DSCI-560

Lecture 4: Data Science Dev Platforms/Languages

Data Science Professional Practicum

Young Cho
Department of Electrical Engineering
University of Southern California

Reproducibility

- **Self-contained** configuration/program/data
  - Dependencies may have incompatibilities
    - May work in Python 3.6 but does not in Python 3.7.
  - Some libraries may require different versions
    - In one project, library A requires numpy 1.0, but numpy 2.0 for project B
- **Isolated** from machine dependencies
  - Certain SW functions might depend on HW configuration
- **Portable** from one machine to another
  - May need to move from one machine to another
    - May have run out of storage or memory
    - May need faster performance
  - May need to demonstrate on another machine

Development

- Self-contained Environments
  - Helps to manage working environments better
  - For each new project, create a new environment
- Validation and Repeatability
  - Immediate Validation and Working Starting Point
  - Reduces erroneous instruction interpretation
  - Reduces the possibility of using the wrong data
  - Presents exact details of the experiments

Virtual Environments

- Virtual Machines
  - Computer Emulation Software
  - HW/SW Operations are Emulated with SW
  - VMs interact HW via Light-weight SW Layer
  - Examples: VMWare, VirtualBox, QEMU, etc.
- Containers
  - Not SW Emulation
  - Application Layer Abstraction for SW and their dependencies
  - Containers are natively run on the HW with Light-weight Virtualization Layer
  - Examples: Docker, CoreOS rkt, Mesos, etc.

Virtual Machines

- Hypervisors
  - Light-weight software layer between VM and HW
  - Separates VMs from each other
  - Allocates physical processors, memory, and storage
  - Manages hosting operating systems
- Software on VM
  - Operating System with system binaries and libraries
  - All applications managed by the OS
- Most Modern Processors have VM Support
  - VM specific HW operations for acceleration

