The Deep Dive into Command Line, Environment Variables, and Python Modules

Let me take you on an extensive journey through these fundamental computing concepts. I'll break down every layer, from the highest level abstractions down to the metal.

---

1. HOW THE COMMAND LINE WORKS AND ENVIRONMENT VARIABLES

The Command Line Interface (CLI) - The Foundation

The command line is a **text-based interface** that allows you to communicate directly with your operating system. Unlike graphical user interfaces (GUIs) where you click buttons and icons, the CLI requires you to type commands as text strings.

The Shell - Your Command Interpreter

When you open a terminal or command prompt, you're actually launching a program called a **shell**. The shell is a special program that:

1. **Reads** your input (commands you type)
2. **Interprets** what you mean
3. **Executes** the appropriate programs
4. **Returns** the output back to you

Common shells include:

- **Bash** (Bourne Again Shell) - Default on most Linux systems
- **Zsh** (Z Shell) - Default on modern macOS
- **PowerShell** - Modern Windows shell
- **CMD** (Command Prompt) - Traditional Windows shell
- **Fish** - A user-friendly alternative shell

The Command Execution Lifecycle - When You Type `python main.py`

Let's break down exactly what happens when you type `python main.py` and press Enter:

Step 1: Input Reading and Parsing

python main.py

The shell receives this string and begins parsing it. It identifies:

- **Command**: `python` (the program to execute)
- **Arguments**: `main.py` (data passed to the program)

The shell uses **whitespace** (spaces and tabs) as delimiters to separate the command from its arguments. If you had typed:

python main.py arg1 arg2 --flag

The shell would parse this as:

- Command: `python`
- Argument 0: `main.py`
- Argument 1: `arg1`
- Argument 2: `arg2`
- Argument 3: `--flag`

Step 2: Command Resolution - Finding the Program

Now the shell needs to find the actual `python` program. It doesn't know where Python is installed on your system, so it uses the **PATH environment variable** to search for it.

The shell performs these steps:

1. **Checks if it's a built-in command** - Some commands are built directly into the shell (like `cd`, `echo`, `exit`)
2. **Checks if it's an absolute path** - If you typed `/usr/bin/python`, it would use that exact location
3. **Checks if it's a relative path** - If you typed `./python`, it would look in the current directory
4. **Searches the PATH** - If it's just `python`, the shell searches directories listed in PATH

Step 3: The PATH Environment Variable - The Search Mechanism

The PATH is an **environment variable** (more on this shortly) that contains a list of directory paths separated by colons (Unix/Linux/macOS) or semicolons (Windows):

Unix/Linux/macOS example:

/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/home/user/.local/bin

Windows example:

C:\Python39;C:\Windows\System32;C:\Program Files\Git\cmd

When you type `python`, the shell:

1. Takes the first directory from PATH: `/usr/local/bin`
2. Checks if `/usr/local/bin/python` exists and is executable
3. If not found, moves to the next directory: `/usr/bin`
4. Checks if `/usr/bin/python` exists and is executable
5. Continues until it finds the executable or exhausts all directories
6. If not found anywhere, returns: `command not found: python`

You can see where a command is located using:

which python   # Unix/Linux/macOS
where python   # Windows

Step 4: File Permissions and Executable Check

On Unix-like systems, the shell verifies:

- The file exists
- You have **execute permissions** on the file
- The file is actually executable (not just a regular file)

File permissions look like this:

-rwxr-xr-x 1 user staff 12345 Oct 19 10:30 python

The `x` indicates execute permission. Without it, the shell would refuse to run the program.

Step 5: Process Creation - The Fork and Exec

Once the shell finds `python`, it needs to create a new process to run it. This involves two fundamental system calls:

On Unix/Linux/macOS:

1. **fork()** - The shell creates a copy of itself (child process)
    - The child process is an exact duplicate of the parent
    - It has its own memory space
    - It inherits environment variables from the parent
2. **exec()** - The child process replaces itself with the Python program
    - The Python executable is loaded into memory
    - The child's memory is overwritten with Python's code
    - The process now runs Python instead of the shell

On Windows:

- Windows uses `CreateProcess()` which combines these steps

Step 6: Passing Arguments and Environment

The operating system passes several things to the new Python process:

1. **Command-line arguments**: `['main.py']`
    - These become `sys.argv` in Python
    - `sys.argv[0]` is the script name
    - `sys.argv[1:]` are additional arguments
2. **Environment variables**: A copy of all environment variables
    - These become accessible via `os.environ` in Python
3. **Standard streams**:
    - **stdin** (standard input) - connected to your keyboard
    - **stdout** (standard output) - connected to your terminal screen
    - **stderr** (standard error) - also connected to your terminal
4. **File descriptors**: Open files that should be inherited
5. **Working directory**: The current directory (`pwd` or `cd` location)

Step 7: Python Interpreter Startup

Now Python begins its own startup process:

1. **Initialize the interpreter**
    - Set up the Python runtime environment
    - Initialize memory management (heap, garbage collector)

    - Load the Python standard library modules
2. **Check the Python path** (different from PATH!)
    - Python has its own `sys.path` list
    - This tells Python where to look for modules
    - Includes: current directory, site-packages, standard library
3. **Parse the script name**: `main.py`
    - Convert to an absolute path if needed
    - Verify the file exists
4. **Compile the script**
    - Python reads `main.py`
    - Parses it into an Abstract Syntax Tree (AST)
    - Compiles to bytecode
    - Stores bytecode in `__pycache__` (more later)
5. **Execute the bytecode**
    - The Python Virtual Machine (PVM) runs the bytecode
    - Your code finally executes!

Step 8: Output and Completion

- Any `print()` statements send data to stdout
- The terminal displays this output
- When Python finishes, it returns an **exit code**:
    - `0` means success
    - Non-zero means an error occurred
- The shell receives this exit code
- The shell displays the prompt again, ready for the next command

---

ENVIRONMENT VARIABLES - The Deep Dive

Environment variables are **key-value pairs** stored in the process's memory that affect how programs behave. They're a form of **inter-process communication** and **configuration management**.

