Python Anti-Patterns

The Little Book of Python Anti-Patterns and Worst Practice

QuantifiedCode

Contents

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.

You can also download this book as a PDF.

Why did we write this?

Short answer: We think that you can learn as much from reading bad code as you can from reading good one.

Long answer: There is an overwhelming amount of Python books that show you how to do things by focusing on best practices and examples of good code. There are only very few books out there that show you how **not** to do things. We wanted to change that by providing you with an **anti-book** that teaches you things which you should **never** do in practice.

Who are we?

We're QuantifiedCode, a Berlin-based startup. Our mission is to help programmers write better code! Our first product is an online tool for automated, data-driven code review. When building this tool we learned a lot about code quality in Python and decided to compile our knowledge into this book.

How is this book organized?

This book contains anti- and migrations pattern for Python and for popular Python frameworks, such as Django. We categorized the patterns as follows:

- **Correctness**: Anti-patterns that will literally break your code or make it do the wrong things.
- **Maintainability**: Anti-patterns that will make your code hard to maintain or extend.
- **Readability**: Anti-patterns that will make your code hard to read or understand.
- **Performance**: Anti-patterns that will unnecessarily slow your code down.
- **Security**: Anti-patterns that will pose a security risk to your program.
- **Migration**: Patterns that help you migrate faster to new versions of a framework

Some patterns can belong in more than one category, so please don't take the choice that we've made too serious. If you think a pattern is grossly misplaced in its category, feel free to create an issue on Github.

References

Whenever we cite content from another source we tried including the link to the original article on the bottom of the page. If you should have missed one, please feel free to add it and make a pull request on Github. Thanks!

Licensing

Contributing

If you think this collection can be improved or extended, please contribute! You can do this by simply forking our Github project and sending us a pull request once you're done adding your changes. We will review and merge all pull requests as fast as possible and be happy to include your name on the list of authors of this document.

We would also like to thank all contributors to this book for their effort. A full list of contributors can be found at Github.

List of Maintainers

If you have any questions concerning this project, please contact one of the maintainers:

- Andreas Dewes
- Christoph Neumann

Index Of Patterns

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

1 Correctness

1.1 Accessing a protected member from outside the class

Accessing a protected member (a member prefixed with _) of a class from outside that class usually calls for trouble, since the creator of that class did not intend this member to be exposed.

1.1.1 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: {:d}".format(r._width))
```

1.1.2 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.

1.1.3 References

- PyLint - W0212, protected-access

1.2 Assigning a *lambda* expression to a variable

The sole advantage that a `lambda` expression has over a `def` is that the `lambda` can be anonymously embedded within a larger expression. If you are going to assign a name to a `lambda`, you are better off just defining it as a `def`.

From the PEP 8 Style Guide:

Yes:

```
def f(x): return 2*x
```

No:

```
f = lambda x: 2*x
```

The first form means that the name of the resulting function object is specifically 'f' instead of the generic '<lambda>'. This is more useful for tracebacks and string representations in general. The use of the assignment statement eliminates the sole benefit a lambda expression can offer over an explicit def statement (i.e. that it can be embedded inside a larger expression)

1.2.1 Anti-pattern

The following code assigns a `lambda` function which returns the double of its input to a variable. This is functionally identical to creating a `def`.

```
f = lambda x: 2 * x
```

1.2.2 Best practice

Use a `def` for named expressions

Refactor the `lambda` expression into a named `def` expression.

```
def f(x): return 2 * x
```

1.2.3 References

- PEP 8 Style Guide - Programming Recommendations
- Stack Overflow - Do not assign a lambda expression

1.3 Assigning to built-in function

Python has a number of built-in functions that are always accessible in the interpreter. Unless you have a special reason, you should neither overwrite these functions nor assign a value to a variable that has the same name as a built-in function. Overwriting a built-in might have undesired side effects or can cause runtime errors. Python developers usually use built-ins 'as-is'. If their behaviour is changed, it can be very tricky to trace back the actual error.

1.3.1 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 called 'list'
list = [1, 2, 3]
# Defining a list 'cars', will now raise an error
cars = list()
# Error: TypeError: 'list' object is not callable
```