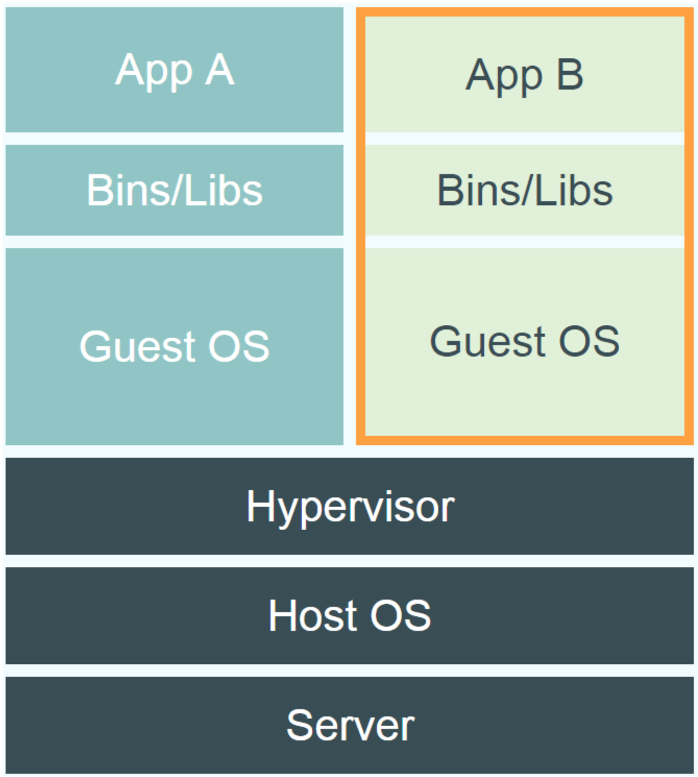
Containers

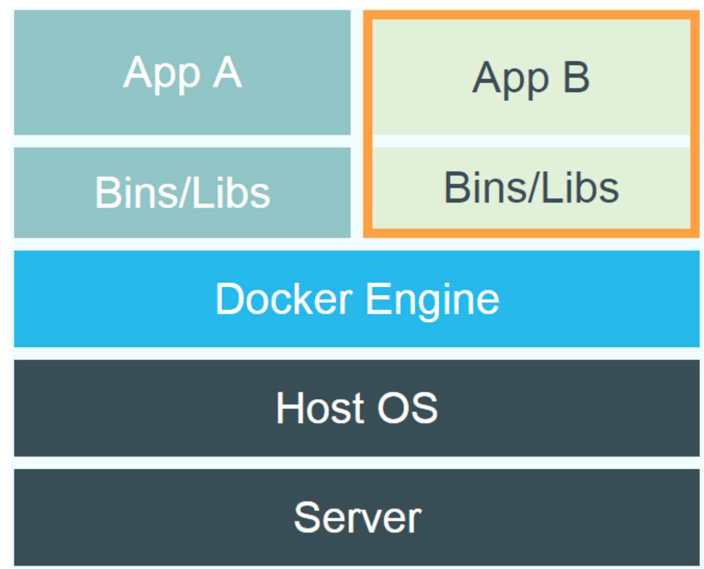
- Container Manager (i.e. Docker)
  - Virtualizes Operating System
  - Enables Application Layer Abstractions
- Container Operations
  - Host OS kernel and libraries are shared
  - Template of an environment is created within
  - Runs a snapshot of the system
  - Consistent behavior of an app.

Images for Containers

- Read-only template for Containers
- Sets up all software and libraries with specific environment configurations
- Enables the creation of various containers
- Can be used for reproducible data science results

VM and Containers



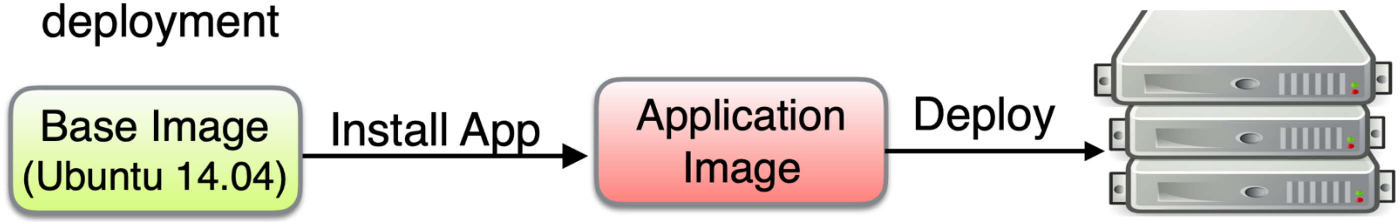


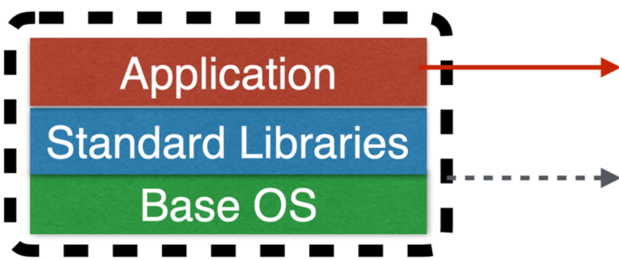
What's the Diff: VMs vs Containers

VMs	Containers
Heavyweight	Lightweight
Limited performance	Native performance
Each VM runs in its own OS	All containers share the SAME host OS
Hardware-level virtualization	OS virtualization
Startup time in minutes	Startup time in milliseconds
Allocates required memory	Requires less memory space
Fully isolated and hence more secure	Process-level isolation, possibly less secure

Performance comparison

- Getting applications from development to production involves creating disk images
- Fast image creation enables rapid testing and continuous deployment