What Are Environment Variables Exactly?

Think of environment variables as a **global configuration dictionary** that every process has access to. They exist in the **process's memory space** and are inherited by child processes.

Structure:

NAME=value

Examples:

HOME=/home/username
USER=john
LANG=en_US.UTF-8
PATH=/usr/bin:/bin

How Environment Variables Are Stored

In memory, environment variables are stored as an **array of strings** (called `environ` in C). Each string has the format `KEY=VALUE`.

When you write a C program, you can access them through:

```
extern char **environ;

// environ[0] might be "HOME=/home/user"
// environ[1] might be "PATH=/usr/bin:/bin"
// environ[n] is NULL (end marker)
```

In Python, they're accessible through `os.environ`, which presents them as a dictionary-like object.

Scope and Inheritance

Environment variables follow a **hierarchical inheritance model**:

```
Operating System (Boot)
   ↓
Login Shell (when you log in)
   ↓
Terminal/Shell (when you open terminal)
   ↓
Your Program (when you run python main.py)
```

Each level can:

- **Inherit** variables from its parent

- **Add** new variables
- **Modify** existing variables
- **Remove** variables

Crucially: Changes only affect the current process and its children. You **cannot** modify the parent's environment.

Example:

```
# In terminal 1
export MY_VAR="hello"
python script.py  # Can see MY_VAR

# In terminal 2 (different process)
python script.py  # Cannot see MY_VAR
```

Common Environment Variables

PATH - The Program Search Path

The most critical environment variable. It tells the shell where to look for executable programs.

Viewing PATH:

```
echo $PATH  # Unix/Linux/macOS
echo %PATH%  # Windows
```

Modifying PATH:

```
# Temporarily (current session only)
export PATH="/new/directory:$PATH"  # Unix/Linux/macOS
set PATH=C:\new\directory;%PATH%   # Windows

# Permanently - edit configuration files:
# ~/.bashrc or ~/.bash_profile or ~/.zshrc (Unix/Linux/macOS)
# System Properties → Environment Variables (Windows)
```

Why PATH matters:

- Without the Python directory in PATH, typing `python` wouldn't work
- You'd have to use the full path: `/usr/local/bin/python main.py`
- PATH makes commands portable and convenient

HOME - User's Home Directory

Points to the user's home folder:

HOME=/home/username # Linux
HOME=/Users/username # macOS
USERPROFILE=C:\Users\Username # Windows

Used by programs to:

- Store user-specific configuration files
- Find documents and downloads
- Store cache and temporary files

PYTHONPATH - Python Module Search Path

Tells Python where to look for modules:

export PYTHONPATH="/my/custom/modules:/another/path"

This **adds** to `sys.path`, which Python uses when you `import` modules.

VIRTUAL_ENV - Active Virtual Environment

When you activate a Python virtual environment:

source venv/bin/activate

It sets:

VIRTUAL_ENV=/path/to/venv

And prepends the venv's `bin` directory to PATH:

PATH=/path/to/venv/bin:/usr/local/bin:/usr/bin...

Now `python` points to the venv's Python, not the system Python!

Other Important Variables

- **LANG/LC_ALL** - Locale and language settings
- **EDITOR** - Default text editor (`vim`, `nano`, `code`)

- **USER/USERNAME** - Current user's name
- **PWD** - Present working directory
- **OLDPWD** - Previous working directory
- **SHELL** - Path to the current shell program
- **TERM** - Terminal type
- **DISPLAY** - X11 display server (Linux GUI)

Setting and Using Environment Variables

In Shell:

```
# Set temporarily (current session)
export MY_VAR="value"

# Use in commands
echo $MY_VAR
python -c "import os; print(os.environ['MY_VAR'])"

# Set for single command only
MY_VAR="value" python main.py
```

In Python:

```
import os

# Read environment variable
path = os.environ.get('PATH')
home = os.environ['HOME']  # Raises KeyError if not found

# Set environment variable (affects child processes only)
os.environ['MY_VAR'] = 'value'

# Check if variable exists
if 'MY_VAR' in os.environ:
    print("Variable exists!")

# Remove variable
del os.environ['MY_VAR']

# Get all environment variables
all_vars = dict(os.environ)
```

In Configuration Files:

Unix/Linux/macOS:

```
# ~/.bashrc or ~/.zshrc
export PATH="$HOME/.local/bin:$PATH"
export EDITOR=vim
export PYTHONPATH="/my/modules"
```

Windows:

- System Properties → Advanced → Environment Variables
- Or use `setx` command:

```
setx PATH "C:\Python39;%PATH%"
```

Environment Variables vs. Shell Variables

There's a subtle but important distinction:

Shell Variables - Local to the shell, not inherited:

```
MY_VAR="value"  # Shell variable
echo $MY_VAR    # Works in shell
python -c "import os; print(os.environ.get('MY_VAR'))"  # Returns None!
```

Environment Variables - Exported, inherited by children:

```
export MY_VAR="value"  # Environment variable
python -c "import os; print(os.environ['MY_VAR'])"  # Works!
```

The `export` command promotes a shell variable to an environment variable.

The Complete Flow: `python main.py` from Start to Finish

Let me trace every single step:

1. **You type**: `python main.py` and press Enter
2. **Terminal captures input**: Raw keyboard input sent to the shell process
3. **Shell reads the line**: Stored in a buffer, parsing begins

4. **Tokenization**: Split into `["python", "main.py"]`

5. **Command type check**: Is it built-in? No. Is it a function? No. Is it an executable? Check PATH...