1.3.2 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()
```

1.3.3 References

- Python Documentation: Built-in functions

1.4 Bad except clauses order

When an exception occurs, Python will search for the first exception clause which matches the exception type that occurred. It doesn't need to be an exact match. If the exception clause represents a base class of the raised exception, then Python considers that exception clause to be a match. E.g. if a `ZeroDivisionError` exception is raised and the first exception clause is `Exception`, then the `Exception` clause will execute because `ZeroDivisionError` is a sub class of `Exception`. Therefore, more specific exception clauses of sub classes should always be placed before the exception clauses of their base classes to ensure that exception handling is as specific and as helpful as possible.

1.4.1 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")
```

1.4.2 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")
```

1.4.3 References

- Pylint - E0701, bad-except-order

1.5 Bad first argument given to `super()`

`super()` enables you to access the methods and members of a parent class without referring to the parent class by name. For a single inheritance situation the first argument to `super()` should be the name of the current child class calling `super()`, and the second argument should be `self` (that is, a reference to the current object calling `super()`).

Note: This anti-pattern only applies to Python versions 2.x, see "Super in Python 3" at the bottom of the page for the correct way of calling `super()` in Python 3.x.

1.5.1 Anti-pattern

Python raises a `TypeError` when it attempts to execute the call to `super()` below. The first argument should be the name of the child class that is calling `super()`. The author of the code mistakenly provided `self` as the first argument.

```
class Rectangle(object):
    def __init__(self, width, height):
        self.width = width
        self.height = height
        self.area = width * height

class Square(Rectangle):
    def __init__(self, length):
        # bad first argument to super()
        super(self, Square).__init__(length, length)

s = Square(5)
print(s.area)  # does not execute
```

1.5.2 Best practice

Insert name of child class as first argument to `super()`

In the modified code below the author has fixed the call to `super()` so that the name of the child class which is calling `super()` (`Square` in this case) is the first argument to the method.

```
class Rectangle(object):
    def __init__(self, width, height):
        self.width = width
        self.height = height
        self.area = width * height

class Square(Rectangle):
    def __init__(self, length):
        # super() executes fine now
        super(Square, self).__init__(length, length)

s = Square(5)
print(s.area)  # 25
```

1.5.3 Super in Python 3

Python 3 adds a new simpler `super()`, which requires no arguments. The correct way to call `super()` in Python 3 code is as follows.

```
class Rectangle(object):
    def __init__(self, width, height):
        self.width = width
        self.height = height
        self.area = width * height

class Square(Rectangle):
    def __init__(self, length):
        # This is equivalent to super(Square, self).__init__(length, length)
        super().__init__(length, length)

s = Square(5)
print(s.area)  # 25
```

1.5.4 References

- Python Standard Library - super([type[, object-or-type]])
- Stack Overflow - What is a basic example of single inheritance using super()?
- Stack Overflow - Python super() inheritance and arguments needed
- PyLint - E1003, bad-super-call
- PEP 3135 - New Super

1.6 `else` clause on loop without a `break` statement

The `else` clause of a loop is executed when the loop sequence is empty. When a loop specifies no `break` statement, the `else` clause will always execute, because the loop sequence will eventually always become empty. Sometimes this is the intended behavior, in which case you can ignore this error. But most times this is not the intended behavior, and you should therefore review the code in question.

1.6.1 Anti-pattern

The code below demonstrates some potential unintended behavior that can result when a loop contains an `else` statement yet never specifies a `break` statement. `contains_magic_number()` iterates through a list of numbers and compares each number to the magic number. If the magic number is found then the function prints `The list contains the magic number`. If it doesn't then the function prints `This list does NOT contain the magic number`. When the code calls the function with a list of `range(10)` and a magic number of 5, you would expect the code to only print `The list contains the magic number`. However, the code also prints `This list does NOT contain the magic number`. This is because the `range(10)` list eventually becomes empty, which prompts Python to execute the `else` clause.

```
def contains_magic_number(numbers, magic_number):
    for i in numbers:
        if i == magic_number:
            print("This list contains the magic number")
    else:
        print("This list does NOT contain the magic number")

contains_magic_number(range(10), 5)
# This list contains the magic number.
# This list does NOT contain the magic number.
```

1.6.2 Best practices

Insert a `break` statement into the loop

If the `else` clause should not always execute at the end of a loop clause, then the code should add a `break` statement within the loop block.