Container images have only application layer

VMs contain entire OS, and have larger images

Time (s)	VM (Vagrant)	Docker
MySQL	236	129
NodeJS	304	49

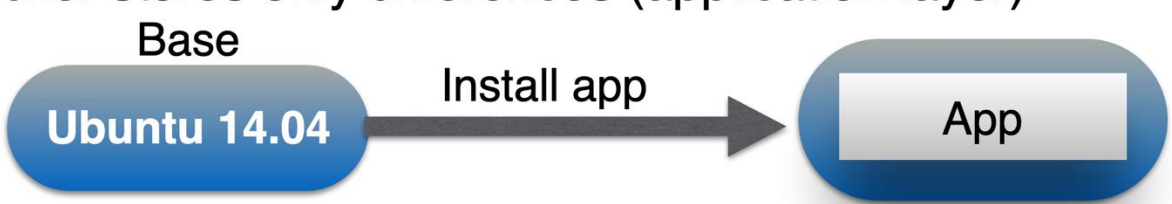
- Docker: 2-6x faster

Size comparison

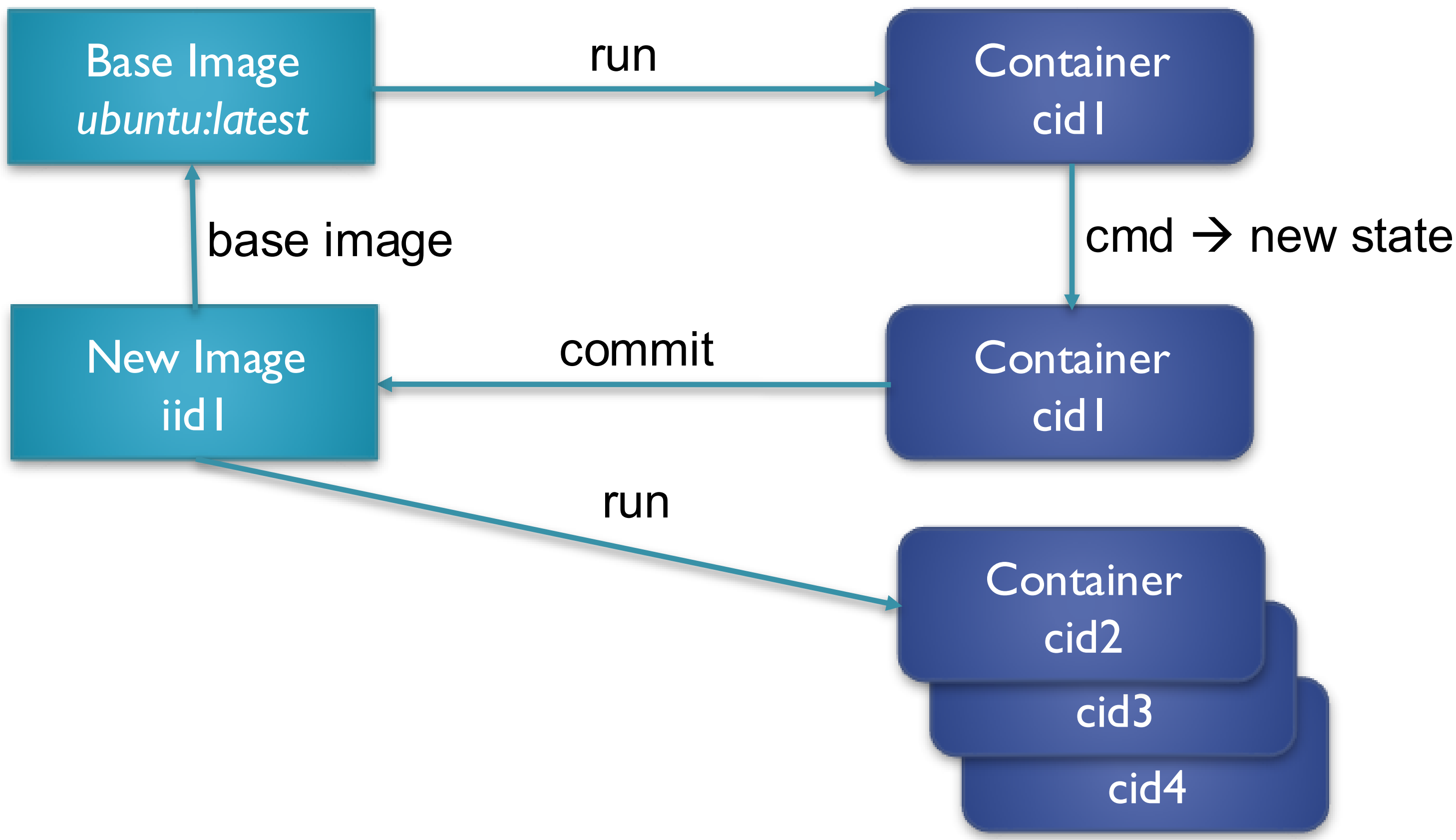
Image size	VM	LXC	Docker
MySQL	1.68 GB	0.4 GB	**112 KB**
NodeJS	2.05 GB	0.6 GB	**72 KB**

Docker: 2-6x smaller

- VMs contain entire OS, and have larger images
- Docker stores only differences (application layer)



Container Images



Dockerfile

- Create images automatically using a build script: «Dockerfile»
- Can be versioned in a version control system like Git or SVN, along with all dependencies
- Docker Hub can automatically build images based on dockerfiles on Github

