Programming for Automation

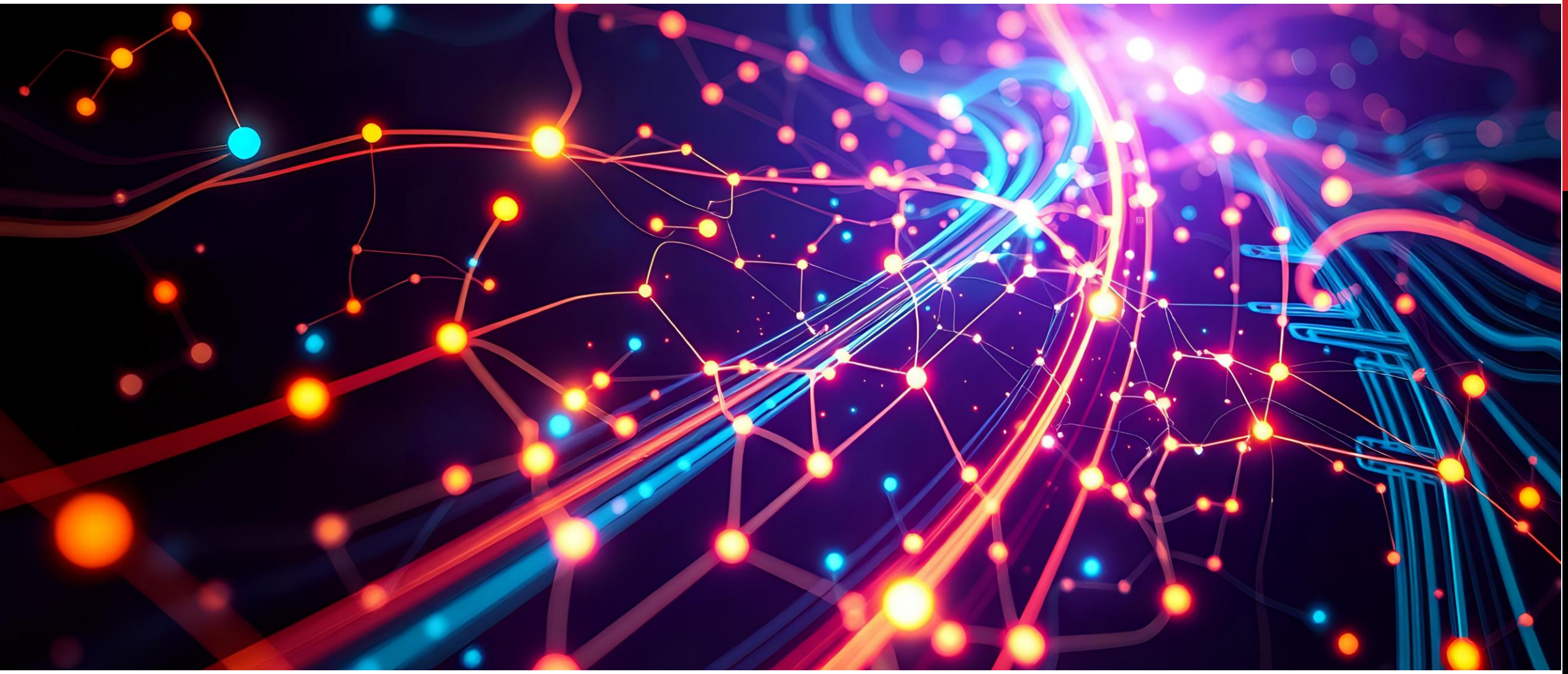
TEKsystems Global Services

WORKFORCE DEVELOPMENT

Python

Develop a passion for learning.

Networking



Programming for Automation

Automation is the act of writing code that performs repeatable tasks consistently, accurately, and at scale without human intervention.

Instead of clicking, typing, or configuring systems manually, we **encode intent into logic** and let software execute it.

Automation focuses on:
- Reducing human error
- Increasing speed and consistency
- Enforcing standards and best practices
- Freeing engineers to solve higher value problems

At its core, automation turns **process knowledge into executable code**.

What is Python?

Python is a **high-level programming language** designed to be easy to read, write, and understand.

It is commonly used for automation, scripting, data processing, web services, and tooling.

Python focuses on **clarity and simplicity**, making it accessible even for first-time programmers.

Python is interpreted

Python is an **interpreted** language, not a compiled one. This means Python:

- Reads code **line by line**
- Executes instructions immediately
- Does not require a build step before running

This makes Python fast to experiment with and ideal for automation and scripting. It runs slower than a compiled language (Java/C++), but faster to develop with!

Dynamically Typed

In Python, you do not need to declare variable types.
Python determines types **at runtime**, based on the value being assigned.

This reduces boilerplate and allows you to focus on **logic instead of syntax**. It also adds ambiguity as to the true data type of a variable and can require the programmer to perform checks.

Python for Network Automation

Why Python for Network Automation?

Python is one of the most effective languages for automation because it balances **power, simplicity, and ecosystem support**.

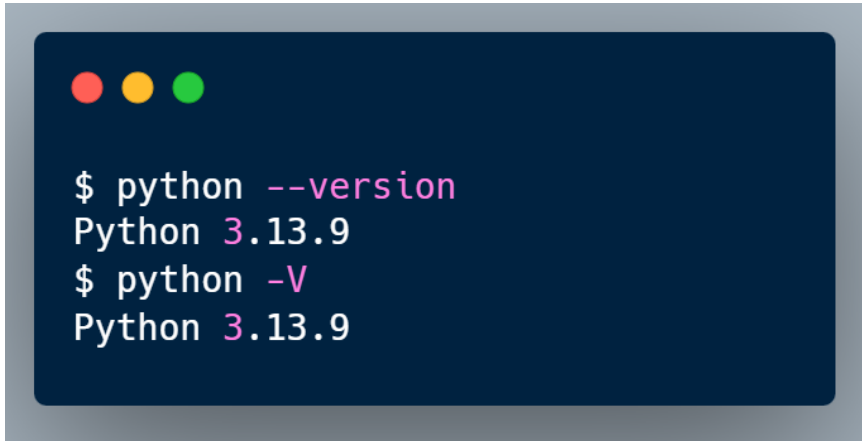
- Easy to read and write, even for non developers
- Massive ecosystem of networking libraries and SDKs
- Excellent support for APIs, SSH, CLI, and data formats
- Works across vendors and platforms
- Ideal for scripting, tooling, and orchestration

Python lets engineers focus on **what they want done**, not language complexity.

Common Automation Tasks with Python

- Device configuration and validation
- Network inventory and discovery
- Backup and restore of configs
- Compliance checks and drift detection
- API driven infrastructure changes
- Alerting, remediation, and reporting

Python Version



Python can be checked from any terminal environment:

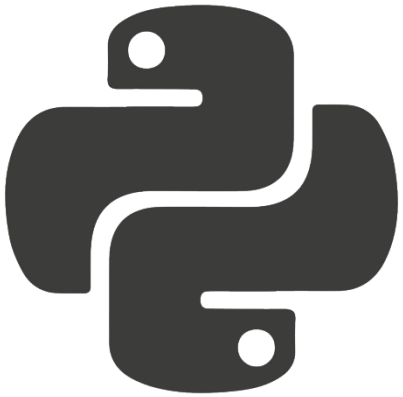
- Linux shell (bash)
- macOS Terminal
- Windows PowerShell

Use the terminal to verify whether Python is already available on your system.

