

COMP 543: Tools & Models for Data Science

Introduction to Python

Chris Jermaine & Risa Myers

Rice University



- Old language, first appeared in 1991
 - But updated often over the years
- Important characteristics
 - Interpreted
 - Dynamically-typed
 - High level
 - Multi-paradigm (imperative, functional, OO)
 - Generally compact, readable, easy-to-use
- Boom in popularity last five years
 - Now the first PL learned in many CS departments

Python: Why So Popular for Data Science?

- Dynamic typing/interpreted
 - Type a command, get a result
 - No need for compile/execute/debug cycle
- Quite high-level: easy for non-CS people to pick up
 - Statisticians, mathematicians, physicists...
- More of a general-purpose PL than R
 - More reasonable target for larger applications
 - More reasonable as API for platforms such as Spark
- Can be used as lightweight wrapper on efficient numerical codes
 - Unlike Java, for example

- Since Python is interpreted, can just fire up Python shell
 - Then start typing
- A first Python program

```
def Factorial (n):  
    if n == 1 or n == 0:  
        return 1  
    else:  
        return n * Factorial (n - 1)
```

```
Factorial (12)
```

- Spacing and indentation
 - Indentation is important
 - No begin/end nor {}
 - Indentation signals code block
- Variables
 - No declaration
 - All type checking is dynamic
 - Just use them

■ Dictionaries

- Standard container type is dictionary/map
- Example: `wordsInDoc = {}` creates an empty dictionary

■ Adding Data

- Add data by saying `wordsInDoc[23] = 16`
- Now can write something like `if wordsInDoc[23] == 16: ...`
- What if `wordsInDoc[23]` is not there? Will crash
- Protect with `wordsInDoc.get(23, 0)` ... returns 0 if key 23 not defined

Encapsulation

- Functions/Procedures

- Defined using `def myFunc (arg1, arg2):`
- Make sure to indent!
- Procedure: no return statement
- Function: return statement

- Remember:

- No marker to end function or procedure
- It ends when you stop indenting

```
def Factorial (n):  
    if n == 1 or n == 0:  
        return 1  
    else:  
        return n * Factorial (n - 1)
```

```
Factorial (12)
```

- Several common forms
- Looping through a range of values
 - Of form `for var in range (0, 50)`
 - Loops for `var in {0, 1, ..., 49}`
- Looping through data structures
 - Example: `for var in dataStruct`
 - loops through each entry in `dataStruct`
 - `dataStruct` can be an array, or a dictionary
 - If array, you loop through the entries
 - If dictionary, you loop through the keys

■ An example

```
a = {}  
a[1] = 'this'  
a[2] = 'that'  
a[3] = 'other'  
for b in a:  
    print(a[b])
```

```
this  
that  
other
```

- NumPy is a Python package
- Most important one for data science!
 - Can use it to do super-fast math, statistics
 - Most basic type is NumPy array
 - Used to store vectors, matrices, tensors
- You will get some reasonable experience with NumPy
- Load with `import numpy as np`

NumPy Arrays: What Are They?

- Multi-dimensional array data structure
- And associated API
- Widely used for data intensive programming...
 - Linear algebra
 - Data science
 - ML

NumPy Arrays: Your Best Friend In DS

- Writing control flow code in DS programming is BAD
 - Kind of like in SQL
- Python is interpreted
 - Time for each statement execution generally large
 - And in DS, you have a lot of data
 - So this code can take a long time:

```
for b in range(0, BIG):  
    a[b] = b
```

```
sum = 0  
for b in a:  
    sum += a[b]
```

- Fewer statements executed, even if the work is the same...
 - ...means better performance!

To Reduce Number of Statements...

- Use NumPy arrays where possible
- Goal: one line of Python to process entire array!
- Some guidelines:
 - Try to replace dictionaries with NumPy arrays
 - Try to replace loops with bulk array operations
 - Backed by efficient, low-level implementations
 - This is known as “vectorized” programming

Creating and Filling NumPy Arrays

- To create a 2 by 5 array, filled with 3.14

```
>>> np.full((2, 5), 3.14)

array([[ 3.14,  3.14,  3.14,  3.14,  3.14],
       [ 3.14,  3.14,  3.14,  3.14,  3.14]])
```

- To create a 2 by 5 array, filled with 0

```
>>> np.zeros((2, 5), 0)

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

More Complicated Creation Examples

- To create an array with odd numbers thru 10

```
>>> np.arange(1, 11, 2)
array([1, 3, 5, 7, 9])
```

- To “tile” an array

```
>>> np.tile (np.arange(1, 11, 2), (1, 2))
array([[1, 3, 5, 7, 9, 1, 3, 5, 7, 9]])

>>> np.tile (np.arange(1, 11, 2), (2, 1))
array([[1, 3, 5, 7, 9],
       [1, 3, 5, 7, 9]])
```

Accessing Subparts of Arrays

■ First we create a 2-d array (matrix)

```
>>> a1 = np.arange(1, 6, 1)
>>> a2 = np.arange(2, 7, 1)
>>> a3 = np.arange(3, 8, 1)
>>> a = np.row_stack ((a1, a2, a3))
>>> a
```

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

a = $\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 6 \\ 3 & 4 & 5 & 6 & 7 \end{bmatrix}$

row
0
1
2

column
0 1 2 3 4

Accessing Subparts of Arrays (cont)

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

- Say we want first two rows:

