Programming for Automation

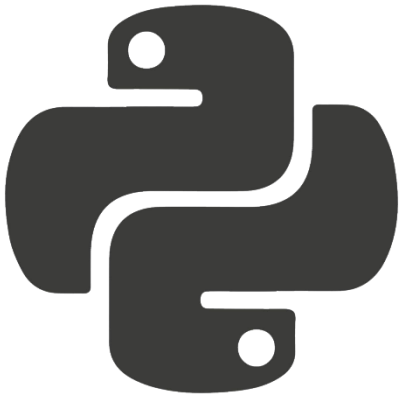
TEKsystems Global Services

WORKFORCE DEVELOPMENT

Python

Develop a passion for learning.

Exceptions



An **exception** occurs when something unexpected happens while a program is running and Python cannot continue executing normally.

Instead of crashing the program, Python raises an exception, giving us a chance to handle the problem gracefully and decide what should happen next.

Exceptions

- Exceptions happen at **runtime**, not when the code is written.
- Common examples include dividing by zero, missing files, or invalid input
- Python uses try, except, else, and finally blocks to handle exceptions
- Proper exception handling keeps programs stable and user-friendly

Takeaway: Exceptions are not failures, they are signals that let us safely control how our program responds to problems.

Exception Examples

```
>>> 10 / 0
Traceback (most recent call last):
  File "<python-input-0>", line 1, in <module>
    10 / 0
    ~~~^~~
ZeroDivisionError: division by zero
>>> a==b
Traceback (most recent call last):
  File "<python-input-1>", line 1, in <module>
    a==b
    ^
NameError: name 'a' is not defined
>>> int("seven")
Traceback (most recent call last):
  File "<python-input-2>", line 1, in <module>
    int("seven")
    ~~~^^^^^^^^^
ValueError: invalid literal for int() with base 10: 'seven'
>>> d=dict()
>>> d['missing_key']
Traceback (most recent call last):
  File "<python-input-4>", line 1, in <module>
    d['missing_key']
    ~^^^^^^^^^^^^^^^
KeyError: 'missing_key'
>>>
```

On the left you can see common exceptions, such as **ZeroDivisionError, NameError, ValueError** and **KeyError**

Exception Examples

```
>>> 10 / 0
Traceback (most recent call last):
  File "<python-input-0>", line 1, in <module>
    10 / 0
    ~~~^~~
ZeroDivisionError: division by zero
>>> a==b
Traceback (most recent call last):
  File "<python-input-1>", line 1, in <module>
    a==b
    ^
NameError: name 'a' is not defined
>>> int("seven")
Traceback (most recent call last):
  File "<python-input-2>", line 1, in <module>
    int("seven")
    ~~~^^^^^^^^^
ValueError: invalid literal for int() with base 10: 'seven'
>>> d=dict()
>>> d['missing_key']
Traceback (most recent call last):
  File "<python-input-4>", line 1, in <module>
    d['missing_key']
    ~^^^^^^^^^^^^^^^
KeyError: 'missing_key'
>>>
```

If an exception is raised and nothing catches it, Python passes the error up the call stack. When it reaches the top without being handled, the program stops running and displays a traceback.

Key points

- The exception moves **up the call stack**
- No handler means **program execution stops**
- Python prints a **traceback** for debugging

Takeaway: Unhandled exceptions halt your program, so they should be handled intentionally.

Exception Handling

```
>>> def risky_operations():
...     try:
...         x = int("abc")          # ValueError
...         y = 10 / 0              # ZeroDivisionError
...         numbers = [1, 2, 3]
...         print(numbers[5])       # IndexError
...     except:
...         print("Something went wrong, but the
program did not crash.")
...
... risky_operations()
... print("Program continues running...")
...
Something went wrong, but the program did not crash.
Program continues running...
>>>
```

To handle exceptions, place code that **may fail** inside a try block. If an exception occurs, Python immediately jumps to the matching except block instead of stopping the program.

Key points

- Code that might fail goes in try
- except runs only if an exception is raised
- A bare except: catches **all exceptions**
- Useful for demos, logging, or last-resort safety nets

Takeaway: try / except lets your program recover instead of crashing.

Python - Finally

The finally block runs **no matter what**, whether an exception occurs or not. It is commonly used for cleanup tasks like closing files, releasing resources, or logging that an operation finished.

The only time finally does not run is when the program is forcefully exited or terminated.
Takeaway: finally is your guarantee that cleanup code runs.

```
>>> def demo_finally():
...     try:
...         x = 10 / 0
...     except:
...         print("Exception caught")
...     finally:
...         print("This always runs")
...
... demo_finally()
... print("Program continues...")
...
Exception caught
This always runs
Program continues...
>>>
```

Base Exception Class

When we catch exceptions with “except”, we prevent a crash but lose the reason the error occurred. A better approach is to catch the generic Exception type and inspect its message so we can understand what went wrong.

All built-in Python exceptions inherit from the base Exception class, which means catching Exception safely captures most runtime errors while still exposing useful details.

Takeaway: Catching Exception as e keeps your program running *and* tells you why it failed.

```
>>> def demo_generic_exception():
...     try:
...         x = int("abc")   # ValueError
...         print(10 / x)
...     except Exception as e:
...         print("Something went wrong:", e)
...
... demo_generic_exception()
...
Something went wrong: invalid literal for int()
with base 10: 'abc'
>>>
```

Catch Specific Exceptions First

Catching specific exceptions makes your code clearer and lets you respond differently depending on the failure.