If Python is installed, a version number will be displayed.

If Python is **not** installed, download it from the official source: **python.org/downloads**

Python



The current **stable Python version** is **3.14.2**.You may already have a different Python 3 version installed on your system. That is completely fine, and you may update to the latest stable release if you choose.

Version Compatibility For automation and scripting:

- Most Python 3 versions behave the same
- Major breaking changes are rare
- New language features are usually optional
- Common automation code works across versions

If you encounter missing modules or library issues, they are typically resolved using **pip** to install dependencies.

Python REPL

To use Python, you can simply type **python** (or **python3**) in your shell and press Enter. This opens the **REPL**, which stands for **Read, Evaluate, Print, Loop**.

The REPL is an interactive Python environment that executes code immediately as you type it.

- Test small code snippets quickly
- Experiment without creating files
- Learn Python syntax interactively
- Validate logic before automating

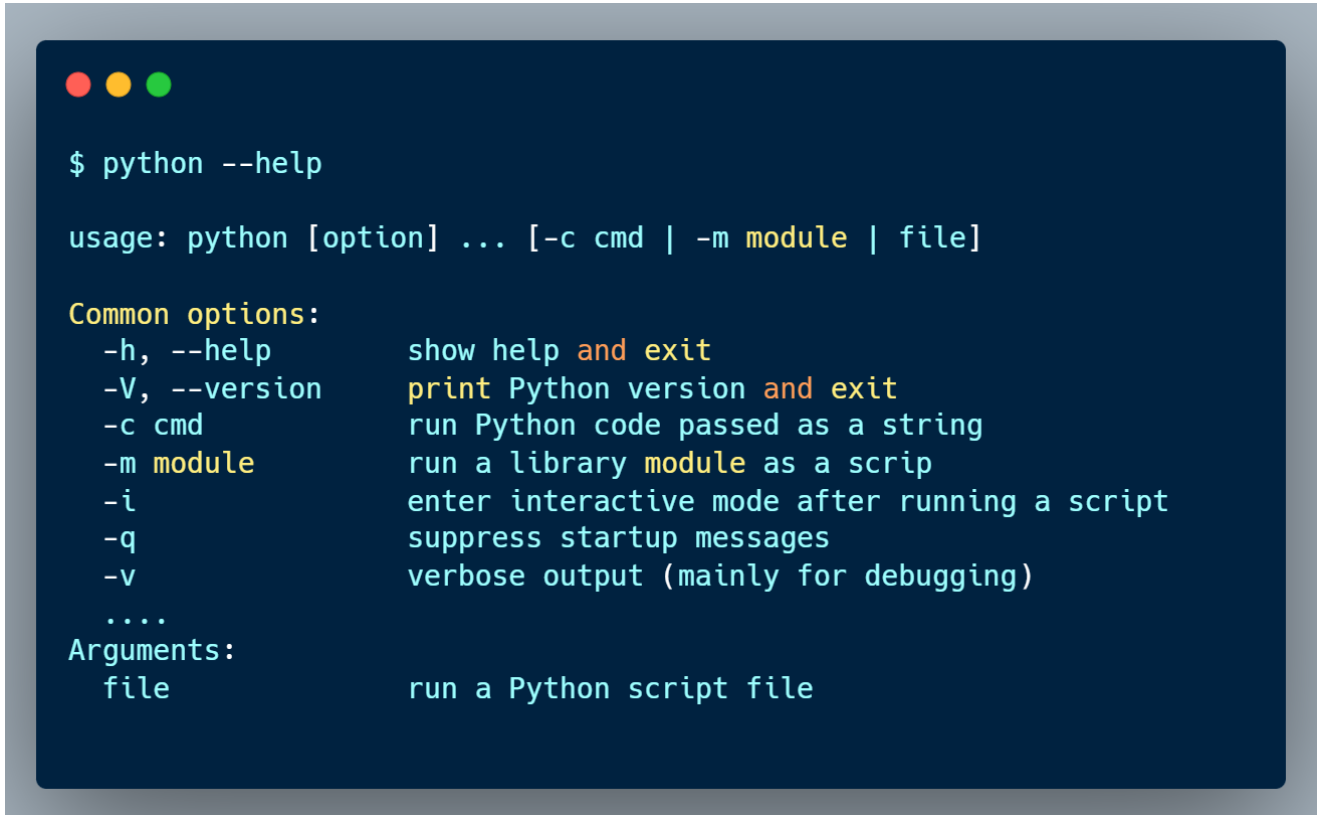
The REPL is one of the fastest ways to explore and understand Python.

Python - Help

Running **python --help** prints the built-in help page for the Python command.

This output shows that Python can be started in **multiple ways**, not just by opening the REPL or running a script.

- Run Python interactively
- Execute a script file
- Run a module as a program
- Pass code directly on the command line



Python – Hello World

Run Python File

```
$ echo "print('hello world')" > hello.py
$ python hello.py
hello world
```

Run Python Code (-c)

```
$ python -c "print('hello world')"
hello world
```

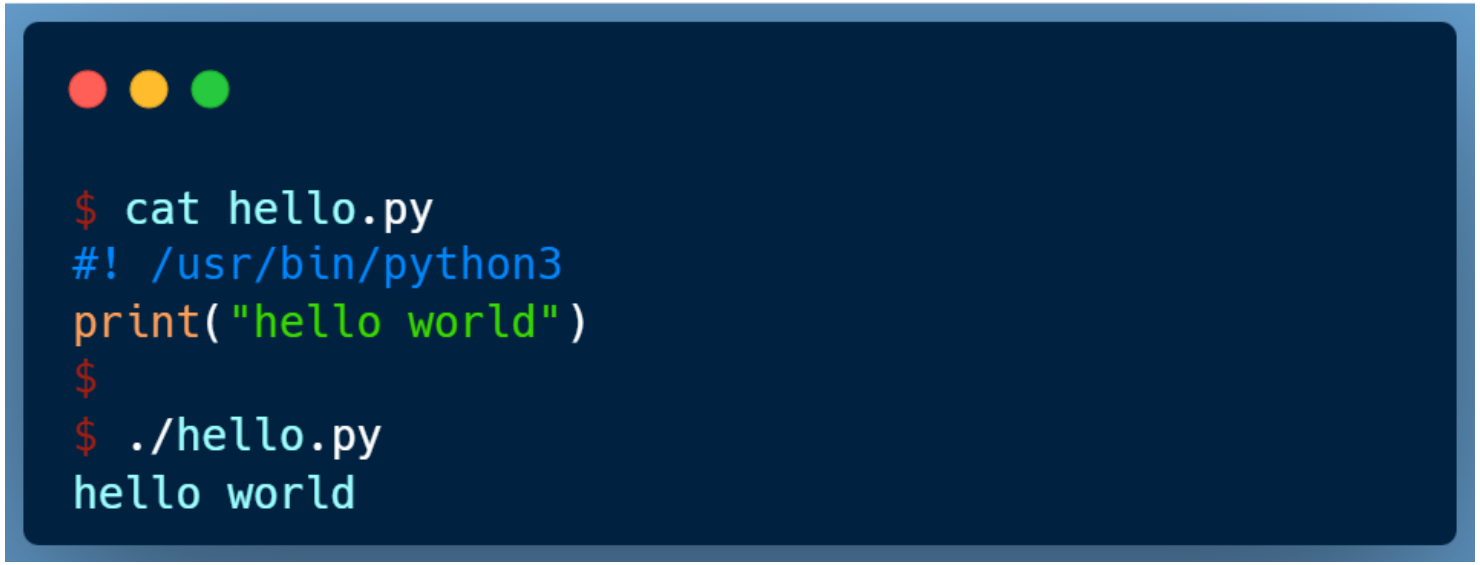
Run Python in REPL

```
$ python
> msg = "hello world"
> msg
hello world
```

