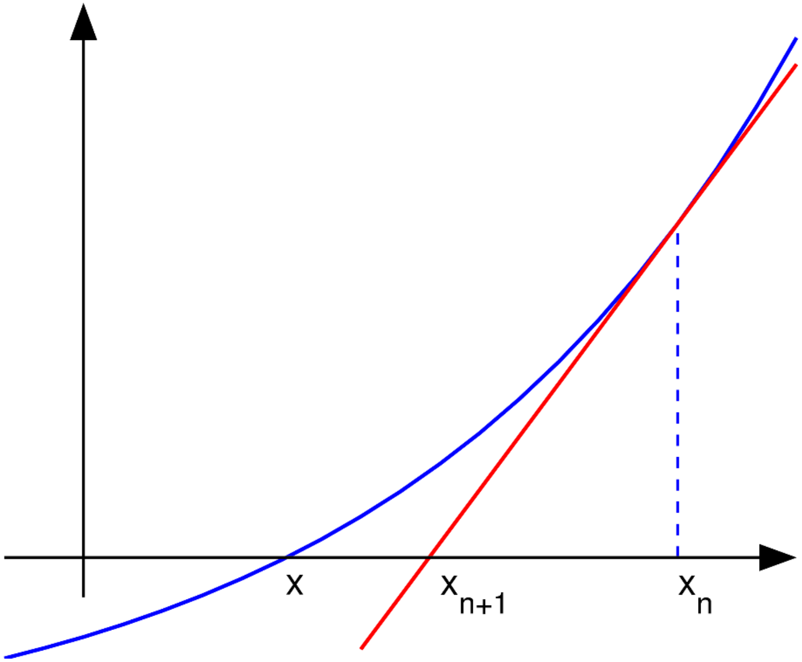
ENGR 1050
Intro to Scientific Computation

Lecture 06 – NumPy

Prof. Nat Trask
Mechanical Engineering & Applied Mechanics
University of Pennsylvania

Last time: `nonlinear solvers + functions`

- A lot of the trouble with implementing the newton solver came from how awkward it is to described mathematical operations with lists
- Today we'll introduce a new python package for applying mathematical operations to vectors of numbers
- First, we'll review a few takeaway thoughts about functions



$$x = x_n - \frac{f(x_n)}{f'(x_n)}$$

Why use functions? Generalization

Same code can be used more than once with parameters to allow for differences

BEFORE

```
diameter_large = 2.54 * 1.65
print('Large ball: ', diameter_large, 'cm')

diameter_med = 2.54 * 1.01
print('Medium ball: ', diameter_med, 'cm')

diameter_small = 2.54 + 0.46
print('Small ball: ', diameter_small, 'cm')
```

would not have made this error

AFTER

```
def print_as_cm(inches, name):
    cm = 2.54 * inches
    print(name, ':', cm, 'cm')

print_as_cm(1.65, 'Large ball')
print_as_cm(1.01, 'Medium ball')
print_as_cm(0.46, 'Small ball')
```

only type these lines once

Why use functions? Maintenance

Much easier to make changes

BEFORE

```
diameter_large = 2.54 * 1.65
print('Large ball: ', diameter_large, 'cm')

diameter_med = 2.54 * 1.01
print('Medium ball: ', diameter_med, 'cm')

diameter_small = 2.54 + 0.46
print('Small ball: ', diameter_small, 'cm')
```

AFTER

```
def print_as_cm(inches, name):
    cm = 2.54 * inches
    print(name, ':', cm, 'cm')

print_as_cm(1.65, 'Large ball')
print_as_cm(1.01, 'Medium ball')
print_as_cm(0.46, 'Small ball')
```

Can change to 'centimeter' with only one change

Why use functions? Encapsulation

Much easier to debug!

BEFORE

```
diameter_large = 2.54 * 1.65
print('Large ball: ', diameter_large, 'cm')

diameter_med = 2.54 * 1.01
print('Medium ball: ', diameter_med, 'cm')

diameter_small = 2.54 + 0.46
print('Small ball: ', diameter_small, 'cm')
```

What are we doing here?

