Python Review Session

CS224N - Winter 25

Stanford University

Two entwined snakes, based on Mayan representations.
However, named after Monty Python’s Flying Circus 😅

Charting a Course

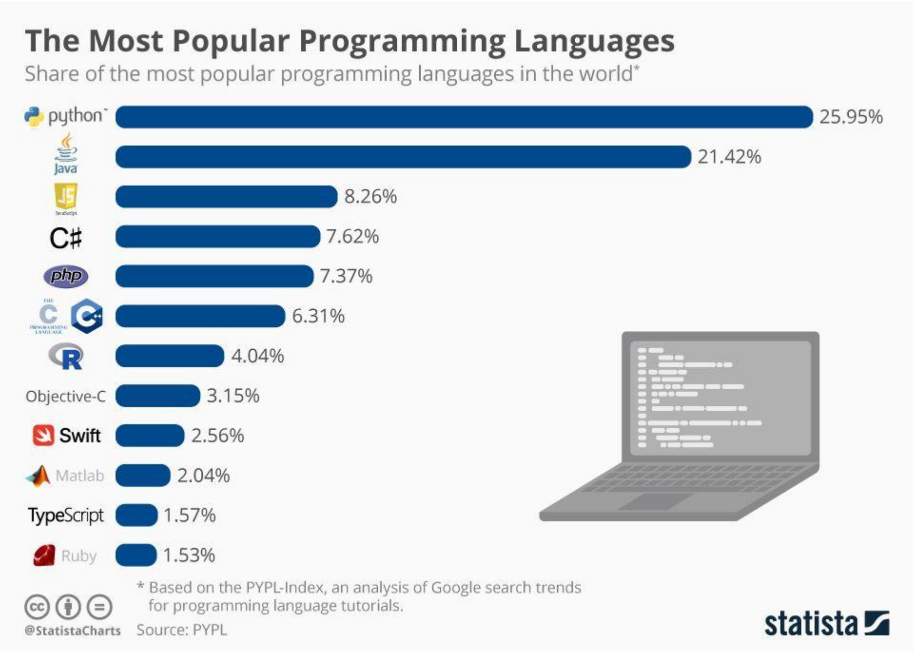
1. Why Python?
2. Setting Up
3. Python Basics
4. Data Structures
5. Numpy
6. Practical Tips

Charting a Course

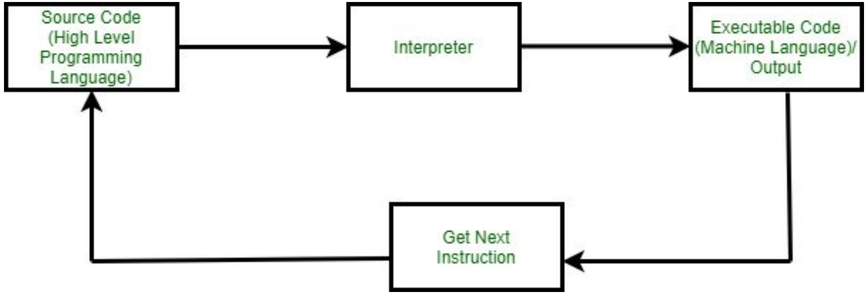
1. Why Python?
2. Setting Up
3. Python Basics
4. Data Structures
5. Numpy
6. Practical Tips

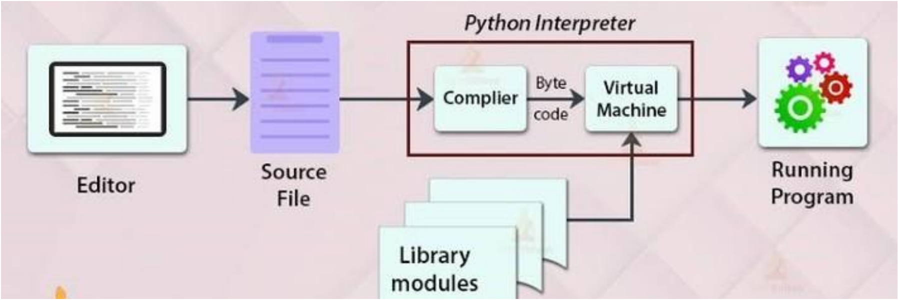
Why Python?

