COMP101 — Week 12

Introduction to Computer Systems

Hardware, operating systems, and what Python is really doing

UM6P — SASE

January 16, 2026

Roadmap

Today's questions

- When you run `python script.py`, what actually happens?

Today's questions

- When you run `python script.py`, what actually happens?
- Why do encoding bugs and file bugs happen in "correct" code?

Today's questions

- When you run `python script.py`, what actually happens?
- Why do encoding bugs and file bugs happen in "correct" code?
- Why can a tiny Python program be slow or memory-hungry?

Today's questions

- When you run `python script.py`, what actually happens?
- Why do encoding bugs and file bugs happen in "correct" code?
- Why can a tiny Python program be slow or memory-hungry?
- How do you debug by layer instead of guessing?

Today's questions

- When you run `python script.py`, what actually happens?
- Why do encoding bugs and file bugs happen in "correct" code?
- Why can a tiny Python program be slow or memory-hungry?
- How do you debug by layer instead of guessing?
- What changes when code is compiled / JIT / on GPU?

Roadmap: the layers we will use

Your Python code
Python runtime (interpreter)
Operating System (OS)
Hardware (CPU, memory, disk, devices)

- Most bugs are “wrong layer” bugs

Roadmap: the layers we will use

Your Python code
Python runtime (interpreter)
Operating System (OS)
Hardware (CPU, memory, disk, devices)

- Most bugs are “wrong layer” bugs
- Goal: learn a small mental model that predicts failures.

Checkpoint: if you had to bet, where is the bug?

- Your script prints weird characters from a file. (Which layer?)

Keep these in mind; we will solve them later.

Checkpoint: if you had to bet, where is the bug?

- Your script prints weird characters from a file. (Which layer?)
- Your script is slow when reading data. (Which layer?)

Keep these in mind; we will solve them later.

Checkpoint: if you had to bet, where is the bug?

- Your script prints weird characters from a file. (Which layer?)
- Your script is slow when reading data. (Which layer?)
- Your script works on your laptop but not your friend's. (Which layer?)

Keep these in mind; we will solve them later.

The Stack: Code, OS, Hardware

The layer cake

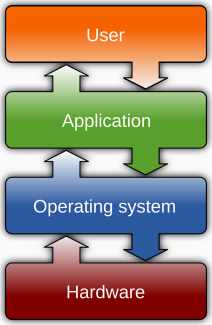

Figure 1: Interacting with the hardware

- **Your code requests services; the OS decides how to provide them.**

The layer cake

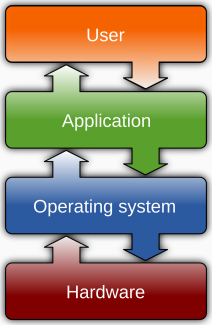

Figure 1: Interacting with the hardware

- **Your code requests services; the OS decides how to provide them.**
- **If you confuse layers, debugging becomes random.**

From terminal to running program

When you type `python script.py`, the OS roughly does:

1. **Locate the `python` executable** (PATH, file system).

From terminal to running program

When you type `python script.py`, the OS roughly does:

1. **Locate the python executable** (PATH, file system).
2. **Create a new process** (a running instance of a program).

From terminal to running program

When you type `python script.py`, the OS roughly does:

1. **Locate the python executable** (PATH, file system).
2. **Create a new process** (a running instance of a program).
3. **Load code and libraries** into memory.

From terminal to running program

When you type `python script.py`, the OS roughly does:

1. **Locate the python executable** (PATH, file system).
2. **Create a new process** (a running instance of a program).
3. **Load code and libraries** into memory.
4. **Give CPU time slices** (scheduling).

From terminal to running program

When you type `python script.py`, the OS roughly does:

1. **Locate the python executable** (PATH, file system).
2. **Create a new process** (a running instance of a program).
3. **Load code and libraries** into memory.
4. **Give CPU time slices** (scheduling).
5. **Allow file/keyboard/screen access** via OS rules (permissions).

Definition

Program = a file containing instructions.

Process = a running instance of a program with its own memory and resources.

- **You can run the same program multiple times ⇒ multiple processes.**

Definition

Program = a file containing instructions.

Process = a running instance of a program with its own memory and resources.

- **You can run the same program multiple times ⇒ multiple processes.**
- **A process has an ID (PID) that the OS uses to manage it.**

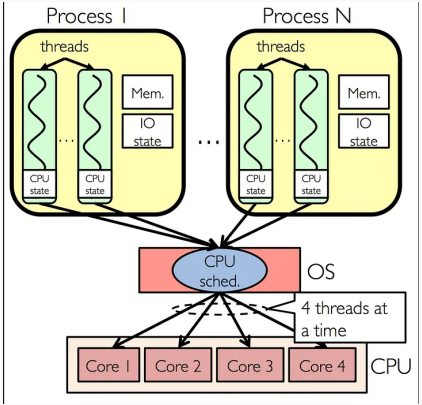

Figure 2: Processes and CPU

Fact: your Python code does not touch hardware directly

Theorem / Fact

In normal user programs, Python does not "poke the hardware".
It asks the OS for services (open a file, print text, get time, allocate memory), and the OS talks to hardware.