6. **PATH search**:

    - Check `/usr/local/bin/python` → Not found
    - Check `/usr/bin/python` → Found! ✓

7. **Permission check**: Can I execute `/usr/bin/python`? Yes ✓

8. **Process creation**: Shell calls `fork()` → Creates child process

9. **Environment setup**: Child inherits all environment variables

10. **Stream setup**: Connect stdin/stdout/stderr to terminal

11. **Working directory**: Set to current directory

12. **Execution**: Child calls `exec()` → Becomes Python interpreter

13. **Python startup**:

    - Initialize interpreter
    - Load built-in modules
    - Parse `main.py`
    - Compile to bytecode
    - Store in `__pycache__/main.cpython-39.pyc`

14. **Run bytecode**: Python VM executes your code

15. **Output**: Any `print()` goes to stdout → displayed in terminal

16. **Exit**: Python finishes, returns exit code (0 for success)

17. **Shell resumes**: Gets exit code, shows prompt again

---

2. HOW MODULES WORK AND WHAT THEY ARE

What is a Module - The Fundamental Concept

A **module** in Python is simply a **file containing Python code**. That's it. Any file ending in `.py` is a module. But this simple concept enables powerful code organization and reuse.

Why Modules Exist

Imagine writing all your code in one massive file:

```
# Everything in one file - 50,000 lines!
def function1():
    pass

def function2():
    pass

class MyClass:
    pass

# ... 49,950 more lines ...
```

This becomes:

- **Unmanageable** - Hard to find anything
- **Unmaintainable** - Changes break everything
- **Unreusable** - Can't share code between projects
- **Collaborative nightmare** - Multiple people editing the same file

Modules solve this by allowing you to **organize code into separate files** and **import** what you need.

The Module System - Deep Architecture

Types of Modules

Single-file modules: One `.py` file

```
math_utils.py  # This is a module
```

1.

Package modules: A directory with `__init__.py`

```
mypackage/
    __init__.py  # Makes this directory a package
```

module1.py
module2.py

2.
3. **Built-in modules**: Written in C, compiled into Python
   - Examples: `sys`, `os`, `math`, `time`
   - Extremely fast, part of Python interpreter
4. **Extension modules**: Third-party, often with C extensions
   - Examples: `numpy`, `pandas`, `tensorflow`
   - Installed via `pip`

The Import Mechanism - How `import` Works

When you write `import math`, a complex process unfolds:

Step 1: Check `sys.modules` Cache

Python maintains a dictionary called `sys.modules` that caches all imported modules:

```
import sys
print(sys.modules)
# {'sys': <module 'sys'>, 'os': <module 'os'>, ...}
```

First check: Is `math` already in `sys.modules`?

- **Yes**: Return the cached module object (very fast!)
- **No**: Proceed with import process

This is why subsequent imports are instant:

```
import math  # First import - slow (microseconds)
import math  # Second import - instant (nanoseconds)
import math  # Third import - instant
```

Step 2: Module Location - The `sys.path` Search

Python needs to find the module file. It searches directories listed in `sys.path`:

```
import sys
```

```
print(sys.path)
# [
#     '/current/directory',          # Where you ran python
#     '/home/user/.local/lib/python3.9/site-packages',  # User packages
#     '/usr/lib/python3.9',          # Standard library
#     '/usr/lib/python3.9/lib-dynload',  # Dynamic libraries
#     ...
# ]
```

Search order:

1. **Current directory** (where `main.py` is)
2. **PYTHONPATH** directories (if set)
3. **Standard library** directories
4. **site-packages** (third-party packages)

For `import math`, Python searches:

- `./math.py` → Not found
- `./math/__init__.py` → Not found
- `/usr/lib/python3.9/math.py` → Not found
- `/usr/lib/python3.9/lib-dynload/math.cpython-39-x86_64-linux-gnu.so` → **Found!**

Note: `math` is a built-in module written in C, so it's a `.so` (shared object) file, not `.py`.

Step 3: Module Loading - Different Strategies

Depending on the module type, Python uses different loaders:

For .py files (Source Code):

1. **Read the file**: Open and read `module.py`

2. **Check for bytecode**: Look for `__pycache__/module.cpython-39.pyc`

    - **Bytecode exists and is up-to-date**: Use it (faster)
    - **No bytecode or outdated**: Compile source code
3. **Compile to bytecode**:

    - Parse source code into Abstract Syntax Tree (AST)
    - Compile AST to Python bytecode
    - Save bytecode to `__pycache__` for next time

4. **Execute bytecode**: Run the compiled code

For built-in modules (C code):

1. **Load shared library**: Call operating system to load `.so` or `.dll`
2. **Find initialization function**: Look for `PyInit_modulename`
3. **Initialize module**: Call the initialization function
4. **Register in sys.modules**: Cache the module object

For packages (directories):

1. **Find `__init__.py`**: Must exist (or use namespace packages)
2. **Execute `__init__.py`**: This defines the package's public API
3. **Create package object**: Store in `sys.modules`

Step 4: Module Object Creation

Python creates a **module object** with these attributes:

```
import math
print(type(math))  # <class 'module'>
print(dir(math))   # ['__name__', '__file__', 'sqrt', 'cos', ...]
```

Module object attributes:

- `__name__`: Module name (`'math'`)
- `__file__`: Path to the module file
- `__package__`: Package name (for packages)
- `__doc__`: Module docstring
- `__dict__`: Dictionary containing all module attributes
- All functions, classes, variables defined in the module

Step 5: Namespace Binding

The module is bound to a name in the current namespace:

```
import math
# Creates binding: 'math' -> module object
print(math.sqrt(16))  # Access via the name 'math'
```

Different import styles create different bindings:

```
# Style 1: Import module
import math
# Binding: 'math' -> module object

# Style 2: Import specific name
from math import sqrt
# Binding: 'sqrt' -> function object

# Style 3: Import with alias
import numpy as np
# Binding: 'np' -> module object

# Style 4: Import everything (discouraged!)
from math import *
# Bindings: 'sqrt', 'cos', 'sin', ... -> function objects
```

Step 6: Code Execution

When a module is imported, **all code in the module runs immediately**:

```
# module.py
print("Loading module!")  # This prints during import
x = 10                    # This executes during import

def func():
    print("Called!")      # This only executes when func() is called

# main.py
import module  # Prints: "Loading module!"
module.func()  # Prints: "Called!"
```