Put specific handlers first, then a generic Exception at the bottom as a fallback

```
def demo_specific_then_generic(value):
    try:
        x = int(value)          # ValueError possible
        result = 10 / x         # ZeroDivisionError possible
        print("Result:", result)

    except ValueError as e:
        print("Invalid number input:", e)

    except ZeroDivisionError as e:
        print("Cannot divide by zero:", e)

    except Exception as e:
        print("Unexpected error:", e)

demo_specific_then_generic("0")
demo_specific_then_generic("abc")
demo_specific_then_generic("2")
```

Raising Exceptions

Sometimes your code should stop and clearly signal that something went wrong.

In Python, we use raise to trigger an exception on purpose, either because an input is invalid or because a rule was violated.

Takeaway: raise is how your code communicates failures intentionally.

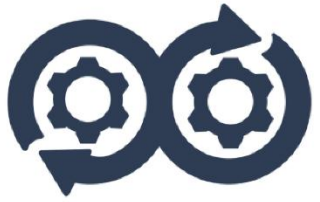
```
>>> # my_api_client.py
...
... class APILimitReached(Exception):
...     pass
...
... def call_api(status_code: int):
...     # Simulating what would happen after an HTTP request
...     if status_code == 429:
...         raise APILimitReached("Too many requests (429).
Slow down or retry later.")
...
...     return {"ok": True}
...
...
... # user_code.py
...
... try:
...     data = call_api(429)
...     print(data)
... except APILimitReached as e:
...     print("Handle throttling:", e)
...
Handle throttling: Too many requests (429). Slow down or
retry later.
>>>
```

Raising Exceptions

Consider the code module shown. Our module enforces a limit, and when the limit is hit, we raise a custom exception.

```
>>> # my_api_client.py
...
... class APILimitReached(Exception):
...     pass
...
... def call_api(status_code: int):
...     # Simulating what would happen after an HTTP request
...     if status_code == 429:
...         raise APILimitReached("Too many requests (429).
Slow down or retry later.")
...
...     return {"ok": True}
...
...
... # user_code.py
...
... try:
...     data = call_api(429)
...     print(data)
... except APILimitReached as e:
...     print("Handle throttling:", e)
...
Handle throttling: Too many requests (429). Slow down or
retry later.
>>>
```

Lab: Python IO



IO Serialization

Why Serialization and IO Matter

Modern automation and network tooling depends on reading configuration files, storing state, and exchanging structured data between systems. Python programs rarely operate only in memory, they interact with files, APIs, and external tools.

- Automation requires **reading and writing data**
- Data often lives outside the program
- Different systems expect different formats

Real-world Python programs must work with files and data formats.

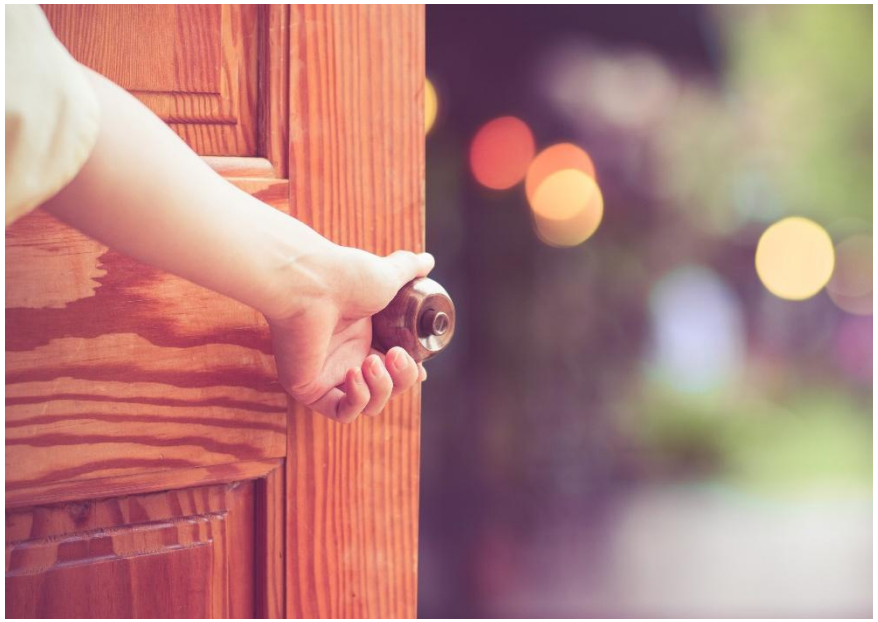
Files vs Data Formats

Files are how data is stored, but **formats define structure**. Python works with many formats, each designed for different use cases such as automation, APIs, and configuration management.

- Files store raw data
- Formats describe **how the data is structured**
- Some formats are for humans, others for machines

Choosing the right format matters as much as writing the code.

Python – Open()



Python provides the with open() pattern to safely work with files. It ensures files are properly closed, even if an error occurs.

- open() gives access to a file
- with automatically closes the file
- Prevents resource leaks and file corruption

Always use with open() when working with files.

Built-in Functions — Python 3.14.2 documentation

Python File IO

When opening a file, you must specify **how** you want to interact with it. The mode controls whether you are reading, writing, or appending data.