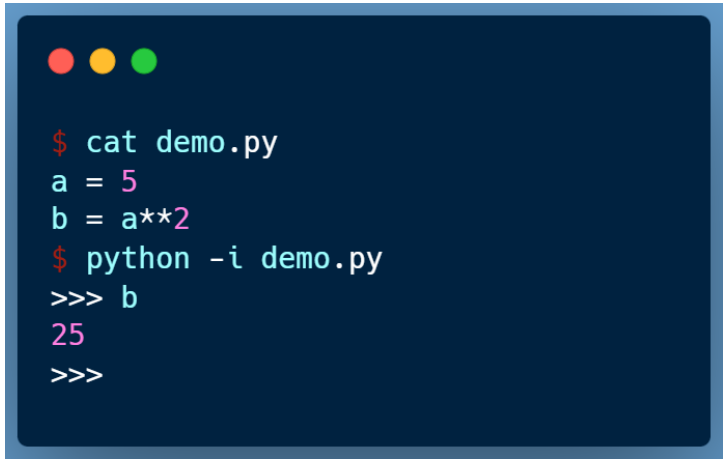
Shebang

You can add a **shebang** to the top of a Python file to make it directly executable. The shebang tells the operating system **which interpreter** should be used to run the file.

This is especially useful when pointing to a specific Python interpreter, such as one inside a virtual environment.
With a shebang in place, the script can be executed like a regular command instead of calling python explicitly.



Running a script interactively



Python can run a script and then drop you into an interactive session using the **-i** flag.

This allows you to inspect variables and experiment after the script has executed.

In this example, the script runs first, then the REPL starts with all variables still in memory.

This is useful for debugging, learning, and exploring program state.

Python help() function

```
$ python
>>> from sys import getsizeof
>>> help(getsizeof)
Help on built-in function getsizeof in module sys:

getsizeof(...)
    getsizeof(object [, default]) -> int

    Return the size of object in bytes.

>>> def test():
...     """ This is a DocString"""
...     return None
...
>>> help(test)
Help on function test in module __main__:

test()
    This is a DocString
```

Python includes a built-in **help()** function that displays documentation for objects. The help output is generated from the object's **docstring**, which means the object must be **imported first**.

- Works on modules, classes, and functions
- Can be run on a class or on an instance
- Useful for learning APIs without leaving Python

The help() function is a simple way to explore Python while you work, without memorizing everything upfront.

Python - pip

pip is the package manager for Python.
It is typically installed automatically with Python, though on some Linux systems it may need to be installed separately.

- Installs Python modules from **PyPI**
- Resolves dependency requirements automatically
- Installs the latest supported versions by default
- Allows version pinning for consistency
- Supports backing up and restoring dependencies

pip makes it easy to manage the libraries your automation code depends on.

```
$ pip install requests
Collecting requests
  Using cached requests-2.32.5-py3-none-any.whl.metadata
(4.9 kB)
Collecting charset_normalizer<4,>=2 (from requests)
  Using cached charset_normalizer-3.4.4-cp313-cp313-
win_amd64.whl.metadata (38 kB)
Collecting idna<4,>=2.5 (from requests)
  Using cached idna-3.11-py3-none-any.whl.metadata (8.4
kB)
Collecting urllib3<3,>=1.21.1 (from requests)
  Using cached urllib3-2.6.2-py3-none-any.whl.metadata
(6.6 kB)
Collecting certifi>=2017.4.17 (from requests)
  Using cached certifi-2025.11.12-py3-none-
any.whl.metadata (2.5 kB)
Using cached requests-2.32.5-py3-none-any.whl (64 kB)
Using cached charset_normalizer-3.4.4-cp313-cp313-
win_amd64.whl (107 kB)
Using cached idna-3.11-py3-none-any.whl (71 kB)
Using cached urllib3-2.6.2-py3-none-any.whl (131 kB)
Using cached certifi-2025.11.12-py3-none-any.whl (159
kB)
Installing collected packages: urllib3, idna,
charset_normalizer, certifi, requests
Successfully installed certifi-2025.11.12
charset_normalizer-3.4.4 idna-3.11 requests-2.32.5
urllib3-2.6.2

[notice] A new release of pip is available: 25.2 -> 25.3
[notice] To update, run: python.exe -m pip install --
upgrade pip
$
```

Python – pip

In this example, we use **pip freeze** to display all installed Python modules and their versions, then save that output to a file so it can be reused later.

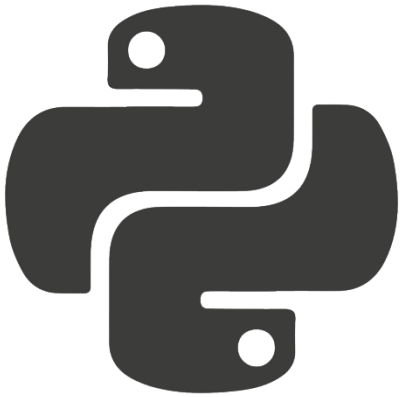
- Shows every installed module and version
- Creates a backup using a requirements file
- Can be shared across systems or environments
- Reinstalls the exact same versions elsewhere

This makes dependency management predictable and repeatable across environments.

```
$ pip freeze
certifi==2025.11.12
charset-normalizer==3.4.4
idna==3.11
requests==2.32.5
urllib3==2.6.2
$
$ pip freeze > requirements.txt
$
$ pip install -r requirements.txt
Requirement already satisfied: certifi==2025.11.12 in
c:\users\blanc\venv\lib\site-packages (from -r
requirements.txt (line 1)) (2025.11.12)
Requirement already satisfied: charset-normalizer==3.4.4
in c:\users\blanc\venv\lib\site-packages (from -r
requirements.txt (line 2)) (3.4.4)
Requirement already satisfied: idna==3.11 in
c:\users\blanc\venv\lib\site-packages (from -r
requirements.txt (line 3)) (3.11)
Requirement already satisfied: requests==2.32.5 in
c:\users\blanc\venv\lib\site-packages (from -r
requirements.txt (line 4)) (2.32.5)
Requirement already satisfied: urllib3==2.6.2 in
c:\users\blanc\venv\lib\site-packages (from -r
requirements.txt (line 5)) (2.6.2)

[notice] A new release of pip is available: 25.2 -> 25.3
[notice] To update, run: python.exe -m pip install --
upgrade pip
$
```