This is why you often see:

```
if __name__ == '__main__':
    # This only runs if the file is executed directly
    # Not when imported as a module
    main()
```

Module Internals - The Module Object

Let's examine what a module actually is:

```python
import math

# Module is an object of type 'module'
print(type(math))  # <class 'module'>

# It has a dictionary of all its attributes
print(math.__dict__.keys())
# dict_keys(['__name__', '__doc__', '__package__', '__loader__',
#           '__spec__', '__file__', 'acos', 'sqrt', ...])

# You can access attributes via dot notation
print(math.sqrt)  # <built-in function sqrt>

# Or via the dictionary
print(math.__dict__['sqrt'])  # <built-in function sqrt>

# They're the same object
assert math.sqrt is math.__dict__['sqrt']
```

When you write `math.sqrt(16)`, Python:

1. Looks up `math` in the current namespace
2. Gets the module object
3. Looks up `sqrt` in the module's `__dict__`
4. Gets the function object
5. Calls it with argument `16`

Creating Your Own Module - Step by Step

Let's create a module and understand everything that happens:

```python
# calculator.py
"""A simple calculator module.

This module provides basic arithmetic operations.
"""

# Module-level variable
PI = 3.14159

# Private variable (by convention)
```

```
_internal_value = 42

# Public function
def add(a, b):
    """Add two numbers."""
    return a + b

def multiply(a, b):
    """Multiply two numbers."""
    return a * b

# Private function (by convention)
def _helper():
    """Internal helper function."""
    return _internal_value

# Class definition
class Calculator:
    """A calculator class."""

    def __init__(self):
        self.result = 0

    def add(self, value):
        self.result += value
        return self.result

# Code that runs on import
print(f"Calculator module loaded! PI = {PI}")

# Main guard
if __name__ == '__main__':
    # This only runs when executing: python calculator.py
    # Not when: import calculator
    print("Running calculator directly!")
    print(add(5, 3))
```

Now let's use it:

```
# main.py
import calculator

# Module-level access
print(calculator.PI)  # 3.14159
```

```python
# Function access
result = calculator.add(10, 5)  # 15

# Class access
calc = calculator.Calculator()
calc.add(5)
calc.add(3)
print(calc.result)  # 8

# Check what was printed during import
# Output: "Calculator module loaded! PI = 3.14159"
```

Package System - Organizing Multiple Modules

When projects grow, you organize modules into packages:

```
myproject/
    __init__.py         # Makes myproject a package
    math_ops/
        __init__.py     # Makes math_ops a package
        basic.py        # Module with basic operations
        advanced.py     # Module with advanced operations
    utils/
        __init__.py
        helpers.py
```

The `__init__.py` File

This special file:

1. **Marks a directory as a package**
2. **Runs when the package is imported**
3. **Defines the package's public API**

```python
# myproject/math_ops/__init__.py
"""Math operations package."""

# Import commonly used functions to package level
from .basic import add, subtract
from .advanced import power, factorial
```

```
# Define what's exported with "from math_ops import *"
__all__ = ['add', 'subtract', 'power', 'factorial']

# Package-level variable
VERSION = "1.0.0"

print("Math operations package initialized!")
```

Now you can import:

```
# Import the package
import myproject.math_ops
# Prints: "Math operations package initialized!"

# Access through package
result = myproject.math_ops.add(5, 3)

# Import specific function
from myproject.math_ops import add
result = add(5, 3)

# Import from submodule
from myproject.math_ops.basic import add, subtract
```

Relative vs. Absolute Imports

Within a package, you can use relative imports:

```
# myproject/math_ops/advanced.py

# Absolute import
from myproject.math_ops.basic import add

# Relative import (preferred within packages)
from .basic import add        # Same directory
from ..utils import helpers   # Parent directory
from .advanced import power   # Same directory
```

Relative imports use dots:

- `.` = current package
- `..` = parent package

- `...` = grandparent package

The Import Hooks System - Advanced Customization

Python allows you to customize the import mechanism through **import hooks**:

```
import sys

class CustomImporter:
    """Custom module finder and loader."""

    def find_module(self, fullname, path=None):
        """Find the module."""
        if fullname.startswith('special_'):
            return self  # This object will load it
        return None  # Let normal import handle it

    def load_module(self, fullname):
        """Load the module."""
        # Create module object
        module = type(sys)(fullname)
        module.__file__ = "<generated>"
        module.__loader__ = self

        # Add custom content
        module.greeting = "Hello from custom importer!"

        # Register in sys.modules
        sys.modules[fullname] = module
        return module

# Install the custom importer
sys.meta_path.insert(0, CustomImporter())

# Now you can import special_* modules
import special_module
print(special_module.greeting)
# Output: "Hello from custom importer!"
```

This is how tools like:

- **pytest** - Modifies imports for test discovery

- **coverage.py** - Instruments code during import
- **Django** - Provides lazy module loading
- **PyInstaller** - Bundles modules into executables

Circular Imports - The Problem and Solutions

A common issue when modules import each other:

```
# module_a.py
from module_b import func_b

def func_a():
    return func_b() + 1

# module_b.py
from module_a import func_a

def func_b():
    return func_a() + 1
```

This causes an ImportError!

Why it fails:

1. `main.py` imports `module_a`
2. `module_a` tries to import `module_b`
3. `module_b` tries to import `module_a`
4. But `module_a` isn't finished loading yet!
5. Error: "cannot import name 'func_a'"

Solutions:

1. **Restructure code** - Extract shared code to a third module
2. **Import inside functions** - Delay import until needed
3. **Import the module, not names** - Use `import module_a` instead of `from module_a import ...`

```
# Solution 2: Import inside function
# module_a.py
def func_a():
    from module_b import func_b  # Import here, not at top
    return func_b() + 1
```

Module Reloading - For Development

Normally, modules are loaded once and cached. To reload:

```
import importlib
import my_module

# Reload the module (useful during development)
importlib.reload(my_module)
```

Note: Reloading is tricky:

- Existing references to old module objects remain
- State is not preserved
- Not recommended for production code

---

3. THE `__pycache__` FOLDER AND ITS CONTENTS

What is `__pycache__`?

