Topics

1. Why Python?
2. Language Basics
3. Introduction to Numpy
4. Practical Python Tips
5. Other Great References
Why Python?

+ Python is a widely used, general purpose programming language.
+ Easy to start working with.
+ Scientific computation functionality similar to Matlab and Octave.
+ Used by major deep learning frameworks such as PyTorch and TensorFlow.
Topics

1. Why Python?
2. Language Basics
3. Introduction to Numpy
4. Practical Python Tips
5. Other Great References
Topics

1. Why Python?

2. **Language Basics**

3. Introduction to Numpy

4. Practical Python Tips

5. Other Great References

*Note: Code is in Courier New. Console output is prefixed with ‘>>>’*
Language Basics

Does anyone want to guess what this function[^1] (or any line of code) does?

def someGreatFunction(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return someGreatFunction(left) + middle + someGreatFunction(right)

print(someGreatFunction([3,6,8,10,1,2,1]))

[^1]: Example code from Andrej Karpathy’s tutorial: http://cs231n.github.io/python-numpy-tutorial/
Does anyone want to guess what this function\(^1\) (or any line of code) does?

```python
def QuickSort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return QuickSort(left) + middle + QuickSort(right)

print(someGreatFunction([3,6,8,10,1,2,1]))
```

\(^1\) Example code from Andrej Karpathy’s tutorial:  [http://cs231n.github.io/python-numpy-tutorial/](http://cs231n.github.io/python-numpy-tutorial/)
Common Operations

```python
x = 10
y = 3

x + y
x - y
x ** y
x / y
x / float(y)
str(x) + " + " + str(y)
```
Common Operations

\[ x = 10 \quad \# \text{Declaring two integer variables} \]
\[ y = 3 \quad \# \text{Comments start with the hash symbol} \]

\[ x + y \quad \gg 13 \quad \# \text{Addition} \]
\[ x - y \quad \gg 7 \quad \# \text{Subtraction} \]
\[ x ^ {\ast} y \quad \gg 1000 \quad \# \text{Exponentiation} \]
\[ x \div y \quad \gg 3 \quad \# \text{Dividing two integers} \]
\[ x \div \text{float}(y) \gg 3.333.. \quad \# \text{Type casting for float division} \]

\[ \text{str}(x)+\ "\ + \" + \text{str}(y) \quad \gg \"10 + 3\" \quad \# \text{Casting and string concatenation} \]
Built-in Values

True, False  # Usual true/false values
None  # Represents the absence of something
       # A valid object -- can be used like one

x = None  # Variables can be None
array = [1, 2, None]  # Lists can contain None

def func():
    return None  # Functions can return None

if [1, 2] != [3, 4]:  # Can check for equality
    print 'Error!'
Brackets → Indents

- Code blocks are created using indents.
- Indents can be 2 or 4 spaces, but should be consistent throughout the file.
- If using Vim, set this value to be consistent in your .vimrc

```python
def fib(n):
    # Indent level 1: function body
    if n <= 1:
        # Indent level 2: if statement body
        return 1
    else:
        # Indent level 2: else statement body
        return fib(n-1)+fib(n-2)
```
Language Basics

Python is a strongly-typed and dynamically-typed language.

**Strongly-typed**: Interpreter always “respects” the types of each variable.[1]

**Dynamically-typed**: “A variable is simply a value bound to a name.” [1]

**Execution**: Python is first interpreted into bytecode (.pyc) and then compiled by a VM implementation into machine instructions. (Most commonly using C.)

Language Basics

Python is a strongly-typed and dynamically-typed language.

**Strongly-typed**: Interpreter always “respects” the types of each variable.[1]  

**Dynamically-typed**: “A variable is simply a value bound to a name.” [1]  

**Execution**: Python is first interpreted into bytecode (.pyc) and then compiled by a VM implementation into machine instructions. (Most commonly using C.)

What does this mean for me?

Language Basics

Python is a strongly-typed and dynamically-typed language.

**Strongly-typed**: Types will not be coerced silently like in JavaScript.

**Dynamically-typed**: Variables are names for values or object references. Variables can be reassigned to values of a different type.

**Execution**: Python is “slower”, but it can run highly optimized C/C++ subroutines which make scientific computing (e.g. matrix multiplication) really fast.

Language Basics

Python is a strongly-typed and dynamically-typed language.

Strongly-typed: 1 + ‘1’ → Error!

Dynamically-typed: foo = [1,2,3] ...later... foo = ‘hello!’

Execution: np.dot(x, W) + b → Fast!

