Environment Variables in Python

Introduction

When we build real applications, we often need information that should **not be written directly inside our Python code**.

Examples:

- Database passwords
- API keys
- Secret tokens
- Application configuration
- Port numbers
- Debug settings
- File paths
- Environment-specific settings

Writing sensitive information directly in code is risky.

Instead, we can store this information in **environment variables** and read it from Python when the program runs.

This gives us a simple separation:

Code application logic

Environment variables configuration and secrets

---

1. What Is an Environment Variable?

An environment variable is a **key-value pair stored outside your Python program**.

For example:

```
DATABASE_URL=postgresql://localhost/mydb
API_KEY=abc123
DEBUG=true
PORT=8000
```

Here:

```
DATABASE_URL
API_KEY
DEBUG
PORT
```

are variable names.

Their corresponding values are:

```
postgresql://localhost/mydb
abc123
true
8000
```

Your Python program can read these values when it starts.

---

2. Why Do We Need Environment Variables?

Imagine writing this:

`API_KEY = "sk-123456789"`

This creates several problems.

Problem 1: Secrets can leak

If the code is uploaded to GitHub, the API key may become visible.

Problem 2: Different environments need different values

Your development environment might use:

`localhost`

while production might use:

`production_database`

You do not want to rewrite your Python code every time.

Problem 3: Configuration should be separate from logic

Your program should focus on:

`connect_to_database()`

rather than containing passwords and deployment-specific settings everywhere.

---

3. Reading Environment Variables with `os`

Python provides the `os` module for interacting with environment variables.

```
import os

api_key = os.getenv("API_KEY")

print(api_key)
```

If the environment contains:

```
API_KEY=abc123
```

Python reads:

```
abc123
```

---

4. os.getenv()

The most common way to read an environment variable is:

```
os.getenv("VARIABLE_NAME")
```

Example:

```
import os

username = os.getenv("USERNAME")

print(username)
```

If `USERNAME` exists, Python returns its value.

If it does not exist, Python returns:

```
None
```

---

5. Checking for a Missing Variable

Consider:

```
import os

api_key = os.getenv("API_KEY")

print(api_key)
```

If `API_KEY` does not exist:

```
None
```

That may cause problems later.

A better approach is to validate important configuration.

```
import os

api_key = os.getenv("API_KEY")

if api_key is None:
    raise ValueError("API_KEY is not configured")

print("API key loaded")
```

Now the program fails with a meaningful message instead of producing a confusing error later.

---

6. Providing a Default Value

`os.getenv()` can also receive a default value.

```
import os

port = os.getenv("PORT", "8000")

print(port)
```

If `PORT` exists:

```
5000
```

Python uses:

```
5000
```

If it does not exist, Python uses:

```
8000
```

The structure is:

```
os.getenv("NAME", "default_value")
```

---

7. Environment Variables Are Strings

One important rule:

Environment variables are read as strings.

Suppose:

```
PORT=8000
```

Python does not automatically treat `8000` as an integer.

```
import os

port = os.getenv("PORT")

print(type(port))
```

Output:

```
<class 'str'>
```

If you need an integer:

```
port = int(os.getenv("PORT", "8000"))
```

Now:

```
print(type(port))
```

Output:

```
<class 'int'>
```

The same idea applies to booleans.

```
debug = os.getenv("DEBUG", "false").lower() == "true"
```

---

8. Common Environment Variable Types

Although environment variables are stored as text, your application may convert them.

Environment value	Python conversion
`"8000"`	`int(...)`
`"3.14"`	`float(...)`
`"true"`	Boolean conversion
`"false"`	Boolean conversion
`"hello"`	Keep as string

Example:

```
import os

port = int(os.getenv("PORT", "8000"))
timeout = float(os.getenv("TIMEOUT", "5.0"))
debug = os.getenv("DEBUG", "false").lower() == "true"
```

---

9. Setting Environment Variables Temporarily

You can create an environment variable from Python using:

```
import os

os.environ["APP_NAME"] = "haas.dev"

print(os.getenv("APP_NAME"))
```

Output:

```
haas.dev
```

However, this normally affects the **current process and its environment**, not your permanent system configuration.