AFTER

```
def print_as_cm(inches, name):
    cm = 2.54 * inches
    print(name, ':', cm, 'cm')

print_as_cm(1.65, 'Large ball')
print_as_cm(1.01, 'Medium ball')
print_as_cm(0.46, 'Small ball')
```

Oh, printing as centimeters!

Where do we get more functions? Modules

A **module** is a Python file with a collection of related functions.

`import` a module to use its functions

```
import MODULE
```

Ex:

```
import math

radians = (90.0/360.0) * 2 * math.pi
print(math.sin(radians))
```

https://docs.python.org/3/library/math.html
https://docs.python.org/3/py-modindex.html

Today

Numpy – a module for scientific computing

Many folks have run into this

Properties [edit]

Scalar multiplication obeys the following rules *(vector in boldface)*:

- Additivity in the scalar: $(c + d)\mathbf{v} = c\mathbf{v} + d\mathbf{v}$;
- Additivity in the vector: $c(\mathbf{v} + \mathbf{w}) = c\mathbf{v} + c\mathbf{w}$;
- Compatibility of product of scalars with scalar multiplication: $(cd)\mathbf{v} = c(d\mathbf{v})$;
- Multiplying by 1 does not change a vector: $1\mathbf{v} = \mathbf{v}$;
- Multiplying by 0 gives the zero vector: $0\mathbf{v} = \mathbf{0}$;
- Multiplying by −1 gives the additive inverse: $(-1)\mathbf{v} = -\mathbf{v}$.

```
my_vector = [1,2,3]
twice_the vector = 2*[1,2,3]
print(twice_the vector)

>> ???
```

What is the output of this?

Many folks have run into this

Properties [edit]

Scalar multiplication obeys the following rules *(vector in boldface)*:

- Additivity in the scalar: $(c + d)\mathbf{v} = c\mathbf{v} + d\mathbf{v}$;
- Additivity in the vector: $c(\mathbf{v} + \mathbf{w}) = c\mathbf{v} + c\mathbf{w}$;
- Compatibility of product of scalars with scalar multiplication: $(cd)\mathbf{v} = c(d\mathbf{v})$;
- Multiplying by 1 does not change a vector: $1\mathbf{v} = \mathbf{v}$;
- Multiplying by 0 gives the zero vector: $0\mathbf{v} = \mathbf{0}$;
- Multiplying by −1 gives the additive inverse: $(-1)\mathbf{v} = -\mathbf{v}$.

```
my_vector = [1,2,3]
twice_the vector = 2*[1,2,3]
print(twice_the vector)

>> [1,2,3,1,2,3]
```

Why?

Many folks have run into this

Properties [edit]

Scalar multiplication obeys the following rules *(vector in boldface)*:

- Additivity in the scalar: $(c + d)\mathbf{v} = c\mathbf{v} + d\mathbf{v}$;
- Additivity in the vector: $c(\mathbf{v} + \mathbf{w}) = c\mathbf{v} + c\mathbf{w}$;
- Compatibility of product of scalars with scalar multiplication: $(cd)\mathbf{v} = c(d\mathbf{v})$;
- Multiplying by 1 does not change a vector: $1\mathbf{v} = \mathbf{v}$;
- Multiplying by 0 gives the zero vector: $0\mathbf{v} = \mathbf{0}$;
- Multiplying by −1 gives the additive inverse: $(-1)\mathbf{v} = -\mathbf{v}$.

```
my_vector = [1,2,3]
twice_the vector = 2*[1,2,3]
twice_the vector = [1,2,3]+[1,2,3]
print(twice_the vector)

>> [1,2,3,1,2,3]
```

Why? Because lists define addition as **concatenation**

Introducing numpy

```
import numpy as np
my_1d_array = np.array([1, 2, 3])
twice_the vector = 2*my_1d_array
print(twice_the vector)

>> [2 4 6]
```

An extension of the base python math libraries to handle scalars, vectors, matrices and other building blocks of linear algebra

NumPy

Main namespaces

Regular/recommended user-facing namespaces for general use:

- numpy
- numpy.exceptions
- numpy.fft
- numpy.linalg
- numpy.polynomial
- numpy.random
- numpy.strings
- numpy.testing
- numpy.typing

