

# Anti-Patterns

O que são e como se livrar deles em Python

Caio Carrara  
eu@caiocarrara.com.br

# Caio Carrara

Tech Lead @ **Loadsmart**

- Desenvolvedor de Software
    - Loadsmart
    - RedHat
    - Olist
    - ThoughtWorks
  - Pythonista
  - Tech Lead
-

# Anti-patterns

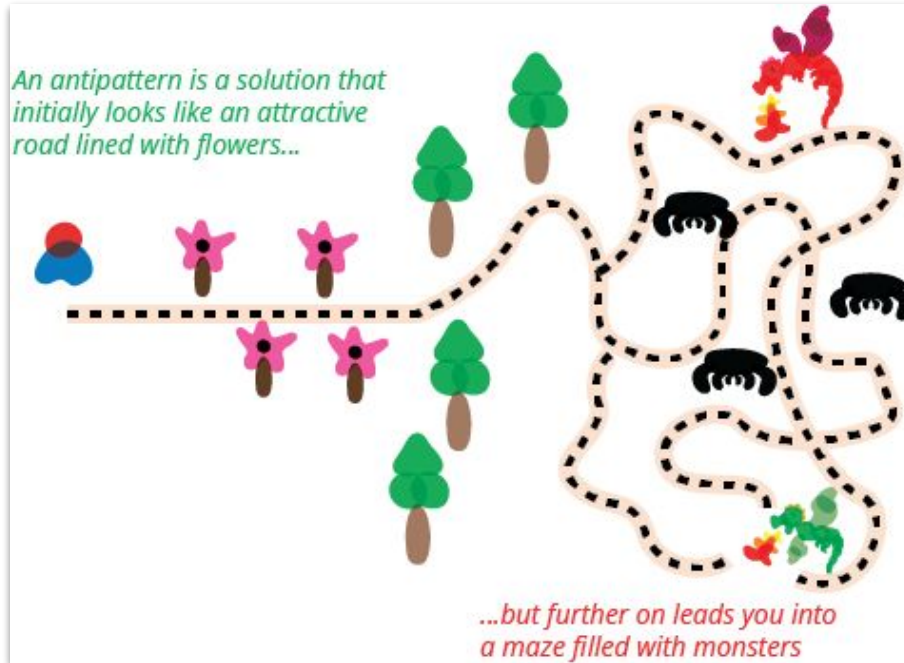


# O que são anti-patterns

“Um antipadrão é como um padrão, exceto que ele parece somente superficialmente como uma solução, mas na verdade não é.”

- Andrew Koenig

# Anti-patterns



# O que são anti-patterns

Anti-patterns podem ser até mais perigosos do que erros tradicionais



# Python Anti-Patterns

- Acessar membros internos (protegidos) de uma classe

## Anti-pattern

```
class Rectangle(object):
    def __init__(self, width, height):
        self._width = width
        self._height = height

r = Rectangle(5, 6)
# direct access of protected member
print("Width: {}".format(r._width))
```

## Best practice

If you are absolutely sure that you need to access the protected member from the outside, do the following:

- Make sure that accessing the member from outside the class does not cause any inadvertent side effects.
- Refactor it such that it becomes part of the public interface of the class.