- "r" read existing files
- "w" write and overwrite
- "a" append to existing files
- "b" for binary data

The file mode determines what your program is allowed to do.

File IO

```
>>> # Write to a file (overwrites if it exists)
... with open("devices.txt", "w") as f:
...     f.write("Router-1\n")
...     f.write("Switch-1\n")
...
... # Append to a file
... with open("devices.txt", "a") as f:
...     f.write("Firewall-1\n")
...
... # Read entire file
... with open("devices.txt", "r") as f:
...     content = f.read()
...     print(content)
...
... # Read line by line
... with open("devices.txt", "r") as f:
...     for line in f:
...         print(line.strip())
...
Router-1
Switch-1
Firewall-1

Router-1
Switch-1
Firewall-1
```

Writing, appending, and reading files all use the same with open() pattern.

Serialization

Serialization is the process of converting in-memory data structures into a format that can be stored or transmitted, then reconstructed later.

- Python objects live in memory
- Serialized data can be saved or sent
- Deserialization recreates the object

Serialization lets programs share and persist structured data.

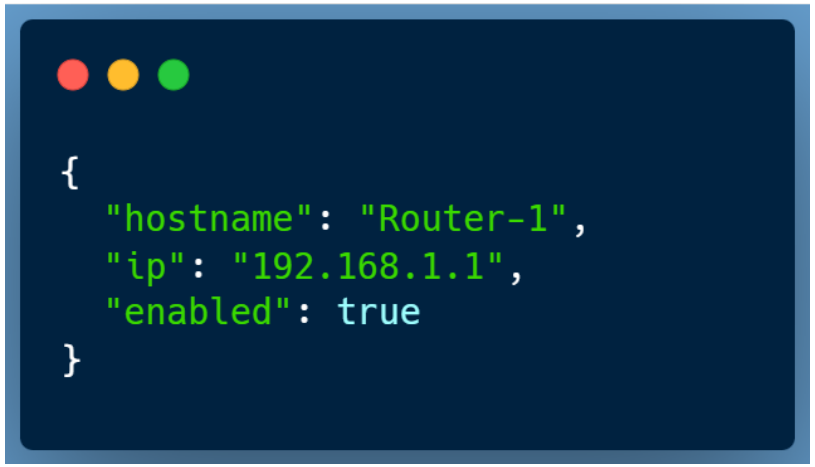
Serialization

Different formats exist for different use cases, balancing readability, compatibility, and performance.

- JSON for APIs and data exchange
- YAML for human-readable configs
- XML for legacy and network systems
- Pickle for Python-only objects

Each format solves a different problem.

JSON



JSON (JavaScript Object Notation) is a lightweight, structured data format commonly used by APIs and automation tools. Python dictionaries and lists map naturally to JSON.

- Text-based and human-readable
- Language-agnostic format
- Direct mapping to Python dict and list

JSON is the most common way systems exchange structured data.

Python and JSON

Python data structures translate directly into JSON, which makes serialization simple and predictable.

- dict → JSON object
- list → JSON array
- str, int, bool → JSON primitives
- Keys must be strings

If you understand Python dictionaries, you already understand JSON.

```python
# Python data structures
data = {
    "hostname": "Router-1",        # str
    "ports": [22, 80, 443],        # list of int
    "enabled": True,               # bool
    "metadata": {
        "location": "DC-1",        # nested dict
        "rack": 42
    }
```

```json
{
  "hostname": "Router-1",
  "ports": [22, 80, 443],
  "enabled": true,
  "metadata": {
    "location": "DC-1",
    "rack": 42
  }
}
```

JSON Serialization Example

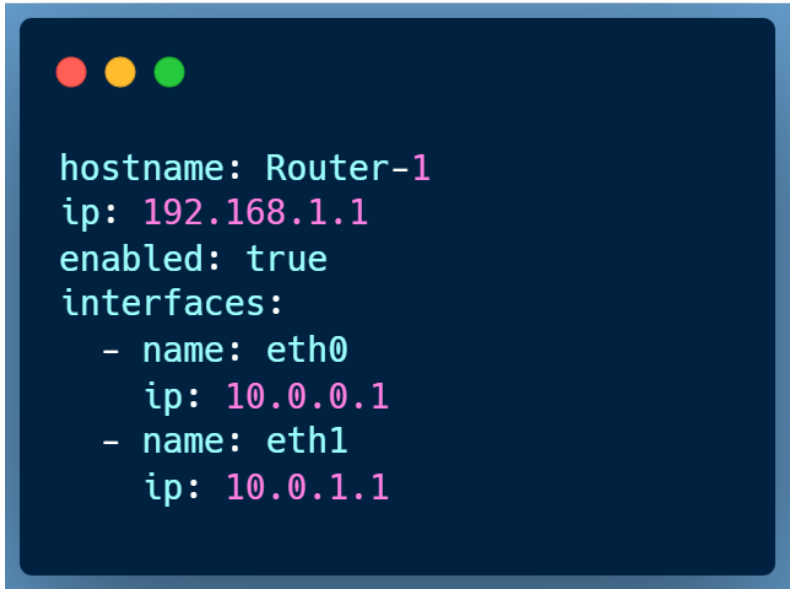
```python
import json

device = {
    "hostname": "Router-1",
    "ip": "192.168.1.1",
    "ports": [22, 80, 443],
    "enabled": True
}

# Write JSON to file
with open("device.json", "w") as f:
    json.dump(device, f, indent=2)

# Read JSON from file
with open("device.json", "r") as f:
    loaded_device = json.load(f)

print(loaded_device["hostname"])
```

Python uses the built-in **json** module to serialize dictionaries to files and load them back into memory.