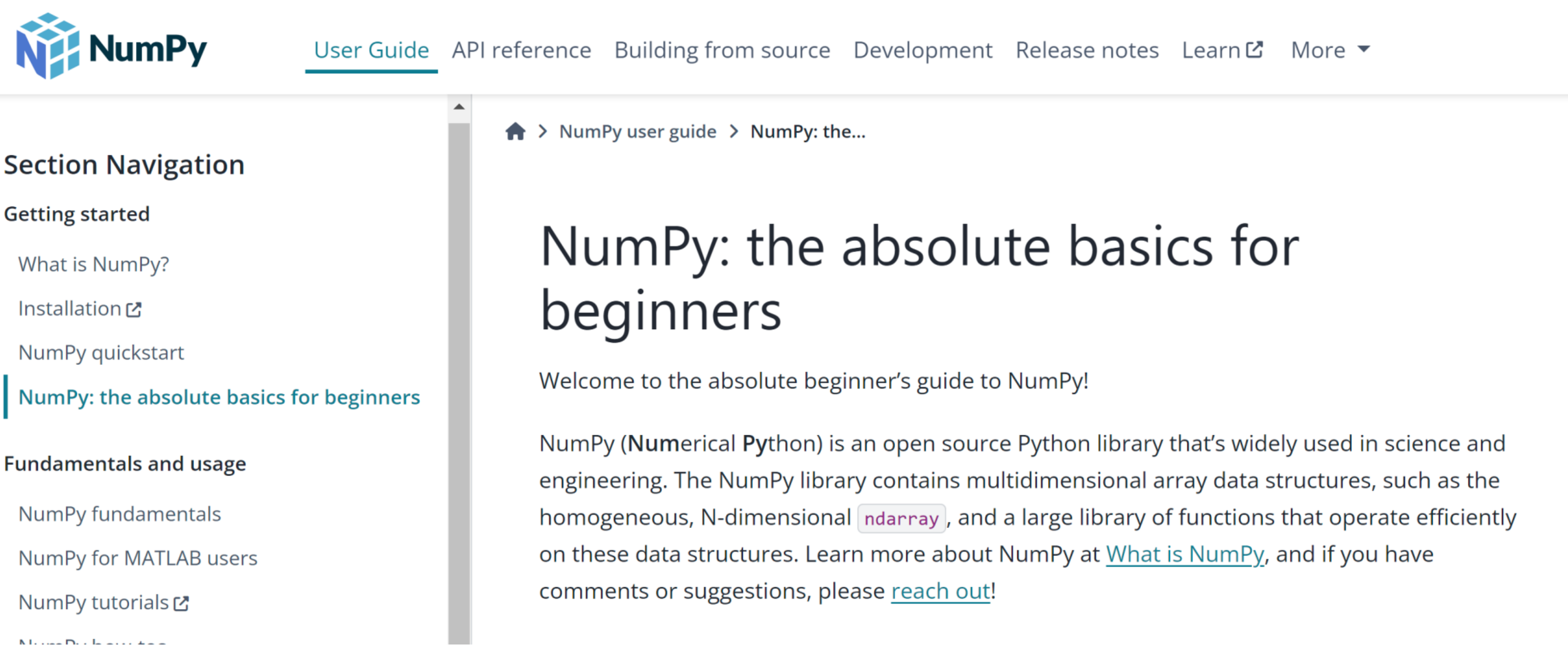
Step 1 when using a new library

Read the docs!

https://numpy.org/doc/stable/user/whatisnumpy.html

Today we will run through key points you need to know – but this is a terrible way to learn!

Go open the docs and read the user guide for more detailed examples. This will pay off, since we will use numpy for everything!



What is NumPy?

NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.

At the core of the NumPy package, is the `ndarray` object. This encapsulates *n*-dimensional arrays of homogeneous data types, with many operations being performed in compiled code for performance. There are several important differences between NumPy arrays and the standard Python sequences:

- NumPy arrays have a fixed size at creation, unlike Python lists (which can grow dynamically). Changing the size of an `ndarray` will create a new array and delete the original.
- The elements in a NumPy array are all required to be of the same data type, and thus will be the same size in memory. The exception: one can have arrays of (Python, including NumPy) objects, thereby allowing for arrays of different sized elements.
- NumPy arrays facilitate advanced mathematical and other types of operations on large numbers of data. Typically, such operations are executed more efficiently and with less code

Why is NumPy fast?