Lists are **mutable arrays** (think `std::vector`)

```python
names = ['Zach', 'Jay']
names[0] == 'Zach'
names.append('Richard')
len(names) == 3
print names >> ['Zach', 'Jay', 'Richard']
names.extend([['Abi', 'Kevin']])
print names >> ['Zach', 'Jay', 'Richard', 'Abi', 'Kevin']
names = []  # Creates an empty list
names = list()  # Also creates an empty list
stuff = [1, ['hi','bye'], -0.12, None]  # Can mix types
```
List Slicing

List elements can be accessed in convenient ways.
Basic format: `some_list[start_index:end_index]`

```python
def main():
    numbers = [0, 1, 2, 3, 4, 5, 6]
    print(numbers[0:3] == numbers[:3])  # True
    print(numbers[5:] == numbers[5:7])  # True
    print(numbers[:] == numbers)        # True
    print(numbers[-1] == 6)             # True
    print(numbers[-3:] == [4, 5, 6])    # True
    print(numbers[3:-2] == [3, 4])      # True

if __name__ == '__main__':
    main()
```

```
Collections: Tuple

Tuples are immutable arrays

```python
collections = ('Zach', 'Jay')  # Note the parentheses
collections[0] == 'Zach'
len(collections) == 2
print(collections >> ('Zach', 'Jay'))
collections[0] = 'Richard'
>> TypeError: 'tuple' object does not support item assignment
```

```python
eempty = tuple()  # Empty tuple
esingle = (10,)  # Single-element tuple. Comma matters!
```
Dictionaries are **hash maps**

```
phonebook = dict()             # Empty dictionary
phonebook = {'Zach': '12-37'}  # Dictionary with one item
phonebook['Jay'] = '34-23'     # Add another item
print('Zach' in phonebook)     >> True
print('Kevin' in phonebook)    >> False
print(phonebook['Jay'])        >> `34-23`
del phonebook['Zach']          # Delete an item
print(phonebook)               >> {'Jay' : '34-23'}
for name, number in phonebook.iteritems():
    print name, number          >> Jay 34-23
```
Loops

```python
for name in ['Zack', 'Jay', 'Richard']:
    print 'Hi ' + name + '!

>> Hi Zack!
    Hi Jay!
    Hi Richard!

while True:
    print 'We’re stuck in a loop...'
    break # Break out of the while loop

>> We’re stuck in a loop...
```
Loops (cont’d)

What about for (i=0; i<10; i++)? Use range():

```python
for i in range(10):
    print 'Line ' + str(i)  # Want an index also?
```

Looping over a list, unpacking tuples:

```python
for x, y in [(1,10), (2,20), (3,30)]:
    print x, y
```

```plaintext
>> 1 10
   2 20
   3 30
```
Classes

class Animal(object):
    def __init__(self, species, age):
        # Constructor `a = Animal('bird', 10)`
        self.species = species
        # Refer to instance with `self`
        self.age = age
        # All instance variables are public

    def isPerson(self):
        # Invoked with `a.isPerson()`
        return self.species == "Homo Sapiens"

    def ageOneYear(self):
        self.age += 1

class Dog(Animal):
    # Inherits Animal’s methods
    def ageOneYear(self):
        # Override for dog years
        self.age += 7
Importing Modules

Install packages in terminal using `pip install [package_name]`

# Import `os` and `time` modules
import os, time

# Import under an alias
import numpy as np
np.dot(x, y)  # Access components with pkg.fn

# Import specific submodules/functions
from numpy import linalg as la, dot as matrix_multiply

# Not really recommended b/c namespace collisions...
Topics

1. Why Python?
2. Language Basics
3. Introduction to Numpy
4. Practical Python Tips
5. Other Great References
Numpy

Optimized library for matrix and vector computation.

Makes use of C/C++ subroutines and memory-efficient data structures.

(Lots of computation can be efficiently represented as vectors.)

**Main data type:** np.ndarray

This is the data type that you will use to represent matrix/vector computations.

*Note:* constructor function is np.array()
np.ndarray

x = np.array([1, 2, 3])
y = np.array([[3, 4, 5]])
z = np.array([[6, 7], [8, 9]])

print x, y, z

print x.shape

print y.shape

print z.shape
np.ndarray

x = np.array([1,2,3])
>> [1 2 3]
y = np.array([[3,4,5]])
>> [[3 4 5]]
z = np.array([[6,7],[8,9]])
>> [[6 7]
   [8 9]]

print x,y,z

print x.shape
>> (3,)

A list of scalars!

print y.shape
>> (1,3)

A (row) vector!

print z.shape
>> (2,2)

A matrix!
np.ndarray Operations

Reductions: `np.max`, `np.min`, `np.argmax`, `np.sum`, `np.mean`, ...

Always reduces along an axis! (Or will reduce along all axes if not specified.)

(You can think of this as “collapsing” this axis into the function’s output.)

