Introduction to Python

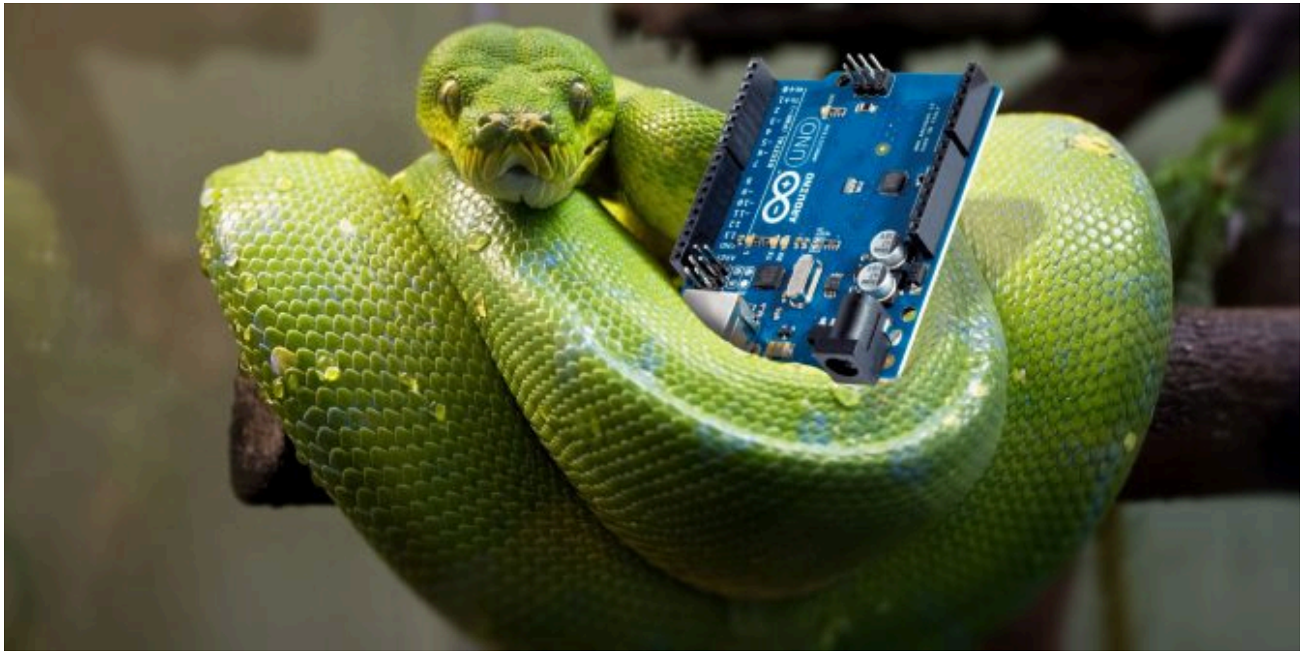
Melody Y. Huang

m.huang@ucla.edu

UCLA Masters of Applied Economics Bootcamp

Motivation

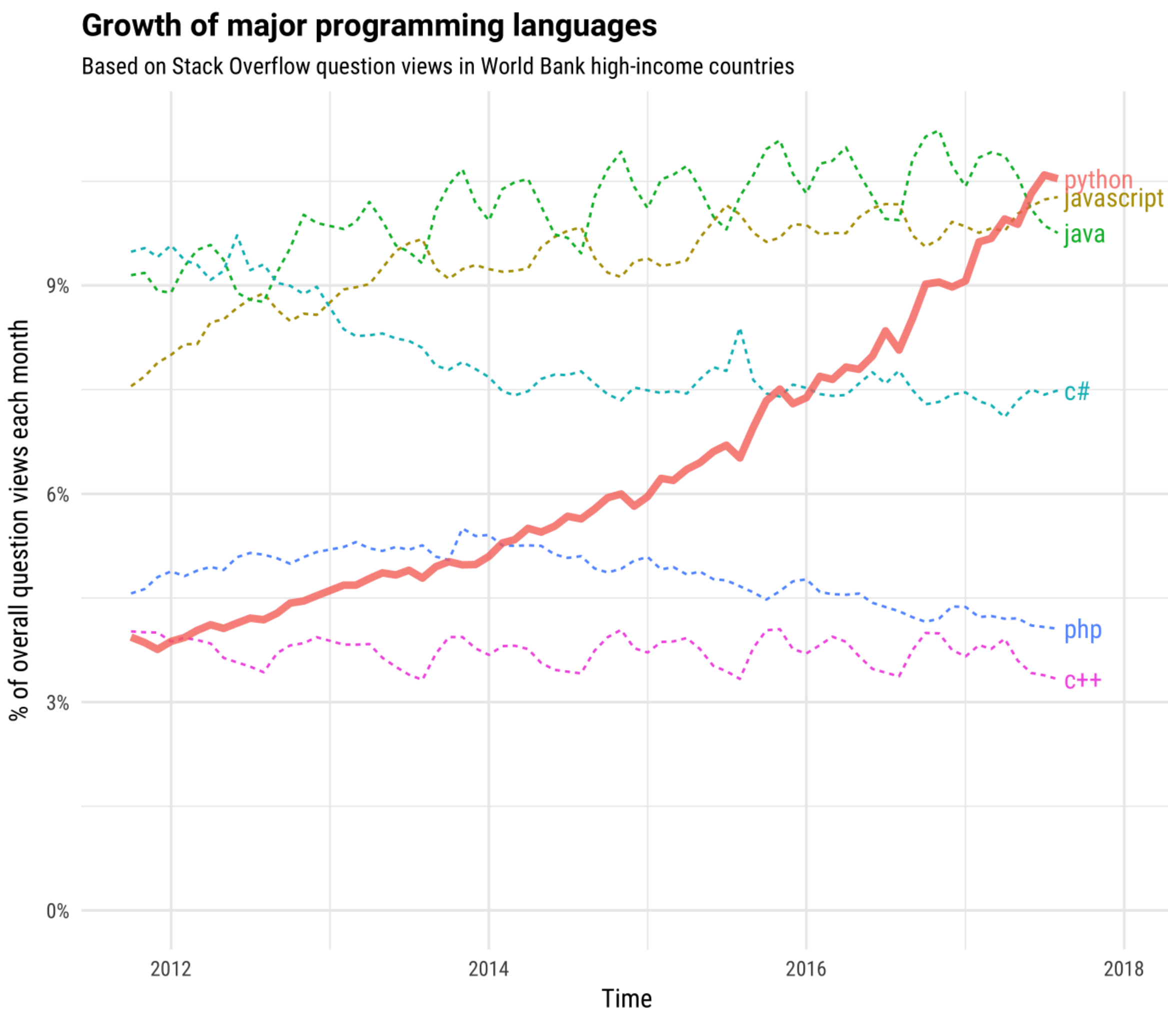
- What is Python?



Motivation (cont.)

- Why learn Python?
  - Data Science
  - Machine Learning

Motivation (cont.)



Installation

- Recommended:
    - anaconda distribution
        - Will provide access to popular IDE's like Spyder and Jupyter Notebook, and will auto-install many popular packages
- Alternatives:
    - Google Colab
    - Terminal/Console (in line, or as a script)
    - PyCharm
    - iPython

Installation (cont.)

- Other helpful things to install:
  - brew 🍺 (Link: https://brew.sh/)
  - pip (`pip install [package_name]`)
    - To install pip: `easy_install pip`

Packages in Python

- Packages (or libraries) in Python provide pre-written functions and objects
- If we want to use a function from a specific library, we have to import that specific library into our workspace

Packages in Python (cont.)