```
>>> a[1:,:] 
```

- ? Will this work?

Accessing Subparts of Arrays (cont)

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

```
>>> a[1:,:] 
```

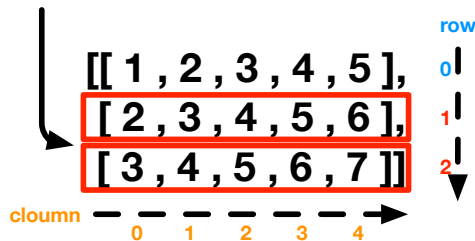
```
array([[2, 3, 4, 5, 6],  
       [3, 4, 5, 6, 7]])
```

```
>>> a[1:]
```

```
array([[2, 3, 4, 5, 6],  
       [3, 4, 5, 6, 7]])
```

Elements of **a** from element 1 to the end

a[1:,:] or **a**[1:]



- Indices start with 0
- Gets rows 1, 2, 3, and so on

Accessing Subparts of Arrays (cont)

- Say we want the last row:

```
>>> a[2:3,]  
array([[3, 4, 5, 6, 7]])
```

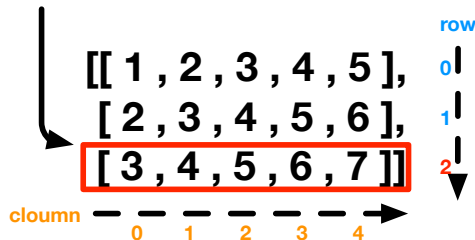
```
>>> a[2:3]  
array([[3, 4, 5, 6, 7]])
```

- Note: still a 2-d array. Want a vector?

```
>>> a[2:3][0]  
  
array([3, 4, 5, 6, 7])
```

Elements of **a** that ≥ 2 but < 3

a[2:3,] or **a**[2:3]



Accessing Subparts of Arrays (cont)

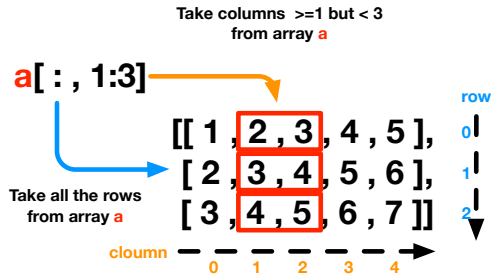
- Now we want the second, third columns:

```
>>> a[:,1:3]
```

```
array([[2, 3],  
       [3, 4],  
       [4, 5]])
```

```
>>> a[:,np.array((1,2))]
```

```
array([[2, 3],  
       [3, 4],  
       [4, 5]])
```



Aggregations Over Arrays

- In statistical/data analytics programming...
 - Tabulations: max, min, etc. over NumPy arrays are ubiquitous
- Key operation allowing this is sum

```
>>> a = np.arange(1, 6, 1)
```

```
>>> a
```

```
array([1, 2, 3, 4, 5])
```

```
>>> a.sum ()
```

```
15
```

Aggregations Over Arrays (cont)

- Can sum along dimension(s) of higher-d array

```
>>> a
```

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

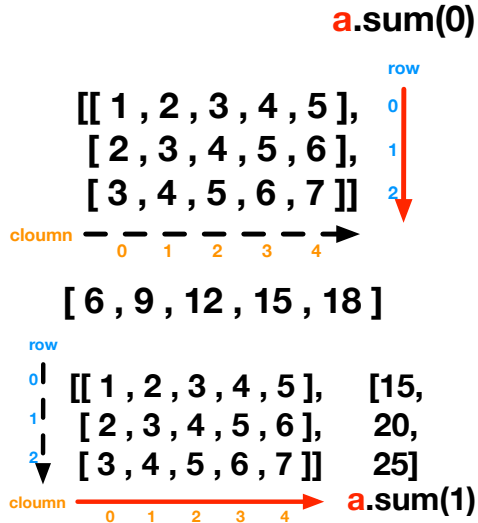
```
>>> a.sum (0)
```

```
array([6, 9, 12, 15, 18])
```

```
>>> a.sum (1)
```

```
array([15, 20, 25])
```

- Think about sum collapsing the array along the specified axis



Aggregations Over Arrays (cont)

- Can find the maximum the same way

```
>>> a  
array([[10, 2, 3, 4, 5],  
       [ 2, 3, 13, 5, 6],  
       [ 3, 4, 5, 6, 7]])
```

```
>>> a.max ()
```

```
13
```

```
>>> a.max (0)
```

```
array([10, 4, 13, 6, 7])
```

```
>>> a.max (1)
```

```
array([10, 13, 7])
```

Aggregations Over Arrays (cont)

- Can find the position of the max as well

```
>>> a
```

```
array([[10,  2,  3,  4,  5],  
       [ 2,  3, 13,  5,  6],  
       [ 3,  4,  5,  6,  7]])
```

```
>>> a.argmax ()
```

```
7
```

```
>>> a.argmax (1)
```

```
array([0, 2, 4])
```


■ Get a tuple of array dimensions

```
>>> a
```

```
array([[10, 2, 3, 4, 5],  
       [ 2, 3, 13, 5, 6],  
       [ 3, 4, 5, 6, 7]])
```

```
>>> a.shape
```

```
(3, 5)
```

Questions?