Reproducibility

- Dependencies
  - Less than 50% of software could be built or installed
  - Difficult to reproduce computational environment
    - Possible Solution: Docker Container
  - Software dependencies change, affecting results
    - Possible Solution: Software versioning
- Imprecise Documentation
  - Difficult to figure out how to install
    - Possible Solution Dockerfile records for dependencies
- Barriers to Adoption and Reuse
  - Difficulty to coordinate build tools/package managers
  - Persistent problem with this point

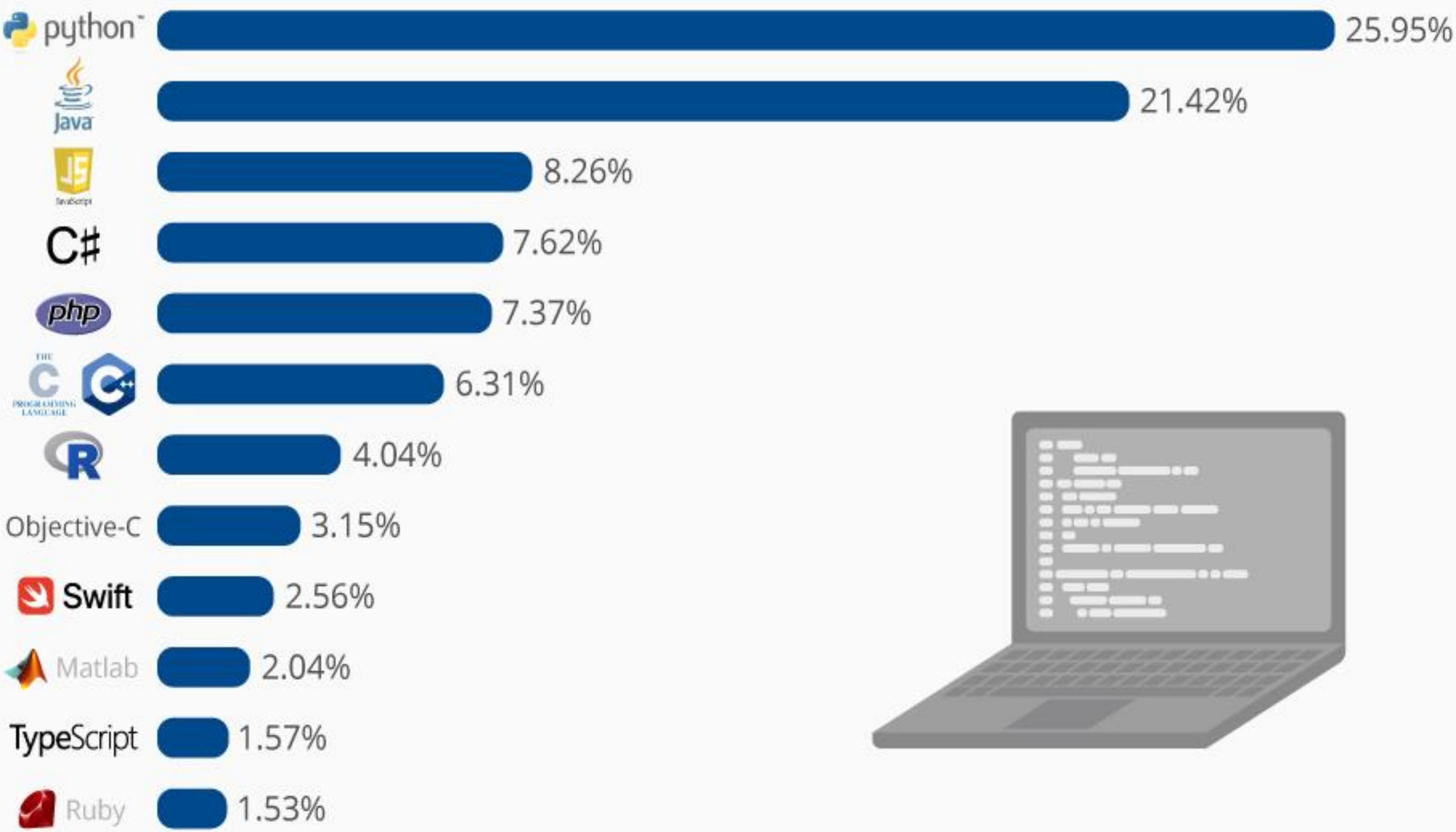
Development Tool: Python

- Started by Guido Van Rossum as a hobby
- Now, the Most Popular Language
  - Also means everyone already knows
- Open Source
- Versatile
- Lots of prebuilt packages
  - This is the KEY
- BUT, only used like a wrapper
  - Need to go beyond!!

