20 Python Anti-Patterns

Learn Python by knowing what not to do.

gevik.ai

20 Python Anti-Patterns Summary with pictures.

Emoji	Anti-Pattern	Description
🙊	AP-01: Ignoring Errors and Exceptions	Ignoring exceptions can lead to unexpected behavior
🏰	AP-02: Writing God Objects or Classes	Violates Single Responsibility Principle
⚠️	AP-03: Using 'eval' Unnecessarily	Security risks and code execution
🔒	AP-04: Hardcoding Passwords or Secrets	Security risks due to exposed credentials
🌍	AP-05: Using Global Variables	Makes code hard to understand and maintain
🧨	AP-06: Using Mutable Default Arguments	Leads to unexpected behavior and bugs
❓	AP-07: Wildcard Imports	Can lead to name conflicts
🗄️	AP-08: Not Using Context Managers	Resource leaks and unexpected behaviors
❗	AP-09: Comparing to Singletons with '=='	Error-prone, use is for comparison
🚦	AP-10: Returning Multiple Variable Types	Makes code less predictable
🧩	AP-11: Using Type Comments	Less readable than modern type hints
🌀	AP-12: Using Double Leading Underscores ('__')	Leads to name mangling, can be confusing
💥	AP-13: Unnecessary List Comprehensions	Consumes more memory, less readable
🎲	AP-14: Unnecessary Use of Generators	Makes code more complex
🌪️	AP-15: Excessive Nested Functions	Increases complexity and cognitive load
🕸️	AP-16: Misusing Default Arguments	Can create tangled and unexpected behaviors
😵💫	AP-17: Using camelCase	Not Pythonic, use snake_case
✂️	AP-18: Overusing Shortcuts like Lambda	Reduces readability for complex logic
🤷	AP-19: Not Commenting Your Code	Makes code harder to understand
⏳	AP-20: Using Outdated Version of Python	May lead to compatibility and security issues

AP-01

Ignoring Errors and Exceptions

Silently ignoring exceptions can lead to hidden bugs and make debugging difficult

Anti-pattern

```
try:
    x = 1 / 0
except:
    pass  # Ignoring the exception, hiding potential bugs
```

Proper Code

```
try:
    x = 1 / 0
except ZeroDivisionError as e:
    print(f"An error occurred: {e}")  # Handling the specific exception
```

AP-02

Writing God Objects or Classes

God objects make the code less maintainable and testable by combining unrelated responsibilities.

Anti-pattern

```
class Everything:
    # ...  # Combining too many responsibilities
```

Proper Code

```
class DatabaseHandler:
    # ...

class NetworkHandler:
    # ...  # Separate responsibilities into different classes
```

AP-03

Using ‘eval’ Unnecessarily

Using ‘eval’ can introduce security risks and code that is difficult to understand.

See bonus material on the security concern with ‘eval’ in the end.

Anti-pattern

```
user_input = "2 + 2"
result = eval(user_input)  # Unsafe usage of eval
```

Proper Code

```
user_input = "2 + 2"
result = int(user_input.split('+')[0]) + int(user_input.split('+')[1])
```

AP-04

Hardcoding Passwords or Secrets

I know this is a “doh”, but you’d be surprised how often I see this.

Anti-pattern

```
password = "SuperSecretPassword"  # Hardcoded sensitive information
authenticate(user="user", password=password)
```

Proper Code

```
from cryptography.fernet import Fernet
import json

def connect_to_database():
    key = load_encryption_key()
    cipher_suite = Fernet(key)

    with open('/path/to/secure/config.json', 'r') as file:
        encrypted_config = file.read()

    decrypted_config = cipher_suite.decrypt(encrypted_config.encode())
    config = json.loads(decrypted_config)

    username = config['database']['username']
    password = config['database']['password']

    connection = database.connect(username, password)
    return connection
```

AP-05

Using Global Variables

Yet another no-brainer. But there are Python developers who don’t write production ready code. This is for them.

Anti-pattern

```python
global_var = 5
def function():
    global global_var
    global_var += 10  # Modifying global variable
```

Proper Code

```python
def function(global_var):
    return global_var + 10  # Pass the variable as an argument
global_var = 5
global_var = function(global_var)
```

AP-06

Using Mutable Default Arguments

Mutable default arguments persist between function calls, leading to unexpected behavior.