The `__pycache__` directory is Python's **bytecode cache**. It stores compiled versions of your `.py` files to make subsequent imports faster.

Why Bytecode Compilation?

Python is an **interpreted language**, but it doesn't directly interpret your source code. Instead:

1. **Source code** (`.py`) → Human-readable Python code
2. **Bytecode** (`.pyc`) → Low-level, platform-independent instructions
3. **Python VM** → Executes bytecode

The compilation process:

```
your_module.py  →  [Parser]  →  [Compiler]  →  bytecode  →  [PVM]  →  Execution
                    ↓                  ↓
                   AST              __pycache__/
```

When is `__pycache__` Created?

Python creates `__pycache__` when:

1. You **import** a module
2. The module hasn't been compiled yet, or
3. The source file is newer than the cached bytecode

Python does NOT create `__pycache__` when:

- You run a script directly: `python main.py`
- The script is not imported by another module

Example:

No __pycache__ created
python main.py

# __pycache__ IS created for helper.py
# main.py:
import helper # This triggers bytecode compilation of helper.py

---

Structure of `__pycache__`

project/
  main.py
  helper.py
  utils.py
  __pycache__/
    helper.cpython-39.pyc
    utils.cpython-39.pyc

Filename format: `{module}.cpython-{version}.pyc`

- `helper` - Module name
- `cpython` - Python implementation (CPython, PyPy, Jython)
- `39` - Python version (3.9)
- `.pyc` - Python compiled bytecode

Why include version in filename?

- Multiple Python versions can coexist
- Python 3.8 bytecode differs from Python 3.9
- Prevents conflicts

Contents of `.pyc` Files - Deep Dive

A `.pyc` file contains:

1. Magic Number (4 bytes)

Identifies Python version and bytecode format:

0x0a0d 0d0a # Python 3.9
0x0a0d 0e0a # Python 3.10

If Python tries to load bytecode with a wrong magic number, it recompiles the source.

2. Timestamp or Hash (4-8 bytes)

Two modes:

Timestamp-based (default):

- Stores source file's modification time
- On import, Python checks if source is newer
- If yes, recompile

Hash-based (deterministic):

- Stores source file's hash (SipHash)
- Generated with `python -

The `exec()` System Call - An Extremely Deep Dive

Let me explain one of the most fundamental operations in Unix/Linux systems: the `exec()` family of system calls. This is absolutely crucial to understanding how programs are launched.

What is `exec()` and Why Does It Exist?

The `exec()` system call is how a running process **transforms itself into a different program**. It's not about starting a new process - it's about **replacing** the current process with a new program.

The Problem It Solves

Imagine you're the shell, and the user types:

```
python main.py
```

You need to run Python. You have two fundamental problems:

1. **The shell is already running** - You can't just "become" Python while still being a shell
2. **You need to stay alive** - After Python finishes, the user needs the shell back

The solution? A two-step dance:

1. **`fork()`** - Make a copy of yourself (create a child process)
2. **`exec()`** - Have the child transform into Python

The `exec()` Family of Functions

There isn't just one `exec()` - there's a whole family! Each member differs in **how you pass arguments** and **how you specify the program**.

The Six Variants

```
int execl(const char *path, const char *arg0, ..., NULL);
int execv(const char *path, char *const argv[]);
int execle(const char *path, const char *arg0, ..., NULL, char *const envp[]);
int execve(const char *path, char *const argv[], char *const envp[]);
int execlp(const char *file, const char *arg0, ..., NULL);
int execvp(const char *file, char *const argv[]);
```

The naming convention:

- `exec` - Base name
- `l` vs `v` - How arguments are passed
    - `l` = **list** (variable number of arguments)
    - `v` = **vector** (array of arguments)
- `e` - Environment variables explicitly passed
- `p` - Use **PATH** to find the executable

Only `execve()` is an actual system call! All others are library wrappers that eventually call `execve()`.

`execve()` - The Real System Call

Let's focus on `execve()` since it's what actually talks to the kernel:

```
int execve(const char *pathname, char *const argv[], char *const envp[]);
```

Parameters Explained

1. `pathname` - The Program to Execute

This is the **absolute path** to the executable file:

```
execve("/usr/bin/python3", ...);  // Correct
execve("python3", ...);           // WRONG - needs full path
```

What happens with this parameter:

1. **Kernel opens the file** at `pathname`
2. **Checks permissions** - Must have execute permission
3. **Reads the file header** - First few bytes determine file type
4. **Identifies the format**:
    - **ELF binary** (Executable and Linkable Format) - Native executable
    - **Shebang script** (`#!/usr/bin/python3`) - Interpreted script
    - **Other formats** - Returns error

2. `argv[]` - Argument Vector

This is an **array of strings** (NULL-terminated) that becomes the program's command-line arguments.

Structure:

```
char *argv[] = {
    "python3",          // argv[0] - Program name (by convention)
    "main.py",          // argv[1] - First actual argument
    "--verbose",        // argv[2]
    NULL                // Must end with NULL!
};
```

Critical details:

- **`argv[0]` is the program name** (by convention, not enforced)
    - Usually the same as the executable name
    - But can be **anything** - the program won't know the difference
    - Some programs check `argv[0]` to change behavior (e.g., `busybox`)
- **Array must be NULL-terminated**
    - The last element must be `NULL`
    - This tells the program where arguments end
    - Without it, program reads garbage memory
- **All elements are strings**
    - Even numbers must be strings: `"42"`, not `42`
    - The program must parse them

Example of `argv` in memory:

```
Memory Address    Content
--------------    -------
0x1000            → "python3\0"
0x1008            → "main.py\0"
0x1010            → "--verbose\0"
0x1018            → NULL

argv array:
argv[0] = 0x1000  (points to "python3")
argv[1] = 0x1008  (points to "main.py")
argv[2] = 0x1010  (points to "--verbose")
argv[3] = 0x1018  (NULL)
```

3. `envp[]` - Environment Variables

This is an **array of strings** (NULL-terminated) containing environment variables in `KEY=VALUE` format.