```
def contains_magic_number(numbers, magic_number):
    for i in numbers:
        if i == magic_number:
            print("This list contains the magic number.")
            # added break statement here
            break
    else:
        print("This list does NOT contain the magic number.")

contains_magic_number(range(10), 5)
# This list contains the magic number.
```

1.6.3 References

- PyLint - W0120, useless-else-on-loop
- Python Standard Library - else Clauses on Loops

1.7 `__exit__` must accept 3 arguments: type, value, traceback

A `contextmanager` class is any class that implements the `__enter__` and `__exit__` methods according to the Python Language Reference's context management protocol. Implementing the context management protocol enables you to use the `with` statement with instances of the class. The `with` statement is used to ensure that setup and teardown operations are always executed before and after a given block of code. It is functionally equivalent to `try...finally` blocks, except that `with` statements are more concise.

For example, the following block of code using a `with` statement...

```
with EXPRESSION:
    BLOCK
```

... is equivalent to the following block of code using `try` and `finally` statements.

```
EXPRESSION.__enter__()
try:
    BLOCK
finally:
    EXPRESSION.__exit__(exception_type, exception_value, traceback)
```

In order for `__exit__` to work properly it must have exactly three arguments: `exception_type`, `exception_value`, and `traceback`. The formal argument names in the method definition do not need to correspond directly to these names, but they must appear in this order. If any exceptions occur while attempting to execute the block of code nested after the `with` statement, Python will pass information about the exception into the `__exit__` method. You can then modify the definition of `__exit__` to gracefully handle each type of exception.

1.7.1 Anti-pattern

The `__exit__` method defined in the `Rectangle` class below does not conform to Python's context management protocol. The method is supposed to take four arguments: `self`, exception type, exception value, and traceback. Because the method signature does not match what Python expects, `__exit__` is never called even though it should have been, because the method `divide_by_zero` creates a `ZeroDivisionError` exception.

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height
    def __enter__(self):
        print("in __enter__")
        return self
    def __exit__(self):
        # never called because
        # argument signature is wrong
        print("in __exit__")
    def divide_by_zero(self):
        # causes ZeroDivisionError exception
```

(continues on next page)

(continued from previous page)

```
        return self.width / 0

with Rectangle(3, 4) as r:
    r.divide_by_zero()
    # __exit__ should be called but isn't

# Output:
# "in __enter__"
# Traceback (most recent call last):
#   File "e0235.py", line 27, in <module>
#     r.divide_by_zero()
# TypeError: __exit__() takes exactly 1 argument (4 given)
```

1.7.2 Best practices

Modifying `__exit__` to accept four arguments ensures that `__exit__` is properly called when an exception is raised in the indented block of code following the `with` statement. Note that the argument names do not have to exactly match the names provided below. But they must occur in the order provided below.

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height
    def __enter__(self):
        print("in __enter__")
        return self
    def __exit__(self, exception_type, exception_value, traceback):
        print("in __exit__")
    def divide_by_zero(self):
        # causes ZeroDivisionError exception
        return self.width / 0

with Rectangle(3, 4) as r:
    # exception successfully pass to __exit__
    r.divide_by_zero()

# Output:
# "in __enter__"
# "in __exit__"
# Traceback (most recent call last):
#   File "e0235.py", line 27, in <module>
#     r.divide_by_zero()
```

1.7.3 References

- PyLint - E0235,unexpected-special-method-signature
- Python Language Reference - The with statement
- Python Language Reference - With Statement Context Managers
- Stack Overflow - Python with. . . as

1.8 Explicit return in __init__

`__init__` is a special Python method that is automatically called when memory is allocated for a new object. The sole purpose of `__init__` is to initialize the values of instance members for the new object. Using `__init__` to return a value implies that a program is using `__init__` to do something other than initialize the object. This logic should be moved to another instance method and called by the program later, after initialization.

1.8.1 Anti-pattern

The `__init__` method of the `Rectangle` class below attempts to return the area of the rectangle within the `__init__` method. This violates the rule of only using `__init__` to initialize instance members.

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height
        self.area = width * height
        # causes "Explicit return in __init__" error
        return self.area
```

1.8.2 Best practices

Remove the *return* statement from the `__init__` method

Remove the `return` statement in the `__init__` method that is returning a value.