- json.dump() writes JSON to a file
- json.load() reads JSON from a file
- Keeps data structured and portable

JSON files make it easy to persist and reload structured data.

YAML

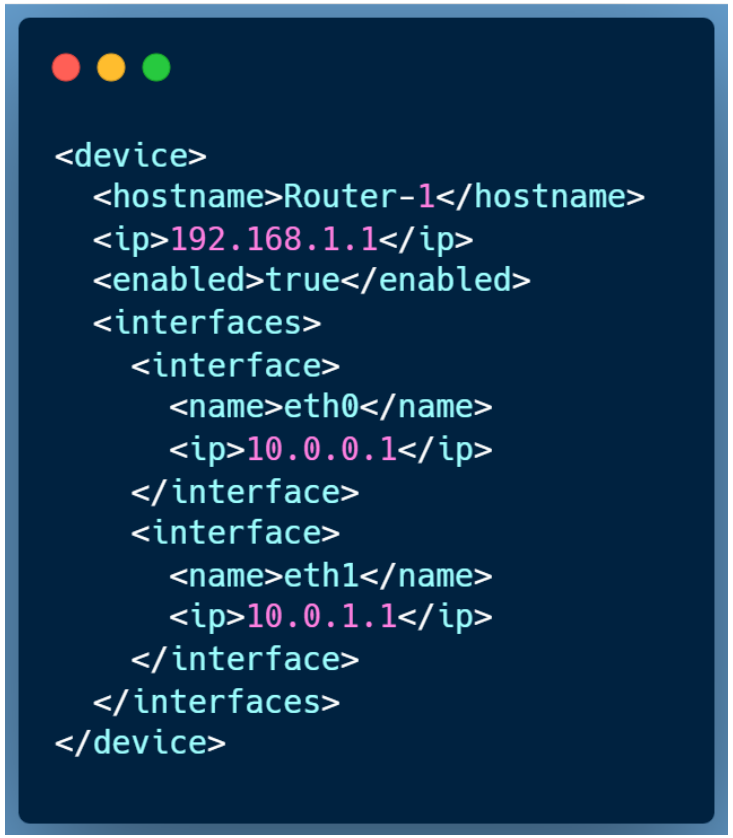


```yaml
hostname: Router-1
ip: 192.168.1.1
enabled: true
interfaces:
  - name: eth0
    ip: 10.0.0.1
  - name: eth1
    ip: 10.0.1.1
```

YAML is a human-readable data format commonly used for configuration files in automation and infrastructure tools. Its structure is designed to be easy for people to read and write while still being machine-friendly.

Frequently used in Ansible, Docker Compose, and Kubernetes
Emphasizes readability over strict syntax
Supports nested data and comments. YAML is often chosen when humans are expected to read and edit configuration files.

XML



```xml
<device>
  <hostname>Router-1</hostname>
  <ip>192.168.1.1</ip>
  <enabled>true</enabled>
  <interfaces>
    <interface>
      <name>eth0</name>
      <ip>10.0.0.1</ip>
    </interface>
    <interface>
      <name>eth1</name>
      <ip>10.0.1.1</ip>
    </interface>
  </interfaces>
</device>
```

XML is a structured and verbose data format that has been widely used in networking and enterprise systems for many years. While less readable than JSON or YAML, it provides strict structure and validation.

Common in NETCONF and some network device APIs
Strongly structured with opening and closing tags
Still present in many legacy systems

XML remains important because many networking tools and protocols still depend on it.

Pickle

```python
import pickle

data = {
    "hostname": "Router-1",
    "ports": [22, 80, 443],
    "enabled": True
}

# Save object to file
with open("device.pkl", "wb") as f:
    pickle.dump(data, f)

# Load object from file
with open("device.pkl", "rb") as f:
    loaded_data = pickle.load(f)

print(loaded_data)
```

Pickle is a built-in Python module that serializes Python objects into a **binary format**. It allows you to save complex in-memory objects to disk and load them back later, exactly as they were.

- Python-specific format
- Binary, not human-readable
- Useful for caching or temporary storage
- Should only be used with **trusted data**

Pickle is convenient, but it is not designed for cross-language data sharing or long-term storage.

Data Formats in Practice

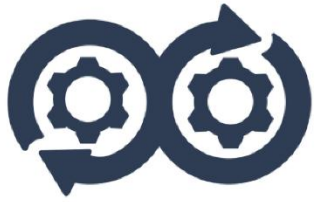
JSON maps naturally to Python dictionaries and lists and is commonly used to exchange data between systems, especially through APIs.

XML is still used in some network configurations and legacy APIs where strict structure and long-standing standards are required.

YAML focuses on human readability and is often used for configuration files in tools like Kubernetes and Ansible, where clarity matters more than compactness.