- Functions within the library can be accessed by calling the library name, followed by a period, and then the function name:

```
#Import library
import [library_name]
#Access function
[library_name].[function_name]
```

Packages in Python (cont.)

- If you just want to import a single function:

  ```
  from [library_name] import [function_name]
  ```

  - If you do that, you can simply call the function name.

- You can even rename functions:

  ```
  from [library_name] import [function_name]
  as MyFunction
  ```

Packages in Python (cont.)

- If you want to import an entire library, you can do so by:

```
from [library_name] import *
```

- This is usually considered bad practice due to naming conflicts.
- **Don't do this!**

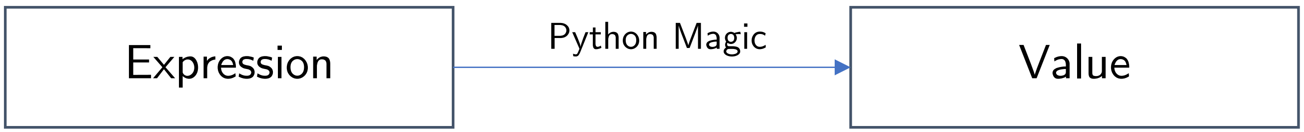
Expressions

- Python is really a fancy calculator:

```
3+4
```

```
7
```

- When we type stuff into the console, this is known as an *expression*.
- Expressions are evaluated by Python to return a *value*.



Expressions (cont.)