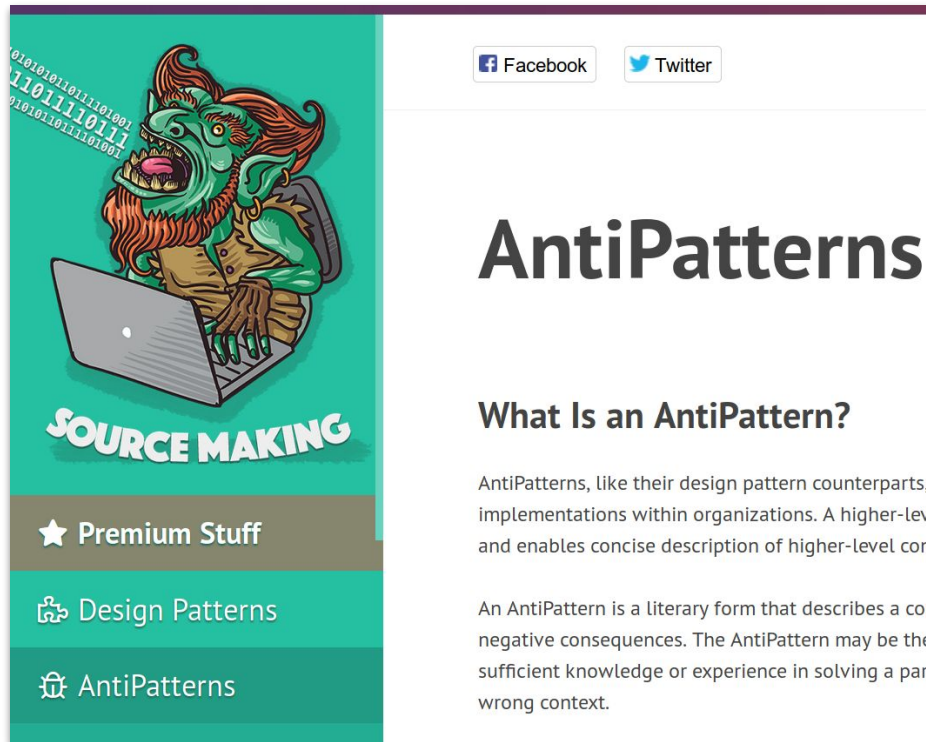
**Por que aprender anti-patterns?**





# Anti-Patterns - para saber mais

<https://sourcemaking.com/antipatterns>



The image shows the header of a web article titled 'AntiPatterns'. On the left is a vertical sidebar with a teal background. It features a cartoon illustration of a green, bearded creature with horns and a backpack, sitting at a laptop. Above the creature are lines of binary code (0s and 1s). Below the illustration, the text 'SOURCE MAKING' is written in a stylized font. Further down the sidebar are three menu items: '★ Premium Stuff', '🔗 Design Patterns', and '🔗 AntiPatterns'. The main content area on the right has a white background. At the top right, there are social media sharing buttons for Facebook and Twitter. The main title 'AntiPatterns' is displayed in a large, bold, black font. Below the title is a sub-heading 'What Is an AntiPattern?' followed by two paragraphs of text.

Facebook Twitter

## AntiPatterns

### What Is an AntiPattern?

AntiPatterns, like their design pattern counterparts, are implemented within organizations. A higher-level concept and enables concise description of higher-level concepts.

An AntiPattern is a literary form that describes a common negative consequence. The AntiPattern may be the result of insufficient knowledge or experience in solving a particular problem in a wrong context.

# Python Anti-Patterns

## The Little Book of Python Anti-Patterns



Welcome, fellow Pythoneer! This is a small book of Python **anti-patterns** and **worst practices**.

Learning about these anti-patterns will help you to avoid them in your own code and make you a better programmer (hopefully). Each pattern comes with a small description, examples and possible solutions. You can check many of them for free against your project at [QuantifiedCode](#).

# Python Anti-Patterns

- Corretude
- Manutenabilidade
- Legibilidade
- Segurança
- Performance\*
- *Django*

## Python Anti-Patterns

Search docs

✓ Correctness

✚ Maintainability

👁 Readability

🔒 Security

🚀 Performance

📖 Django

## Index Of Patterns

Here's the full index of all anti-patterns in this book.

### ✓ Correctness

[Accessing a protected member from outside the class](#)

[Assigning a `lambda` expression to a variable](#)

[Assigning to built-in function](#)

[Bad except clauses order](#)

[Bad first argument given to `super\(\)`](#)

[`else` clause on loop without a `break` statement](#)

[`\_exit` must accept 3 arguments: type, value, traceback](#)

[Explicit return in `\_\_init\_\_`](#)

[`future` import is not the first non-docstring statement](#)

[Implementing Java-style getters and setters](#)

[Indentation contains mixed spaces and tabs](#)

[Indentation contains tabs](#)

[Method could be a function](#)