- Widely used, general purpose
- Easy to **learn**, **read**, and **write**
- Scientific computation functionality similar to Matlab and Octave
- Used by major **deep learning** frameworks (PyTorch, TensorFlow)
- Active **open-source**, many **libraries**!



The Python Interpreter





Ex. Interactive Mode (line-by-line)

Ex. Script Mode (.py file)

Python code → **interpreted** into **bytecode** (.pyc) → compiled by a VM implementation into machine instructions (most commonly using C.)

“Slower”, but can run highly **optimized C/C++ subroutines** to make operations **fast**

Language Basics

Strongly Typed

Interpreter always "**respects**" the **types** of each **variable**.

Interpreter keeps track of all variable types (strict handling)

Types will **not be coerced** silently like in JavaScript, Perl

```
1 + '1' → Error!

[1, 2] + set([3]) → Error!
```

*Cases like float and int addition are allowed by **explicit implementation** (no auto conversion)*

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

Language Basics

Dynamically Typed

*A variable is simply a **value** or **object reference** bound to a **name**.*

Data types of variables are determined at runtime (flexible!)

```
def find(required_element, sequence):
    for index, element in enumerate(sequence):
        if element == required_element:
            return index
    return -1
```

Variables can be assigned to values of a different type.

```
print(find(2, [1, 2, 3]))  # Outputs: 1
print(find("c", ("a", "b", "c", "d")))  # Outputs: 2
```

✅

```
num = 1  # int
num = "One"  # str
```

✅

https://medium.com/@pavel.loginov.dev/typing-in-python-strong-dynamic-implicit-c3512785b863

A Quick Check-In 🥳

🎯 In Python, what will the following code output?

```
x = 5
y = "3"
print(x + y)
```

A. 8

B. "53"

C. TypeError

D. "53.0"

A Quick Check-In 🥳

🎯 In Python, what will the following code output?

```
x = 5
y = "3"
print(x + y)
```

A. 8

B. "53"

C. TypeError

D. "53.0"

Charting a Course

1. Why Python?
2. Setting Up
3. Python Basics
4. Data Structures
5. Numpy
6. Practical Tips

Syntax Going Forward

`Code is in Courier New`.

Command line input is prefixed with **‘$’**.

Output is prefixed with **‘>>’**.

Python Installation

https://www.python.org/downloads/ 🥳

Active Python Releases

For more information visit the Python Developer's Guide.

Python version	Maintenance status	First released	End of support	Release schedule
3.14	pre-release	2025-10-01 (planned)	2030-10	PEP 745
3.13	bugfix	2024-10-07	2029-10	PEP 719
3.12	bugfix	2023-10-02	2028-10	PEP 693
3.11	security	2022-10-24	2027-10	PEP 664
3.10	security	2021-10-04	2026-10	PEP 619

Helpful Commands

Print out Version

$python --version

$python -v

$python -vv

Print out Location

$which python (mac, linux)

$where python (windows)

See Installed Libraries

$python -m pip list

pip is Python's package installer

-m runs a module (ex. pip) as a script

Run in Different Modes

$python script.py

$python -i script.py

-i remains in interactive mode after running .py

$python -c "print('hello there!')"

-c runs one-liner code snippet

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!