Pydoc

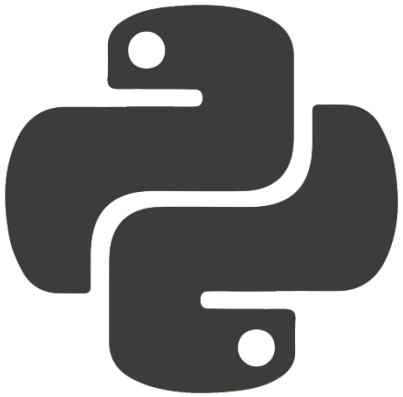


pydoc is a Python documentation tool similar to the **help()** function that reads and displays documentation for Python modules.

- Displays documentation from docstrings
- Works for installed modules and libraries
- Can generate text-based documentation
- Can start a local web server to browse docs

pydoc provides an offline way to explore Python documentation directly from your system.

Python Built-Ins



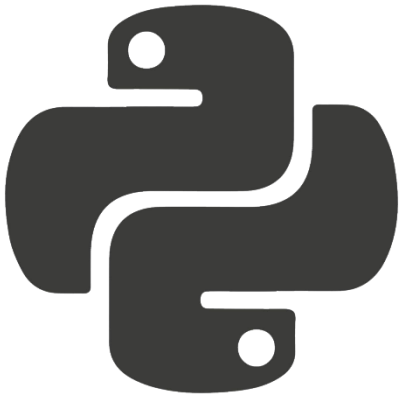
Python includes a set of **built-in functions and objects** that are always available without importing anything.
These built-ins form the foundation of everyday Python programming and are used constantly in automation and scripting.

- Available by default
- No imports required
- Core to reading, writing, and logic

Understanding built-ins makes Python feel simpler and more predictable.

Read the Docs
Built-in Functions — Python 3.14.2 documentation

Python Common Built-Ins

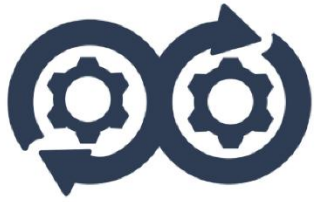


Below are some of the more common Python Built-Ins.

- print()
- len()
- type()
- input()
- int(), str(), float()
- range()
- list(), dict(), set()
- Id(), hash()

Built-ins handle everyday tasks so you can focus on solving real problems instead of reinventing basics.

Lab: Python Basics



POP QUIZ:

What command prints the current version of Python?

A. python
B. python -v
C. python -V
D. pyhon version

POP QUIZ:

What command prints the current version of Python?

A. python
B. python -v
C. **python -V**
D. pyhon version

POP QUIZ:

What Python option will run a module named pydoc?

A. Python module pydoc
B. python run pydoc
C. python pydoc
D. pyhon -m pydoc

POP QUIZ:

What Python option will run a module named pydoc?

A. Python module pydoc
B. python run pydoc
C. python pydoc
D. **pyhon -m pydoc**

POP QUIZ:

What does the shebang allow us to do?

A. Execute the python file directly
B. Run the python script
C. Import the python script
D. Run the python script in interactive mode

POP QUIZ:

What does the shebang allow us to do?

A. **Execute the python file directly**
B. Run the python script
C. Import the python script
D. Run the python script in interactive mode

POP QUIZ:

What tool lets us run Python code interactively?

A. Python CLI
B. Python Shell
C. Python REPL
D. VS Code

POP QUIZ:

What tool lets us run Python code interactively?

A. Python CLI
B. Python Shell
C. **Python REPL**
D. VS Code

Virtual Environments

Virtual Environment

A virtual environment is an isolated Python environment with its own interpreter and dependencies.
It allows different projects to use different libraries and versions without interfering with each other.

Virtual Environment

Virtual environments make Python projects safer, cleaner, and more reproducible

- Keeps project dependencies isolated
- Prevents version conflicts
- Works alongside system Python
- Commonly used with pip and requirements files

Virtual Environments

To create a virtual environment, Python provides the built-in **venv** module. You can name the environment anything, though **venv** or **.venv** are common since they are usually included in .gitignore files.

- Created using the venv module
- Contains an isolated Python interpreter
- Includes setup scripts for activation
- Stores packages installed with pip

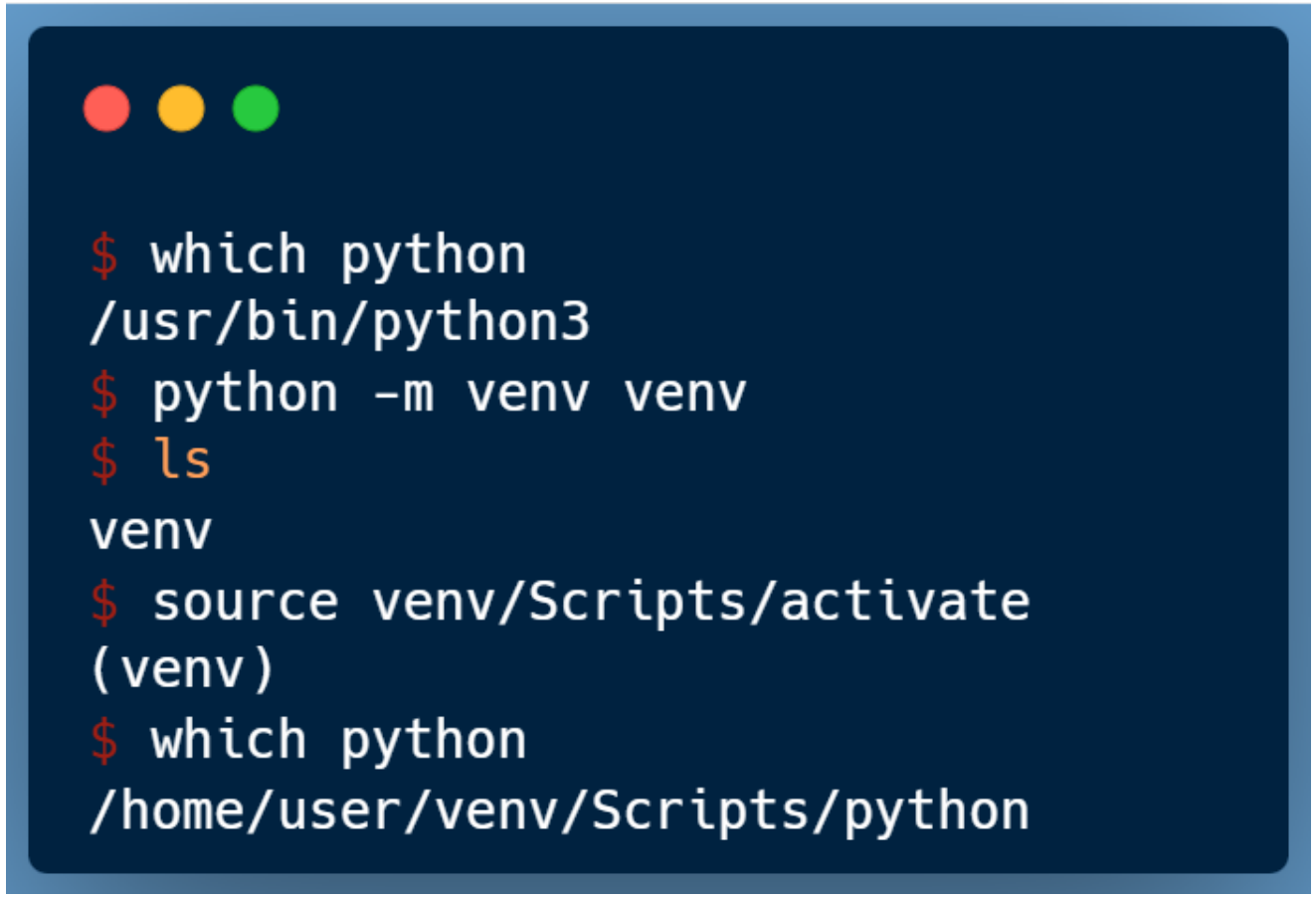
Virtual environments keep project dependencies isolated and reproducible.