- We can use *variables* to help hold expressions.

```
x = 3+4
print(x)
```

```
7
```

- When we tell Python to do something explicitly (i.e., print, create a variable), we call this a *statement*.

Data Types

- Expressions can take on different data types.
- Common data types:
  - Numerics (numbers)
  - Strings (characters)
  - Boolean (true/false)
  - Lists

Data Types (cont.)

- Different data types will have various attributes/methods that are associated with them.
- We can access these attributes by using a period:
  `[var_name].[attribute]`
- There are mutable and immutable types
  - Fancy way to say that you can change objects by accessing

Strings

- Strings contain characters
- Example of some useful string attributes:
  - split – splits a string on particular characters
  - upper – converts everything to uppercase
  - find – identifies specific elements in the string

Strings (cont.)

```
#Small example
x = "Hello World!"
x.split(" ")
```

```
"Hello" "World!"
```

Basic Functions

- `len(x)`
  - Returns the length of an object
  - Equivalent to R's `length()` function

Basic Functions

- `len(x)`
  - Returns the length of an object
  - Equivalent to R's `length()` function
- `set(x)`
  - Returns unique items
  - Equivalent to R's `unique()` function
- Assignment operator: = (**Unlike in R:** `<-`)

Range Function

- `range(x)`:
  - Returns a sequence of numbers of length x, starting from 0
  - Will create a range object – to view the contents, you have to write a for loop
  - Example:
    - `range(10)`
      - Actually returns `0, 1, 2, 3, 4, 5, 6, 7, 8, 9`
      - NOT: `0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10`

Range Function

- `range(x)`:
    - Returns a sequence of numbers of length x, starting from 0
    - Will create a range object – to view the contents, you have to write a for loop
    - Example:
        - `range(10)`
            - Actually returns `0, 1, 2, 3, 4, 5, 6, 7, 8, 9`
            - **NOT**: `0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10`

> Note: The range() object is an iterable, which means to see the contents of it, you have to use a for loop. For example, try: `print(range(10))`.

White Spaces

- White spaces matter!
- There are no brackets in Python, so your computer determines the code folding by indentation/spaces

```
#In R:
for(i in 1:10){
    print(i)
}
var1<-i
print(var1)

#You can also write:
for(i in 1:10){
    print(i)
} var1<-i; print(var1)
```

```
#In Python:
for i in range(10):
    print(i+1)
var1=i
print var1

for i in range(10):
    print(i+1)
    var1=i
    print(var1)
```

Looping over Strings

```
#Try these on your own
for letter in "hello":
  print(letter)

my_string = "UCLA MAE"
for letter in my_string:
  print(letter)

count = 0
for letter in my_string:
  count = count + 1
print(count)
```

Lists [...]

- Similar to R's numeric vectors (`x <- c(1, 2, 3, 4)`)
- We declare lists using brackets: `[1, 2, 3, 4]`
- Example:

```
#Numbers:
num_list = [1,2,3,4]

#Strings
str_list = ["hello", "world"]

#Lists
list_ception = [[1,2,3], [4,5,6], [7,8,9]]
```

Lists (cont.)

- Common methods:
  - `x.append()`
    - Adds stuff to the end of a list
  - `x.insert(i, x2)`
    - i = index at which we insert x2
    - Inserts element at index i
  - `x.pop()`
    - Takes last element from list and remove it to store it elsewhere
    - Like a stack in C++

Lists (cont.)

- A simple example:

```
list1 = [1, 2, 3, 4, 5]
list1.append([6, 7, 8])
print(list1)
```

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

- To add elements 6, 7, 8, you can use `.extend()`

```
list1.extend([6,7,8])
```