Vectorization describes the absence of any explicit looping, indexing, etc., in the code - these things are taking place, of course, just “behind the scenes” in optimized, pre-compiled C code. Vectorized code has many advantages, among which are:

- vectorized code is more concise and easier to read
- fewer lines of code generally means fewer bugs
- the code more closely resembles standard mathematical notation (making it easier, typically, to correctly code mathematical constructs)
- vectorization results in more “Pythonic” code. Without vectorization, our code would be littered with inefficient and difficult to read `for` loops.

Broadcasting is the term used to describe the implicit element-by-element behavior of operations; generally speaking, in NumPy all operations, not just arithmetic operations, but logical, bit-wise, functional, etc., behave in this implicit element-by-element fashion, i.e., they broadcast. Moreover, in the example above, `a` and `b` could be multidimensional arrays of the same shape, or a scalar and an array, or even two arrays with different shapes, provided that the smaller array is “expandable” to the shape of the larger in such a way that the resulting broadcast is unambiguous. For detailed “rules” of broadcasting see Broadcasting.

How to import

Shortcut for frequently used packages

```python
import numpy as np
import numpy.linalg as la

# Example: Generate a vector and a matrix
my_vec = np.array([1,2,3])
my_matrix = np.array([[1, 2], [3, 4]])

# Example: Calculating the determinant of a matrix
determinant = np.linalg.det(my_matrix)
determinant = la.det(my_matrix)
print(f"The determinant of the matrix is: {determinant}")
```

Remember the syntax for calling functions

object.method_name(arguments, …)

Generate a 1-Dimensional NumPy Array

Matrix: a set of numbers arranged in rows and columns to form a rectangular array.

Generating a matrix:

variable name square brackets around your array

my_1d_array = np.array([1, 2, 3, 4])

```
my_1d_array = np.array([1, 2, 3, 4])
print(my_1d_array)
print(type(my_1d_array))

>> [1 2 3 4]
>> <class 'numpy.ndarray'>
```

Generate a 1-Dimensional NumPy Array

Generating a matrix with a data type:

variable name   square brackets around your array

my_1d_array = np.array([1, 2, 3, 4], dtype=np.int8)

```
my_1d_array = np.array([1, 2, 3, 4],
dtype=np.int8)
print(my_1d_array)
print(type(my_1d_array[0]))

>> [1 2 3 4]
>> <class ‘numpy.int8>
```

Generate a 1-Dimensional NumPy Array

Generating a matrix with a boolean type:

variable name

square brackets around your array

my_1d_array = np.array([0, 0, 1, 1], dtype='bool')

```
my_1d_array = np.array([0, 0, 1, 1],
dtype = 'bool')
print(my_1d_array)
print(type(my_1d_array[0]))

>> [False False True True]
>> <class 'numpy.bool_'>
```

Generate a 2-Dimensional NumPy Array

Generating a 2D matrix:

TWO square brackets for a matrix

```
my_2d_array = np.array([[1, 2, 3, 4]])
```

Use .ndim and .shape for info on array:

.ndim: dimension of your array

.shape: shape of array

Generate a 2-Dimensional NumPy Array

Generating a 2D matrix:

TWO square brackets for a matrix

my_2d_array = np.array([[1, 2, 3, 4]])

```
my_2d_array = np.array([[1, 2, 3, 4]])
print(my_2d_array)
print(my_2d_array.ndim)
print(my_2d_array.shape)

>> [[1 2 3 4]]
>> 2
>> (1, 4)
```

Generate a 2-Dimensional NumPy Array

Generating a 2D matrix:

use square brackets to separate rows

my_2d_array = np.array([[1], [2], [3], [4]])

```
my_2d_array = np.array([[1],[2],[3],[4]])
print(my_2d_array)
print(my_2d_array.ndim)
print(my_2d_array.shape)

>> [[1]
    [2]
    [3]
    [4]]
>> 2
>> (4, 1)
```