For application secrets and deployment configuration, environment variables are usually set outside the application.

---

10. Reading Multiple Variables

A real application may need several configuration values.

```
import os

database_url = os.getenv("DATABASE_URL")
api_key = os.getenv("API_KEY")
debug = os.getenv("DEBUG", "false").lower() == "true"
port = int(os.getenv("PORT", "8000"))
```

Now your application can use these values without hardcoding them.

---

11. Environment Variables vs Hardcoded Values

Hardcoded

```
API_KEY = "abc123"
DATABASE_PASSWORD = "mypassword"
```

Environment variables

```
import os

API_KEY = os.getenv("API_KEY")
DATABASE_PASSWORD = os.getenv("DATABASE_PASSWORD")
```

The second approach keeps configuration outside the source code.

---

12. `.env` Files

During local development, developers commonly use a file named:

```
.env
```

Example:

```
API_KEY=abc123
DATABASE_URL=postgresql://localhost/mydb
DEBUG=true
PORT=8000
```

This makes local configuration easier.

However, Python does **not automatically read `.env` files**.

A popular third-party package for this is:

`python-dotenv`

Install it with:

```
pip install python-dotenv
```

---

13. Loading a `.env` File

Suppose your project contains:

```
project/
□
□□□ .env
□□□ app.py
□□□ requirements.txt
```

`.env`:

```
API_KEY=abc123
DEBUG=true
PORT=8000
```

Python:

```
from dotenv import load_dotenv
import os
```

```
load_dotenv()

api_key = os.getenv("API_KEY")
debug = os.getenv("DEBUG")
port = os.getenv("PORT")

print(api_key)
print(debug)
print(port)
```

`load_dotenv()` loads values from `.env` into the environment used by the application.

---

14. A Better `.env` Example

A realistic development `.env` file might contain:

```
APP_NAME=haas.dev
DEBUG=true
PORT=8000
DATABASE_URL=postgresql://localhost/haas
API_KEY=your_api_key_here
```

Python:

```
import os
from dotenv import load_dotenv

load_dotenv()

app_name = os.getenv("APP_NAME")
debug = os.getenv("DEBUG", "false").lower() == "true"
port = int(os.getenv("PORT", "8000"))
```

```
database_url = os.getenv("DATABASE_URL")
api_key = os.getenv("API_KEY")
```

Now the application configuration is separated from the application logic.

---

15. Never Commit Secrets to Git

A `.env` file can contain sensitive information.

For example:

```
API_KEY=real_secret_key
DATABASE_PASSWORD=real_password
```

You generally should **not upload this file to GitHub**.

Add it to `.gitignore`:

```
.env
```

A typical project might have:

```
project/
□
□□□ .env
□□□ .gitignore
□□□ app.py
□□□ requirements.txt
```

`.gitignore`:

```
.env
```

This helps prevent accidental commits.

---

16. .env.example

If other developers need to know which environment variables are required, create:

```
.env.example
```

Example:

```
API_KEY=
DATABASE_URL=
DEBUG=false
PORT=8000
```

The real `.env` contains actual values.

The `.env.example` contains only the expected variable names.

This gives developers a template without exposing secrets.

---

17. Environment Variables in Development and Production

Imagine your application has two environments.

Development

```
DATABASE_URL=localhost
DEBUG=true
```

Production

```
DATABASE_URL=production_database
DEBUG=false
```

The Python code can remain the same:

```
import os

database_url = os.getenv("DATABASE_URL")
debug = os.getenv("DEBUG", "false").lower() == "true"
```

Only the environment configuration changes.

This is one of the biggest advantages of environment variables.

---

18. Configuration Module

As an application grows, repeatedly calling `os.getenv()` everywhere can become messy.

You can create:

```
project/
□
□□□ config.py
□□□ app.py
□□□ .env
```

`config.py`:

```
import os
from dotenv import load_dotenv

load_dotenv()

API_KEY = os.getenv("API_KEY")
DATABASE_URL = os.getenv("DATABASE_URL")
DEBUG = os.getenv("DEBUG", "false").lower() == "true"
PORT = int(os.getenv("PORT", "8000"))
```

Then:

```
import config

print(config.PORT)
print(config.DEBUG)
```

This gives the application one central place for configuration.