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

Indexing

- In Python, we index starting from zero
- Example:

```
x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

  - To access the first element of `x` (i.e., 1), what do we type?

    ```
    print(x[0])
    ```

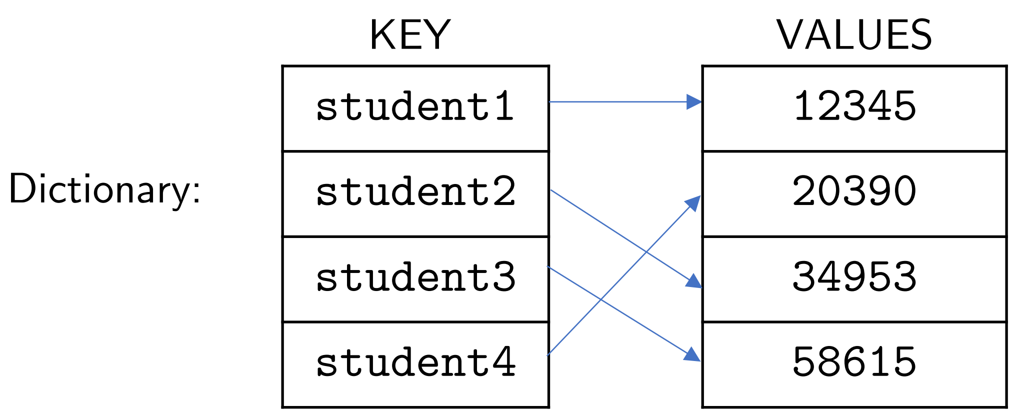
  What if we want numerous elements?

  We can use a colon to access slices of the data.

    ```
    print(x[:5])
    ```

Dictionaries {}

- Python's version of a hashtable
- Each item in your dictionary has both a key and a value associated with it
- Defined by braces: { }
    - More specifically: {key1: value1, key1: value1, …}



Dictionaries (cont.)

Example:

- Let's say we have a dictionary containing student names and their UID. Call this the registrar.

```
registrar = {'Student1': 12345,
             'Student2': 34953,
             'Student3': 58615,
             'Student4': 20390}
```

Dictionaries (cont.)

Example:

- Let's say we have a dictionary containing student names and their UID. Call this the registrar.

```
registrar = {‘Student1’: 12345,
             ‘Student2’: 34953,
             ‘Student3’: 58615,
             ‘Student4’: 20390}
```

- So, to retrieve the UID associated with Student 4, we would simply type: `registrar[‘Student4’]`.

Dictionaries (cont.)

- Why use a dictionary?
  - Very computationally efficient to retrieve information!
  - We can input a key, and retrieve the value without iterating through the entire list!

Iterables

- Certain objects in Python are **iterables**, which means that we can automatically loop through them
  - No need to manually iterate through the list with indexing.
- Let `x = [1,3,0,5,9]`. We don't have to call each item of `x` by `x[i]`, we can just say:

```
for i in x:
    print(i)
#Compare the output to this:
for i in range(len(x)):
    print(x[i])
```

Checkpoint

- Print a sequence of numbers from 0 to 100.
- Print a sequence of even numbers from 2 to 100.
- Create a list of even numbers from 2 to 12.

Checkpoint

- Print a sequence of numbers from 0 to 100.

```
for x in range(101):
    print(x)
```

- Print a sequence of even numbers from 2 to 100.

```
for x in range(99):
    print(x+2)
```

- Create a list of even numbers from 2 to 12:

```
numbers = []
for x in range(6):
    numbers.append((x+1)*2)
```

List Comprehension

- List comprehension is a way to define lists and dictionaries in a way where you have a nested for loop within your list.
- Let's say we want to create a list from 1 to 10.
- You could write:

```
Numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

- Alternatively, using list comprehension:

```
Numbers = [x+1 for x in range(10)]
```

List Comprehension (cont.)

- Provides an elegant way to set up lists
- Let's say you have a really messed up list that contains lists within itself (a list-ception)
- The inner list contains 6 elements:
  1. Open Price
  2. High
  3. Low
  4. Close
  5. Volume Traded
  6. Market Cap

List Comprehension (cont.)

- To separate each element so that you can have just a list that contains all opening prices, you can use list comprehension:

```
data = [[42, 53, 10, 25, 2300, 5112346],
        [12, 32, 12, 31, 2600, 1094529],
                 …
        [23, 51, 23, 25, 2015, 1034951]]
```

List Comprehension (cont.)

- To separate each element so that you can have just a list that contains all opening prices, you can use list comprehension:

```
data = [[42, 53, 10, 25, 2300, 5112346],
        [12, 32, 12, 31, 2600, 1094529],
                  …
        [23, 51, 23, 25, 2015, 1034951]]
```

List Comprehension (cont.)

- To separate each element so that you can have just a list that contains all opening prices, you can use list comprehension:

```
data = [[42, 53, 10, 25, 2300, 5112346],
        [12, 32, 12, 31, 2600, 1094529],
                 …
        [23, 51, 23, 25, 2015, 1034951]]
```

This is the **first inner list** of the larger list.
So to call it, we write: `data[0]`.
But this returns the entire inner list!
We only want the first element of the inner list.

List Comprehension (cont.)

- To separate each element so that you can have just a list that contains all opening prices, you can use list comprehension:

```
data = [[42, 53, 10, 25, 2300, 5112346],
        [12, 32, 12, 31, 2600, 1094529],
                  …
        [23, 51, 23, 25, 2015, 1034951]]
```

- To call the first element, we treat `data[0]` as if it is any other list, and write: `data[0][0]`.