Anti-pattern

```
def add_item(item, items=[]):  # Mutable default argument
    items.append(item)
    return items
```

Proper Code

```
def add_item(item, items=None):
    if items is None:
        items = []  # Use an immutable default argument
    items.append(item)
    return items
```

AP-07

Wildcard Imports

Wildcard imports can lead to name conflicts and make code harder to read.

Anti-pattern

```
from math import *  # It's unclear what's imported
```

Proper Code

```
import math  # Import the whole module or specific objects
math.sqrt(16)
```

```
import numpy as np

array = np.array([1, 2, 3])
```

```
from scipy import stats, integrate

result1 = stats.norm()
result2 = integrate.quad(lambda x: x**2, 0, 1)
```

AP-08

Not Using Context Managers to Handle Resources

Not using context managers can lead to resource leaks, where resources (e.g.; files, database connections, network connections) are not properly closed. This can cause issues such as performance degradation, unexpected errors, or system instability.

Anti-pattern

```
def read_file_bad(filename):
    file = open(filename, 'r')
    content = file.read()
    # Missing file.close() can lead to resource leak
    return content
```

Proper Code

```
def read_file_good(filename):
    with open(filename, 'r') as file:
        content = file.read()
        # file will be automatically closed when exiting this block
    return content
```

AP-09

Comparing to Singletons like None Using ‘==‘

‘==‘ checks if two objects are equivalent, not if they're the same object. Since ‘None’ is a singleton (i.e.; there's only one instance of ‘None’), it’s better to check identity vs equivalence. Classes can define custom __eq__ methods, may leading to unexpected behavior when comparing with None using ==.

Anti-pattern

```
class AlwaysEquals:
    def __eq__(self, other):
        return True


value = AlwaysEquals()

# Will not evaluate to True, even though you might expect it to
if value == None:
    print("This won't be printed")
```

Proper Code

```
if value is None:  # Correctly checks whether value is None
    print("This will be printed if value is None")
```

AP-10

Returning more than one variable type from Function calls

Returning inconsistent types makes code less predictable and harder to work with.

Anti-pattern

```
def function(flag):
    if flag:
        return 1
    else:
        return "Error"  # Inconsistent return types
```

Proper Code

```
def function(flag):
    if flag:
        return 1
    else:
        raise ValueError("Error") # Raise an exception for error handling
```

AP-11

Using type comments instead of type hints

Modern type hints are more readable and provide better tooling support (for linters, IDEs, type checkers, and code completion suggestion) than type comments. They further provide forward compatibility.

Anti-pattern

```
def add(a, b):  # type: (int, int) -> int  # Old style type comments
    return a + b
```

Proper Code

```
def add(a: int, b: int) -> int:  # Modern type hints
    return a + b
```

AP-12

Using double leading underscores (__var)

Use '_' to indicate a variable is used internally within a class.

Use '__' for name mangling ***only*** when you need to avoid name clashes with subclasses that are out of your control. Even then, this should be used sparingly and documented clearly.

Anti-pattern

```python
class MyClass:
    def __init__(self):
        self.__secret_var = 5  # Name mangling can be confusing
```

Proper Code

```python
class MyClass:
    def __init__(self):
        self._secret_var = 5  # Single underscore for protected variables
```

AP-13

Unnecessary List Comprehensions

Unnecessary list comprehensions can consume more memory and make code less readable.

Anti-pattern

```python
# List comprehension creates a list in memory
squares_sum = sum([x ** 2 for x in range(10)])
```

Proper Code

```python
# Generator expression doesn't create a list in memory
squares_sum = sum(x ** 2 for x in range(10))
```

AP-14

Unnecessary use of generators

Overusing generators when simple loops would suffice can make code more complex. If you are working with large data sets that might not fit in memory, generators are an excellent tool. But for small ranges or simple tasks, using a generator might add unnecessary complexity.

Anti-pattern

```python
def square_numbers():
    for i in range(10):
        yield i ** 2

for square in square_numbers():
    print(square)
```

Proper Code

```python
for i in range(10):
    print(i ** 2)
```

AP-15

Excessive use of nested functions or conditionals

Deeply nested code is harder to read and maintain.

Anti-pattern

```
def outer():
    def inner():
        def inner_inner():
            pass  # Too much nesting
```

Proper Code

```
def outer():
    pass

def inner():
    pass

def inner_inner():
    pass  # Separate functions to reduce complexity
```

AP-16

Missing default arguments

Misusing default arguments, especially with mutable defaults, can lead to unexpected behavior.