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height
        self.area = width * height
        # return statement removed from here
```

Move the program logic to another instance method

There is no reason why the `Rectangle` class MUST return the area immediately upon initialization. This program logic should be moved to a separate method of the `Rectangle` class. The program can call the method later, after the object has successfully initialized.

```
class Rectangle(object):
    def __init__(self, width, height):
        self.width = width
        self.height = height
        self._area = width * height

    @property
    # moved the logic for returning area to a separate method
    def area(self):
        return self._area
```

Note that the class must inherit from `object` now, since the `property` decorator only works for new style classes.

1.8.3 References

- PyLint - E0101, return-in-init
- Python Language Reference - object.__init__(self[, ...])

1.9 `__future__` import is not the first non-docstring statement

The `__future__` module enables a module to use functionality that is mandatory in future Python versions. If it was possible to place the `__future__` module in the middle of a module, then that would mean that one half of the module could use the old Python functionality for a given feature, and the other half (after the `__future__` import) could use the new Python functionality of the feature. This could create many strange and hard-to-find bugs, so Python does not allow it.

1.9.1 Anti-pattern

The code below attempts to place a `__future__` import statement in the middle of the module. When Python encounters the `from __future__ import division` statement it raises a `SyntaxError` and halts execution. However, if the code were to execute, the first `print` statement would print out `1` (which is how the division operator behaves in Python versions 2 and below), but the second `print` statement would print out a decimal value, which is how the division operator functions in Python versions 3 and later. As you can see, this could create very strange behavior, so Python does not allow `__future__` import statements in the middle of a module. The module can use either version of the division operator, but it can't use both.

```
print(8 / 7)  # 1

# SyntaxError
from __future__ import division

# 1.1428571428571428
print(8 / 7)
```

1.9.2 Best practice

Remove `__future__` import

In the modified code below, the author decides that the module needs to use the old functionality of the division operator. The only solution in this case is to remove the `__future__` import statement from the module.

```
# removed __future__ import statement
print(8 / 7)  # 1
```

Place `__future__` import before all other statements

In the modified code below, the author decides that the module needs the new functionality of the division operator. The only solution then is to place the `__future__` import statement at the beginning of the module

```
from __future__ import division

# 1.1428571428571428
print(8 / 7)
```

1.9.3 References

- PyLint - W0410, misplaced-future
- Simeon Visser - How does 'from __future__ import ...' work?
- Python Standard Library - __future__

1.10 Implementing Java-style getters and setters

Python is not Java. If you need to set or get the members of a class or object, just expose the member publicly and access it directly. If you need to perform some computations before getting or setting the member, then use Python's built-in `property` decorator.

1.10.1 Anti-pattern

The programmer below comes to Python from a long career as a Java programmer. For every class member that he wants to expose publicly, he defines a `get` and `set` method for that member. This is common practice in Java, but is frowned upon in Python as a waste of time and a cause of unnecessary code.

```
class Square(object):
    def __init__(self, length):
        self._length = length
    # Java-style
    def get_length(self):
        return self._length
    # Java-style
    def set_length(self, length):
        self._length = length

r = Square(5)
r.get_length()
r.set_length(6)
```

1.10.2 Best practice

Access the members directly

In Python it is acceptable to simply access class or object members directly. The modified code below exposes the `length` member as a public member. This is signified by the fact that there is no underscore character at the beginning of the member name. The `get_length()` and `set_length()` methods are no longer necessary so they have been deleted.

```
class Square(object):
    def __init__(self, length):
        self.length = length
```

(continues on next page)

(continued from previous page)

```
r = Square(5)
r.length
r.length = 6
```

Use built-in `property` decorator

When a member needs to be slightly protected and cannot be simply exposed as a public member, use Python's `property` decorator to accomplish the functionality of getters and setters.

```
class Square(object):
    def __init__(self, length):
        self._length = length

    @property
    def length(self):
        return self._length

    @length.setter
    def length(self, value):
        self._length = value

    @length.deleter
    def length(self):
        del self._length

r = Square(5)
r.length  # automatically calls getter
r.length = 6  # automatically calls setter
```

1.10.3 References

- Python Built-in Functions - property
- dirtSimple - Python Is Not Java
- Stack Overflow - What's the Pythonic Way to use getters and setters?

1.11 Indentation contains mixed spaces and tabs

Per the PEP 8 Style Guide, all Python code should be consistently indented with 4 spaces, never tabs.

1.11.1 Anti-pattern

The following code mixes spaces and tabs for indentation. The `print("Hello, World!")` statement is indented with a tab. The `print("Goodybye, World!")` statement is indented with 4 spaces.