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: Virtual Envs

- 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

Solution 1: venv

- Created on top of existing installation, known as the virtual env's "base" Python
- Directory contains a specific Python interpreter and libraries, binaries which are needed to support a project
- Isolated from software in other virtual envs and interpreters and libraries installed in OS

```
$python -m venv /path/to/new/virtual/env
```

Creates a new directory → can activate (differs based on OS)

OS	Shell	Activation Command
Windows	Command Prompt	`path\to\venv\Scripts\activate`
Windows	PowerShell	`.\path\to\venv\Scripts\Activate`
macOS/Linux	Bash	`source path/to/venv/bin/activate`
macOS/Linux	Fish	`source path/to/venv/bin/activate.fish`
macOS/Linux	PowerShell	`path\to\venv\Scripts\Activate`

https://docs.python.org/3/library/venv.html

Solution 2: Anaconda (or Miniconda)

https://www.anaconda.com/download/

Very popular Python env/package manager

- Supports Windows, Linux, MacOS
- Can create and manage different isolated envs

Basic Workflow

Choose specific Python version

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
```

Export/create from env files!

Installing Packages

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

🚨 Best to first use conda to install as many packages as possible and use pip to install remaining packages after.

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

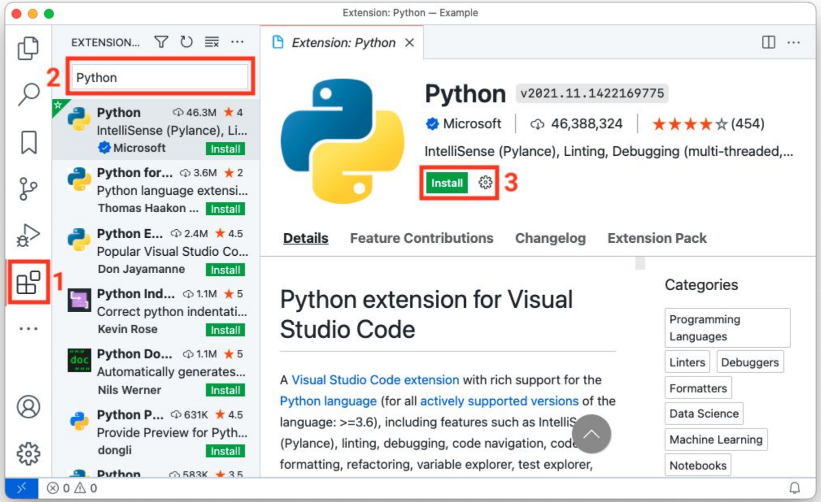
Install packages using pip in a conda environment (necessary when package not available through conda):

```
conda install -n myenv pip                          # Install pip in environment
conda activate myenv                                # Activate environment
pip install                                         # Install package individually OR
[package_name][==optional version number]
pip install -r <requirements.txt>                   # Install packages from file
```

IDEs / Text Editors

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

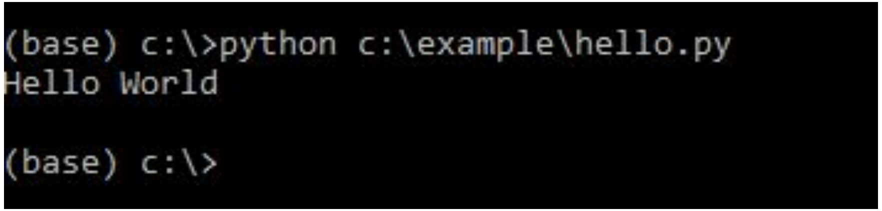
- PyCharm
- Visual Studio Code
- Sublime Text
- Atom
- Vim (for Linux or Mac)



IDEs often have useful extensions! (ex. VS Code)

In terminal, just activate virtual environment and run command:

$ python <filename.py>



Python Notebooks

https://colab.research.google.com/

Jupyter Notebook

- .ipynb → write and execute Python locally in web browser
- Interactive, re-execute code, result storage, can interleave text, equations, and images
- Can add conda environments
- Read-Eval-Print-Loop (REPL)

Google Colab

- Hosted Jupyter notebooks, run in cloud, requires no setup to use, provides free access to GPUs
- Comes with many Python libraries pre-installed
- Can integrate with Git (pull/run), Google Drive, local storage
- Tools > Settings > Misc > 😉😁

🎯 Matching time!

1. venv
2. Anaconda
3. Jupyter Ntbk
4. pip

A. Python package manager used to install and manage libraries.

B. Tool for creating isolated Python environments for dependency management.

C. Distribution that simplifies package and environment management, designed for data science.

D. An interactive platform for writing and running code alongside visualizations and notes.

🎯 Matching time!

1. venv

2. Anaconda

3. Jupyter Ntbk

4. pip

A. Python package manager used to install and manage libraries.

B. Tool for creating isolated Python environments for dependency management.

C. Distribution that simplifies package and environment management, designed for data science.

D. An interactive platform for writing and running code alongside visualizations and notes.

Language Basics

1. Why Python?
2. Setting Up
3. Python Basics
4. Data Structures
5. Numpy
6. Practical Tips

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             # 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():             # Functions can return None

    return None
```

