Everything about PATH variable, PYTHONPATH, symlinks, python installation and version management on macOS

System Python: macOS comes with Python pre-installed, but it's typically an older version (often 3.8 or 3.9) located at `/usr/bin/python3`. You should generally avoid modifying this system Python as it's used by macOS itself.

User-installed Python: You'll want to install and manage your own Python versions for development work.

PYTHON INSTALLATION METHODS ON MACOS

1. Homebrew (Recommended for simplicity)

```
# Install latest Python 3

brew install python


# Install specific version

brew install python@3.11

brew install python@3.12


# List available versions

brew search python
```

Details:

- **Location**: `/opt/homebrew/opt/python@3.x/` (Apple Silicon) or `/usr/local/opt/python@3.x/` (Intel)
- **Symlinks**: `/opt/homebrew/bin/python3.x`

- **PATH:** `/opt/homebrew/bin` added to PATH

2. Official Python.org Installer

   Installation:

   # Download from https://www.python.org/downloads/
   # Run the .pkg installer

   Details:

- **Location**: /usr/local/bin/python3.x and /Applications/Python 3.x/
- **PATH**: Usually adds itself to PATH
- **Versions**: Latest stable releases

3. pyenv (Recommended for version management)

```
# Install pyenv

brew install pyenv


# Add to your shell profile (.zshrc or .bash_profile)

echo 'export PATH="$HOME/.pyenv/bin:$PATH"' >> ~/.zshrc

echo 'eval "$(pyenv init --path)"' >> ~/.zshrc

echo 'eval "$(pyenv init -)"' >> ~/.zshrc
```

```
# Install Python versions
pyenv install 3.11.8
pyenv install 3.12.2
pyenv install 3.13.0

# List available versions to install
pyenv install --list
```

Details:

- **Location**: `~/.pyenv/versions/`
- **Shims**: Uses `~/.pyenv/shims` in PATH
- **Version switching**: Global, local, or shell-specific

Understanding `python -m pip`

The `-m` flag tells Python to run a module as a script. `python -m pip` means:

- Run pip as a module using the specific Python interpreter
- Ensures pip installs packages for that exact Python version
- More reliable than calling `pip` directly

```
# These can be different!
pip install package          # Might use different Python version
python3 -m pip install package  # Uses the specific python3 version
```

```
# Even more specific

python3.11 -m pip install package

/opt/homebrew/bin/python3.12 -m pip install package
```

Best Practices

1. Virtual Environments

```
Always use virtual environments for projects:

# Create virtual environment with specific Python

python3.11 -m venv myproject_env

source myproject_env/bin/activate
```

2. Check your environment

```
# Verify your setup

which python3

python3 --version

which pip3

pip3 --version

python3 -m pip --version
```

3. Upgrade pip safely

```
# Upgrade pip for specific Python version

python3.11 -m pip install --upgrade pip

python3.12 -m pip install --upgrade pip
```

What is the PATH variable?

The PATH variable is one of the most important environment variables in Unix-like systems (macOS, Linux). It tells your shell where to look for executable programs when you type a command.

What PATH Does?

When you type a command like python3, your shell doesn't magically know where to find it. Instead, it:

1. Splits PATH into individual directories
2. Searches each directory in order from left to right
3. Runs the first match it finds
4. Stops searching once it finds a match

```
# See your current PATH

echo $PATH

# Make it more readable (one directory per line)

echo $PATH | tr ':' '\n'
```

How PATH Works

Example: What happens when you type python3

Ans - If your PATH is this -

```
# /Users/machbluex/opt/anaconda3/bin

# /opt/homebrew/bin

# /opt/homebrew/sbin

# /usr/local/bin

# /usr/bin

# /bin

# /usr/sbin

# /sbin
# Shell searches in this order:
# 1. /Users/machbluex/opt/anaconda3/bin/python3 ← Found! Uses this one
# 2. /opt/homebrew/bin/python3                  ← Skipped
# 3. /usr/local/bin/python3                     ← Skipped
# 4. /usr/bin/python3                           ← Skipped
# etc.
```

Finding which executable is used:

```
# See which python3 will be executed
which python3
/Users/machbluex/opt/anaconda3/bin/python3

# See all python3 executables in PATH
which -a python3
# Shows all matches in PATH order
/Users/machbluex/opt/anaconda3/bin/python3
/opt/homebrew/bin/python3
/Library/Frameworks/Python.framework/Versions/3.10/bin/python3
/usr/local/bin/python3
/usr/bin/python3
```

Why PATH Order Matters

First match wins! This is why you might get unexpected behavior:

```
# Your PATH might look like:
# /Users/machbluex/opt/anaconda3/bin:/opt/homebrew/bin:/usr/bin


# So typing python3 uses Anaconda's version
python3 --version
```

```
# Python 3.11.5 (from Anaconda)


# Even though you have Homebrew's python3 installed too

/opt/homebrew/bin/python3 --version

# Python 3.13.0 (from Homebrew)

# If Anaconda comes first:

# /Users/machbluex/opt/anaconda3/bin ← python3 from here

# /opt/homebrew/bin ← python3 ignored

# To temporarily use Homebrew's python: /opt/homebrew/bin/python3

# To permanently change priority, modify ~/.zshrc
```