List Comprehension (cont.)

- To separate each element so that you can have just a list that contains all opening prices, you can use list comprehension:

```
data = [[42, 53, 10, 25, 2300, 5112346],
        [12, 32, 12, 31, 2600, 1094529],
                   …
        [23, 51, 23, 25, 2015, 1034951]]
```

- To access the first elements of the i-th list: `data[i][0]`.

List Comprehension (cont.)

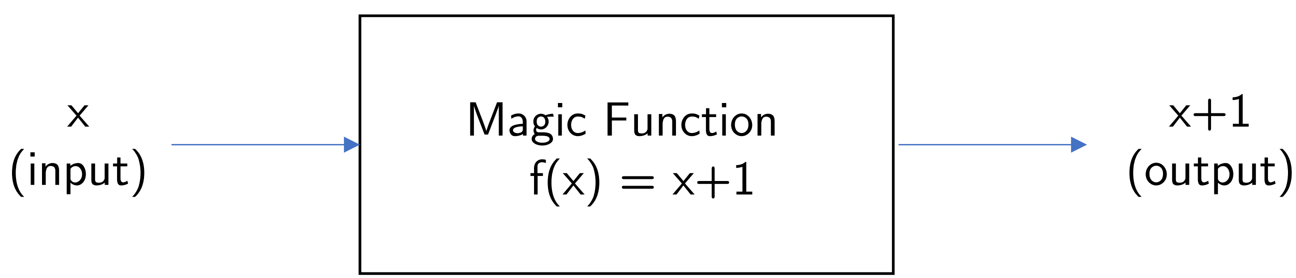
- To separate each element so that you can have just a list that contains all opening prices, you can use list comprehension:

```
#List comprehension:
open = [data[i][0] for i in range(len(data))]

#Equivalent to the following for loop:
open = [] #Empty list
for i in range(len(data)):
    open.append(data[i][0])
```

Functions

- In programming, we use functions so that we can perform repetitive tasks easily.
- Functions take things in as inputs and then produces some sort of result

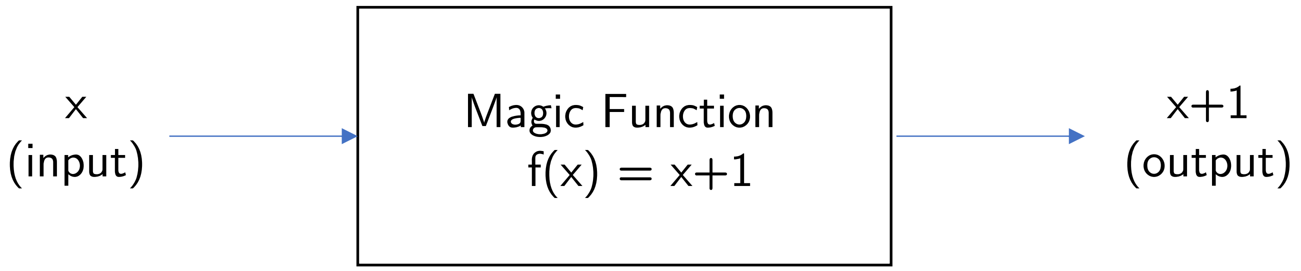


Functions (cont.)

- To declare a function:

```
def functionName(inputVariables):
    [function body]
    return(someVariable)
```

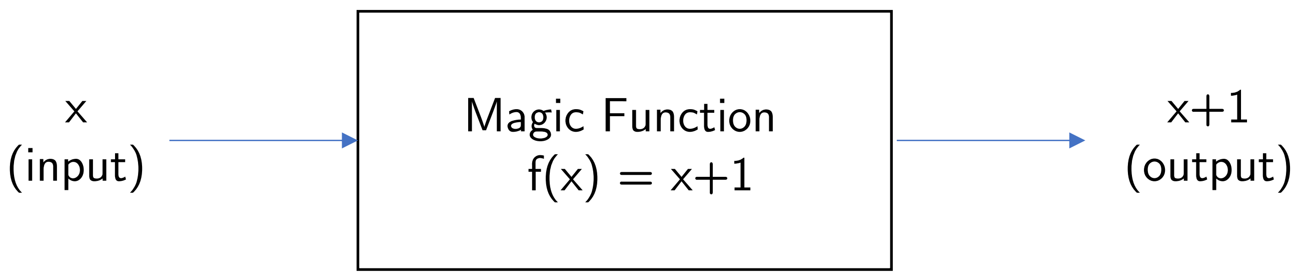
Functions (cont.)