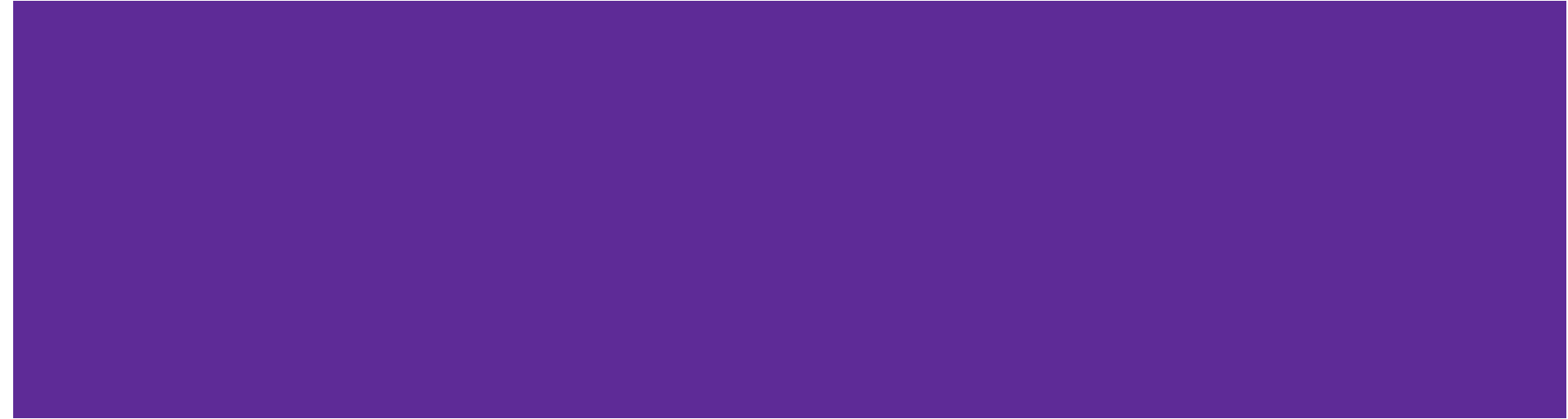
[Method has no argument](#)

[Missing argument to `super\(\)`](#)

[Using a mutable default value as an argument](#)

[No exception type\(s\) specified](#)

# Corretude



# Python Anti-Patterns - Corretude

- Atribuição para built-ins

## Anti-pattern

In the code below, the `list` built-in is overwritten. This makes it impossible, to use `list` to define a variable as a list. As this is a very concise example, it is easy to spot what the problem is. However, if there are hundreds of lines between the assignment to `list` and the assignment to `cars`, it might become difficult to identify the problem.

```
# Overwriting built-in 'list' by assigning values to a variable call  
list = [1, 2, 3]  
# Defining a list 'cars', will now raise an error  
cars = list()  
# Error: TypeError: 'list' object is not callable
```

# Python Anti-Patterns - Corretude

- Atribuição para built-ins

## Best practice

Unless you have a very specific reason to use variable names that have the same name as built-in functions, it is recommended to use a variable name that does not interfere with built-in function names.

```
# Numbers used as variable name instead of 'list'  
numbers = [1, 2, 3]  
# Defining 'cars' as list, will work just fine  
cars = list()
```

# Python Anti-Patterns - Corretude

- Má ordenação de *except*

## Anti-pattern

The code below performs a division operation that results in a `ZeroDivisionError`. The code contains an `except` clause for this type of error, which would be really useful because it pinpoints the exact cause of the problem. However, the `ZeroDivisionError` exception clause is unreachable because there is a `Exception` exception clause placed before it. When Python experiences an exception, it will linearly test each exception clause and execute the first clause that matches the raised exception. The match does not need to be identical. So long as the raised exception is a sub class of the exception listed in the exception clause, then Python will execute that clause and will skip all other clauses. This defeats the purpose of exception clauses, which is to identify and handle exceptions with as much precision as possible.

```
try:
    5 / 0
except Exception as e:
    print("Exception")
# unreachable code!
except ZeroDivisionError as e:
    print("ZeroDivisionError")
```

# Python Anti-Patterns - Corretude

- Má ordenação de *except*

## Best practice

### Move sub class exception clause before its ancestor's clause

The modified code below places the `ZeroDivisionError` exception clause in front of the `Exception` exception clause. Now when the exception is triggered the `ZeroDivisionError` exception clause will execute, which is much more optimal because it is more specific.

```
try:
    5 / 0
except ZeroDivisionError as e:
    print("ZeroDivisionError")
except Exception as e:
    print("Exception")
```



# Python Anti-Patterns - Corretude

- Valores mutáveis como padrão em argumentos

## Anti-pattern

A programmer wrote the `append` function below under the assumption that the `append` function would return a new list every time that the function is called without the second argument. In reality this is not what happens. The first time that the function is called, Python creates a persistent list. Every subsequent call to `append` appends the value to that original list.

```
def append(number, number_list=[]):  
    number_list.append(number)  
    print(number_list)  
    return number_list  
  
append(5) # expecting: [5], actual: [5]  
append(7) # expecting: [7], actual: [5, 7]  
append(2) # expecting: [2], actual: [5, 7, 2]
```