Structure:

```
char *envp[] = {
    "PATH=/usr/bin:/bin",
    "HOME=/home/user",
    "PYTHONPATH=/my/modules",
    "USER=john",
    NULL                    // Must end with NULL!
};
```

Each string has format: `"VARIABLE_NAME=value"`

Example in memory:

```
Memory Address    Content
--------------    -------
0x2000          → "PATH=/usr/bin:/bin\0"
0x2020          → "HOME=/home/user\0"
0x2040          → "USER=john\0"
0x2050          → NULL

envp array:
envp[0] = 0x2000  (points to "PATH=...")
envp[1] = 0x2020  (points to "HOME=...")
envp[2] = 0x2040  (points to "USER=...")
envp[3] = 0x2050  (NULL)
```

What Happens During `execve()` - Step by Step

When you call `execve("/usr/bin/python3", argv, envp)`, here's the **complete journey**:

Step 1: Kernel Entry

```
execve("/usr/bin/python3", argv, envp);
```

1. **System call transition**:
    - User space → Kernel space
    - CPU switches to privileged mode
    - Execution continues in kernel code
2. **Parameter validation**:
    - Check if pointers are valid
    - Check if memory is accessible
    - Verify no NULL pointers (except array terminators)

Step 2: File Loading and Validation

Open the file: `/usr/bin/python3`

```
// Kernel code (simplified)
int fd = open("/usr/bin/python3", O_RDONLY);
if (fd < 0) return -ENOENT;  // File not found
```

1.
2. **Permission checks**:
    - Must have execute permission (`x` bit)
    - Check user/group/other permissions
    - Check setuid/setgid bits (special permissions)

```
-rwxr-xr-x  1 root root  5234576  python3
 ^
 This 'x' means executable
```

3.

Read file header (first ~128 bytes):

```
char header[128];
read(fd, header, 128);
```

4.

Identify file format:

Option A: ELF Binary (Native executable)

```
Header: 7F 45 4C 46 ...  (ELF magic number)
```

5.
    - This is a compiled native program
    - Written in C, C++, Rust, Go, etc.
    - Runs directly on CPU

Option B: Shebang Script (Interpreted)

Header: #! /usr/bin/python3

6.
    - This is a script that needs an interpreter
    - Kernel extracts the interpreter path: `/usr/bin/python3`
    - **Kernel recursively calls `execve()` with the interpreter!**

Example transformation:

Original call:
execve("script.py", ["script.py"], envp)

Becomes:
execve("/usr/bin/python3", ["python3", "script.py"], envp)

7.

Step 3: Memory Demolition

This is where things get dramatic. **The current process's memory is completely destroyed:**

1. **Text segment** (program code) - Erased
2. **Data segment** (global variables) - Erased
3. **Heap** (dynamically allocated memory) - Erased
4. **Stack** (local variables, call stack) - Erased
5. **Memory mappings** - Closed (except specific ones)

What survives:

- **Process ID (PID)** - Stays the same
- **Parent Process ID (PPID)** - Stays the same
- **File descriptors** - Unless marked close-on-exec
    - stdin (0), stdout (1), stderr (2) remain open
    - Other open files remain unless `FD_CLOEXEC` flag set
- **Current directory** - Stays the same
- **Signal handlers** - Reset to default
- **Process credentials** - UID, GID, groups

This is crucial: After `execve()`, the process has the **same PID**, but it's running **completely different code**. It's like a **body transplant** for the process.

Step 4: New Program Loading

Now the kernel loads the new program into memory:

Create new address space:

Memory Layout (for ELF binary):

```
High addresses
┌─────────────────────────────┐
│ Kernel Space   │ (not accessible to program)
├─────────────────────────────┤
│   Stack        │ (grows downward)
│     ↓        │
├─────────────────────────────┤
│              │
│  (unmapped)    │
│              │
├─────────────────────────────┤
│     ↑        │
│   Heap         │ (grows upward)
├─────────────────────────────┤
│ BSS segment    │ (uninitialized data)
├─────────────────────────────┤
│ Data segment   │ (initialized data)
├─────────────────────────────┤
│ Text segment   │ (program code)
└─────────────────────────────┘
Low addresses
```

1.
2. **Load program segments**:
    - **Text segment**: Load executable code (read-only)
    - **Data segment**: Load initialized variables
    - **BSS segment**: Allocate space for uninitialized variables
3. **Map shared libraries**:
    - Load dynamic linker (`ld-linux.so`)
    - Dynamic linker will load shared libraries later
    - Examples: `libc.so` (standard C library), `libpython3.9.so`

Step 5: Stack Setup

The kernel creates a **new stack** and carefully arranges data for the program:

Top of Stack (high address)

```
┌─────────────────────────────────────┐
│ Environment strings                 │
│ "PATH=/usr/bin:/bin\0"              │
│ "HOME=/home/user\0"                 │
│ ...                                 │
├─────────────────────────────────────┤
│ Argument strings                    │
│ "python3\0"                         │
│ "main.py\0"                         │
│ "--verbose\0"                       │
├─────────────────────────────────────┤
│ Auxiliary vector (AT_* values)      │
│ - Random value for ASLR             │
│ - Entry point address               │
│ - Program headers address           │
├─────────────────────────────────────┤
│ NULL                                │
├─────────────────────────────────────┤
│ envp[N]   (pointers)                │
│ ...                                 │
│ envp[1]                             │
│ envp[0]                             │
├─────────────────────────────────────┤
│ NULL                                │
├─────────────────────────────────────┤
│ argv[N]   (pointers)                │
│ ...                                 │
│ argv[2] → "--verbose"               │
│ argv[1] → "main.py"                 │
│ argv[0] → "python3"                 │
├─────────────────────────────────────┤
│ argc (number of arguments)          │
│ 3                                   │
└─────────────────────────────────────┘
```

Bottom of stack setup (stack pointer)

This stack layout is mandated by the System V ABI (Application Binary Interface).