```
def print_hello_world():
# indented with tab
    print("Hello, World!")
def print_goodbye_world():
    # indented with 4 spaces
    print("Goodbye, World!")
```

1.11.2 Solutions

Consistently indent with spaces

All Python code should be consistently indented with 4 spaces.

```
def print_hello_world():
    print("Hello, World!")  # indented with 4 spaces
def print_goodbye_world():
    print("Goodbye, World!")  # indented with 4 spaces
```

1.11.3 References

- PEP 8 Style Guide - Tabs or Spaces?
- PEP 8 Style Guide - Indentation

1.12 Indentation contains tabs

Per the PEP 8 Style Guide, all Python code should be consistently indented with 4 spaces for each level of indentation, not tabs.

1.12.1 Anti-pattern

The following code uses tabs for indentation. Python code should be indented with four spaces for each level of indentation.

```
def print_hello_world():
    # indented with tab
    print("Hello, World!")
def print_goodbye_world():
    # indented with tab
    print("Goodbye, World!")
```

1.12.2 Best practice

Consistently indent with spaces

All Python code should be consistently indented with 4 spaces.

```
def print_hello_world():
    # indented with 4 spaces
    print("Hello, World!")
def print_goodbye_world():
    # indented with 4 spaces
    print("Goodbye, World!")
```

1.12.3 References

- PEP 8 Style Guide - Tabs or Spaces?
- PEP 8 Style Guide - Indentation

1.13 Method could be a function

When a method is not preceded by the `@staticmethod` or `@classmethod` decorators and does not contain any references to the class or instance (via keywords like `cls` or `self`), Python raises the `Method could be a function` error. This is not a critical error, but you should check the code in question in order to determine if this section of code really needs to be defined as a method of this class.

1.13.1 Anti-pattern

In the `Rectangle` class below the `area` method calculates the area of any rectangle given a width and a height.

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height
        self.area = width * height
    # should be preceded by @staticmethod here
    def area(width, height):
        return width * height
```

`area` causes the `Method could be a function` error because it is ambiguous. It does not reference the instance or class using the `self` or `cls` keywords and it is not preceded by the `@staticmethod` decorator.

Class method is not preceded by `@classmethod` decorator

In the `Rectangle` class below the `print_class_name` method prints the name of the class. Again, Python raises the `Method could be a function` error because the method does not reference any class members or methods and is not preceded by the `@classmethod` decorator.

Furthermore, the first argument of a class method must be a reference to the class itself.

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height
        self.area = width * height
    # should be preceded by @classmethod here
    # missing required first argument "cls"
    def print_class_name():
        print("class name: Rectangle")
```

1.13.2 Best practices

Add the `@staticmethod` decorator before the static method

All static methods must be preceded by the `@staticmethod` decorator.

```
class Rectangle:
    # clarifies that this is a static method and belongs here
    @staticmethod
    def area(width, height):
        return width * height
```

Add the `@classmethod` decorator before the class method

All class methods must be preceded by the `@classmethod` decorator. Furthermore, the first argument of any class method must be `cls`, which is a reference to the class itself.

```
class Rectangle:
    @classmethod
    def print_class_name(cls):
        # "class name: Rectangle"
        print("class name: {0}".format(cls))
```

1.13.3 References

- PyLint - R0201, no-self-use

1.14 Method has no argument

Unlike some programming languages, Python does not pass references to instance or class objects automatically behind the scenes. So the program must explicitly pass them as arguments whenever it wants to access any members of the instance or class within a method.

1.14.1 Anti-pattern

In the `Rectangle` class below the `area` method attempts to return the value of the `area` instance variable. However, `self.area` is undefined because a reference to the instance object has not been explicitly passed as an argument to the method.

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height
        self.area = width * height
    # missing first argument "self"
    def area():
        # self is undefined here
        return self.area
```

Class method is missing the `cls` keyword

The method `print_class_name` attempts to print the name of the class. However, to programmatically access a class name, a method needs to have a reference to the class object. This is accomplished by passing the keyword `cls` as the first argument to the method. Because `print_class_name` does not do this, its reference to `cls` in the body of the method is undefined.

```
class Rectangle:
    @classmethod
    # missing first argument "cls"
    def print_class_name():
        # cls is undefined here
        print("Hello, I am {0}!".format(cls))
```