---

19. Validating Configuration

Important variables should be checked when the application starts.

```
import os
from dotenv import load_dotenv

load_dotenv()

api_key = os.getenv("API_KEY")

if not api_key:
    raise RuntimeError("API_KEY is missing")
```

This is better than discovering the missing key much later.

For multiple required values:

```
required = [
    "API_KEY",
    "DATABASE_URL"
]

missing = [
    name
    for name in required
    if not os.getenv(name)
]

if missing:
    raise RuntimeError(
        f"Missing environment variables: {', '.join(missing)}"
    )
```

---

20. Environment Variables and API Keys

Suppose your application calls an external API.

Avoid:

```
API_KEY = "abc123secret"
```

Instead:

```
import os
```

```
API_KEY = os.getenv("API_KEY")

if not API_KEY:
    raise RuntimeError("API_KEY is missing")
```

Then your application can use:

```
response = call_api(API_KEY)
```

The secret remains outside the source code.

---

21. Environment Variables and Database Connections

A database connection may require several values:

```
DB_HOST
DB_PORT
DB_NAME
DB_USER
DB_PASSWORD
```

Python:

```
import os

host = os.getenv("DB_HOST")
port = int(os.getenv("DB_PORT", "5432"))
database = os.getenv("DB_NAME")
user = os.getenv("DB_USER")
password = os.getenv("DB_PASSWORD")
```

The application can then construct its database connection using these values.

---

22. Environment Variables in Web Applications

Web applications commonly use environment variables for:

`DATABASE_URL`
`SECRET_KEY`
`API_KEY`
`PORT`
`DEBUG`
`CORS_ORIGINS`

For example:

```
import os

secret_key = os.getenv("SECRET_KEY")
port = int(os.getenv("PORT", "8000"))
```

The same application can run locally, on a test server, or in production without changing the source code.

---

23. Environment Variables Are Not Automatically Secure

An important misconception is:

> "If something is an environment variable, it is automatically secure."

Not necessarily.

Environment variables can still be exposed through:

- Logs
- Debug output
- Misconfigured servers
- CI/CD systems
- Container configuration
- Error messages
- Accidentally committed configuration files

Never do:

```
print(os.getenv("API_KEY"))
```

in production logs.

Treat secrets as sensitive regardless of where they are stored.

---

24. Common Mistakes

Mistake 1: Hardcoding secrets

```
API_KEY = "real_secret"
```

Better:

```
API_KEY = os.getenv("API_KEY")
```

Mistake 2: Forgetting that values are strings

```
port = os.getenv("PORT")

if port > 5000:
    ...
```

This can fail because `port` is a string.

Use:

```
port = int(os.getenv("PORT", "8000"))
```

Mistake 3: Assuming `.env` loads automatically

This:

```
.env
```

does not automatically become available to Python.

When using `python-dotenv`:

```
from dotenv import load_dotenv

load_dotenv()
```

Mistake 4: Committing `.env`

Never casually commit:

`.env`

when it contains real secrets.

Use:

`.env.example`

for documentation.

Mistake 5: No validation

This:

```
api_key = os.getenv("API_KEY")
```

may produce:

`None`

if the variable is missing.

For required values, validate them early.

Mistake 6: Exposing secrets in logs

Avoid:

```
print("API KEY:", api_key)
```

Use:

```
print("API key loaded")
```

if you need a safe diagnostic message.

---

25. Environment Variables vs `.env`

These are related but not identical.

Environment Variables	**.env File**
Stored in the process/system environment	Stored in a file
Available to applications	Used to provide local configuration
Common in production	Common during development
Can be configured by deployment platforms	Usually loaded using a package
No file required	Requires `.env` file

Think of `.env` as a **convenient local source for environment variables**.

---

26. os.environ vs os.getenv()

You can read variables using:

```
os.getenv("API_KEY")
```

or:

```
os.environ["API_KEY"]
```

There is an important difference.

os.getenv()

```
api_key = os.getenv("API_KEY")
```

If the variable is missing:

```
None
```

os.environ[]

```
api_key = os.environ["API_KEY"]
```

If the variable is missing, Python raises:

```
KeyError
```

Use `getenv()` when a missing value can be handled gracefully.

