Python Review Session

CS224N
Stanford University

Topics

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

Topics

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

`Code is in Courier New`.
Command line input is prefixed with **`'$'`**.
Output is prefixed with **`'>>'`**.

Topics

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

Common Operations

```
x = 10                          # Declaring two integer variables
y = 3                           # Comments start with hash
x + y           >> 13           # Arithmetic operations
x ** y          >> 1000         # Exponentiation
x / y           >> 3            # Dividing two integers
x / float(y)    >> 3.333…       # Type casting for float division
str(x) + "+"    >> "10 + 3"     # Casting integer as string and
+ str(y)                        string concatenation
```

Built-in Values

```
True, False

None

x = None

array = [1, 2, None]

def func():

    return None
```

```
# Usual true/false values

# Represents the absence of something

# Variables can be assigned None

# Lists can contain None

# Functions can return None
```

Built-in Values

```
and
or
not
```

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

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

```
# 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`**

```
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.)

[1] https://wiki.python.org/moin/Why%20is%20Python%20a%20dynamic%20language%20and%20also%20a%20strongly%20typed%20language

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?

[1] https://wiki.python.org/moin/Why%20is%20Python%20a%20dynamic%20language%20and%20also%20a%20strongly%20typed%20language

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.

[1] https://wiki.python.org/moin/Why%20is%20Python%20a%20dynamic%20language%20and%20also%20a%20strongly%20typed%20language

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!`

[1] https://wiki.python.org/moin/Why%20is%20Python%20a%20dynamic%20language%20and%20also%20a%20strongly%20typed%20language

Topics

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

Collections: List

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

```
names = ['Zach', 'Jay']
names[0] == 'Zach'
names.append('Richard')
print(len(names) == 3) >> True
print(names) >> ['Zach', 'Jay', 'Richard']
names += ['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]`**

```
numbers = [0, 1, 2, 3, 4, 5, 6]
numbers[0:3] == numbers[:3] == [0, 1, 2]
numbers[5:] == numbers[5:7] == [5, 6]
numbers[:] == numbers == [0, 1, 2, 3, 4, 5, 6]
numbers[-1] == 6 # Negative index wraps around
numbers[-3:] == [4, 5, 6]
numbers[3:-2] == [3, 4] # 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**.

```
phonebook = {} # Empty dictionary
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()`

```
for i in range(10):
    print(i)
>> 1
   2…
   9
   10
```

Loops

To iterate over a list

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

```
>> Hi Zach!
   Hi Jay!
   Hi Richard!
```

To iterate over indices and values

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

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

```
>> 1 Zach
   2 Jay
   3 Richard
```

Loops

To iterate over a dictionary

```
phonebook = {'Zach': '12-37', 'Jay': '34-23'}
for name in phonebook:                          >> Jay
    print(name)                                    Zach

for number in phonebook.values():               >> 12-37
    print(number)                                  34-23

for name, number in phonebook.items():          >> Zach 12-37
    print(name, number)                            Jay 34-23
```

Note: Whether dictionary iteration order is guaranteed depends on the version of Python.

Classes

```
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.

```
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                                         # Install package
[package_name][==optional version number]


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

[1] https://docs.conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html#using-pip-in-an-environment

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. Common Data Structures
5. **Introduction to NumPy**
6. Practical Python Tips
7. 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)
```

```
>> [1 2 3]
   [[3 4 5]]
   [[6 7]
    [8 9]]
```

```
print(x.shape)
print(y.shape)
print(z.shape)
```

```
>> (3,)
>> (1,3)
>> (2,2)
```

A 1-D vector!

A (row) vector!

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.)

```
# 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: `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 ° B, can be computed:
`A * B`

Dot product and **matrix vector product** (between 1-D array vectors), can be computed:
`np.dot(u, v)`
`np.dot(x, W)`

Matrix product / multiplication of matrix A and B, AB, can be computed:
`np.matmul(A, B)` or `A @ B`
`np.dot()` can also be used for matrix multiplication, but if A and B are both 2-D arrays, `np.matmul()` is **preferred**.

Transpose with `x.T`

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

Indexing

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

y = np.random.random((3, 1))      # Random (3, 1) vector

z = np.random.random((1, 4))      # Random (1, 4) vector

x + y  # Adds y to each column of x

x * z  # 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)

1	2	3	4
5	6	7	8
9	10	11	12

x

\+

1	1	1	1
2	2	2	2
3	3	3	3

y

=

2	3	4	5
7	8	9	10
12	13	14	15

1	2	3	4
5	6	7	8
9	10	11	12

x

*

1	2	3	4
1	2	3	4
1	2	3	4

z

=

1	2	3	4
5	12	21	32
9	30	33	48

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)

```
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).

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

```
x **= 2
```

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

```
x[np.arange(100,1000), :] += 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:

```
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

```
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

Code	What it does
`array.shape`	Get shape of NumPy array
`array.dtype`	Check data type of array (for precision, for weird behavior)
`type(stuff)`	Get type of variable
`import pdb; pdb.set_trace()`	Set a breakpoint [1]
`print(f'My name is {name}')`	Easy way to construct a string to print

[1] 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.

`array.shape`

Topics

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