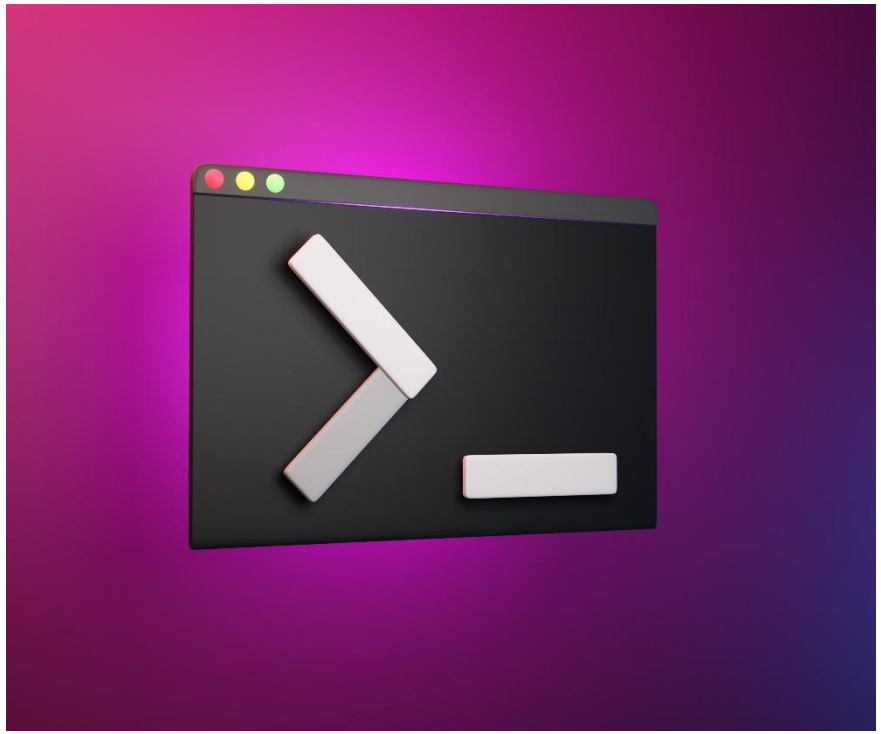
Understanding when and why each format is used is more important than memorizing their syntax.

Lab: Python Serialization



Python and PowerShell

PowerShell

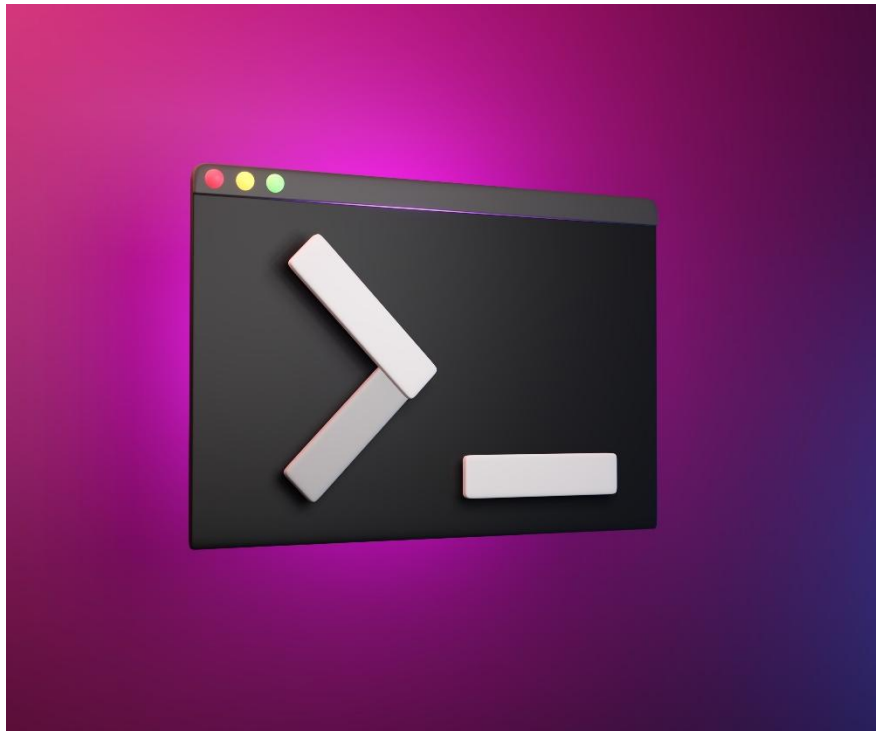


PowerShell is a task automation and scripting environment built for managing Windows systems and infrastructure. It is widely used by system and network engineers working in Windows-based environments.

- Designed for administration and automation
- Built on top of .NET
- Common in Windows servers and enterprise networks

PowerShell is a core skill for working in Windows-heavy environments.