- Programmatically:

```
def magicFunction(x):
    return x+1
```

Functions (cont.)



- Programmatically:

Input variable
(also known as parameters)

```
def magicFunction(x):
    return x+1
```

Indicates to computer that
this is the result (i.e., output)

Functions (cont.)

- When you modify variables within a function, you do not modify them in the global environment.
- As an example, what gets printed out?

```
x = 3

magicFunction(x)

print(x)
```

Functions (cont.)

- So what if we want to store the value that the function outputs?
- Assign the output to a new variable:

```
x = 3
x_new = magicFunction(x)
print(x)
print(x_new)
```

NumPy

- What is NumPy?
  - Scientific computing library on Python
  - Provides the basis of much of the machine learning and stats packages in Python
- To import:

```
import numpy as np
```

NumPy array - np.array()

- Represents vectors and matrices
- An array can only contain elements that are of the same data type
- Each array has:
  - shape – tuple that represents the dimension of the object
  - dtype – an object that tells us what data type the array is
  - ndim – how many dimensions there are

NumPy array (cont.)

Creating arrays:

```
data1 = [6, 7.5, 8, 0, 1]
arr1 = np.array(data1)
print(arr1)
```

```
array([ 6., 7.5, 8., 0., 1.])
```

NumPy array (cont.)

- If you input lists of lists, you end up with a multidimensional array:

```
data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]
arr2 = np.array(data2)
print(arr2.ndim)
```

```
2
```

```
print(arr2.shape)
```

```
(2,4)
```

NumPy array (cont.)

Other ways to create NumPy arrays:

- `np.zeros(x)` – creates an array with dimension/shape specified by x (Note: x can be multidimensional)

```
np.zeros(5)
```

```
array([ 0., 0., 0., 0., 0.])
```

```
np.zeros((3, 6))
```

```
array([[0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.]])
```

Intuition

- What does a NumPy array represent in the context of data analytics?

```
array([[1, 3, … 5, 2],
       [1, 6, … 2, 7],
             …
       [2, 1, … 8, 5]
       [5, 6, … 3, 1]])
```

x_1

NumPy array (cont.)

- np.zeros()
- np.ones()
- np.empty()
- np.eye() – creates an N by N identity matrix
- np.arange() – like the range() function but stores the result in an array object
- np.random.random() – generates random numbers

NumPy Operations

- Arrays are great because we can bypass writing for loops to apply batch operations across all the elements within the array
  - We call this vectorization

- For example:

```
arr = np.array([[1., 2., 3.], [4., 5., 6.]])
print(1 / arr)
```

```
array([[ 1.    , 0.5   , 0.3333],
       [0.25  , 0.2   , 0.1667]])
```

NumPy Operations (cont.)

- Everything happens elementwise usually
- Basic operations:
  - Adding a scalar will just add the scalar to each element in the matrix
  - All arithmetic operates elementwise (same with multiplying b a scalar)
- Array multiplication will **not** be matrix multiplication!
  - For matrix multiplication: `np.dot(x1, x2)`

Common Mathematical Operations

- np.dot(arr.T, arr)
  - Note: the .T operation transposes the matrix
- np.mean(), np.var()
- np.sum()
- np.cumsum(), np.cumprod()
- np.min/np.max

Note: we can compute these quantities over a specific axis only by specifying the axis in the function (i.e., np.sum(axis=1))

Checkpoint

1. Create a 3x3 matrix with values ranging from 0 to 8.
2. Create a 10x10 array with random values and find the minimum and maximum values
3. Create a null vector (vector of all zeroes) of size 10 but the fifth value which is 1
4. Generate a random vector of size 30 and find the mean.

Indexing

- One dimensional arrays behave like lists:

```
arr = np.arange(10)
print(arr)
```

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

```
print(arr[5])
```

```
5
```

```
print(arr[5:8])
```

```
array([5, 6, 7])
```

Indexing (cont.)

- When we access several elements from a list or array (i.e., `arr[5:8]`), we refer to that as *slicing* (because we have a slice 🍰 of data)

- If we assign a scalar value (i.e., just 1 number) to a slice of data, we call this *broadcasting*

Indexing (cont.)

```
#Example continued
arr[5:8] = 12
print(arr[5:8])
```

```
array([ 0, 1, 2, 3, 4, 12, 12, 12, 8, 9])
```

- We see that the value 12 has been propagated (or broadcast) to the entire selection

- Any modifications we make to the slices of the array are also made on the original array!

Indexing (cont.)