The method `area` computes the value of any rectangle. Currently this method is ambiguous. It is defined as a method of the `Rectangle` class, yet it does not reference any instance or class members. The method needs to explicitly state that it is a static method via the `@staticmethod` decorator.

```
class Rectangle:
    # "@staticmethod" should be here
    def area(width, height):
        return width * height
```

1.14.2 Best practices

Add the `self` parameter to instance methods

To access the `area` member of a `Rectangle` instance the first argument of the `area` method needs to be a reference to the instance object, signified by the keyword `self`.

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height
        self.area = width * height
    # instance members now accessible because of "self"
    def area(self):
        return self.area
```

Add the `cls` parameter to class methods

To access the name of the class the `print_class_name` method needs to explicitly pass an argument to the class object. This is done by adding the keyword `cls` as the first argument of the method.

```
class Rectangle:
    @classmethod
    # class members now accessible, thanks to "cls"
    def print_class_name(cls):
        print("Hello, I am {0}!".format(cls))
```

Add the `@staticmethod` decorator to static methods

If the method is a static method that does not need access to any instance members, then the method should be preceded by the `@staticmethod` decorator. This improves readability by helping clarify that

the method should never rely on any instance members.

```
class Rectangle:
    # clarifies that the method does not need any instance members
    @staticmethod
    def area(width, height):
        return width * height
```

1.14.3 References

- PyLint - E0211, no-method-argument

1.15 Missing argument to `super()`

`super()` enables you to access the methods and members of a parent class without referring to the parent class by name. For a single inheritance situation the first argument to `super()` should be the name of the current child class calling `super()`, and the second argument should be `self`, that is, a reference to the current object calling `super()`.

Note: This error is only raised for Python versions 2.x which don't support new-style classes.

1.15.1 Anti-pattern

The author of the code below provides no arguments for the child class' call to `super()`. Python raises a `TypeError` at runtime because it expects at least 1 argument for `super()`.

```
class Rectangle(object):
    def __init__(self, width, height):
        self.width = width
        self.height = height
        self.area = width * height

class Square(Rectangle):
    def __init__(self, length):
        # no arguments provided to super()
        super().__init__(length, length)

s = Square(5)
print(s.area)  # does not execute
```

1.15.2 Best practice

Insert name of child class as first argument to `super()`

In the modified code below the author has fixed the call to `super()` so that the name of the child class which is calling `super()` (`Square` in this case) is the first argument to the method, and a reference to the object calling `super()` is the second argument.

```
class Rectangle(object):
    def __init__(self, width, height):
        self.width = width
        self.height = height
        self.area = width * height

class Square(Rectangle):
    def __init__(self, length):
        # super() executes fine now
        super(Square, self).__init__(length, length)

s = Square(5)
print(s.area)  # 25
```

1.15.3 References

- PyLint - E1004, missing-super-argument
- Python Standard Library - super([type[, object-or-type]])
- Stack Overflow - What is a basic example of single inheritance using super()?
- Stack Overflow - Python super() inheritance and arguments needed

1.16 Using a mutable default value as an argument

Passing mutable lists or dictionaries as default arguments to a function can have unforeseen consequences. Usually when a programmer uses a list or dictionary as the default argument to a function, the programmer wants the program to create a new list or dictionary every time that the function is called. However, this is not what Python does. The first time that the function is called, Python creates a persistent object for the list or dictionary. Every subsequent time the function is called, Python uses that same persistent object that was created from the first call to the function.

1.16.1 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]
```

1.16.2 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]
```

1.16.3 References

- PyLint - W0102, dangerous-default-value
- Stack Overflow - Hidden Features of Python

1.17 No exception type(s) specified

The function *divide* simply divides *a* by *b*. To avoid invalid calculations (e.g., a division by zero), a *try-except* block is added. This is valid and ensures that the function always returns a result. However, by securing your code with the try clause, you might hide actual programming errors, e.g., that you pass a string or an object as *b*, instead of a number. By not specifying an exception type, you not only hide this error but you also lose information about the error itself.

1.17.1 Anti-pattern

```
def divide(a, b):

    try:
        result = a / b
    except:
        result = None

    return result
```

1.17.2 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}' occured. Arguments {1}.".format(e.message, e.args))
    else:
        # Excecutes if no exception occured
        print("No errors")
    finally:
        # Executes always
        if result is None:
            result = 0

    return result
```

With this pattern, you are able to handle exceptions based on their actual exception-type. The first exception type that matches the current error is handled first. Thus, it is recommended to handle specific exception types first (e.g., ZeroDivisionError) and generic error types (e.g., Exception) towards the end of the try-except block.