Generate a 2-Dimensional NumPy Array

Generating a 2D matrix:

Choose the shape you want by placing brackets accordingly

my_2d_array = np.array([[1, 2], [3, 4]])

$$a2D = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

```
my_2d_array = np.array([[1,2],[3,4]])
print(my_2d_array)
print(my_2d_array.shape)
```

✓ 0.0s

```
[[1 2]
 [3 4]]
(2, 2)
```

Generate a 3-Dimensional NumPy Array

Generating a 3D tensor:

THREE square brackets for a tensor

my_3d_array = np.array([[[1, 2], [3, 4]], [[5, 6],[7, 8]], [[9, 10],[11, 12]]])

```
my_3d_array = np.array([[[1, 2],[3, 4]],[[5, 6],[7,
8]],[[9, 10],[11, 12]]])
print(my_3d_array)
print(my_3d_array.ndim)
print(my_3d_array.shape)

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

Generate an array with np.arange()

We can auto populate an array, similar to for loops.

Use np.arange(start, stop, step_size)

```
my_1st_array = np.arange(5)
my_2nd_array = np.arange(0,10)
my_3rd_array = np.arange(0,10,0.5)
print(my_1st_array)
print(my_2nd_array)
print(my_3rd_array)
my_2d_array = np.array([np.arange(4), np.arange(4)])

>> [0 1 2 3 4]
>> [0 1 2 3 4 5 6 7 8 9]
>> [0 0.5 1 1.5 2 2.5 3 3.5 4 4.5 5 …]
>> [[0 1 2 3]
    [0 1 2 3]]
```

Generate an array with np.linspace()

We can auto populate an array, similar to for loops.

Similar to np.arange, np.linspace can generate an array, but instead of the *step size*, provide **number of steps** for your array.

Use np.linspace(start, stop, number_of_steps)

```
my_1st_array = np.linspace(0, 10, 5)
print(my_1st_array)
my_2d_array = np.array([np.linspace(0,10,5),
np.linspace(0,10,5)])
print(my_2d_array)

>> [0 2.5 5 7.5 10]
>> [[0 2.5 5 7.5 10]
    [0 2.5 5 7.5 10]]
```

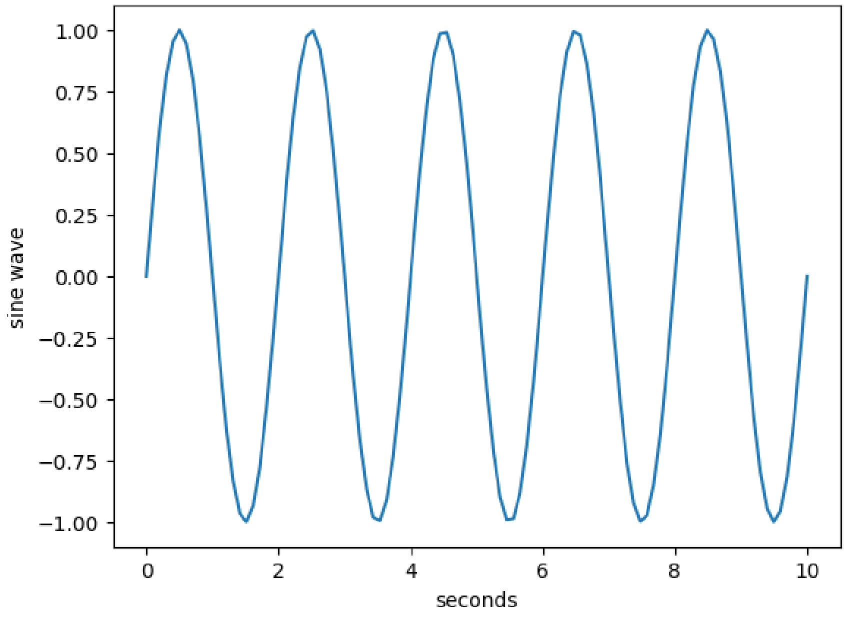
Generate a sine graph

With .arange and .linspace, plotting data becomes a little easier. Let’s create a sine graph.