PowerShell

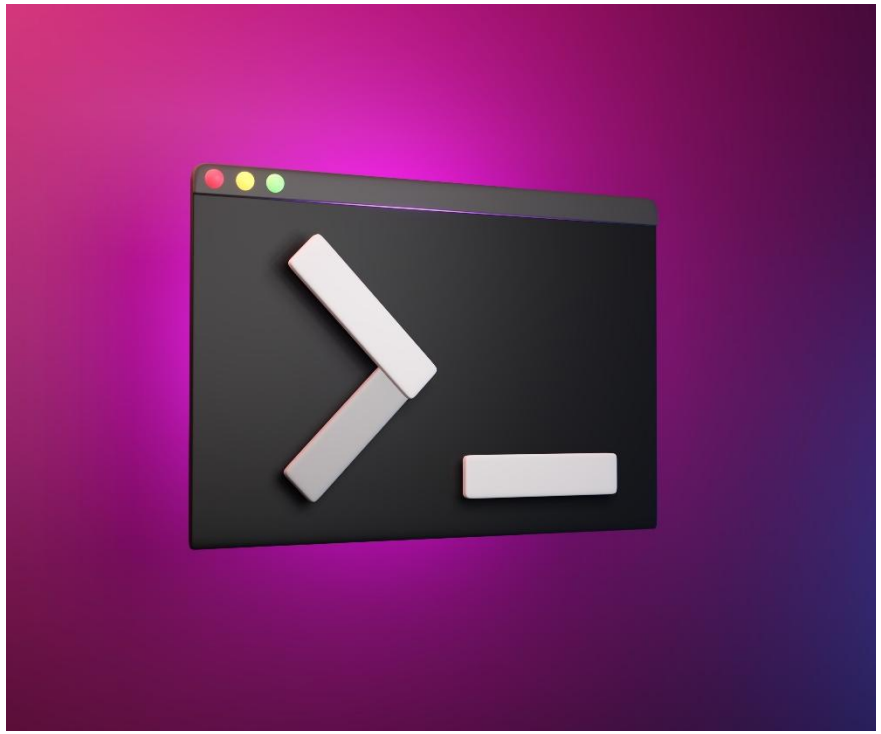


In Windows environments, PowerShell provides direct access to networking, system, and API functionality without installing extra tools. Many networking tasks are built in as native commands.

- Built-in networking cmdlets
- Deep access to the operating system
- Ideal for quick diagnostics and automation

PowerShell shines when working close to the operating system.

PowerShell vs Python

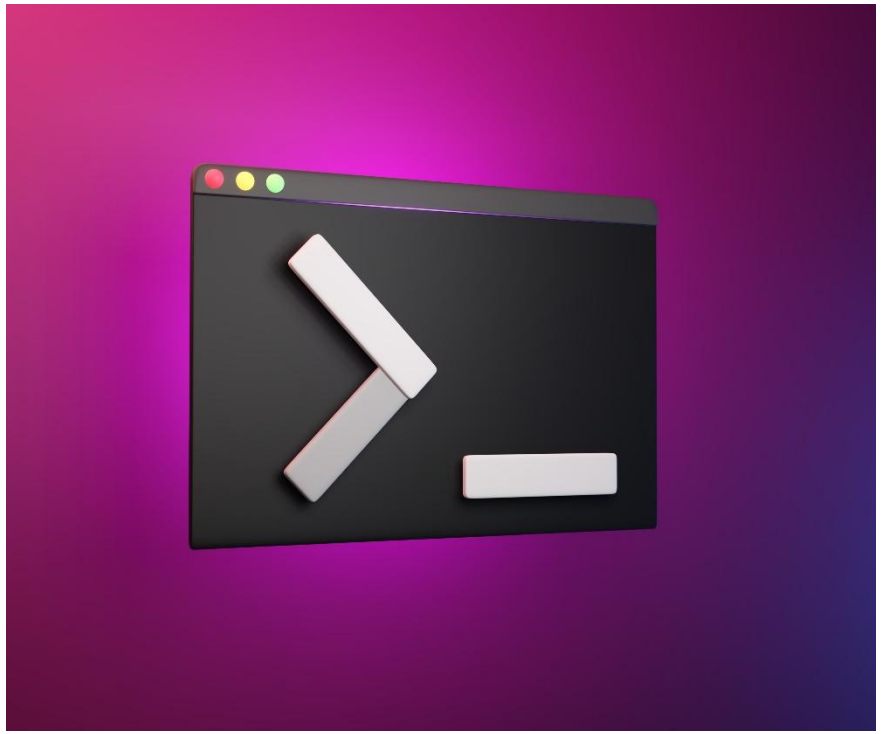


PowerShell and Python can solve similar problems, but they are designed for different purposes. PowerShell is an administrative shell, while Python is a general-purpose programming language.

- PowerShell is command-first and interactive
- Python is code-first and library-driven
- Both are object-oriented

Knowing both makes you effective across environments.

PowerShell Objects



Unlike traditional shells, PowerShell works with objects instead of plain text. Commands return structured data with properties and methods.

- Cmdlets return objects
- Objects have properties you can inspect
- No text parsing required

This object-based design is what makes PowerShell powerful.

Basic Powershell Commands

```powershell
# Go to your home directory
Set-Location ~

# Show current location
Get-Location

# List files and folders
Get-ChildItem

# Create a directory
New-Item -ItemType Directory demo-
folder

# Move into the directory
Set-Location demo-folder

# Create a file
New-Item demo.txt

# View file contents
Get-Content demo.txt
```

Before working with networking tasks, it's important to understand a few basic PowerShell commands and patterns.