```python
x = np.array([[1, 2], [3, 4]])
print(np.max(x, axis=1))
print(np.max(x, axis=1, keepdims=True))
```
np.ndarray Operations

Reductions: \( \text{np.max}, \text{np.min}, \text{np.amax}, \text{np.sum}, \text{np.mean}, \ldots \)

Always reduces along an axis! (Or will reduce along all axes if not specified.)

(You can think of this as “collapsing” this axis into the function’s output.)

\[
x = \text{np.array}([[1,2],[3,4]])
\]

```python
print(np.max(x, axis = 1))        # >> [2  4]
print(np.max(x, axis = 1, keepdims = True)) # >> [[2] [4]]
```
np.ndarray Operations

Matrix Operations: \texttt{np.dot}, \texttt{np.linalg.norm}, .T, +, -, *, ... 

Infix operators (i.e. +, -, *, **, /) are \textbf{element-wise}.

Matrix multiplication is done with \texttt{np.dot(x, W)} or \texttt{x.dot(W)}

Transpose with \texttt{x.T}

\textbf{Note:} Shapes (N,) \neq (1, N)

\begin{verbatim}
print(np.array([1,2,3]).T)     \quad >> [1 2 3]
np.sum(np.array([1,2,3]), axis = 1) \quad >> Error!
\end{verbatim}
np.ndarray Operations

Matrix Operations: `np.dot`, `np.linalg.norm`, `.T`, `+`, `−`, `∗`, `…`

Infix operators (i.e. `+`, `−`, `∗`, `∗∗`, `/`) are **element-wise**.

Matrix multiplication is done with `np.dot(x, W)` or `x.dot(W)`

Transpose with `x.T`

**Note:** Shapes `(N,) != (N, 1)`

```python
print(np.array([1,2,3]).T)  # >> [1 2 3]
np.sum(np.array([1,2,3]), axis = 1)  # >> Error!
```

**Note:** Scipy and np.linalg have many, many other advanced functions that are very useful!
Indexing

```python
x = np.random.random((3, 4))  # Random (3,4) matrix

x[:,:]  # Selects everything in x

x[np.array([0, 2]), :]  # Selects the 0th and 2nd rows

x[1, 1:3]  # Selects 1st row as 1-D vector
            # and 1st through 2nd elements

x[x > 0.5]  # Boolean indexing
```
Indexing

```python
x = np.random.random((3, 4))  # Random (3,4) matrix

x[:] # Selects everything in x

x[np.array([0, 2]), :] # Selects the 0th and 2nd rows

x[1, 1:3] # Selects 1st row as 1-D vector 
# and 1st through 2nd elements

x[x > 0.5] # Boolean indexing
```

Note: Selecting with an ndarray or range will preserve the dimensions of the selection.
Broadcasting

```python
x = np.random.random((3, 4))  # Random (3, 4) matrix
y = np.random.random((3, 1))  # Random (3, 1) matrix
z = np.random.random((1, 4))  # Random (3,) vector

x + y  # Adds y to each column of x
x * z  # Multiplies z element-wise with each row of x

print((y + y.T).shape)  # Can give unexpected results!
```
Broadcasting

```
x = np.random.random((3, 4))  # Random (3, 4) matrix
y = np.random.random((3, 1))  # Random (3, 1) matrix
z = np.random.random((1, 4))  # Random (3,) vector
x + y  # Adds y to each column of x
x * z  # Multiplies z element-wise with each row of x
print((y + y.T).shape)  # Can give unexpected results!
```

Note: If you’re getting an error, print the shapes of the matrices and investigate from there.
Efficient Numpy Code

Avoid explicit for-loops over indices/axes at all costs.

For-loops will *dramatically* slow down your code (~10-100x).