Modifying PATH

Temporary changes (current session only):

```
# Add to beginning (highest priority)

export PATH="/new/directory:$PATH"


# Add to end (lowest priority)

export PATH="$PATH:/new/directory"


# Remove a directory

export PATH=$(echo $PATH | tr ':' '\n' | grep -v '/unwanted/path' | tr '\n' ':')
```

Permanent changes:

Add to your shell configuration file (~/.zshrc for zsh, ~/.bash_profile for bash):

```
# Edit your shell config
nano ~/.zshrc

# Add lines like:
export PATH="/opt/homebrew/bin:$PATH"
export PATH="/Users/machbluex/.local/bin:$PATH"

# Reload your config
source ~/.zshrc
```

SYMLINKS

What is the difference between

```
 file /opt/homebrew/bin/python3.11

*# python3.11: Mach-O 64-bit executable arm64*

file /opt/homebrew/opt/python@3.11/bin/python3.11

*# Similar output*?
```

The Two Locations Explained

/opt/homebrew/bin/python3.11

***This is a symlink** - a shortcut that points to the actual installation*

/opt/homebrew/opt/python@3.11/bin/python3.11

***This is the actual installation** - where the real Python binary lives*

~~~
## Homebrew's Organization Structure

```

/opt/homebrew/

├── bin/ # Symlinks to executables (what's in your PATH)

 | ├── python3.11 -> ../opt/python@3.11/bin/python3.11

| ├── python3.12 -> ../opt/python@3.12/bin/python3.12

| └── python3.13 -> ../opt/python@3.13/bin/python3.13

| └── opt/ # Actual installations

├── python@3.11/

| └── bin/python3.11 # Real binary here

├── python@3.12/

| └── bin/python3.12 # Real binary here

└── python@3.13/

└── bin/python3.13 # Real binary here
~~~

What is a symlink?

A symlink (symbolic link) is essentially a shortcut or pointer to another file or directory. It's like an alias that redirects to the actual location of a file.

Think of it like this:

- *Regular file: Contains actual data*
- *Symlink: Contains a path that points to another file*
- *When you access the symlink, the system automatically follows the pointer to the real file*

```
Real file:    /opt/homebrew/opt/python@3.11/bin/python3.11  [actual binary data]
                              ↑
Symlink:      /opt/homebrew/bin/python3.11 ---------------→ points here
```

How to Identify Symlinks?

Using `ls -la`

```
ls -la /opt/homebrew/bin/python3.11

# Output shows:
# lrwxr-xr-x ... python3.11 -> ../opt/python@3.11/bin/python3.11
# ↑                            ↑
# l = symlink                  arrow shows target
```

What is the PYTHONPATH variable?

PYTHONPATH is an environment variable that tells Python where to look for modules and packages when you import them. It extends Python's default module search path.

What PYTHONPATH Does

When you do `import something` in Python, Python searches for modules in this order:

1. Current directory (where your script is)
2. PYTHONPATH directories (if set)
3. Standard library (built-in Python modules)
4. Site-packages (installed packages via pip)

```
# See Python's complete search path
python3 -c "import sys; print('\n'.join(sys.path))"

# Example output:
# /Users/username/current/directory
# /opt/homebrew/lib/python3.13/site-packages
# /opt/homebrew/lib/python3.13
# /opt/homebrew/lib/python3.13/lib-dynload
# ...
```

How does a virtual environment work?

A virtual environment is essentially a isolated copy of Python with its own set of packages. Here's how it works under the hood:

A virtual environment creates a separate directory structure that contains:

- *A Python interpreter (or link to one)*
- *Its own `site-packages` directory for installed packages*
- *Scripts to modify your environment when activated*

# Create a virtual environment
python3 -m venv myenv

# Let's see what was created
ls -la myenv/
*```*

### Directory Structure Created:
*```*

```
myenv/
├── bin/                # Executables (scripts, python, pip)
│   ├── activate          # Activation script
```

```
│   ├── activate.fish      # Fish shell activation
│   ├── pip                # Virtual env's pip
│   ├── pip3               # Symlink to pip
│   ├── python             # Symlink to python3
│   └── python3            # Symlink to system python3
├── include/               # C headers for compiling extensions
├── lib/
│   └── python3.13/
│       └── site-packages/ # Where packages get installed
├── pyvenv.cfg             # Configuration file
└── share/                 # Shared data
```

```
# Check what python3 executable the venv uses
ls -la myenv/bin/python3

# It's usually a symlink to your system Python
# myenv/bin/python3 -> /opt/homebrew/bin/python3.13

# Virtual env has its own package directory
ls myenv/lib/python3.13/site-packages/

# Initially almost empty (just pip and setuptools)
# pip                # Package installer
# setuptools         # Package building tools
# _distutils_hack    # Internal tools
```