

chapter one

Programs, Processes, and Files

Running a Python file creates operating-system state that the source file does not contain: memory, open files, an execution position, and a process identifier. The shell, database, and server in this book all manipulate some part of that state. We will begin by separating the program stored on disk from the process executing it.

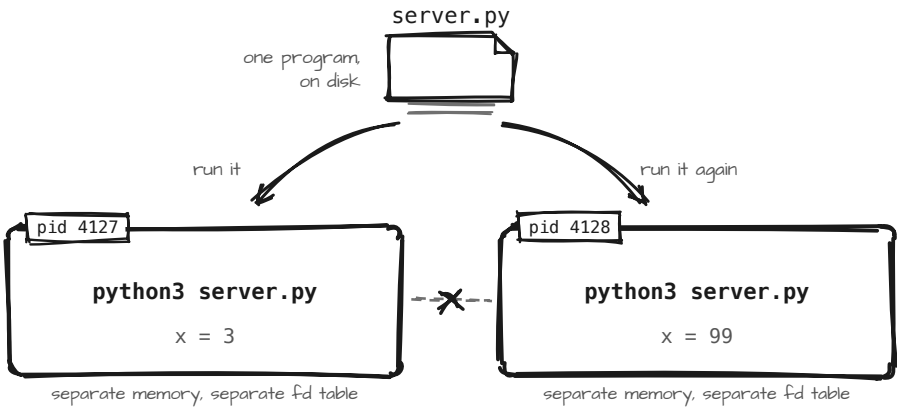
A program is not a process

A **program** is code and associated data in a stored form. Our examples are Python source files, though a program may span several files or be stored as a native executable.

A **process** is an operating-system instance of a running program. When you enter `python3 server.py`, the operating system starts the Python interpreter as a process, and the interpreter reads and executes `server.py`. The process has an address space, execution state, open descriptors, and other metadata maintained by the kernel.

Run the command in two terminals and you get two processes with different process identifiers and separate address spaces. Under ordinary conditions, changing a variable in one process does not change a variable in the other. Processes can communicate or share memory, but only through mechanisms arranged for that purpose.

One program, two PIDs. Each process also has a parent PID, credentials, and other kernel-maintained attributes.



Two processes execute the same program with separate address spaces and different PIDs.

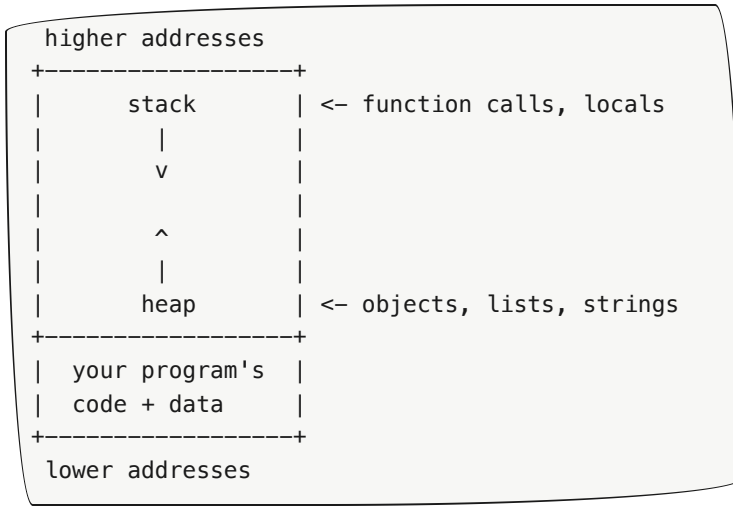
note

A program is stored code. A process is a running instance with state managed by the operating system.

What a process owns

The kernel records a collection of state for each process. This includes an address space, a file-descriptor table, a current working directory, credentials, signal state, and relationships to other processes. This chapter concentrates on memory and file descriptors because the projects use them directly.