Creating and Using Virtual Environment

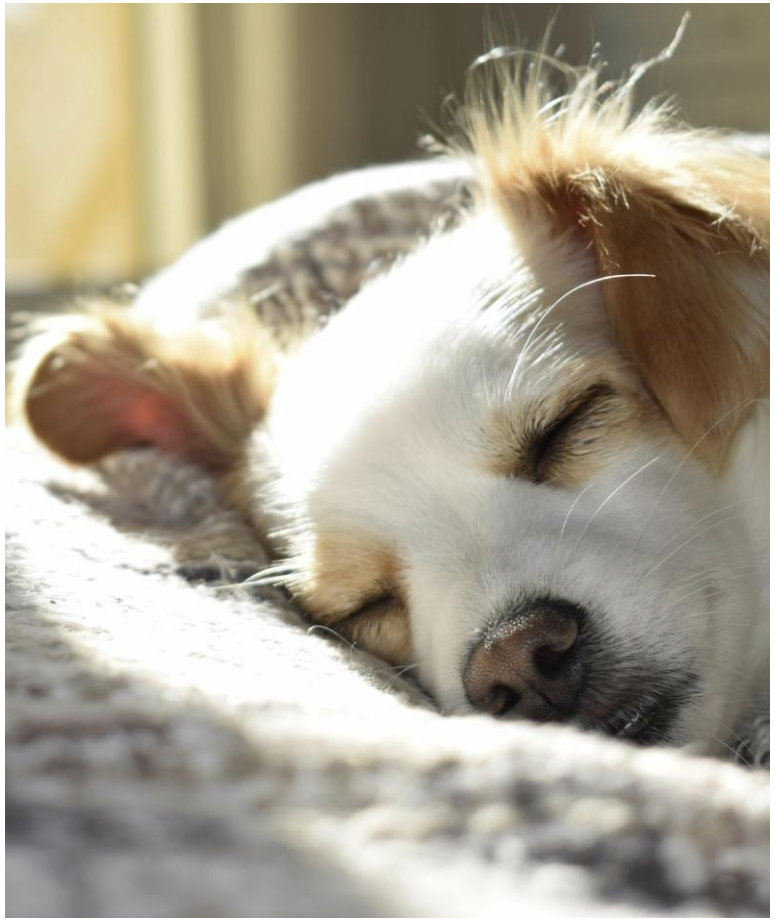
In this example, we create a virtual environment and activate it.
Once activated, the shell uses the Python interpreter inside the virtual environment instead of the system Python.

- python -m venv venv creates the environment
- activate switches the shell to use it
- The prompt shows the active environment
- which python confirms the interpreter changed

Activating a virtual environment ensures your Python commands and packages stay isolated to that project.



Pip and Virtual Environments

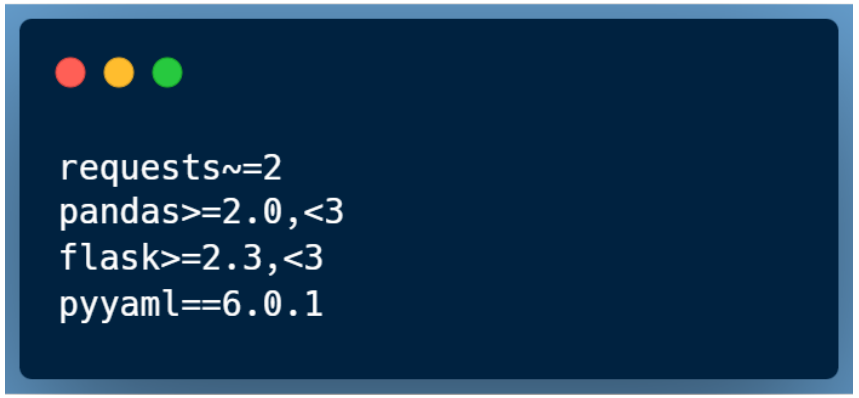


When a virtual environment is active, **pip** installs Python modules into that environment only.

- Each virtual environment has its own pip
- Installed modules do not affect system Python
- Different projects can use different versions
- pip freeze reflects only that environment

Using pip with virtual environments keeps dependencies isolated, controlled, and repeatable.

Version Constraints

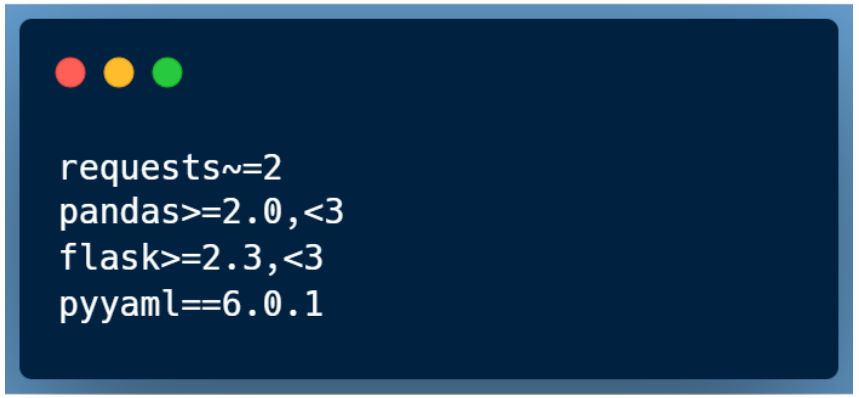


Requirements files allow you to control how strictly Python modules are installed. This is especially useful when sharing environments or supporting multiple Python versions.