Step 6: CPU State Initialization

The kernel sets up CPU registers:

x86-64 Architecture:

```
┌─────────┬──────────────────────────────────┐
│ Register │ Value                    │
├─────────┼──────────────────────────────────┤
│ %rip     │ Entry point address         │ (Instruction pointer)
│ %rsp     │ Top of stack               │ (Stack pointer)
│ %rax     │ 0                      │
│ %rbx     │ 0                      │
│ %rcx     │ 0                      │
│ %rdx     │ 0                      │
│ Other    │ Cleared to 0               │
└─────────┴──────────────────────────────────┘
```

Entry point: Where execution begins

- For ELF: Usually the dynamic linker (`ld-linux.so`)
- Dynamic linker then loads libraries and jumps to program's `main()`

Step 7: Return to User Space

The kernel **never returns** from `execve()` in the traditional sense:

```
// In the CHILD process:
execve("/usr/bin/python3", argv, envp);
// ↑ This point is NEVER reached if execve succeeds!
// The process is now Python - all shell code is GONE

// If execve fails (e.g., file not found):
perror("execve failed");  // This WOULD execute
```

If successful: CPU starts executing at the new program's entry point **If failed**: Returns `-1` and sets `errno`

Complete Example: Shell Executing `python main.py`

Let's trace **every single step** when the shell runs your command:

In the Shell Process (Parent)

```
// Shell code (simplified)
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main() {
    char input[256];

    // Shell prompt
    printf("$ ");
    fgets(input, sizeof(input), stdin);
    // User types: python main.py

    // Parse the input
    // tokens = ["python", "main.py"]

    // Find "python" in PATH
    char *program = find_in_path("python");
    // Returns: "/usr/bin/python3"

    // Prepare arguments
    char *argv[] = {
        "python",       // argv[0] - program name
        "main.py",      // argv[1] - script to run
        NULL            // terminator
    };

    // Get current environment
    extern char **environ;  // Global variable with env vars

    // Create child process
    pid_t pid = fork();

    if (pid == 0) {
        // ========== CHILD PROCESS ==========

        // Replace this process with Python
        execve("/usr/bin/python3", argv, environ);

        // If we reach here, execve failed
        perror("execve failed");
        exit(1);
    }
```

```
    // ========== PARENT PROCESS (Shell) ==========

    // Wait for child to complete
    int status;
    waitpid(pid, &status, 0);

    // Child is done, show prompt again
    printf("$ ");

    return 0;
}
```

What Happens in the Kernel

Let's trace `execve("/usr/bin/python3", argv, environ)`:

Phase 1: System Call Entry

```
// CPU switches to kernel mode
// Registers saved
// Kernel code begins executing

sys_execve() {
    // Validate parameters
    if (!valid_pointer(pathname)) return -EFAULT;
    if (!valid_pointer(argv)) return -EFAULT;
    if (!valid_pointer(envp)) return -EFAULT;

    // Copy strings from user space to kernel space
    // (kernel can't trust user memory)
    char *kpathname = copy_from_user(pathname);
    char **kargv = copy_argv_from_user(argv);
    char **kenvp = copy_envp_from_user(envp);
}
```

Phase 2: File Resolution

```
// Open the executable file
struct file *file = open_exec(kpathname);
// file = handle to /usr/bin/python3

// Check permissions
if (!file_has_execute_permission(file))
```

```
    return -EACCES;

// Read ELF header
struct elf_header hdr;
read_elf_header(file, &hdr);

// Verify it's a valid ELF file
if (hdr.magic != ELF_MAGIC)
    return -ENOEXEC;
```

Phase 3: Memory Destruction

```
// Point of no return - destroy current memory
struct mm_struct *old_mm = current->mm;

// Unmap all memory regions
unmap_all_regions(old_mm);

// Close file descriptors marked FD_CLOEXEC
close_cloexec_fds();

// Reset signal handlers
reset_signal_handlers();

// Create new memory space
struct mm_struct *new_mm = mm_alloc();
current->mm = new_mm;
```

Phase 4: Program Loading

```
// Parse ELF program headers
for each segment in elf_file {
    if (segment.type == PT_LOAD) {
        // Map segment into memory
        mmap_segment(segment.virtual_addr,
                    segment.file_offset,
                    segment.size,
                    segment.permissions);
    }
}

// Typical segments:
// 1. Text segment:   0x00400000 - 0x00500000 (r-x)
// 2. Data segment:   0x00600000 - 0x00601000 (rw-)
```

```
// 3. BSS segment:    0x00601000 - 0x00602000 (rw-)

// Load dynamic linker if needed
if (elf_has_interpreter) {
    char *interp = "/lib64/ld-linux-x86-64.so.2";
    load_elf(interp);
    entry_point = interp_entry_point;
} else {
    entry_point = elf_entry_point;
}
```

Phase 5: Stack Construction

```
// Allocate stack (typically 8MB)
void *stack_base = mmap(NULL, STACK_SIZE,
                        PROT_READ | PROT_WRITE,
                        MAP_PRIVATE | MAP_ANONYMOUS,
                        -1, 0);

char *sp = stack_base + STACK_SIZE;  // Stack pointer

// Push environment strings
for (int i = 0; envp[i] != NULL; i++) {
    sp -= strlen(envp[i]) + 1;
    strcpy(sp, envp[i]);
    env_pointers[i] = sp;
}

// Push argument strings
for (int i = 0; argv[i] != NULL; i++) {
    sp -= strlen(argv[i]) + 1;
    strcpy(sp, argv[i]);
    arg_pointers[i] = sp;
}

// Align stack pointer (16-byte boundary on x86-64)
sp = ALIGN_DOWN(sp, 16);

// Push auxiliary vector (random values, etc.)
push_auxv(sp);

// Push NULL terminator for envp
push_ptr(sp, NULL);
```

```
// Push environment pointers
for (int i = envc - 1; i >= 0; i--) {
    push_ptr(sp, env_pointers[i]);
}

// Push NULL terminator for argv
push_ptr(sp, NULL);

// Push argument pointers
for (int i = argc - 1; i >= 0; i--) {
    push_ptr(sp, arg_pointers[i]);
}

// Push argument count
push_long(sp, argc);

// Stack is now ready!
```

