Python Review Session

CS224N
Stanford University
Topics

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

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

Code is in Courier New.
Command line input is prefixed with `'$'`.
Output is prefixed with `'''`.
Environment Management

● Problem
  ○ Different versions of Python
  ○ 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 version
    ■ Keeps its own set of packages (can specify package versions)
    ■ Can be easily replicated
Anaconda

● Anaconda is a popular Python environment/package manager
  ○ Install from https://www.anaconda.com/download/
  ○ Supports Windows, Linux, MacOS
  ○ Can create and manage different isolated environments

Basic workflow

Create a new environment

$ conda create -n <environment_name>
$ conda create -n <environment_name> python=3.7
$ conda env create -f <environment.yml>

Activate/deactivate environment

$ conda activate <environment_name>
<...do stuff...>
$ conda deactivate

Export environment

$ conda activate <environment_name>
$ conda env export > environment.yml
IDEs / text editors for Python

- PyCharm
- Visual Studio Code
- Sublime Text
- Atom
- Vim (for Linux)

Write a Python program in your IDE or text editor of choice.

In terminal, activate conda environment and run program with command:

$ python <filename.py>
Jupyter Notebook / Google Colab

- **Jupyter Notebook**
  - A Jupyter notebook lets you write and execute Python code locally in your web browser
  - Interactive, code re-execution, result storage, can interleave text, equations, and images
  - Can add conda environments to Jupyter notebook

- **Google Colab**
  - [https://colab.research.google.com/](https://colab.research.google.com/)
  - Google’s hosted Jupyter notebook service, runs in the cloud, requires no setup to use, provides free access to computing resources including GPUs
  - Comes with many Python libraries pre-installed
Topics

1. Why Python?
2. Setup
3. **Language Basics**
4. Introduction to NumPy
5. Practical Python Tips
6. Other Great References
Common Operations

\[
x = 10 \quad \text{# Declaring two integer variables}
\]
\[
y = 3 \quad \text{# Comments start with hash}
\]
\[
x + y \quad \text{# Arithmetic operations} \quad \text{>> 13}
\]
\[
x ** y \quad \text{# Exponentiation} \quad \text{>> 1000}
\]
\[
x / y \quad \text{# Dividing two integers} \quad \text{>> 3}
\]
\[
x / \text{float}(y) \quad \text{# Type casting for float division} \quad \text{>> 3.333…}
\]
\[
\text{str}(x) + "+" \quad \text{# Casting integer as string and} \quad \text{>> "10 + 3"}
\]
\[
+ \text{str}(y) \quad \text{string concatenation}
\]
Built-in Values

- **True, False**  
  # Usual true/false values

- **None**  
  # Represents the absence of something

- **x = None**  
  # Variables can be assigned None

- **array = [1, 2, None]**  
  # Lists can contain None

- **def func():**  
  
  return None  
  
  # Functions can return None
Built-in Values

and

or

not

if [] != [None]:
    print("Not equal")

# Boolean operators in Python written as plain English, as opposed to &&, ||, ! in C++

# Comparison operators == and != check for equality/inequality, return true/false values
Brackets → Indents

- Code blocks are created using indents, instead of brackets like in C++
- Indents can be 2 or 4 spaces, but should be consistent throughout file
- If using Vim, set this value to be consistent in your `.vimrc`

```python
def sign(num):
    # Indent level 1: function body
    if num == 0:
        # Indent level 2: if statement body
        print("Zero")
    elif num > 0:
        # Indent level 2: else if statement body
        print("Positive")
    else:
        # Indent level 2: else statement body
        print("Negative")
```
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
collections = ['Zach', 'Jay']
collections[0] == 'Zach'
collections.append('Richard')
print(len(collections) == 3)  # True
print(collections)  # ['Zach', 'Jay', 'Richard']
collections += ['Abi', 'Kevin']
print(collections)  # ['Zach', 'Jay', 'Richard', 'Abi', 'Kevin']
collections = []  # Creates an empty list
collections = 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: \( \text{some-list[start-index:end-index]} \)

\[
\text{numbers} = [0, 1, 2, 3, 4, 5, 6] \\
\text{numbers[0:3]} == \text{numbers[:3]} == [0, 1, 2] \\
\text{numbers[5:]} == \text{numbers[5:7]} == [5, 6] \\
\text{numbers[:]} == \text{numbers} == [0, 1, 2, 3, 4, 5, 6] \\
\text{numbers[-1]} == 6 \quad \# \quad \text{Negative index wraps around} \\
\text{numbers[-3:]} == [4, 5, 6] \\
\text{numbers[3:-2]} == [3, 4] \quad \# \quad \text{Can mix and match}
\]
Collections: Tuples