Anti-pattern

```
def my_function(a=[]):  # Mutable default value
    a.append(5)
    return a
```

Proper Code

```
def my_function(a=None):
    if a is None:
        a = []  # Immutable default value
    a.append(5)
    return a
```

AP-17

Using CamelCase in Functions and Variable names

This may be a religious argument. CamelCase is used in other languages, but Pythoneers want us to follow PEP 8 naming conventions.

Anti-pattern

```
def MyFunction():
    myVariable = 10  # Inconsistent naming
```

Proper Code

```
def my_function():
    my_variable = 10  # Following PEP 8 naming conventions
```

AP-18

Overusing shortcuts like Lambda functions.

This may be another religious argument. However, while shortcuts can make code concise, overusing them may reduce readability, especially when defining a reusable function is more appropriate than peppering lambda's everywhere in your code.

Anti-pattern

```
result = (lambda x: x*2)(10)  # Overuse of lambda
```

Proper Code

```
def double(x):
    return x * 2  # Defining a proper function enhances readability

result = double(10)
```

AP-19

Not commenting your code

Now this applies to every coding language but for completeness, no matter how readable a language is, commenting code is an absolute must.

AP-20

Using outdated python versions

Outdated versions may have security risks and lack the benefits of newer features.

Bonus: The good and the bad of 'eval'

'eval' security risks explained:

`eval` takes a string and evaluates it as an expression. This can lead to serious security risks if not handled with extreme care, especially if the input string can be influenced by an untrusted source. Here's why:

1. **Arbitrary Code Execution**: If an attacker can influence the string being passed to `eval`, they can execute any arbitrary code. This can include code that deletes files, sends data to a remote server, or any other malicious actions that the Python process has the permissions to carry out.

2. **Information Disclosure**: An attacker could use `eval` to access sensitive information. For instance, they might craft a string that references or calls functions that expose private or secure information.

3. **Denial of Service (DoS) Attacks**: By passing specially crafted strings to `eval`, an attacker might cause the program to consume excessive resources, leading to slowdowns or crashes, thus denying service to legitimate users.

4. **Bypassing Security Controls**: If you're using `eval` in a context where certain operations are supposed to be restricted (such as in a sandboxed environment), an attacker might craft input that escapes those restrictions, leading to unauthorized activities.

5. **Code Injection**: This is a specific form of arbitrary code execution where the attacker can inject malicious code into your application. It's especially concerning if the code is persistently stored and executed on other users' sessions as well.

Here's an example to illustrate the risk:

```
user_input = input("Enter a mathematical expression: ")
result = eval(user_input)
```

If an attacker enters something like `__import__('os').system('rm -rf /')`, `eval` will execute the command, potentially deleting all files on the system (if run with sufficient permissions).

Because of these and other risks, it's generally best to avoid `eval` whenever possible, especially with untrusted input. If you need to evaluate mathematical expressions from user input, consider using a library designed for safely parsing and evaluating such expressions, or write specific parsing code that tightly controls what kinds of input are allowed.

When should you use 'eval'

The \`**eval**\` function in Python, despite its potential security risks, exists for legitimate use cases, especially in controlled environments where the inputs are well-understood and trusted. Here are some reasons why it's included in the language:

1. **Rapid Prototyping**: \`eval\` can be useful for quickly testing ideas and prototyping code. It allows dynamic execution of code from strings, which can be handy in development environments.

2. **Dynamic Code Generation**: In some advanced programming scenarios, there might be a need to dynamically generate and execute code based on certain conditions. \`**eval**\` allows for this kind of flexibility.

3. **Interactive Shells and Tools**: Tools like debuggers, REPLs (Read-Eval-Print Loop), and other interactive shells can use \`**eval**\` to interpret and execute user commands.

4. **Scripting and Automation**: In controlled environments, \`**eval**\` can be used in scripting and automation tasks, where you know exactly what kind of input will be provided.

5. **Educational Purposes** \`**eval**\` might be used in educational settings where the instructor wants to demonstrate how interpretation and evaluation work in Python.

6. **Custom DSLs (Domain-Specific Languages)**: \`**eval**\` can be used to interpret custom languages or config files within a well-controlled domain.

While \`**eval**\` can be powerful and useful in these and other controlled situations, its misuse, especially with untrusted or uncontrolled input, can lead to the significant security risks mentioned earlier. Therefore, it should be used with extreme caution, and alternatives should be considered whenever possible, particularly in production code or when dealing with untrusted inputs.

Learning by Teaching

Learn Python by knowing what not to do.

gevik.ai