- Get-Command to discover commands
- Get-Help to learn how commands work
- Get-Member to inspect object properties

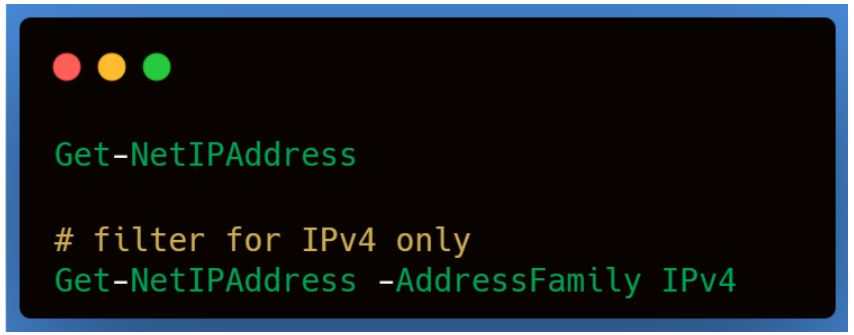
Learning these basics makes every PowerShell task easier.

Network Adapters

The command “Get-NetAdapter” Shows all network interfaces. Includes status, interface speed, and MAC address. First command to run when diagnosing network issues. This command shows information related to OSI layer 2.

This command answers “what network interfaces does this machine have?”

Viewing IP addresses



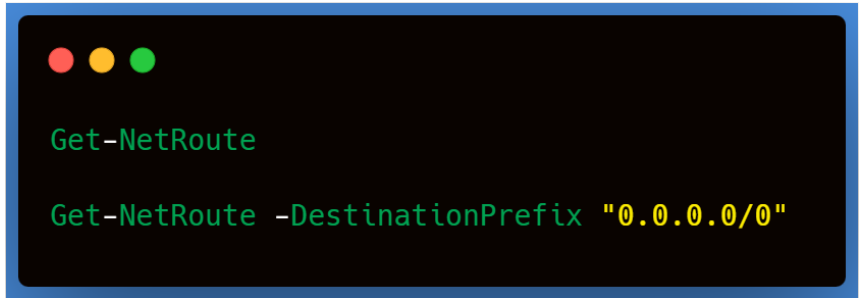
```
Get-NetIPAddress

# filter for IPv4 only
Get-NetIPAddress -AddressFamily IPv4
```

Next logical question after adapters: what IPs are assigned?

"**Get-NetIPAddress**" command shows IP addresses, subnet, and interface.

This shows how the system is addressed on the network, focused on the layer 3 of OSI model.

Routing Table



Once you know IPs, the next question is: where does traffic go?

Command shows how to display routing table. Default route determines where non-local traffic goes

Routing decides how packets leave the system.

Testing Connectivity (ICMP)

“Test-Connection google.com -Count 4”

“Test-Connection google.com -Count 1 –Quiet”

- ICMP reachability test
- Quiet mode returns true/false (great for scripts)

This answers “can I reach the destination at all?”

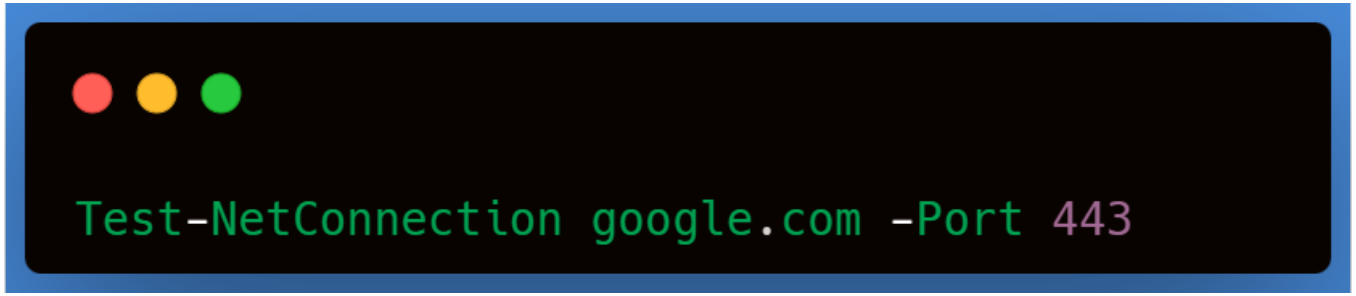
Testing Ports

This is where PowerShell really shines.

"**Test-NetConnection google.com -Port 443"**

- Tests TCP connectivity
- Shows DNS, IP, and port status
- No extra tools required

This is one of PowerShell's most powerful built-in networking features.



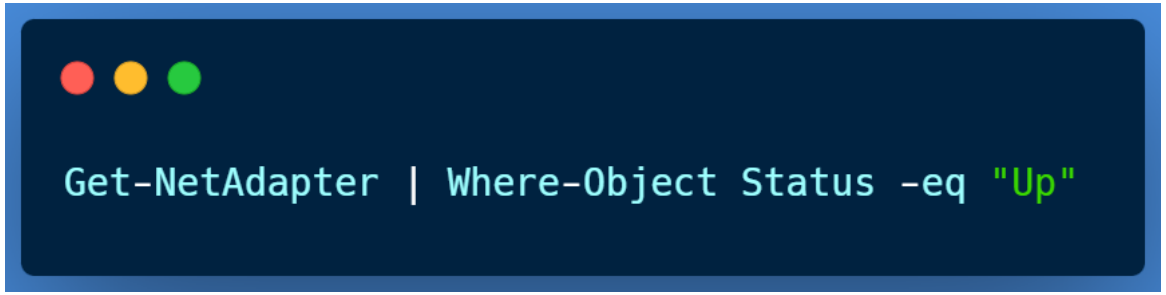
Warning: Only test ports on systems you own or have explicit permission to test.