By default, **pip freeze records exact versions**, which maximizes reproducibility but can be restrictive.

Version constraints let you decide how much change you are willing to allow.

Version Constraints Example



requests==2.31.0
Exact version pinned (typical output of pip freeze)

requests~=2
Allows any 2.x release, blocks 3.x

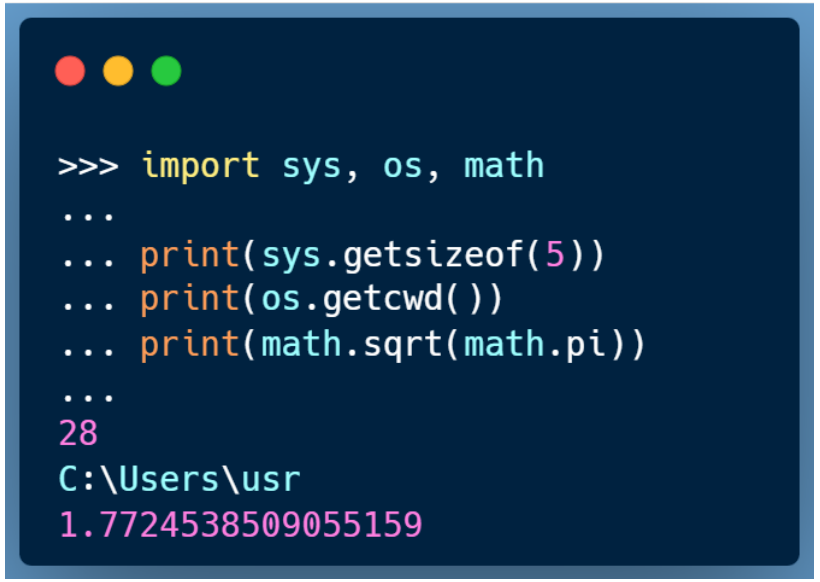
requests>=2,<3
Explicit version range with the same behavior as ~=2

requests>=2,<4
Allows multiple major versions, higher flexibility and risk
These options let you balance reproducibility with long-term compatibility.

requests
Pip will try to find a compatible version with your Python version and other libraries

Python Libraries

Common built-in Libraries



Python includes many **built-in libraries** that ship with Python but still must be explicitly imported before use.
These libraries provide core system and utility functionality.

- **os** – operating system interactions
- **sys** – Python runtime and arguments
- **math** – mathematical functions
- **time** – time and sleep utilities
- **datetime** – dates and timestamps
- **json** – JSON parsing and serialization

Built-in libraries extend Python's capabilities without requiring external dependencies.

Common Library from PyPi (requests)

```
>>> import requests
...
... response = requests.get("https://example.com")
...
... print(response.status_code)
... print(response.text)
...
200
<!doctype html><html lang="en"><head><title>Example
Domain</title>...Learn more</a></div></body></html>
```

requests is a third-party Python library used to make HTTP requests, similar to tools like **curl**. In this example, we perform an HTTP GET request to retrieve data from a website or API.

- Installed using pip
- Supports GET, POST, PUT, DELETE, and more
- Commonly used for API and automation tasks
- Returns a response object with status and data

This pattern is foundational for interacting with web services and APIs in Python.

Common Library from PyPi (bs4)

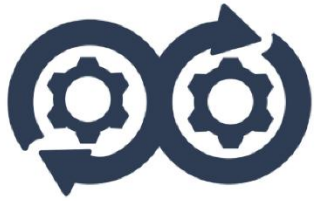
```
>>> import requests
... from bs4 import BeautifulSoup
...
... response = requests.get("https://example.com")
...
... soup = BeautifulSoup(response.text, "html.parser")
...
... print(soup.h1.text)          # "Example Domain"
... print(soup.a.text)           # "Learn more"
...
Example Domain
Learn more
```

Beautiful Soup is a Python library used to **parse and extract data from HTML and XML**. It works on top of raw text returned by tools like **requests**, turning it into a structured object that is easy to navigate.

- Parses HTML and XML documents
- Allows searching by tags and structure
- Simplifies data extraction
- Commonly used with requests

Beautiful Soup makes working with web content readable and practical.

Lab: Python venv



POP QUIZ:

What is will create a virtual environment named api?

A. Python –m venv venv
B. python –i venv venv
C. python api
D. python –m venv api

POP QUIZ:

What is will create a virtual environment named api?

A. Python –m venv venv
B. python –i venv venv
C. python api
D. **python –m venv api**

POP QUIZ:

Considering a venv named .venv, how can we use an interpreter in this venv (select all)?

A. .venv/Scripts/python myfile
B. Add !# /home/user/.venv/Scripts/python as first line of file
C. source .venv/Scripts/activate # then use python myfile
D. .venv myfile

POP QUIZ:

Considering a venv named .venv, how can we use an interpreter in this venv (select all)?

A. **.venv/Scripts/python myfile**
B. **Add !# /home/user/.venv/Scripts/python as first line of file**
C. **source .venv/Scripts/activate # then use python myfile**
D. .venv myfile

POP QUIZ:

Which version constraint allows any 2.x release of a package but blocks 3.x?

A. requests==2
B. requests>=2
C. requests~=2
D. requests<2

POP QUIZ:

Which version constraint allows any 2.x release of a package but blocks 3.x?

A. requests==2
B. requests>=2
C. **requests~=2**
D. requests<2

POP QUIZ:

What is the proper way to back up a Python environment's installed packages and versions?