- **So many errors are OS-level: paths, permissions, encodings, missing files.**

Fact: your Python code does not touch hardware directly

Theorem / Fact

In normal user programs, Python does not "poke the hardware".
It asks the OS for services (open a file, print text, get time, allocate memory), and the OS talks to hardware.

- So many errors are OS-level: paths, permissions, encodings, missing files.
- Your job is to identify the layer before changing code.

Python demo: process, PID, and a file read

```python
import os

print("PID:", os.getpid())
print("CWD:", os.getcwd())

with open("example.txt", "w", encoding="utf-8") as f:
    f.write("hello\n")

with open("example.txt", "r", encoding="utf-8") as f:
    print("Read:", f.read().strip())
```

- PID identifies the running process.

Python demo: process, PID, and a file read

```python
import os

print("PID:", os.getpid())
print("CWD:", os.getcwd())

with open("example.txt", "w", encoding="utf-8") as f:
    f.write("hello\n")

with open("example.txt", "r", encoding="utf-8") as f:
    print("Read:", f.read().strip())
```

- PID identifies the running process.
- CWD (current working directory) explains many FileNotFoundError cases.

Checkpoint: what can the OS block?

- **Access to files** (permissions, missing paths).

Checkpoint: what can the OS block?

- **Access to files** (permissions, missing paths).
- **Access to network** (firewall, DNS, no internet).

Checkpoint: what can the OS block?

- **Access to files** (permissions, missing paths).
- **Access to network** (firewall, DNS, no internet).
- **CPU time** (your process may be slowed by other processes).

Checkpoint: what can the OS block?

- **Access to files** (permissions, missing paths).
- **Access to network** (firewall, DNS, no internet).
- **CPU time** (your process may be slowed by other processes).
- **Memory** (OS can kill your process if memory is exhausted).

Rings of Privilege

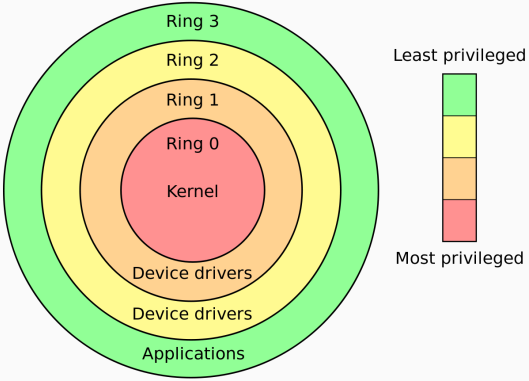

Figure 3: Kernel space VS User space

Representation: Bits, Bytes, Meaning

Bits and bytes: the only thing hardware understands

Definition

Bit = 0 or 1.

Byte = 8 bits (a number from 0 to 255).

Encoding = a rule to interpret bytes as text characters.

- **Files are bytes. Text is an interpretation.**
- **If you interpret bytes with the wrong encoding, text becomes garbage.**

ASCII vs UTF-8

- **ASCII** maps bytes 0–127 to English characters and symbols.
- **UTF-8** is a variable-length encoding for (almost) all written characters.
- **UTF-8 uses 1–4 bytes per character, but keeps ASCII unchanged.**

Why you care

`UnicodeDecodeError` almost always means: **wrong bytes** or **wrong decoding rule**.

Python demo: str vs bytes

```
s = "caf\u00e9"          # Python str (text), ASCII source code
b = s.encode("utf-8")    # bytes on disk / network

print("text:", s)
print("bytes:", b)
print("back:", b.decode("utf-8"))
```

- **str is text (meaning). bytes is raw data.**
- **The OS reads/writes bytes. Python converts bytes <-> str.**

Python demo: ord, chr, and bytes

```
print(ord("A"))           # 65
print(chr(65))            # 'A'

b = b"ABC"
print(list(b))            # [65, 66, 67]
print(b.decode("ascii")) # 'ABC'
```

- A byte is just a number. Encoding gives it meaning as text.

Common trap: default encoding assumptions

- Different machines may have different default encodings.

Common trap: default encoding assumptions

- Different machines may have different default encodings.
- A file written in one encoding can be read with another by mistake.

Common trap: default encoding assumptions

- Different machines may have different default encodings.
- A file written in one encoding can be read with another by mistake.
- Symptoms: weird characters, crashes, or data corruption.