# Python Anti-Patterns - Corretude

- Valores mutáveis como padrão em argumentos

## Best practice

### Use a sentinel value to denote an empty list or dictionary

If, like the programmer who implemented the `append` function above, you want the function to return a new, empty list every time that the function is called, then you can use a [sentinel value](#) to represent this use case, and then modify the body of the function to support this scenario. When the function receives the sentinel value, it knows that it is supposed to return a new list.

```
# the keyword None is the sentinel value representing empty list
def append(number, number_list=None):
    if number_list is None:
        number_list = []
    number_list.append(number)
    print(number_list)
    return number_list

append(5) # expecting: [5], actual: [5]
append(7) # expecting: [7], actual: [7]
append(2) # expecting: [2], actual: [2]
```

# Python Anti-Patterns - Corretude

- Nenhuma exceção especificada

## Anti-pattern

```
def divide(a, b):  
  
    try:  
        result = a / b  
    except:  
        result = None  
  
    return result
```

# Python Anti-Patterns - Corretude

- Nenhuma exceção especificada

## Best practice

Handle exceptions with Python's built in [exception types](#).

```
def divide(a, b):

    result = None

    try:
        result = a / b
    except ZeroDivisionError:
        print("Type error: division by 0.")
    except TypeError:
        # E.g., if b is a string
        print("Type error: division by '{0}'.".format(b))
    except Exception as e:
        # handle any other exception
        print("Error '{0}' occurred. Arguments {1}.".format(e.message))
    else:
        # Executes if no exception occurred
        print("No errors")
    finally:
        # Executes always
        if result is None:
            result = 0

    return result
```

# Manutenibilidade



# Python Anti-Patterns - Manutenibilidade

- *Wildcard imports*

## Anti-pattern

The following code imports everything from the `math` built-in Python module.

```
# wildcard import = bad
from math import *
```

## Best practices

### Make the `import` statement more specific

The `import` statement should be refactored to be more specific about what functions or variables it is using from the `math` module. The modified code below specifies exactly which module member it is using, which happens to be `ceil` in this example.

```
from math import ceil
```

# Python Anti-Patterns - Manutenibilidade

- Abrir arquivos sem *context manager*

## Anti-pattern

The code below does not use `with` to open a file. This code depends on the programmer remembering to manually close the file via `close()` when finished. Even if the programmer remembers to call `close()` the code is still dangerous, because if an exception occurs before the call to `close()` then `close()` will not be called and the memory issues can occur, or the file can be corrupted.

```
f = open("file.txt", "r")
content = f.read()
1 / 0 # ZeroDivisionError
# never executes, possible memory issues or file corruption
f.close()
```

# Python Anti-Patterns - Manutenibilidade

- Abrir arquivos sem *context manager*

## Best practice

### Use `with` to open a file

The modified code below is the safest way to open a file. The `file` class has some special built-in methods called `__enter__()` and `__exit__()` which are automatically called when the file is opened and closed, respectively. Python guarantees that these special methods are always called, even if an exception occurs.

```
with open("file.txt", "r") as f:  
    content = f.read()  
    # Python still executes f.close() even though an exception occur  
1 / 0
```



# Python Anti-Patterns - Manutenibilidade

- Retornar mais de um tipo de dado

## Anti-pattern

In the code below, the function `get_secret_code()` returns a secret code when the code calling the function provides the correct password. If the password is incorrect, the function returns `None`. This leads to hard-to-maintain code, because the caller will have to check the type of the return value before proceeding.

```
def get_secret_code(password):
    if password != "bicycle":
        return None
    else:
        return "42"

secret_code = get_secret_code("unicycle")

if secret_code is None:
    print("Wrong password.")
else:
    print("The secret code is {}".format(secret_code))
```

# Python Anti-Patterns - Manutenabilidade

- Retornar mais de um tipo de dado

## Best practice

### Raise an exception when an error is encountered or a precondition is unsatisfied

When invalid data is provided to a function, a precondition to a function is not satisfied, or an error occurs during the execution of a function, the function should not return any data. Instead, the function should raise an exception. In the modified version of `get_secret_code()` shown below, `ValueError` is raised when an incorrect value is given for the `password` argument.

```
def get_secret_code(password):
    if password != "bicycle":
        raise ValueError
    else:
        return "42"

try:
    secret_code = get_secret_code("unicycle")
    print("The secret code is {}".format(secret_code))
except ValueError:
    print("Wrong password.")
```