```
import matplotlib.pyplot as plt

time = np.linspace(0, 10, 100)
sine_graph = np.sin(time * np.pi)

fig, ax = plt.subplots()
ax.plot(time, sine_graph)
ax.set_xlabel(‘seconds’)
ax.set_xlabel(‘sine wave’)
plt.show()
```



Generate an array of zeros and ones

We can generate an array of zeros and ones.

Use np.zeros(num_of_elements)

Use np.ones(num_of_elements)

```
zeros = np.zeros(5)
2d_zeros = np.zeros((2, 2))
ones = np.ones(5)
2d_ones = np.ones((2, 2))
print(zeros)
print(ones)
print(2d_zeros)
print(2d_ones)

>> [0 0 0 0 0]
>> [[0 0]
    [0 0]]
>> [1 1 1 1 1]
>> [[1 1]
    [1 1]]
```

Generate a random array

We can generate a random array with np.random

np.random.randint(min_num, max_num, size = (size of array))

Note* max_num is **not** included in the matrix.

```
random_array = np.random.randint(7, size = (3, 3))
print(random_array)

>> [[4 5 2]
    [3 0 5]
    [2 6 0]]
```

Generate a random array

Let's revisit dtype = 'bool'

```
random_array = np.random.randint(7, size = (3, 3))
truth = random_array > 3
print(random_array)
print(truth)

>> [[4 5 2]
    [3 0 5]
    [2 6 0]]

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

Generate a random array

Let's revisit dtype = 'bool'

```
array = np.random.randint(0, 2, size = (3, 3))
bool_array = np.array(array, dtype='bool')
print(array)
print(bool_array)

>> [[1 0 0]
    [0 0 0]
    [1 1 0]]

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

Indexing a NumPy array

Similar to lists, we can index NumPy arrays:

```
index_1d_array = np.array([5, 8, 1])
print(index_1d_array[1])
>> 8

index_2d_array = np.array([[1, 2, 3],[3, 4, 5]])
print(index_2d_array[1, 2])
>> 5

index_3d_array = np.array([[[1, 2],[3, 4]],[[5, 6],[7,
8]],[[9, 10],[11, 12]]])
print(index_3d_array[2, 1, 0])
>> 11
```

Indexing a NumPy array

You can index an rows, columns, or subsets

Grab row

```
index_2d_array = np.array([[1,2,3],[4,5,6],[7,8,9]])
print(index_2d_array[1,:])
>> [4,5,6]
```

Grab column

```
index_2d_array = np.array([[1,2,3],[4,5,6],[7,8,9]])
print(index_2d_array[:,2])
>> [3,6,9]
```

Grab submatrix

```
index_2d_array = np.array([[1,2,3],[4,5,6],[7,8,9]])
print(index_2d_array[1:,:])
>> [[4 5 6] [7 8 9]]
```

Use ":" to start from the beginning or until the end.

[1:] -> index 1 to the end. [:, 3] -> beginning **up to** index 3

Indexing a NumPy array

You can index an rows, columns, or subsets

Grab row

```
index_2d_array = np.array([[1,2,3],[4,5,6],[7,8,9]])
print(index_2d_array[1,:])
>> [4,5,6]
```

Grab column

```
index_2d_array = np.array([[1,2,3],[4,5,6],[7,8,9]])
print(index_2d_array[:,2])
>> [3,6,9]
```

Grab submatrix

```
index_2d_array = np.array([[1,2,3],[4,5,6],[7,8,9]])
print(index_2d_array[1:,:])
>> [[4 5 6] [7 8 9]]
```

Careful: slicing will index

<u>up to</u> the number you give. [0:2] -> 0, 1

Finding entries with boolean expressions

A little extra: numpy.array has a special feature built in that will find values for you.