Use `os.environ[]` when the variable is required and you want missing configuration to fail immediately.

---

27. A Practical Configuration Pattern

A small application might use:

```
import os
from dotenv import load_dotenv

load_dotenv()

config = {
    "api_key": os.getenv("API_KEY"),
    "database_url": os.getenv("DATABASE_URL"),
    "debug": os.getenv("DEBUG", "false").lower() == "true",
    "port": int(os.getenv("PORT", "8000"))
}

if not config["api_key"]:
    raise RuntimeError("API_KEY is required")
```

Now the application has one configuration object.

---

28. Real World Example: haas.dev

Imagine a Python backend for haas.dev.

It needs:

```
DATABASE_URL
ADMIN_API_KEY
EMAIL_API_KEY
DEBUG
```

`PORT`

.env:

```
DATABASE_URL=postgresql://localhost/haas
ADMIN_API_KEY=secret_value
EMAIL_API_KEY=secret_value
DEBUG=true
PORT=8000
```

Configuration:

```
import os
from dotenv import load_dotenv

load_dotenv()

DATABASE_URL = os.getenv("DATABASE_URL")
ADMIN_API_KEY = os.getenv("ADMIN_API_KEY")
EMAIL_API_KEY = os.getenv("EMAIL_API_KEY")

DEBUG = os.getenv("DEBUG", "false").lower() == "true"
PORT = int(os.getenv("PORT", "8000"))
```

Application code:

```
def start_server():
    print(f"Starting server on port {PORT}")
```

The application knows how to work.

The environment decides **which configuration it should use**.

This separation becomes especially useful as a project grows.

---

29. The Mental Model

Think of environment variables like a **configuration box outside your application**.

Your program says:

`"I need an API key."`

The environment provides:

`API_KEY = actual_value`

Your program says:

`"I need the database URL."`

The environment provides:

`DATABASE_URL = actual_database`

Your code does not need to know where those values came from.

The important flow is:

```
Environment
    
Configuration
    
```

```
Python Application
    
Application Logic
```

---

30. A Better Problem Solving Workflow

When using environment variables:

Step 1: Identify configuration

Ask:

> What value changes between environments?

Examples:

```
API key
database URL
port
debug mode
```

Step 2: Decide whether it is sensitive

If it is a secret:

```
API_KEY
PASSWORD
SECRET_KEY
TOKEN
```

do not hardcode it.

Step 3: Create the variable

Example:

```
API_KEY=...
```

Step 4: Read it

```
api_key = os.getenv("API_KEY")
```

Step 5: Convert it if necessary

```
port = int(os.getenv("PORT", "8000"))
```

Step 6: Validate required values

```
if not api_key:
    raise RuntimeError("API_KEY is missing")
```

Step 7: Keep secrets out of Git

Use:

```
.env
```

and add it to:

```
.gitignore
```

---

31. Mini Project: Configuration Loader

Create:

```
config_project/
□
□□□ .env
□□□ .gitignore
□□□ config.py
□□□ app.py
```

.env

```
APP_NAME=haas.dev
DEBUG=true
PORT=8000
API_KEY=demo_key
```

config.py

```
import os
from dotenv import load_dotenv

load_dotenv()

APP_NAME = os.getenv("APP_NAME", "My App")
DEBUG = os.getenv("DEBUG", "false").lower() == "true"
PORT = int(os.getenv("PORT", "8000"))
API_KEY = os.getenv("API_KEY")

if not API_KEY:
    raise RuntimeError("API_KEY is required")
```

app.py

```
import config
```

```
print("Application:", config.APP_NAME)
print("Debug:", config.DEBUG)
print("Port:", config.PORT)
print("API key loaded:", bool(config.API_KEY))
```

Notice that the actual API key is never printed.

---

32. Five Minute Challenge

Create a Python program that reads:

```
APP_NAME
PORT
DEBUG
```

from environment variables.