Trigger and fix UnicodeDecodeError

```
# Write UTF-8 bytes
with open("names.txt", "w", encoding="utf-8") as f:
    f.write("na\u00efve caf\u00e9\n")

# Wrong: reading with an incompatible encoding can fail
try:
    with open("names.txt", "r", encoding="ascii") as f:
        print(f.read())
except UnicodeDecodeError as e:
    print("Decode error:", e)

# Right: read with the correct encoding
with open("names.txt", "r", encoding="utf-8") as f:
    print("OK:", f.read().strip())
```

Integers vs floats: the practical truth

- **Integers in Python behave like mathematical integers (arbitrary precision).**

One-liner to remember

`0.1 + 0.2 != 0.3` is not a Python bug; it's representation.

Integers vs floats: the practical truth

- Integers in Python behave like mathematical integers (arbitrary precision).
- Floats are finite precision (IEEE 754 double on most machines).

One-liner to remember

`0.1 + 0.2 != 0.3` is not a Python bug; it's representation.

Integers vs floats: the practical truth

- Integers in Python behave like mathematical integers (arbitrary precision).
- Floats are finite precision (IEEE 754 double on most machines).
- Some decimals cannot be represented exactly in binary floats.

One-liner to remember

`0.1 + 0.2 != 0.3` is not a Python bug; it's representation.

CPU + Memory: Why speed and memory aren't "just Python"

Minimal model: CPU, RAM, disk, I/O

- **CPU**: executes instructions (compute).
- **RAM**: fast working memory (temporary).
- **Disk/SSD**: slower but persistent storage (files).
- **I/O**: moving data in/out (disk, network, screen).

Prediction

If your program reads a huge file, **I/O** often dominates time, not Python syntax.

Fetch–decode–execute (story version)

- The CPU repeats:

Fetch–decode–execute (story version)

- The CPU repeats:
  1. **Fetch** next instruction from memory.

Fetch–decode–execute (story version)

- The CPU repeats:
  1. **Fetch** next instruction from memory.
  2. **Decode** what it means.

Fetch–decode–execute (story version)

- The CPU repeats:
  1. **Fetch** next instruction from memory.
  2. **Decode** what it means.
  3. **Execute** (compute / load / store / branch).

Fetch–decode–execute (story version)

- The CPU repeats:
  1. **Fetch** next instruction from memory.
  2. **Decode** what it means.
  3. **Execute** (compute / load / store / branch).
- **Your Python program becomes many CPU instructions via the interpreter.**

Fetch–decode–execute (story version)

- The CPU repeats:
  1. **Fetch** next instruction from memory.
  2. **Decode** what it means.
  3. **Execute** (compute / load / store / branch).
- **Your Python program becomes many CPU instructions via the interpreter.**
- **CPUs are fast; memory and I/O are usually the bottleneck.**

Cores vs threads

- **Core** = hardware that can execute independently.

Cores vs threads

- **Core** = hardware that can execute independently.
- **Thread** = an execution stream inside a process (scheduled by OS).

Cores vs threads

- **Core** = hardware that can execute independently.
- **Thread** = an execution stream inside a process (scheduled by OS).
- More threads is not automatically faster.

RAM vs disk: why I/O dominates

- RAM is fast; disk/SSD is slower; network is slower and variable.
- Reading large files costs time regardless of how elegant your loop is.
- If RAM is full, swapping to disk can make everything painfully slow.

Definition

Stack = memory for function call frames (locals, return info).
Heap = memory for objects created during execution (lists, dicts, strings, objects).

- **In Python, most data structures live on the heap.**

Definition

Stack = memory for function call frames (locals, return info).
Heap = memory for objects created during execution (lists, dicts, strings, objects).

- In Python, most data structures live on the heap.
- Lots of objects can increase memory usage and slow things down.