```
random_array = np.random.randint(0, 100, size = (3, 3))
print(random_array)
even_and_above_50 = random_array[(random_array > 50) &
(random_array % 2 == 0)]
print(even_and_above_50)

>> [[68 59 25]
     [0 72 68]
     [65 83 28]]

>> [68 72 68]
```

np.where

A little extra: np.where is another way of finding where a specific condition is met; you can either get indices where condition holds, or evaluate an expression on those indices

```
import numpy as np
a = np.arange(2,8)
print(a)
print(np.where(a < 5))            # Get indices where a < 5
print(np.where(a < 5, a, 10*a)) # Replace indices where a < 5 w 10*a
print(a[np.where(a < 5)])         # Grab sub-array where a < 5
```

✓ 0.0s

```
[2 3 4 5 6 7]
(array([0, 1, 2], dtype=int64),)
[ 2  3  4 50 60 70]
[2 3 4]
```

Reshaping NumPy arrays

Given a NumPy array, we can reshape the matrix.

Use .reshape(size). Make sure size can unpack all values evenly.

```
random_matrix = np.random.randint(10, size=(1, 10))
print(random_matrix)
>> [5 2 7 0 3 2 9 6 4 5]

new_random = random_matrix.reshape(5, 2)
print(new_random)
print(new_random.shape)
>> [[5, 2],
    [7, 0],
    [3, 2],
    [9, 6],
    [4, 5]]

>> (5, 2)
```

Universal functions

Numpy comes with a number of functions defined to apply component-wise across an array.

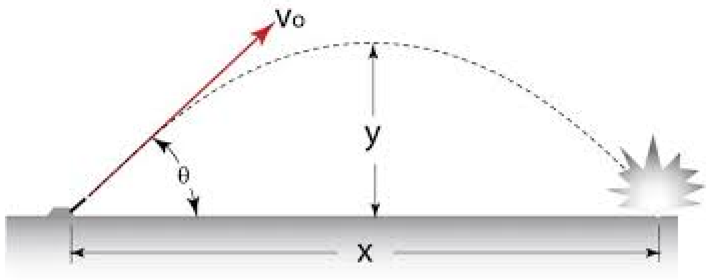
This behavior is referred to as **broadcasting**

```
B = np.arange(3)
print(B)
print(np.exp(B))  # Evaluate exponential applied componentwise
print(np.sqrt(B)) # Evaluate square root applied componentwise
C = np.array([2., -1., 4.])
print(np.add(B, C)) # Add B+C componentwise
```

✓ 0.0s

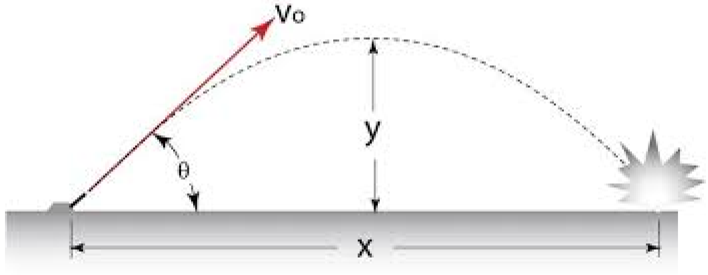
```
[0 1 2]
[1.         2.71828183 7.3890561 ]
[0.         1.         1.41421356]
[2. 0. 6.]
```

Cannon Problem



You are launching a projectile
with an initial velocity of $v_0 = 10$ m/s
at an angle θ towards a target at x = 3 m away. What is the maximum height of the projectile?

Cannon Problem



You are launching a projectile with an initial velocity of $v_0 = 10$ m/s at an angle θ towards a target at x = 3 m away. What is the maximum height of the projectile?

$$x(t) = v_0 \cos(\theta)\, t$$
$$y(t) = v_0 \sin(\theta)\, t - \frac{1}{2} g t^2$$
$$y'(t) = v_0 \sin(\theta) - gt$$
$$y'(t_f) = 0$$

$$\Rightarrow \quad t_f = \frac{v_0 \sin(\theta)}{g}$$

New HW out due the usual time:
We will set aside the following lecture to work through it together, so don't start it yet unless you want a challenge!

In-Class 07: NumPy

Do this with a partner.

Turn in as a pair on Canvas.

Tips for pair programming:

- Switch off who is typing.
- The person who is not typing should:
  - Make comments or suggest potential solutions
  - Be “devil’s advocate”: what are potential issues with what is being typed
  - Suggest other things to explore

At-Home: Complete in-class