Your program should:

1. Read all three values.
2. Convert `PORT` to an integer.
3. Convert `DEBUG` to a Boolean.
4. Provide a default port of `8000`.
5. Print the configuration without exposing any secret values.

Expected output:

```
Application: haas.dev
Port: 8000
Debug: True
```

---

33. Exercises

Exercise 1

Read:

`USERNAME`

from an environment variable.

Exercise 2

Read:

`PORT`

and convert it into an integer.

Exercise 3

Create a default value for:

`APP_NAME`

if it does not exist.

Exercise 4

Create a `.env` file containing:

```
API_KEY=
DEBUG=false
PORT=8000
```

Load it using `python-dotenv`.

Exercise 5

Write a configuration module that validates:

```
DATABASE_URL
API_KEY
```

and raises an error if either is missing.

---

34. Mini Quiz

1. What is an environment variable?

A. A Python function
B. A key-value configuration stored outside application code
C. A database table
D. A Python class

Answer: B

---

2. What does `os.getenv()` return if a variable does not exist?

A. `0`
B. `False`
C. `None`
D. `""`

Answer: C

---

3. What type does `os.getenv()` normally return?

A. Integer
B. String
C. Boolean
D. List

Answer: B

---

4. What package is commonly used to load `.env` files?

A. requests
B. pandas
C. python-dotenv
D. pathlib

Answer: C

---

5. Should real API keys normally be committed to Git?

A. Yes
B. No

Answer: B

---

35. Interview Questions

Beginner

1. What is an environment variable?

A configuration value provided outside the application's source code.

2. How do you read an environment variable in Python?

```
import os

value = os.getenv("NAME")
```

3. What happens when `os.getenv()` cannot find a variable?

It returns `None`, unless a default value is provided.

4. Why are environment variables useful?

They separate configuration and secrets from application code and make it easier to run the same code in different environments.

Intermediate

5. Why are environment variables strings?

Operating systems expose environment variables as text, so applications must convert them when another type is needed.

6. What is the difference between `os.getenv()` and `os.environ[]`?

`os.getenv()` returns `None` when the variable is missing, while `os.environ[]` raises `KeyError`.

7. What is a `.env` file?

A local configuration file commonly used to define environment variables during development.

8. Why should `.env` usually be added to `.gitignore`?

Because it may contain secrets such as API keys and passwords.

Advanced

9. Why should configuration validation happen when an application starts?

It allows missing or invalid configuration to be detected early instead of causing unexpected failures later.

10. Why should an application not print environment secrets?

Logs and debugging output can expose credentials to other people or systems.

---

36. Cheat Sheet

Import

```
import os
```

Read variable

```
os.getenv("NAME")
```

Read with default

```
os.getenv("PORT", "8000")
```

Required variable

```
os.environ["API_KEY"]
```

Convert to integer

```
int(os.getenv("PORT", "8000"))
```

Convert to Boolean

```
os.getenv("DEBUG", "false").lower() == "true"
```

Set variable for current process

```
os.environ["APP_NAME"] = "haas.dev"
```

Load `.env`

```
from dotenv import load_dotenv

load_dotenv()
```

Install `python-dotenv`

```
pip install python-dotenv
```

Ignore `.env`

```
.env
```

Template configuration

```
.env.example
```

---

37. Key Takeaways

- Environment variables store configuration outside application code.
- They are commonly used for secrets, API keys, database URLs, ports, and environment-specific settings.
- Python can read them with `os.getenv()`.
- Environment variable values are normally strings.
- Convert values when you need integers, floats, or Booleans.
- `os.getenv()` returns `None` when a variable is missing.
- `os.environ[]` raises `KeyError` when a variable is missing.
- `.env` files are commonly used for local development.
- `python-dotenv` can load `.env` values.
- Never casually commit real secrets to Git.
- Use `.env.example` to document required configuration.
- Validate required configuration when the application starts.
- Never expose secrets through logs or debug output.
- Environment variables allow the same application code to work across different environments.

The core idea:

```
Keep application logic in code.
Keep environment-specific configuration outside the code.
```

`Keep secrets out of the repository.`

Visit haas.dev for more resources and guides.

https://dev-roast-app.vercel.app