Tuples are immutable arrays.

names = (‘Zach’, ‘Jay’) # Note the parentheses
names[0] == ‘Zach’
print(len(names) == 2) >> True
print(names) >> (‘Zach’, ‘Jay’)
names[0] = ‘Richard’ >> TypeError: 'tuple' object does not support item assignment
empty = tuple() # Empty tuple
single = (10,) # Single-element tuple. Comma matters!
Collections: Dictionary

Dictionaries are hash maps.

```python
dictionaries = {}

# Empty dictionary
phonebook = {}
phonebook = dict()  # Also creates an 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'}
```

Loops

For loop syntax in Python

Instead of `for (i=0; i<10; i++)` syntax in languages like C++, use `range()`

```python
for i in range(10):
    print(i)
```

```plaintext
>> 1
  2...
  9
  10
```
Loops

To iterate over a list

```python
names = ['Zach', 'Jay', 'Richard']
for name in:
    print('Hi ' + name + '!
```

To iterate over indices and values

```python
# One way
for i in range(len(names)):
    print(i, names[i])
```

```python
# A different way
for i, name in enumerate(names):
    print(i, name)
```
Loops

To iterate over a dictionary

```python
define phonebook as a dictionary with 'Zach': '12-37', 'Jay': '34-23'
for name in phonebook:
    print(name)
for number in phonebook.values():
    print(number)
for name, number in phonebook.items():
    print(name, number)
```

>> Jay
    Zach

>> 12-37
    34-23

>> Zach 12-37
    Jay 34-23

Note: Whether dictionary iteration order is guaranteed depends on the version of Python.
class Animal(object):
    def __init__(self, species, age):
        self.species = species
        self.age = age

    def is_person(self):
        return self.species

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

class Dog(Animal):
    def age_one_year(self):
        self.age += 7

# Constructor `a = Animal('human', 10)`
# Refer to instance with `self`
# Instance variables are public
# Invoked with `a.is_person()`

# Inherits Animal’s methods
# Override for dog years
Model Classes

In the later assignments, you’ll see and write model classes in PyTorch that inherit from `torch.nn.Module`, the base class for all neural network modules.

```python
import torch.nn as nn

class Model(nn.Module):
    def __init__():
        ...

    def forward():
        ...
```
Installing Packages

pip installs Python packages, conda installs packages which may contain software written in any language.

Issues may arise when using pip and conda together. It is best practice to first use conda to install as many packages as possible and use pip to install remaining packages after. [1]

conda install -n myenv [package_name][=optional version number]

Install packages using pip in a conda environment (necessary when package not available through conda)

conda install -n myenv pip  # Install pip in environment
conda activate myenv  # Activate environment
pip install [package_name][==optional version number]  # Install package

pip install -r <requirements.txt>  # Install packages from file

Importing Package Modules

# 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
# Can result in namespace collisions...
Topics

1. Why Python?
2. Setup
3. Language Basics
4. Introduction to NumPy
5. Practical Python Tips
6. 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()`
```python
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 1-D vector!
print(y.shape)                  >> (1, 3)
A (row) vector!
print(z.shape)                  >> (2, 2)
A matrix!

Note: shape (N,) != (1, N) != (N, 1)
np.ndarray Operations

Reductions: `np.max`, `np.min`, `np.amax`, `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
# shape: (3, 2)
x = np.array([[1, 2], [3, 4], [5, 6]])
# shape: (3,)
print(np.max(x, axis = 1))                           >> [2 4 6]
# shape: (3, 1)
print(np.max(x, axis = 1, keepdims = True)) >> [[2] [4] [6]]
```
np.ndarray Operations

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

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

**Element-wise product** (Hadamard product) of matrix A and B, $A \odot B$, can be computed: $A \times B$

**Dot product** and **matrix vector product** (between 1-D array vectors), can be computed:
\begin{align*}
\texttt{np.dot(u, v)} \\
\texttt{np.dot(x, W)}
\end{align*}

**Matrix product / multiplication** of matrix A and B, $AB$, can be computed:
\begin{align*}
\texttt{np.matmul(A, B)} \quad \text{or} \quad A @ B \\
\texttt{np.dot()} \quad \text{can also be used for matrix multiplication, but if A and B are both 2-D arrays, np.matmul()} \quad \text{is preferred.}
\end{align*}