We have to be careful:

```
arr_slice = arr[5:8]
arr_slice[1] = 12345
print(arr)
```

```
array([ 0, 1, 2, 3, 4, 12, 12345, 12, 8, 9])
```

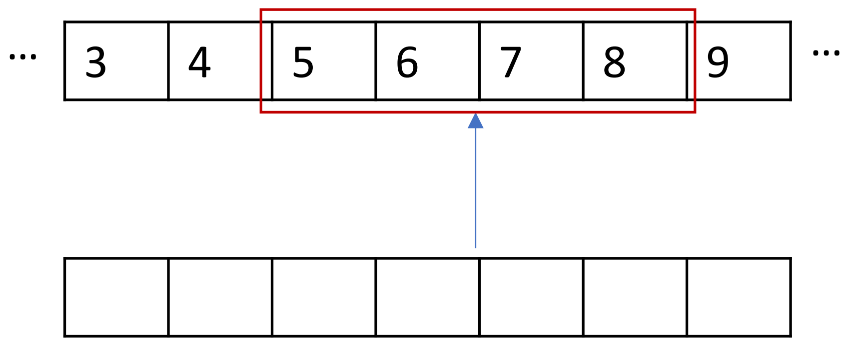
```
arr_slice[:] = 64
print(arr)
```

```
array([ 0, 1, 2, 3, 4, 64, 64, 64, 8, 9])
```

Indexing (cont.)

What's happening here?!

- `arr_slice = arr[5:8]`
- Simply creates a pointer to part of the original object (`arr`).
- As a result, when we modify `arr_slice`, we also modify `arr`!



Indexing in Higher Dimensions

- Similar to how we index nested lists:

```
arr2 = np.array([[1, 2, 3],
                 [4, 5, 6],
                 [7, 8, 9]])
print(arr2[2])
```

```
array([7, 8, 9])
```

  - To access the 3rd element of `arr2[2]`, what would we need to type?

Indexing in Higher Dimensions

- Similar to how we index nested lists:

```
arr2 = np.array([[1, 2, 3],
                 [4, 5, 6],
                 [7, 8, 9]])
print(arr2[2])
```

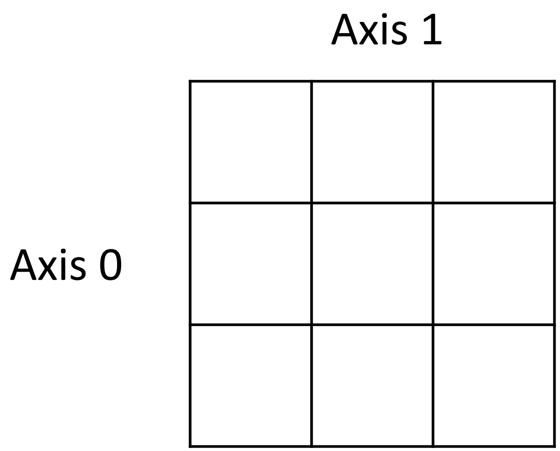
```
array([7, 8, 9])
```

  - To access the 3rd element of `arr2[2]`, what would we need to type?

```
print(arr2[2][2])
```

Indexing with Slices

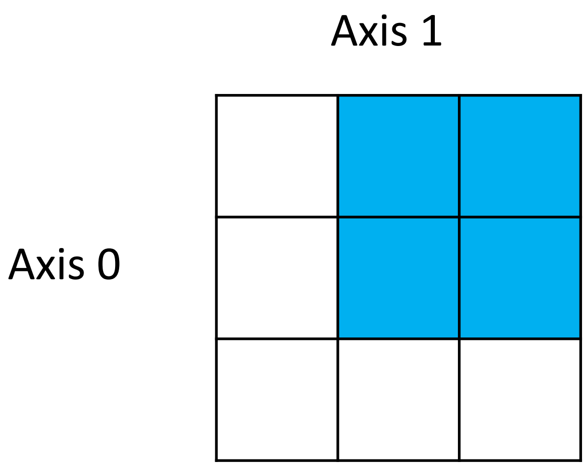
- This seems a little inefficient though
- We can also index and access different dimensions by *slicing* (which is the more common approach)
- In general: `arr[axis0, axis1]`



Indexing with Slices (cont.)