Built-in Values

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

not


if [] != [None]:        # Comparison operators == and !=
                        check for equality/inequality, return
    print("Not equal")  true/false values
```

Spacing: Brackets → Indents

Code blocks are created using indents and newlines, instead of brackets like in C++

- Indents can be 2 or 4 spaces, but should be consistent throughout
- 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")
```

🎯 Debugging Derby

```
0length = 10
float width = 5.0

print "Beginning work..."

area = 0length * Width

if area > 20
    print("Area: " + area)

message = "Completed!'
```

Find the errors!

🎯 Debugging Derby

```
0length = 10                    # can’t start var name with number
float width = 5.0               # no explicit type declaration!

print "Beginning work..."       # parentheses around print

area = 0length * Width          # capitalization mismatch “Width”

if area > 20                    # missing colon after condition
    print("Area: " + area)      # need to cast area to string type

message = "Completed!'          # mismatch in quotation (" vs ')
```

🎯 Debugging Derby

```
length = 10
width = 5.0

print("Beginning work...")

area = length * width

if area > 20:
    print("Area: " + str(area))

message = "Completed!"
```

All fixed! 🥳

Language Basics

1. Why Python?
2. Setting Up
3. Python Basics
4. Data Structures
5. Numpy
6. Practical Tips

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)
>> 0
   1…
   8
   9
```

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])
```

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

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

Loops

To iterate over a dictionary

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

```
>> Jay
   Zach
```

```
for number in phonebook.values():
    print(number)
```

```
>> 12-37
   34-23
```

```
for name, number in phonebook.items():
    print(name, number)
```

```
>> Zach 12-37
   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():
        …
```

🎯 Inner Interpreter

```
v1 = ["Eeyore", "Goofy", "Nemo", "Wall-E"]
v2 = {"Eeyore": 12, "Nemo": 2, "Goofy": 42}

m1 = v1[1:-1]

for n in m1:
    print(f"{n} is {v2[n]} years old.")
```

Output?

🎯 Inner Interpreter

```
v1 = ["Eeyore", "Goofy", "Nemo", "Wall-E"]
v2 = {"Eeyore": 12, "Nemo": 2, "Goofy": 42}

m1 = v1[1:-1]

for n in m1:
    print(f"{n} is {v2[n]} years old.")
```

>> Goofy is 42 years old.
>> Nemo is 2 years old.

Language Basics

1. Why Python?
2. Setting Up
3. Python Basics
4. Data Structures
5. Numpy
6. Practical Tips

Prelude: 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...
```

Now, NumPy!

- 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()`

On average, a task in Numpy is **5-100X** faster than standard list!

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.