Use well-known public services and standard ports such as 80 (HTTP) and 443 (HTTPS).
Unauthorized port scanning may be considered illegal.

The PowerShell Pipeline

PowerShell uses a pipeline to pass **objects** from one command to the next.

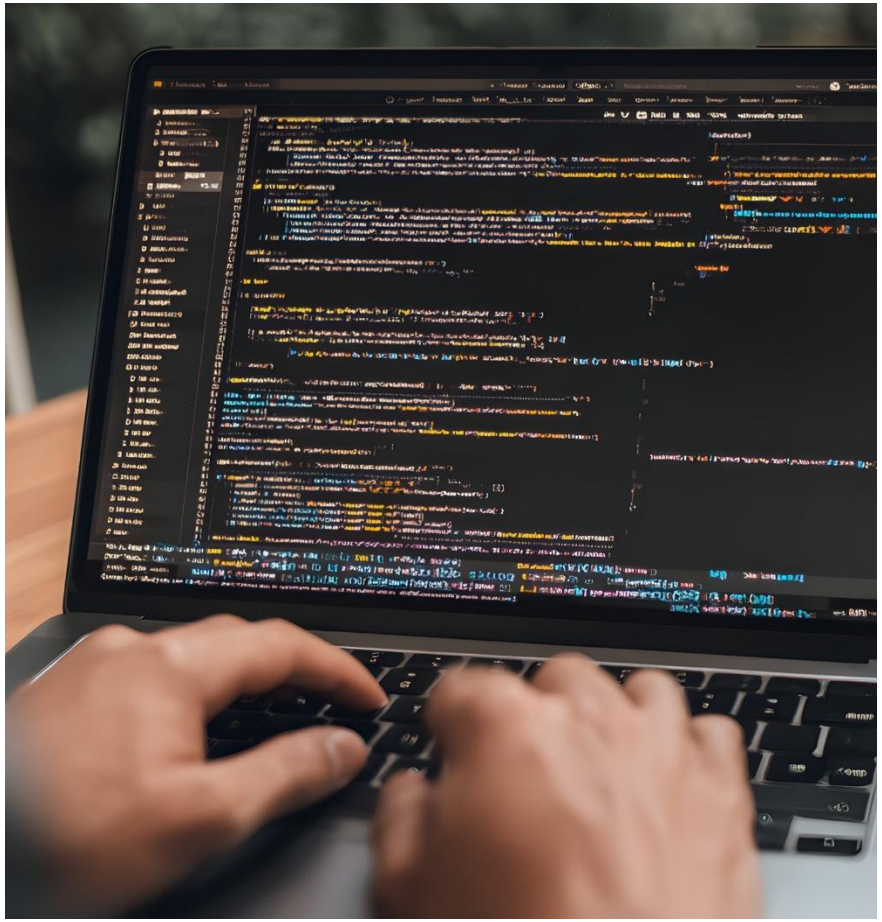
Each command receives structured data, not plain text, which allows filtering and selection without parsing output.



- Output of one command becomes input to the next
- Objects keep their properties through the pipeline
- Commands focus on a single responsibility

The pipeline is what makes PowerShell concise and powerful for administration.

Python for Networking Tasks



Python can perform the same network checks, but it usually relies on libraries instead of built-in cmdlets. Many network automation tools and scripts are written in Python because it works across Windows, Linux, and network appliances.

- PowerShell uses built-in cmdlets and object pipelines
- Python uses libraries like **psutil**, **socket**, and **requests**
- Python scripts are portable across platforms
- Many automation tools and SDKs are Python-first

Python gives you cross-platform automation, while PowerShell gives you deep Windows integration.

Python psutil Module

```python
import psutil

# List network interfaces
interfaces = psutil.net_if_addrs()

for name in interfaces:
    print(name)
```

psutil is a popular Python library used to retrieve system and network information. Many monitoring and automation tools rely on it under the hood.

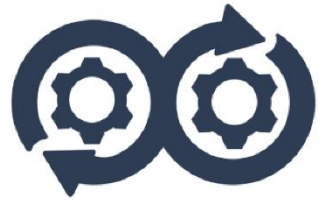
- Provides network interface and IP information
- Works across operating systems
- Returns structured Python objects

psutil allows Python to perform many of the same tasks as PowerShell networking cmdlets.

Python psutil Module

```python
import psutil
import socket

# PowerShell equivalent:
# Get-NetIPAddress
# Get IP addresses for each interface
for iface, addresses in psutil.net_if_addrs().items():
    for addr in addresses:
        if addr.family == socket.AF_INET:
            print(f"{iface} -> {addr.address}")

# PowerShell equivalent:
# Get-NetAdapter
# Get interface status (UP / DOWN)
stats = psutil.net_if_stats()
for iface, stat in stats.items():
    print(f"{iface}: {'UP' if stat.isup else 'DOWN'}")
```

Lab: Python and PowerShell



Congratulations

Congratulations — you've covered the foundations of Python programming.
You learned how to work with core data structures, understand and use basic Object-Oriented Programming concepts, and apply Pythonic patterns such as generators, comprehensions, and decorators.

These fundamentals form the base for writing clean, readable, and scalable Python code.
Great work — you're now ready to build on this foundation.