- What part of the matrix would the following return?

```
print(arr[:2, 1:])
```

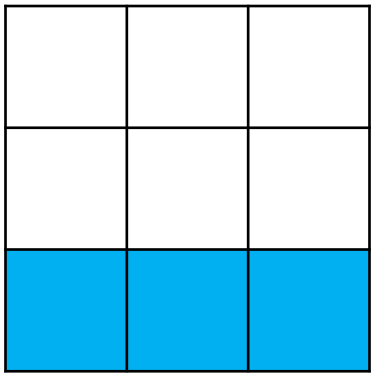


What is the corresponding shape of the matrix?

```
arr.shape
(2,2)
```

Indexing with Slices

- What if I wanted to access the following instead?



Possible solutions:

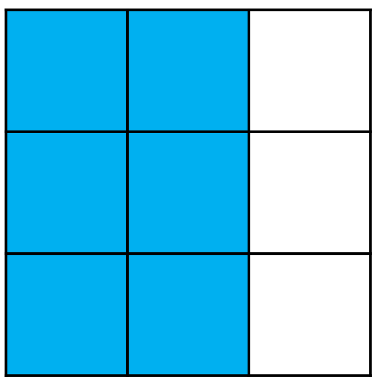
- `arr[2]`
- `arr[2,:]`
- `arr[2:,:]`

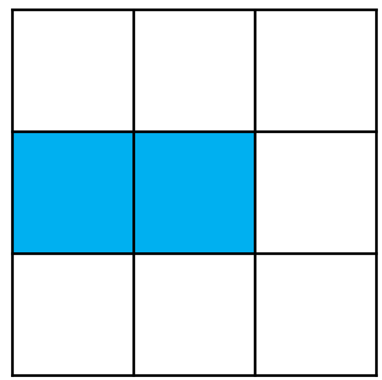
What is the difference?

The shape of `arr[2:,:]` will be (1,3), while the first two calls generate an object that is of shape (3,).

Indexing with Slices

- Try to subset the highlighted parts of the 3x3 matrix using slicing. Check the shapes of each matrix that you end up subsetting.





Boolean Indexing (cont.)

- Similar to R in that we can index using booleans
- In other words, we can target operations to only segments of the array that meet the criteria we input

Boolean Indexing (cont.)

- Example:

```
#Generate data:
data = np.random.random(7, 4)*10-5
print(data)
```

```
array([[-0.048, 0.5433, -0.2349, 1.2792],
       [-0.268, 0.5465, 0.0939, -2.0445],
       [-0.047, -2.026, 0.7719, 0.3103],
       [ 2.1452, 0.8799, -0.0523, 0.0672],
       [-1.0023, -0.1698, 1.1503, 1.7289],
       [ 0.1913, 0.4544, 0.4519, 0.5535],
       [ 0.5994, 0.8174, -0.9297, -1.2564]])
```

Boolean Indexing (cont.)

- Example:

```
#Generate data:
data = np.random.random(7, 4)
print(data)
```

```
array([[-0.048, 0.5433, -0.2349, 1.2792],
       [-0.268, 0.5465, 0.0939, -2.0445],
       [-0.047, -2.026, 0.7719, 0.3103],
       [ 2.1452, 0.8799, -0.0523, 0.0672],
       [-1.0023, -0.1698, 1.1503, 1.7289],
       [ 0.1913, 0.4544, 0.4519, 0.5535],
       [ 0.5994, 0.8174, -0.9297, -1.2564]])
```

Boolean Indexing

- Goal:
  Set all of the elements that are negative to 0.

```
data[data < 0] = 0
print(data)
```

```
array([[ 0., 0.5433, 0., 1.2792],
       [ 0., 0.5465, 0.0939, 0.],
       [ 0., 0., 0.7719, 0.3103],
       [ 2.1452, 0.8799, 0., 0.0672],
       [ 0., 0., 1.1503, 1.7289],
       [ 0.1913, 0.4544, 0.4519, 0.5535],
       [ 0.5994, 0.8174, 0., 0. ]])
```

Boolean Indexing (cont.)

- We can even mask one array using a different boolean array
  - Fancy way to say we can keep data that corresponds to another matrix with true/false entries
  - The nuance is that the length of the boolean array has to match that of the axis from which it is indexing.

Boolean Indexing (cont.)

- `np.where()`
- Equivalent (or at least very similar) to R's ifelse()

```
np.where( [condition],
          [if true, what value to set],
          [else, what value to set])
```

Boolean Indexing (cont.)

```
arr = np.random.random(4, 4)*10-5

array([[ 0.6372, 2.2043, 1.7904, 0.0752],
       [-1.5926, -1.1536, 0.4413, 0.3483],
       [-0.1798, 0.3299, 0.7827, -0.7585],
       [ 0.5857, 0.1619, 1.3583, -1.3865]])

np.where(arr > 0, 2, -2)

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