```python
for i in range(x.shape[0]):
    for j in range(x.shape[1]):
        x[i, j] **= 2

for i in range(100, 1000):
    x[np.arange(100,1000), :] += 5
    for j in range(x.shape[1]):
        x[i, j] += 5
```
Topics

1. Why Python?
2. Language Basics
3. Introduction to Numpy
4. Practical Python Tips
5. Other Great References
List Comprehension

- Similar to `map()` from functional programming languages.
- Can improve readability & make the code succinct.
- Format: `[func(x) for x in some_list]`
- Following are equivalent:
  - `squares = []
   for i in range(10):
     squares.append(i**2)`
  - `squares = [i**2 for i in range(10)]`
- Can be conditional:
  - `odds = [i**2 for i in range(10) if i%2 == 1]`
Convenient Syntax

- Multiple assignment / unpacking iterables
  - \( x, y, z = [\text{'Tensorflow'}, \text{'PyTorch'}, \text{'Chainer'}] \)
  - \( \text{age}, \text{name}, \text{pets} = 20, \text{'Joy'}, [\text{'cat'}] \)
- Returning multiple items from a function
  - \( \text{def some_func():} \)
  
  \( \text{return 10, 1} \)

  \( \text{ten, one = some_func()} \)
- Joining list of strings with a delimiter
  - \( ", \text{.join([1, 2, 3])} == \text{'1, 2, 3'} \)
- String literals with both single and double quotes
  - \( \text{message = 'I like "single" quotes.'} \)
  - \( \text{reply = "I prefer 'double' quotes."} \)
Debugging Tips

- Python has an **interactive shell** where you can execute arbitrary code
  - Great replacement for TI-84 (no integer overflow!)
  - Confused by syntax? Just try it in the shell!
    - `$ python
      Python 2.7.10 (default, Jul 15 2017, 17:16:57)
      >>> 2 ** 5 / 2
      16
      >>> 2 ** (5 / 2)
      4
    - Can import any module (even custom ones in the current directory)
    - Try small test cases in the shell
Debugging Tips (cont’d)

- Unsure of what you can do with an object? Use `type()` and `dir()`!!

```python
>>> class Duck(object):
...     def quack(self): pass
...     
... >>> bird = Duck()
>>> type(bird)
<class '__main__.Duck'>
>>> dir(bird)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__',
'__getattribute__', '__hash__', '__init__', '__module__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', '__weakref__', 'quack']
```
Numpy Debugging

- Print shapes to see if they match what you expect: `print x.shape`
- Print shapes!! Make sure broadcasting is done properly.
- Print types and values.
- Checking if two float arrays are approximately equal (element-wise)
  - `np.allclose(x, y)` # Can also specify tolerance
- Checking if an array is close to zero (e.g. gradient)
  - `np.allclose(x, 0)` # Broadcasting
- Selecting all elements less than 0 from an array
  - `x[x < 0]` # Returns 1-dim array
Environment Management

- Problem:
  - Python 3 is not backward-compatible with Python 2
  - Countless Python packages and their dependencies
  - Different projects require different packages
    - Even worse, different versions of the same package!

- Solution:
  - Keep multiple Python environments that are isolated from each other
  - Each environment...
    - can use different Python versions
    - keeps its own set of packages
    - can be easily replicated (e.g. on a VM, friend’s laptop, etc.)
Anaconda

- Anaconda is a popular Python environment/package manager
  - Install from [https://www.anaconda.com/download/](https://www.anaconda.com/download/)
  - Supports Windows, Linux, macOS
  - Basic workflow
    $ source activate <environment_name>
    <... do stuff ...>
    $ deactivate
  - Other environments won’t be affected by anything you do
  - Allows you to run a different version of Python for each environment
Virtualenv

• Virtualenv is another popular Python environment manager
  ○ Only specifies different packages per environment
  ○ Doesn’t help run different Python version
  ○ Installation from
  ○ Basic workflow
    $ mkdir <environment_directory>
    $ virtualenv <environment_directory>
    $ source <env_dir>/bin/activate
    $ pip install <package>
Topics

1. Why Python?
2. Language Basics
3. Introduction to Numpy/Scipy
4. Practical Python Tips
5. Other Great References
Other Great References

1. Official Python 2 documentation: https://docs.python.org/2/
2. Official Python 2 tutorial: https://docs.python.org/2.7/tutorial/index.html
3. Numpy Quickstart: https://docs.scipy.org/doc/numpy-dev/user/quickstart.html
END OF PRESENTATION
Iterables (cont’d)

Abstraction for *anything you can iterate over*

**Sets**: similar to lists, but without ordering and duplicates

```python
names = set(['Zack', 'Jay'])
names[0]  # TypeError: 'set' object does not support indexing
len(names) == 2
print names  # set(['Zack', 'Jay'])
names.insert('Jay')
print names  # set(['Zack', 'Jay'])  # Ignored duplicate

empty = set()  # Empty set
```