```
# 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]]
```

tl;dr "collapsing" this axis into the func's output.

np.ndarray Operations

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

Element-wise product (Hadamard product) of matrix A and B, A ∘ B, is: `A * B`

Dot product is: `np.dot(u, v)`

Matrix vector product (1-D array vectors) is: `np.dot(x, W)`

Matrix product / multiplication of matrix A and B is: `np.matmul(A, B)` or `A @ B`

`np.dot()` can also be used, but if A and B are both 2-D arrays, `np.matmul()` is **preferred**.

Transpose is: `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	4	9	16
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
```

If the arrays have different ranks (number of dimensions), NumPy implicitly prepends 1s to the shape of the lower-rank array.

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 → (3, 3)
a + c → Broadcast Error
b + c → (3, 3)
```

If the arrays have different ranks (number of dimensions), NumPy implicitly prepends 1s to the shape of the lower-rank array.

Broadcasting Algorithm

```
p = max(m, n)
if m < p:
    left-pad A's shape with 1s until it also has p dimensions
else if n < p:
    left-pad B's shape with 1s until it also has p dimensions

result_dims = new list with p elements

for i in p-1 ... 0:
    A_dim_i = A.shape[i]; B_dim_i = B.shape[i]

    if A_dim_i != 1 and B_dim_i != 1 and A_dim_i != B_dim_i:
        raise ValueError("could not broadcast")
    else:
        # Pick the Array which is having maximum Dimension
         result_dims[i] = max(A_dim_i, B_dim_i)
```

Efficient NumPy Code

Avoid explicit for-loops over indices/axes at all costs. *(~10-100x slowdown).*

```
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
```

🎯 Numpy Knowhow

How do you create a NumPy array with numbers from 1 to 10?

A. `np.arange(1, 10)`
B. `np.arange(1, 11)`
C. `np.array(range(1, 10))`
D. `np.linspace(1, 10)`

What does `np.random.rand(3, 4)` generate?

A. A 3x4 array of random integers
B. A 3x4 array of random values between 0 and 1
C. A 3x4 array of random values between -1 and 1
D. A 3x4 identity matrix

🎯 Numpy Knowhow

How do you create a NumPy array with numbers from 1 to 10?

A. `np.arange(1, 10)`
B. `np.arange(1, 11)`
C. `np.array(range(1, 10))`
D. `np.linspace(1, 10)`

What does `np.random.rand(3, 4)` generate?

A. A 3x4 array of random integers
B. A 3x4 array of random values between 0 and 1
C. A 3x4 array of random values between -1 and 1
D. A 3x4 identity matrix

Language Basics

1. Why Python?
2. Setting Up
3. Python Basics
4. Data Structures
5. Numpy
6. Practical Tips

List Comprehensions

- Similar to map() from functional programming languages (readability + succinct)
- Format: `[func(x) for x in some_list]`

```
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 = (‘TF’, ‘PyTorch’, ‘JAX’)
```

Join list of strings with delimiter

```
“, ”.join([‘1’, ‘2’,
‘3’]) == ‘1, 2, 3’
```

Returning multiple items from a function

```
def some_func():
    return 10, 1
ten, one =
some_func()
```

String literals with both single and double quotes

```
message = ‘I like
“single” quotes.’
reply = “I prefer
‘double’ quotes.”
```

Single-line if else

```
result = "even"
if number % 2
== 0 else "odd"
```

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]]
```

<u>Helpful Commands</u>

Ctrl-d: Exit IPython Session

Ctrl-c: Interrupt current command

Ctrl-l: Clear terminal screen

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

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 s to print out the shape of relevant arrays to see if they match what you expect: **`array.shape`**

[Very Active, Open-Source Community] When debugging, check Ed and forums such as StackOverflow or GitHub Issues → likely that others have encountered the same error!

Other Great References

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

Official Anaconda user guide:
https://docs.conda.io/projects/conda/en/latest/user-guide/index.html

Official NumPy documentation: https://numpy.org/doc/stable/

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

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

Several Python and library-specific (ex. NumPy) “Cheat Sheet” guides online as well!

Yayy, we did it! 🥳

Thanks for listening!