Phase 6: CPU State Setup and Launch

```
// Setup registers
struct pt_regs *regs = task_pt_regs(current);

regs->rip = entry_point;        // Instruction pointer
regs->rsp = sp;                 // Stack pointer
regs->rax = 0;
regs->rbx = 0;
regs->rcx = 0;
regs->rdx = 0;
// ... clear other registers

// Clear CPU flags
regs->eflags = 0x200;  // Interrupt enable flag

// Return to user space
// CPU will begin executing at 'entry_point' with stack 'sp'
return_to_userspace(regs);
```

The New Program Starts: Python Initialization

Now we're in the Python interpreter:

```c
// Python's main() function (in C)
int main(int argc, char **argv) {
    // argc = 2
    // argv[0] = "python"
    // argv[1] = "main.py"

    // Initialize Python interpreter
    Py_Initialize();

    // Set sys.argv from C argv
    PySys_SetArgvEx(argc, argv, 1);

    // Open and compile the script
    FILE *fp = fopen(argv[1], "r");
    PyRun_SimpleFile(fp, argv[1]);

    // Cleanup
    fclose(fp);
    Py_Finalize();

    return 0;
}
```

Inside `PyRun_SimpleFile()`:

```c
int PyRun_SimpleFile(FILE *fp, const char *filename) {
    // Read the file content
    char *source = read_file(fp);

    // Parse source → AST
    PyArena *arena = PyArena_New();
    mod_ty ast = PyParser_ASTFromString(source, filename,
                                        Py_file_input,
                                        NULL, arena);

    // Compile AST → bytecode
    PyCodeObject *code = PyAST_CompileObject(ast, filename,
                                             NULL, -1, arena);

    // Execute bytecode
    PyObject *result = PyEval_EvalCode(code, globals, locals);

    // Your Python code is now running!
```

```
    return result == NULL ? -1 : 0;
}
```

Why This Design? The `fork()` + `exec()` Pattern

You might wonder: Why not have a single "run this program" system call?

Historical reasons:

1. **Unix philosophy**: Small, composable tools
    - `fork()` does ONE thing: duplicate process
    - `exec()` does ONE thing: replace process
    - Combined, they're very powerful

Flexibility:

```
pid = fork();
if (pid == 0) {
    // Child process

    // Can modify environment before exec
    setenv("MY_VAR", "value", 1);

    // Can redirect I/O
    int fd = open("output.txt", O_WRONLY);
    dup2(fd, STDOUT_FILENO);  // stdout → file

    // Can change directory
    chdir("/tmp");

    // Can drop privileges
    setuid(1000);

    // NOW exec the program
    execve("/usr/bin/python3", argv, environ);
}
```

2.

Shell pipelines:

cat file.txt | grep "pattern" | sort

3. The shell:
    - Forks 3 times (for cat, grep, sort)
    - Sets up pipes between them
    - Execs each program

Other `exec()` Variants

`execv()` - Vector of Arguments

```
char *argv[] = {"python", "main.py", NULL};
execv("/usr/bin/python3", argv);
```

Same as `execve()` but uses current environment.

`execl()` - List of Arguments

```
execl("/usr/bin/python3", "python", "main.py", NULL);
//    ^                    ^         ^          ^
//    pathname             argv[0]   argv[1]    terminator
```

Pass arguments as separate parameters (must end with NULL).

`execvp()` - Search PATH

```
char *argv[] = {"python", "main.py", NULL};
execvp("python", argv);
//     ^
//     Searches PATH for "python"
```

No need to specify full path!

`execlp()` - Search PATH + List

```
execlp("python", "python", "main.py", NULL);
//     ^         ^         ^          ^
//     search    argv[0]   argv[1]    terminator
```

Combines PATH search with list-style arguments.

`execle()` - List + Environment

```
char *envp[] = {"PATH=/usr/bin", "HOME=/home/user", NULL};
execle("/usr/bin/python3", "python", "main.py", NULL, envp);
//                                             ^     ^
//                                             term  env
```

Pass custom environment variables.

Error Handling with `exec()`

```
pid_t pid = fork();

if (pid == 0) {
    // Child process
    execve("/usr/bin/python3", argv, envp);

    // Only reaches here if execve FAILS
    switch (errno) {
        case ENOENT:
            fprintf(stderr, "File not found\n");
            break;
        case EACCES:
            fprintf(stderr, "Permission denied\n");
            break;
        case ENOEXEC:
            fprintf(stderr, "Not an executable\n");
            break;
        default:
            perror("execve");
    }

    exit(127);  // Convention: exit with 127 on exec failure
}

// Parent process
int status;
waitpid(pid, &status, 0);
```

```
if (WIFEXITED(status)) {
    int exit_code = WEXITSTATUS(status);
    if (exit_code == 127) {
        printf("Failed to execute command\n");
    }
}
```

Summary: The Complete Picture

When you type `python main.py`:

1. **Shell parses** the command
2. **Shell searches PATH** for `python` executable
3. **Shell calls `fork()`** → Creates child process
4. **Child calls `execve("/usr/bin/python3", ["python", "main.py"], environ)`**
5. **Kernel validates** parameters and permissions
6. **Kernel destroys** child's memory
7. **Kernel loads** Python executable into memory
8. **Kernel sets up** new stack with arguments and environment
9. **Kernel initializes** CPU registers
10. **CPU jumps** to Python's entry point
11. **Python initializes** its interpreter
12. **Python compiles** `main.py` to bytecode
13. **Python executes** your code
14. **Python exits**, returning status to shell
15. **Shell displays** prompt again

The `exec()` call is the **critical transformation point** where one program becomes another - it's the heart of process execution in Unix/Linux systems!