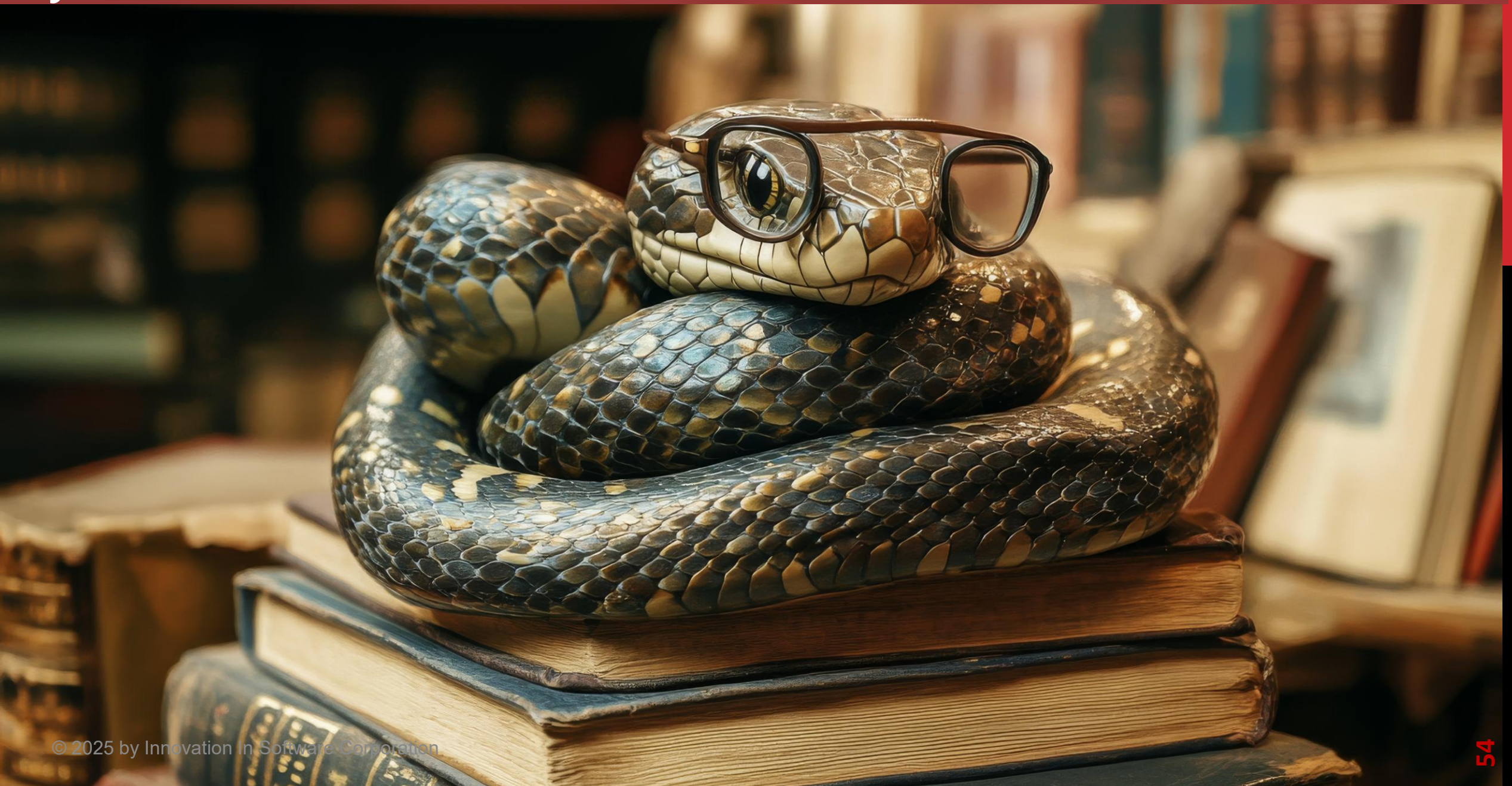
A. Copy the virtual environment directory
B. Run `pip list` and save the output
C. Run `pip freeze > requirements.txt`
D. Reinstall Python using the same version

POP QUIZ:

What is the proper way to back up a Python environment's installed packages and versions?

A. Copy the virtual environment directory
B. Run `pip list` and save the output
C. **Run `pip freeze > requirements.txt`**
D. Reinstall Python using the same version

Python Modules



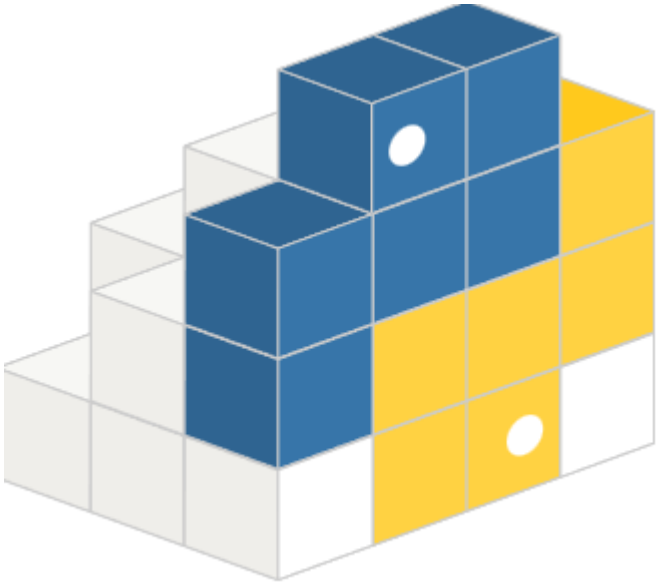
Python Modules

A **module** is a file that contains Python code, such as functions, classes, and variables, that can be reused in other programs. Modules allow code to be organized, shared, and imported instead of rewritten.

- Provide reusable functionality
- Imported using import
- Can be built-in or third-party
- Help structure larger programs

Modules are the basic building blocks of Python applications and automation.

Python Package Index (PyPI)



PyPI is the public repository where Python packages are published and shared. Developers can register and upload their own Python modules so others can install them using **pip**.

- Central repository for Python packages
- Used automatically by pip
- Hosts open source and private tools
- Supports versioning and updates

PyPI makes Python libraries easy to distribute and install across systems.

Packaging Python Projects - Python Packaging User Guide

Python Module Example

```
$ cat utils.py
def moneyf(amount):
    return f"{amount:.2f}"

if __name__ == "__main__":
    print(moneyf(7.899))
    print(moneyf(7.0000001))
    print(moneyf(7.111111))
    print(moneyf(7.489))
    print(moneyf(7.091))

$ python utils.py
7.90
7.00
7.11
7.49
7.09
(venv)
```

This file defines a function that can be reused when the file is imported as a module. When the file is executed directly, Python sets **__name__ to "__main__"**, allowing the test code to run.

- Functions can be imported and reused elsewhere
- The file can also be executed directly
- __name__ == "__main__" checks how the file is being run
- **Prevents execution of test code when imported**

This pattern lets a Python file act as both a **module** and a **standalone script**.

The Import keyword

The **import** keyword is used to load a module so its functions and objects can be used in your code. When you import a module, you access its contents using *module_name.object_name*.

- Loads a module into your program
- Does not execute guarded code (if __name__ == "__main__")
- Access functions via the module name

Importing modules allows you to reuse code instead of duplicating it.

The Import keyword

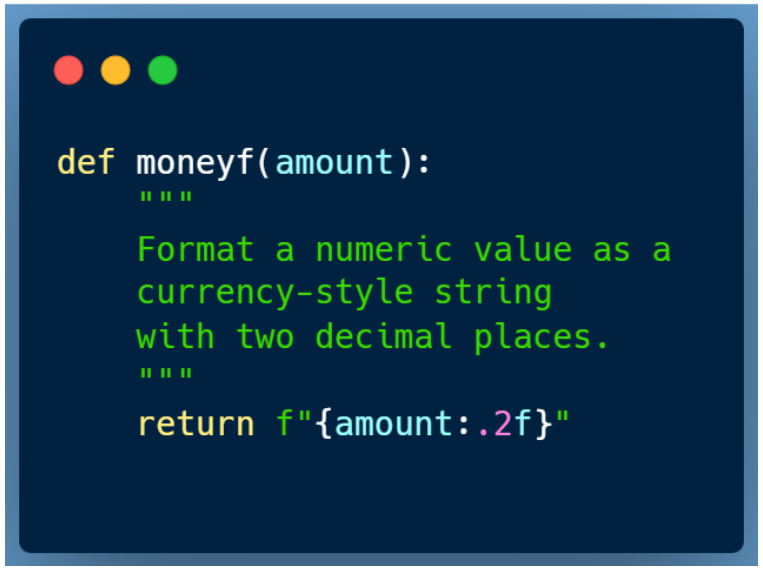
Python supports multiple ways to import modules and objects, each with different tradeoffs. Using explicit imports makes code clearer, safer, and easier to maintain.