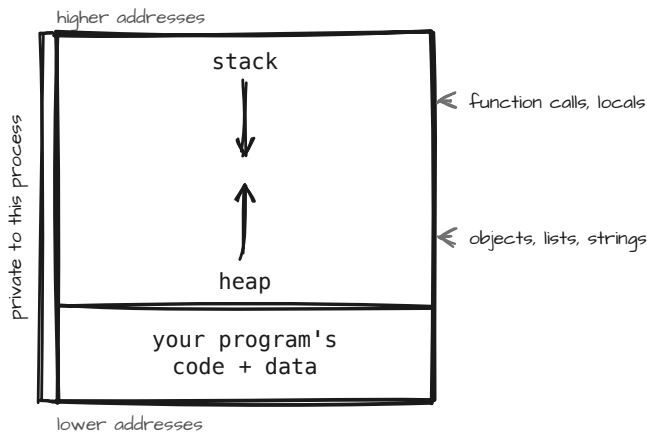
First, a process has a virtual **address space**. It is isolated from other processes by default, although explicit shared-memory mappings are possible. Python objects, interpreter state, native libraries, and call stacks all occupy regions in this address space. A simplified textbook layout looks like this:



This is a conceptual map, not a promise about addresses or growth directions on every platform.

Native code normally uses a stack for call frames and a heap for dynamic allocations. CPython manages Python objects in heap memory and keeps its own interpreter frames and allocator structures. The diagram omits shared libraries, memory-mapped files, guard pages, threads, and platform-specific layout choices. It is useful for distinguishing process memory from files on disk, not for predicting an exact address.

CPython combines reference counting with a cyclic garbage collector; memory management is automatic, but memory can still be retained.



A simplified address-space diagram. Real layouts contain more regions and vary by runtime, executable format, and operating system.

Second — and this is the possession we care about most — the process gets a **file descriptor table**. This is where files, terminals, and network connections all plug in, and it is worth its own section.

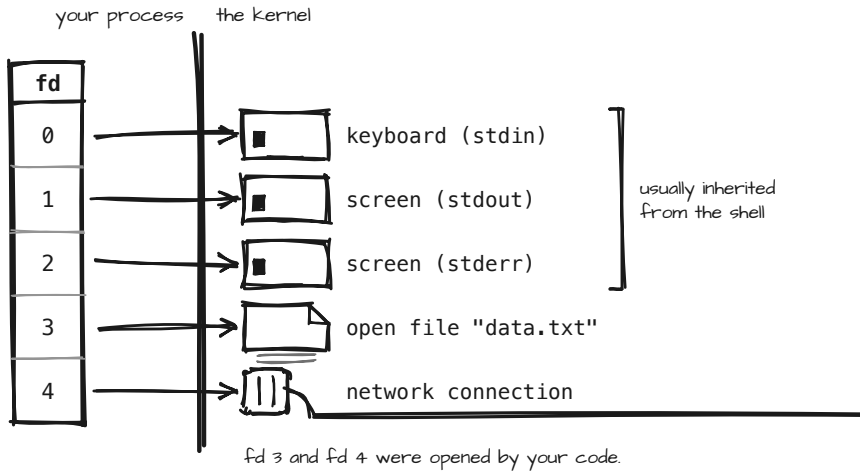
The file descriptor table

On Unix-like systems, opening a file successfully returns a small non-negative integer called a **file descriptor**, or **fd**. The process supplies that number to operations such as `read`, `write`, and `close`. Descriptors can also refer to pipes, terminals, and sockets.

Each process has a descriptor table. A table entry refers to kernel-managed open-file state or another I/O object. A typical interactive process might have entries like these:

The table is per-process. Your fd 3 is not my fd 3.