Guido Van Rossum

The Most Popular Programming Languages

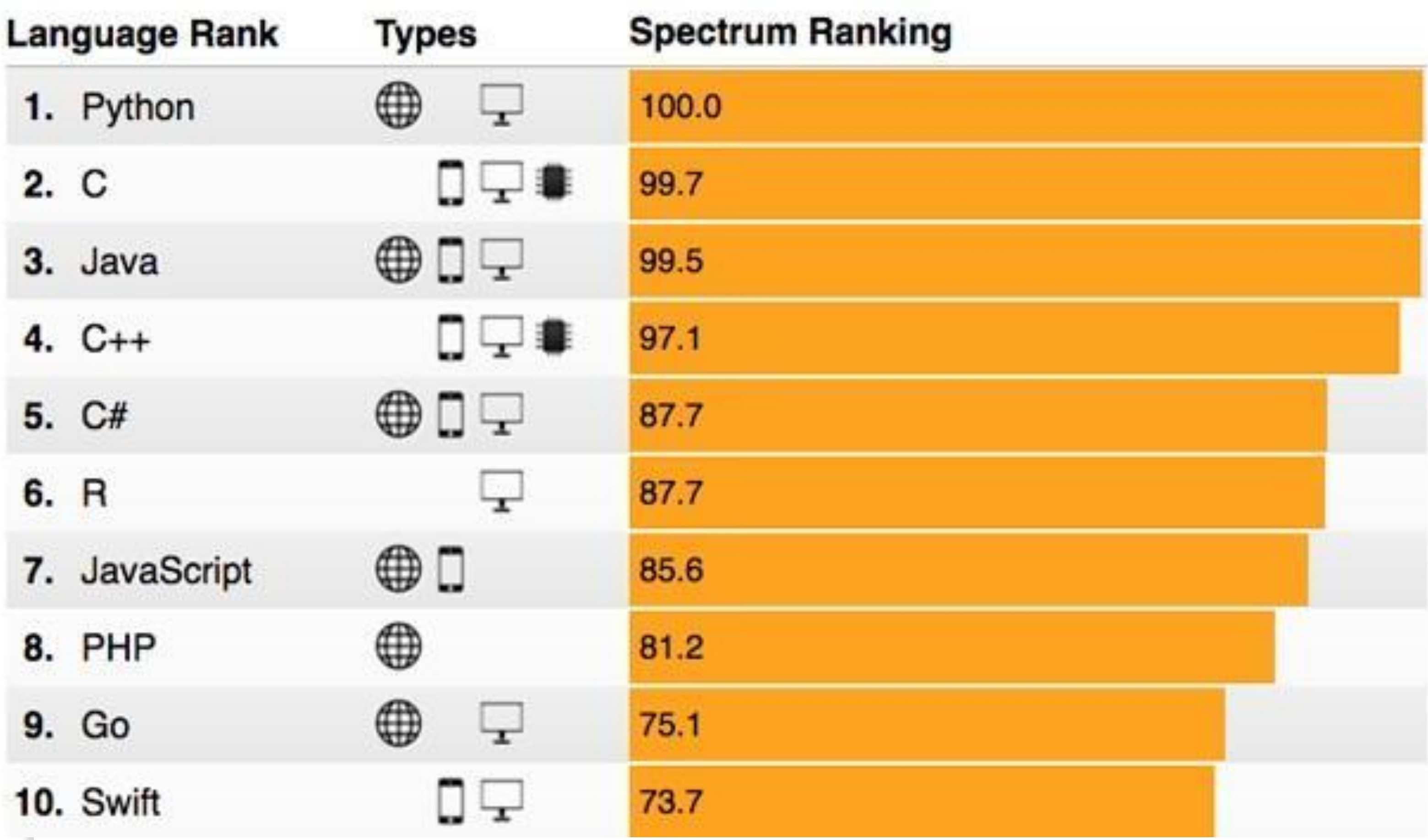
Share of the most popular programming languages in the world*



* Based on the PYPL-Index, an analysis of Google search trends for programming language tutorials.

Source: PYPL

2025 IEEE Ranking based on Types of Developments



Python Today

- Development Today
  - Large and active scientific computing and data analysis community
- One of the most important languages for
  - Data science
  - Machine learning
  - General software development
- Many Prebuilt Packages

Two Modes

1. **IPython**

Python can be run interactively

Used extensively in research

2. **Python scripts**

What if we want to run more than a few lines of code?