# Legibilidade



# Python Anti-Patterns - Legibilidade

- Pedir permissão ao invés de perdão  
(EAFP - *easier to ask for forgiveness than permission*)

## Anti-pattern

The code below uses an `if` statement to check if a file exists before attempting to use the file. This is not the preferred coding style in the Python community. The community prefers to assume that a file exists and you have access to it, and to catch any problems as exceptions.

```
import os

# violates EAFP coding style
if os.path.exists("file.txt"):
    os.unlink("file.txt")
```

# Python Anti-Patterns - Legibilidade

- Pedir permissão ao invés de perdão  
(EAFP - *easier to ask for forgiveness than permission*)

## Best practice

### Assume the file can be used and catch problems as exceptions

The updated code below is a demonstration of the EAFP coding style, which is the preferred style in the Python community. Unlike the original code, the modified code below simply assumes that the needed file exists, and catches any problems as exceptions. For example, if the file does not exist, the problem will be caught as an `OSError` exception.

```
import os

try:
    os.unlink("file.txt")
# raised when file does not exist
except OSError:
    pass
```

# Python Anti-Patterns - Legibilidade

- `map()` ou `filter()` ao invés de *List Comprehensions*

## Anti-pattern

The code below defines a list, and then uses `map()` to create a new list of doubled values from the first list.

```
values = [1, 2, 3]
doubles = map(lambda x: x * 2, values)
```

## Best practice

**Use list comprehension instead of `map()`**

In the modified code below, the code uses a list comprehension to create a new list of doubled values from the first list. Although this is functionally equivalent, it is generally agreed to be more concise and easier to read.

```
values = [1, 2, 3]
doubles = [x * 2 for x in values]
```

# Python Anti-Patterns - Legibilidade

- Não usar o método items() de dicionários

## Anti-pattern

The code below defines a for loop that iterates over a dictionary named `d`. For each loop iteration Python automatically assigns the value of `key` to the name of the next key in the dictionary. Inside of the `for` loop the code uses `key` to access the value of each key of the dictionary. This is a common way for iterating over a dictionary, but it is not the preferred way in Python.

```
d = {"first_name": "Alfred", "last_name": "Hitchcock"}

for key in d:
    print("{} = {}".format(key, d[key]))
```

# Python Anti-Patterns - Legibilidade

- Não usar o método `items()` de dicionários

## Best-practice

### Use `items()` to iterate across dictionary

The updated code below demonstrates the Pythonic style for iterating through a dictionary. When you define two variables in a `for` loop in conjunction with a call to `items()` on a dictionary, Python automatically assigns the first variable as the name of a key in that dictionary, and the second variable as the corresponding value for that key.

```
d = {"first_name": "Alfred", "last_name": "Hitchcock"}

for key, val in d.items():
    print("{} = {}".format(key, val))
```



**Segurança**



# Python Anti-Patterns - Segurança

- Uso de *eval* ou *exec*

## Anti-pattern

### Program uses `exec` to execute arbitrary Python code

The sample code below composes a literal string containing Python code and then passes that string to `exec` for execution. This is an indirect and confusing way to program in Python.

```
s = "print(\"Hello, World!\")"
exec s
```

## Best practice

### Refactor the code to avoid `exec`

In most scenarios, you can easily refactor the code to avoid the use of `exec`. In the example below, the use of `exec` has been removed and replaced by a function.

```
def print_hello_world():
    print("Hello, World!")

print_hello_world()
```

# Anti-Patterns

O que são e como se livrar deles  
em Python

- Anti-patterns são mais do que simples erros
  - Há diversos tipos em Python
    - Corretude
    - Manutenibilidade
    - Legibilidade
    - Segurança
    - Performance
  - É importante que saibamos como identificar e corrigir
-

# Obrigado

Anti-patterns - o que são e como se livrar deles em Python

Caio Carrara

[eu@caiocarrara.com.br](mailto:eu@caiocarrara.com.br)

[speakerdeck.com/cacarrara](https://speakerdeck.com/cacarrara)

# Referências

<https://martinfowler.com/bliki/AntiPattern.html>

<https://docs.quantifiedcode.com/python-anti-patterns/>

<https://sourcemaking.com/antipatterns>

<https://realpython.com/the-most-diabolical-python-antipattern/>