Transpose with $x.T$

**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
x[:, :, np.newaxis]  # 3-D vector of shape (3, 4, 1)
```

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

\[ x = \text{np.random.random}((3, 4)) \quad \# \text{Random (3, 4) matrix} \]
\[ y = \text{np.random.random}((3, 1)) \quad \# \text{Random (3, 1) vector} \]
\[ z = \text{np.random.random}((1, 4)) \quad \# \text{Random (1, 4) vector} \]

\[ x + y \quad \# \text{Adds y to each column of x} \]
\[ x \times z \quad \# \text{Multiplies z (element-wise) with each row of x} \]

**Note**: If you’re getting an error, print the shapes of the matrices and investigate from there.
Broadcasting (visually)

\[
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8 \\
9 & 10 & 11 & 12 \\
\end{array}
\quad \quad
\begin{array}{cccc}
1 & 1 & 1 & 1 \\
2 & 2 & 2 & 2 \\
3 & 3 & 3 & 3 \\
\end{array}
\quad \quad
\begin{array}{cccc}
2 & 3 & 4 & 5 \\
7 & 8 & 9 & 10 \\
12 & 13 & 14 & 15 \\
\end{array}
\]

\[
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8 \\
9 & 10 & 11 & 12 \\
\end{array}
\quad \quad
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
1 & 2 & 3 & 4 \\
1 & 2 & 3 & 4 \\
\end{array}
\quad \quad
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
5 & 12 & 21 & 32 \\
9 & 30 & 33 & 48 \\
\end{array}
\]

\[
\begin{array}{cccc}
x & + & y & z
\end{array}
\]

\[
x \quad + \quad y
\]

\[
x \quad \times \quad z
\]
Broadcasting (generalized)

When operating on two arrays, NumPy compares their shapes element-wise. It starts with the trailing (i.e. rightmost) dimensions and works its way left. Two dimensions are compatible when

1. they are equal, or
2. one of them is 1 (in which case, elements on the axis are repeated along the dimension)

```python
a = np.random.random((3, 4))  # Random (3, 4) matrix
b = np.random.random((3, 1))  # Random (3, 1) vector
c = np.random.random((3, ))   # Random (3, ) vector
```

What do the following operations give us? What are the resulting shapes?

- `b + b.T`
- `a + c`
- `b + c`
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. Setup
3. Language Basics
4. Introduction to NumPy
5. Practical Python Tips
6. Other Great References
List Comprehensions

- 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:

```python
squares = []
for i in range(10):
    squares.append(i**2)
```

```python
squares = [i**2 for i in range(10)]
```

Can be conditional:

```python
odds = [i**2 for i in range(10) if i%2 == 1]
```
Convenient Syntax

Multiple assignment / unpacking iterables
age, name, pets = 20, ‘Joy’, [‘cat’]
x, y, z = (‘Tensorflow’, ‘PyTorch’, ‘Chainer’)

Returning multiple items from a function
def some_func():
    return 10, 1
ten, one = some_func()

Joining list of strings with a delimiter
‘, ‘.join([1, 2, 3]) == ‘1, 2, 3’

String literals with both single and double quotes
message = ‘I like “single” quotes.’
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!)
- Can import any module (even custom ones in the current directory)
- Try out syntax you’re unsure about and small test cases (especially helpful for matrix operations)

```
$ python
Python 3.9.7 (default, Sep 16 2021, 08:50:36)
[Clang 10.0.0 ] :: Anaconda, Inc. on darwin
>> import numpy as np
>> A = np.array([[1, 2], [3, 4]])
>> B = np.array([[3, 3], [3, 3]])
>> A * B
  [[3 6]
   [9 12]]
>> np.matmul(A, B)
  [[9 9]
   [21 21]]
```
# Debugging Tools

<table>
<thead>
<tr>
<th>Code</th>
<th>What it does</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>array.shape</code></td>
<td>Get shape of NumPy array</td>
</tr>
<tr>
<td><code>array.dtype</code></td>
<td>Check data type of array (for precision, for weird behavior)</td>
</tr>
<tr>
<td><code>type(stuff)</code></td>
<td>Get type of variable</td>
</tr>
<tr>
<td><code>import pdb; pdb.set_trace()</code></td>
<td>Set a breakpoint [1]</td>
</tr>
<tr>
<td><code>print(f’My name is {name}’)</code></td>
<td>Easy way to construct a string to print</td>
</tr>
</tbody>
</table>

[1] [https://docs.python.org/3/library/pdb.html](https://docs.python.org/3/library/pdb.html)
Common Errors

**ValueError(s)** are often caused by **mismatch of dimensions** in broadcasting or matrix multiplication.

If you get this type of error, a good first step to debugging the issue is to print out the shape of relevant arrays to see if they match what you expect.

```python
array.shape
```
Topics

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

Official Python 3 documentation: https://docs.python.org/3/


Python tutorial from CS231N: https://cs231n.github.io/python-numpy-tutorial/

Stanford Python course (CS41): https://stanfordpython.com/#/