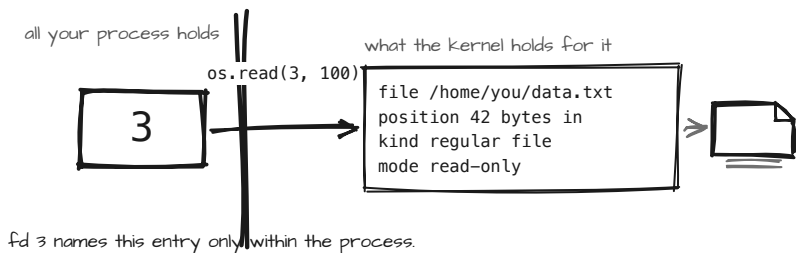
process's file descriptor table		
fd	refers to	
0	keyboard (stdin)	
1	screen (stdout)	
2	screen (stderr)	
3	open file "data.txt"	
4	network connection	



Descriptor numbers index a per-process table; the referenced I/O state is managed by the kernel.

The descriptor selects a table entry. For a regular file, the entry ultimately reaches an open file description that records properties such as the current offset and status flags. More than one descriptor can share that open file description after operations such as `dup` or `fork`. Sockets and other descriptor types keep different kernel state, but user code still names them with small integers.

A descriptor number has meaning only in the process whose table contains it.



fd 3 names this entry only within the process.

Descriptor 3 leads across the boundary to kernel-managed state for an open file.

The first three entries follow a convention used by shells and many other process launchers.

stdin, stdout, and stderr: fds 0, 1, 2

An interactive shell normally gives a new process three inherited descriptors:

- **fd 0 — standard input (stdin):** where the process reads its input from. In an interactive terminal, this usually refers to the terminal rather than directly to a keyboard.
- **fd 1 — standard output (stdout):** where the process writes its normal output. In an interactive terminal, this usually refers to the terminal.
- **fd 2 — standard error (stderr):** where the process writes diagnostics. It is a separate descriptor even when it refers to the same terminal as fd 1.

Two hoses, not one.
That is why
`2>/dev/null`
hides errors and
keeps output.

In a normal command-line Python process, `print("hello")` writes through the buffered `sys.stdout` object associated with fd 1, and an uncaught exception is reported through `sys.stderr`, associated with fd 2. Because the descriptors are separate, a shell can send standard output to a file while leaving diagnostics on the terminal. A program may close, replace, or reconfigure any of these streams, so the convention is common rather than guaranteed.

This convention predates Python. In a usual terminal session, you can inspect the descriptors underneath Python's stream objects:

`fileno()` is the
crowbar. It shows
you the integer under
the object.

```
import sys

print(sys.stdin.fileno())    # 0
print(sys.stdout.fileno())   # 1
print(sys.stderr.fileno())   # 2
```

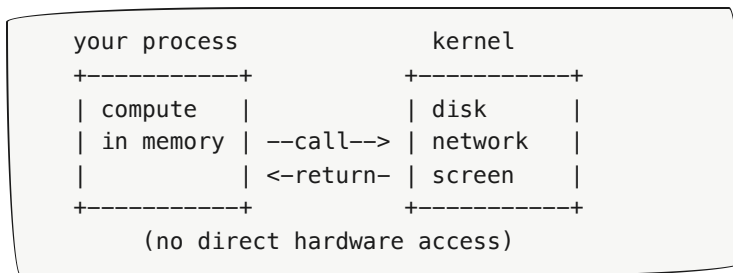
Part II uses this table directly. For `ls > out.txt`, the shell opens `out.txt` and arranges for fd 1 in the child

process to refer to that file before executing `ls`. The program continues writing to standard output; the descriptor table determines the destination. Pipes use the same mechanism to connect one process's output to another's input.