- **import utils**
Access everything using utils.object_name

- **from utils import moneyf**
Imports a specific function directly

- **from utils import ***
Imports all public objects and can cause name collisions. This is the least preferred import method.

Functions



```
def moneyf(amount):
    """
    Format a numeric value as a
    currency-style string
    with two decimal places.
    """
    return f"{amount:.2f}"
```

A **function** is a reusable block of code that performs a specific task. Functions take inputs, run logic, and return a result. They help avoid repetition and make code easier to read and maintain.

- Defined using the def keyword
- Can accept parameters
- Can return a value
- Can include documentation (Docstring)

Functions are core to organizing and reusing Python code.

Docstrings

```
>>> from utils import moneyf
>>> help(moneyf)
Help on function moneyf in module
utils:

moneyf(amount)
    Format a numeric value as a
currency-style string
    with two decimal places.

>>>
```

The **help()** function displays documentation for a function, class, or module. The output you see comes directly from the object's **docstring**.

Docstrings are considered a best practice when writing reusable code. They make modules easier to understand, easier to use, and easier to maintain over time. Pydoc will also read your docstring.

Packing and *args

```python
# Any number of arguments can be "packed" into *argv
def analyze_numbers(*argv):
    total = sum(argv)
    mean = total / len(argv)

    variance = sum((x - mean) ** 2 for x in argv) / len(argv)
    std_dev = variance ** 0.5

    return total, mean, std_dev

# Remaning outputs of function are "packed" into stats
total, *stats = analyze_numbers(1, 2, 3, 4, 5)

print(total)
print(stats)
```

Python allows functions to accept and return a flexible number of values using packing and unpacking.

The * operator groups multiple values together or spreads them apart when assigning variables. This makes functions more flexible while keeping the code readable.

Keyword Arguments **kwargs

```python
## **kwargs packs a dictionary from named args
def do_math(number, **kwargs):
    if "add" in kwargs:
        number += kwargs["add"]

    if "multiply" in kwargs:
        number *= kwargs["multiply"]

    return number

math_dict = {"add": 5, "multiply": 2}
result = do_math(**math_dict) # spreads dict
print(result) # 30

result = do_math(10, add=5)
print(result) # 15
```

Python allows functions to accept an arbitrary number of **named arguments** using **kwargs. These values are passed into the function as a dictionary, allowing behavior to be controlled by keywords.

This pattern is common in configuration driven and extensible functions.

****Note that kwargs and args are not keywords, they are common conventions.**

Module Best Practices

When writing Python modules, a few simple practices make your code easier to reuse, test, and maintain.

1. Use **if __name__ == "__main__"** to guard code that should only run when the file is executed directly, which is useful for testing modules without affecting imports.
2. Use **explicit imports** so dependencies are clear and predictable.
3. Write **docstrings** to document how functions and modules are intended to be used.
4. Use ***args and **kwargs** to make functions more flexible without overcomplicating their interfaces.

Following these practices leads to cleaner, more reusable Python modules.

Modules with __init__.py

The **__init__.py** file tells Python that a **directory should be treated as a package**. It is executed when the package is imported and can be used to control what the package exposes.

You can use __init__.py to:

- Initialize package level state
- Import commonly used modules or functions
- Define what is exported from the package

Placing imports in __init__.py helps simplify how users interact with your package.

Simplifying Imports with __init__.py

The __init__.py file is often left empty, but it can be used to **simplify imports** and define a clean public interface for a package.

By placing imports inside __init__.py, you control what users access when they import the package.

This improves readability and avoids deep import paths.

```
utils/
├── __init__.py
├── math_utils.py
├── string_utils.py
```

```
$ cat utils/__init__.py
from .math_utils import add
from .string_utils import shout
```

```
# the __init__ allow us to import as show below
from utils import add, shout

# instead of the full imports
# from utils.math_utils import add
# from utils.string_utils import shout
```

Command Line Args

Python provides **sys.argv** to access arguments passed to a script from the command line.

Arguments are stored as a list of strings, allowing programs to change behavior based on user input.

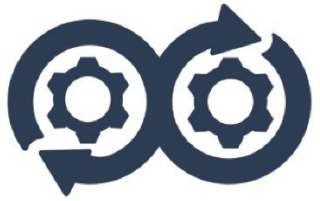
This is commonly used for simple CLI tools and automation scripts.

```python
import sys

for arg in sys.argv[1:]:
    print(arg.upper())
```

```
python script.py hello world python
HELLO
WORLD
PYTHON
```

Lab: Module Basics



POP QUIZ:

What does `if __name__ == "__main__":` do in a Python file?

A. It prevents the file from being imported
B. It runs code only when the file is executed directly
C. It defines the module name
D. It enables command-line arguments

POP QUIZ:

What does `if __name__ == "__main__":` do in a Python file?

A. It prevents the file from being imported
B. **It runs code only when the file is executed directly**
C. It defines the module name
D. It enables command-line arguments

POP QUIZ:

What happens when you use `import utils`?

A. All functions run immediately
B. Only variables are imported
C. The module is loaded and accessed via `utils.object`
D. Guarded code always executes

POP QUIZ:

What happens when you use `import utils`?

A. All functions run immediately
B. Only variables are imported
C. **The module is loaded and accessed via utils.object**
D. Guarded code always executes

POP QUIZ:

What is the purpose of __init__.py?

A. It compiles Python files
B. It defines command-line arguments
C. It marks a directory as a Python package
D. It installs dependencies

POP QUIZ:

What is the purpose of __init__.py?

A. It compiles Python files
B. It defines command-line arguments
C. It marks a directory as a Python package
D. It installs dependencies

POP QUIZ:

What is the purpose of __init__.py?

A. It compiles Python files
B. It defines command-line arguments
C. **It marks a directory as a Python package**
D. It installs dependencies

POP QUIZ:

What does *argv represent in a function definition?

A. A single argument named argv
B. A list of command-line arguments
C. A tuple of positional arguments passed to the function
D. A dictionary of keyword arguments

POP QUIZ:

What does *argv represent in a function definition?

A. A single argument named argv
B. A list of command-line arguments
C**. A tuple of positional arguments passed to the function**
D. A dictionary of keyword arguments

Congratulations

In this section, we covered the core foundations of Python. You learned how the Python interpreter works, how to use the REPL, and how to run basic scripts like a hello world example.

We explored virtual environments, pip, dependency management, and version constraints.

You also learned the basics of creating and structuring Python modules for reuse. These fundamentals form the base for writing practical Python automation.