Then we must write text files in .py

Installing Python

Windows:

- Download Python from http://www.python.org
- Install Python.
- Run **Idle** from the Start Menu.

Mac OS X:

- Python is already installed.
- Open a terminal and run `python` or run Idle from Finder.

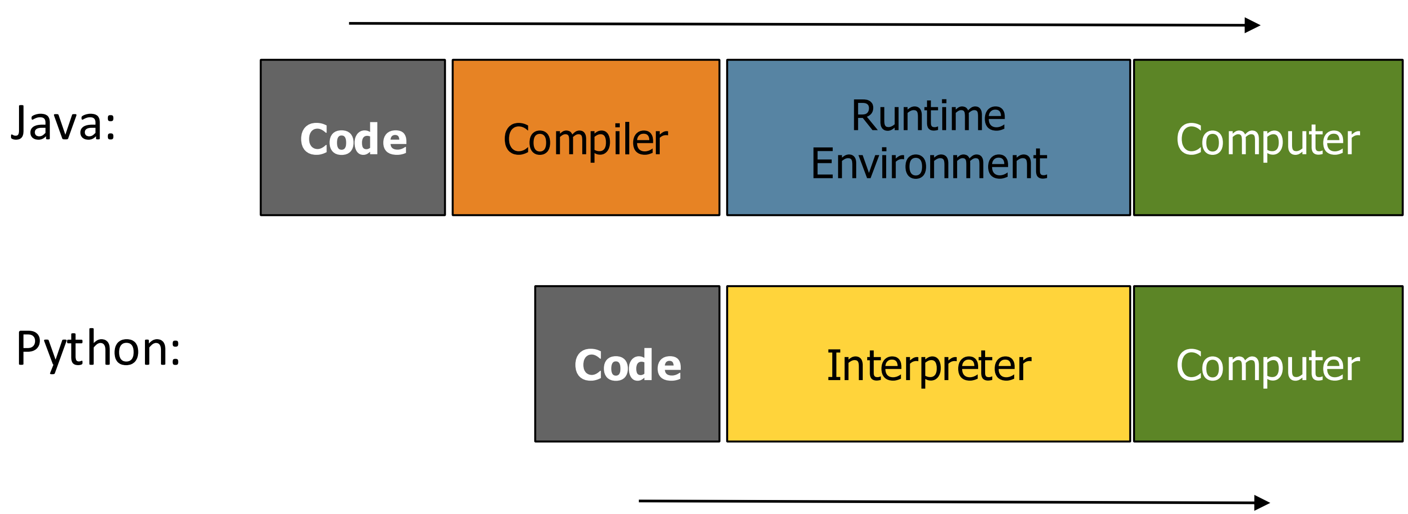
Linux:

- Chances are you already have Python installed. To check, run `python` from the terminal.
- If not, install from your distribution's package system.

Note: For step by step installation instructions, see the course web site.

Interpreted Languages

- **Interpreted**
  - Not compiled like Java
  - Code is written and then directly executed by an **interpreter**
  - Type commands into interpreter and see immediate results



Why Learn C?

- Python is high-level, easy, but slower.
- C is lower-level, compiled, and much faster.
- C gives control over memory and hardware.
- Many Python libraries (NumPy, TensorFlow) are written in C.

Key Differences

- Compiled vs Interpreted
  - Python: interpreted, dynamic
  - C: compiled, static typing
- Manual vs Automatic memory management
- Syntax differences: indentation vs braces {}

First Program

- Python:

```python
print("Hello, World!")
```

- C:

```c
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

Variables & Types

Python:

- x = 10
- y = 3.14

C:

- int x = 10; // 32-bit fixed point number
- float y = 3.14; // 32-bit floating point number

Control Structures

Python:

- if x > 0:
- print("Positive")
- else:
- print("Non-positive")

C:

- if (x > 0) {
- printf("Positive\n");
- } else {
- printf("Non-positive\n");
- }

Loops

Python:

- for i in range(5):
- print(i)

C:

- for (int i = 0; i < 5; i++) {
- printf("%d\n", i);
- }

Functions

Python:

- def add(a, b):
- return a + b

C:

- int add(int a, int b) {
- return a + b;
- }

Arrays vs Lists

Python:

- arr = [1, 2, 3, 4]
- print(arr[2])

C:

- int arr[4] = {1, 2, 3, 4};
- printf("%d\n", arr[2]);

Strings

Python:

- s = "Hello"
- print(len(s))

C:

- char s[] = "Hello";
- printf("%lu\n", strlen(s));

Memory Management

- Python: Automatic garbage collection

- C:

```
int *p = malloc(sizeof(int) * 5);
free(p);
```

Pointers

- C Example:

```
int x = 10;
int *p = &x;
printf("%d\n", *p); // dereference
```

Structs vs Classes

Python:

- class Point:
- def __init__(self, x, y):
- self.x = x
- self.y = y

C:

- struct Point {
- int x;
- int y;
- };

Python ↔ C Interoperability

- Python can call C libraries with ctypes or cffi.
- Example
  - NumPy, TensorFlow use C under the hood.
- Use C for performance-critical code.

Python to C

- Python is high-level, dynamic, memory-safe.
  - C is low-level, explicit, and fast.
  - Knowing both = power + flexibility.
- Next Steps:
  - Write simple C programs.
  - Experiment with memory management.
  - Try calling C from Python.

Why CUDA?

- CPUs
  - A Few Powerful cores
  - Good for sequential tasks.
- GPUs
  - Thousands of lightweight cores
  - Ideal for parallel tasks.
- CUDA
  - NVIDIA's platform to program GPUs using C/C++.
- Used in AI, simulations, graphics, data science.

CUDA vs C

- C: Programs run on CPU.
- CUDA C: C + extensions for GPU programming.
- Functions can run on CPU (host) or GPU (device).
- Explicit memory management between CPU and GPU.

CUDA Programming Model

- Host (CPU) and Device (GPU).
- Functions:

```
__host__ - runs on CPU
__device__ - runs on GPU
__global__ - GPU kernel callable from CPU
```

- Threads grouped into blocks and grids.

Hello CUDA

C Program:

- printf("Hello from CPU\n");

CUDA Program:

- #include <stdio.h>
- __global__ void hello() {
- printf("Hello from GPU!\n");
- }
- int main() {
- hello<<<1, 1>>>();
- cudaDeviceSynchronize();
- return 0;
- }

Thread Hierarchy

- Thread → smallest execution unit.
- Block → group of threads.
- Grid → group of blocks.

- CUDA IDs:

```
int tid = threadIdx.x;
int bid = blockIdx.x;
int gid = bid * blockDim.x + tid;
```

Example: Vector Addition

- CPU C Code:

```
for (int i = 0; i < N; i++) {
    C[i] = A[i] + B[i];
}
```

- CUDA Kernel:

```
__global__ void vecAdd(int *A, int *B, int *C, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) C[i] = A[i] + B[i];
}
```

- Launch:

```
vecAdd<<<(N+255)/256, 256>>>(A, B, C, N);
```

Memory in CUDA

- Host Memory (CPU RAM) vs Device Memory (GPU VRAM).
- Explicit copies:

```
cudaMalloc((void**)&d_A, size);
cudaMemcpy(d_A, h_A,
size,cudaMemcpyHostToDevice);
```

- Types:
  - Global (slow, big)
  - Shared (fast, per block)
  - Registers (fastest, per thread)

Shared Memory Example

```
__global__ void addShared(int *A, int *B, int *C)
{
    __shared__ int temp[256];
    int tid = threadIdx.x;
    temp[tid] = A[tid] + B[tid];
    __syncthreads();
    C[tid] = temp[tid];
}
```

- Shared memory allows fast collaboration within a block.

Performance Considerations

- \- Use many threads to hide latency.
- \- Coalesced memory access = faster.
- \- Minimize CPU↔GPU transfers.
- \- Use shared memory wisely.
- \- Profile with nvprof or nsight.

CUDA vs Multithreading in C

- C with pthreads/OpenMP
  - Parallel on CPU only.
- CUDA: thousands of GPU threads.
- GPU excels at data-parallel problems.

CUDA Applications

- Deep Learning (PyTorch, TensorFlow)
- Image processing
- Physics simulations
- Financial modeling

C to CUDA

- CUDA extends C for GPU programming.
- Kernels, threads, blocks, memory are key concepts.
- Workflow: Allocate → Copy to GPU → Launch kernel → Copy back.
- Learn memory hierarchy, profiling, optimization.