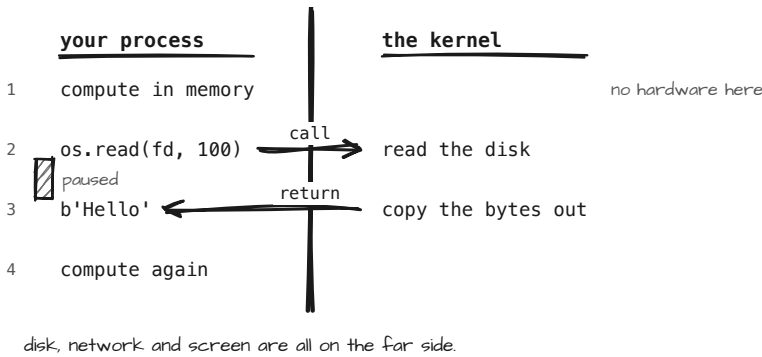
The kernel and system calls

The **kernel** manages processes, virtual memory, filesystems, devices, and networking. Application code normally runs without the privileges needed to manipulate those resources directly.

A **system call** crosses from application code into the kernel to request an operation. Unix provides calls for reading, writing, opening files, mapping memory, creating sockets, and managing processes. Library functions and language runtimes often wrap these calls, validate arguments, buffer data, or combine several calls into one higher-level operation.



After buffering and encoding, ordinary terminal output eventually reaches an OS write operation. One **print** need not equal one syscall.



A simplified system-call sequence: the process enters the kernel, the kernel performs or starts the operation, and control returns. Spacing indicates order, not elapsed time.

Python's `print`, `open`, and networking libraries eventually use operating-system services, but they add useful behavior along the way. To work closer to the descriptor interface, we will often use functions from the `os` module:

`os` is a relatively thin, portable wrapper over many OS facilities, but it is still a Python API rather than a syscall table.

```
import os

fd = os.open("data.txt",
              os.O_RDONLY)
data = os.read(fd, 100)
os.close(fd)
```

Here `fd` is one of the integers in the process's descriptor table. `os.read(fd, 100)` requests up to 100 bytes. It may return fewer, and it returns an empty byte string at end-of-file. Python's higher-level `open()` returns a file object that adds buffering, text encoding, iteration, and automatic cleanup patterns.

note

A system call is a controlled request from application code to the kernel. Python usually reaches it through one or more library layers.

Where the projects use this model

pysh, the shell, creates processes and arranges their file descriptors. To build `ls | grep py`, it connects the write end of a pipe to `ls` on fd 1 and the read end to `grep` on fd 0.

littlekv, the key-value database, appends encoded records to a file and records their locations in memory. File contents can remain after the process exits, but surviving a crash requires more than calling `write`; later chapters examine buffering, flushing, ordering, and recovery.

minihttp, the web server, listens for network connections and exchanges bytes with clients. On Unix-like systems, its sockets have file descriptors and participate in many of the same I/O and readiness interfaces as regular files, though sockets have their own operations and semantics.

On Unix-like systems, descriptors provide a common naming and I/O interface for regular files, terminals, pipes, and sockets. Their behavior is not identical: a seekable file is different from a terminal or a TCP byte stream. Chapter 2 starts with the value all of them can carry: bytes.

Exercises

Exercise 1.1. Write a two-line program that prints `os.getpid()` and then sleeps for thirty seconds. Run it in two terminals at the same time. Predict, before you look, whether the two runs print the same number, and explain your answer in terms of the program-versus-process distinction. What single program file is on disk, and how many processes exist while both are sleeping?

Exercise 1.2. Open two files at once with `os.open` and print the file descriptor each call returns:

```
import os
a = os.open("x.txt", os.O_RDONLY)
b = os.open("y.txt", os.O_RDONLY)
print(a, b)
```

Note the integers. Now `os.close(a)`, open a third file, and print its descriptor. Which number does the OS hand back, and what does that tell you about how the descriptor table is filled?

Exercise 1.3. Using `os.write(1, ...)` and `os.write(2, ...)`, write one line to standard output and one to standard error from the same program. Run it twice: once plainly, and once with `python3 prog.py > out.txt`. Which line still appears on your screen, which lands in the file, and why does redirection separate them?

Exercise 1.4. Read the first ten bytes of a text file using only `os.open`, `os.read`, and `os.close` — no `open()`, no `with`. Then call `os.read` a second time without reopening. Explain why the second read continues where the first stopped rather than starting over.