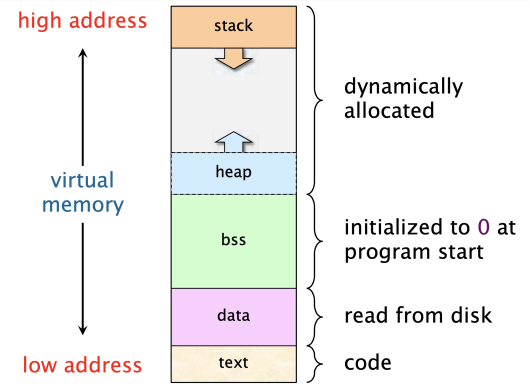

Figure 4: Program Layout in Memory

Stack vs heap

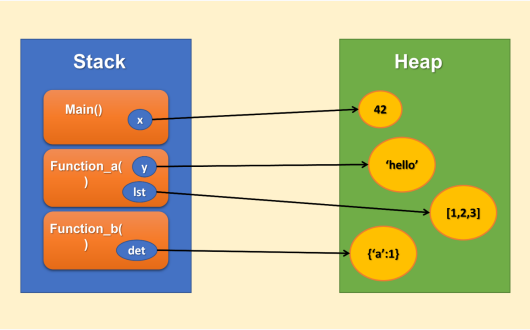

Figure 5: Stack vs Heap

Python demo: object overhead with sys.getsizeof

```python
import sys

x = 123
s = "hello"
lst = [1, 2, 3, 4, 5]

print("int:", sys.getsizeof(x))
print("str:", sys.getsizeof(s))
print("list object:", sys.getsizeof(lst))
print("list elements:", sum(sys.getsizeof(e) for e in lst))
```

- A list is a container + references + objects. Not “just numbers”.

Fact: Big-O does not predict everything

Theorem / Fact

Big-O helps with growth, but real runtime also depends on constants, memory access, and I/O.
A "faster" algorithm can lose if it forces more I/O or creates too many objects.

- **In COMP101: use Big-O as a compass, not a stopwatch.**

The OS: The invisible boss

What the OS provides to your Python program

- **Processes**: start/stop programs, isolate memory.

What the OS provides to your Python program

- **Processes**: start/stop programs, isolate memory.
- **Files and directories**: persistent storage.

What the OS provides to your Python program

- **Processes**: start/stop programs, isolate memory.
- **Files and directories**: persistent storage.
- **Permissions**: who can read/write/execute.

What the OS provides to your Python program

- **Processes**: start/stop programs, isolate memory.
- **Files and directories**: persistent storage.
- **Permissions**: who can read/write/execute.
- **Virtual memory**: the illusion of "large" memory; can swap to disk.

What the OS provides to your Python program

- **Processes**: start/stop programs, isolate memory.
- **Files and directories**: persistent storage.
- **Permissions**: who can read/write/execute.
- **Virtual memory**: the illusion of "large" memory; can swap to disk.
- **Networking**: sockets, DNS, routing.

Scheduling (why your program shares the CPU)

- Many processes run “at the same time” by taking turns quickly.

Scheduling (why your program shares the CPU)

- Many processes run “at the same time” by taking turns quickly.
- Your program can be slowed by other heavy programs.

Scheduling (why your program shares the CPU)

- Many processes run “at the same time” by taking turns quickly.
- Your program can be slowed by other heavy programs.
- On a multicore CPU, multiple processes can truly run in parallel.

- **Relative path** is resolved from the **current working directory**.

File system basics you actually use

- **Relative path** is resolved from the **current working directory**.
- **Absolute path** starts from the root of the system.

File system basics you actually use

- **Relative path** is resolved from the **current working directory**.
- **Absolute path** starts from the root of the system.
- **Permissions** can block reading/writing even if the path is correct.

Python demo: where am I? (CWD and listing files)

```
import os

print("CWD:", os.getcwd())
print("Files here:", os.listdir("."))  # current directory
```

- If you get FileNotFoundError, print CWD first.

Python demo: diagnose `FileNotFoundError`

```python
import os

filename = "data/input.txt"
print("CWD:", os.getcwd())
print("Exists?", os.path.exists(filename))

try:
    with open(filename, "r", encoding="utf-8") as f:
        print(f.readline())
except FileNotFoundError as e:
    print("FileNotFoundError:", e)
    print("Try:", os.listdir("."))
```

- Most of the time: wrong relative path, wrong CWD, or file not created.

Definition: system call

Definition

A **system call** is a request from a program to the OS kernel, such as: open a file, read bytes, write bytes, get time, allocate memory, create a process.

- In Python, many library functions eventually trigger system calls.
- You do not need kernel knowledge; you need the mental model.

Algorithm: debug by layer (stop guessing)

Algorithm

Debug by layer

1. **Representation**: bytes/encoding/format? (print `repr`, set `encoding=`).
2. **File system**: path/CWD/exists? (print `os.getcwd()`, `os.path.exists`).
3. **Permissions**: read/write allowed? (catch `PermissionError`).
4. **Resources**: slow due to I/O/CPU/memory? (use `timeit`, avoid loading all).
5. **Logic last**: only after the layer checks pass.

Summary

What you must remember

- Python runs on top of the OS; the OS controls files, processes, permissions.
- Files are bytes; text is decoding. Always think: bytes + encoding.
- Slow programs are often I/O or memory bound; measure before optimizing.
- Debug by layer: representation $\rightarrow$ file system $\rightarrow$ permissions $\rightarrow$ resources $\rightarrow$ logic.