Cleanup actions (optional): The *else*-clause executes only, if no exception occurred. It is useful to log the success of your code. The *finally*-block executes under all circumstances — no matter if an error occured or not. It is useful to clean up the *try-except* block.

1.17.3 Implement user defined exceptions

In addition to Python's standard exceptions, you can implement your own exception classes.

```
class DivisorTooSmallError(StandardError):
    def __init__(self, arg):
        self.args = arg


def divide(a, b):
    if b < 1:
        raise DivisorTooSmallError
    return a / b


try:
    divide(10, 0)
except DivisorTooSmallError:
    print("Unable to divide these numbers!")
```

1.17.4 References

- PyLint W0702, bare-except
- *Python Built-in Exceptions<https://docs.python.org/2/library/exceptions.html#exceptions.BaseException>*
- *Python Errors and Exceptions<https://docs.python.org/2/tutorial/errors.html>*

1.18 Not using `defaultdict()`

When a dict is created using `defaultdict()`, the value for each key in the dict will default to the value provided as the first argument of `defaultdict()`. This is more concise and less error-prone than manually setting the value of each key.

1.18.1 Anti-pattern

The code below defines an empty dict and then manually initializes the keys of the dict. Although there is nothing wrong with this code, there is a more concise and less error-prone way to achieve the same idea, as explained in the solution below.

```
d = {}

if "k" not in d:
    d["k"] = 6

d["k"] += 1

print(d["k"])  # 7
```

1.18.2 Best practice

Use `defaultdict()` to initialize dict keys

The modified code below uses `defaultdict` to initialize the dict. Whenever a new key is created, the default value for that key is 6. This code is functionally equivalent to the previous code, but this one is more concise and less error-prone, because every key automatically initializes to 6 with no work on the part of the programmer.

```
from collections import defaultdict

d = defaultdict(lambda : 6)
d["k"] += 1

print(d["k"])  # 7
```

1.18.3 References

- Python Standard Library - collections.defaultdict

1.19 Not using `else` where appropriate in a loop

The Python language provides a built-in `else` clause for `for` loops. If a `for` loop completes without being prematurely interrupted by a `break` or `return` statement, then the `else` clause of the loop is executed.

1.19.1 Anti-pattern

The code below searches a list for a magic number. If the magic number is found in the list, then the code prints `Magic number found`. If the magic number is not found, then the code prints `Magic number not found`.

The code uses a flag variable called `found` to keep track of whether or not the magic number was found in the list.

The logic in this code is valid; it will accomplish its task. But the Python language has built-in language constructs for handling this exact scenario and which can express the same idea much more concisely and without the need for flag variables that track the state of the code.

```
l = [1, 2, 3]
magic_number = 4
found = False

for n in l:
    if n == magic_number:
        found = True
        print("Magic number found")
        break

if not found:
    print("Magic number not found")
```

1.19.2 Best practice

Use `else` clause with `for` loop

In Python, you can declare an `else` loop in conjunction with a `for` loop. If the `for` loop iterates to completion without being prematurely interrupted by a `break` or `return` statement, then Python executes the `else` clause of the loop.

In the modified code below, the `for` loop will iterate through all three items in the list. Because the magic number is not contained in the list, the `if` statement always evaluates to `False`, and therefore the `break` statement is never encountered. Because Python never encounters a `break` statement while iterating over the loop, it executes the `else` clause.

The modified code below is functionally equivalent to the original code above, but this modified code is more concise than the original code and does not require any flag variables for monitoring the state of the code.

```
l = [1, 2, 3]
magic_number = 4

for n in l:
    if n == magic_number:
        print("Magic number found")
        break
else:
    print("Magic number not found")
```

Note: Since `else` on a `for` loop is so unintuitive and error-prone, even some experienced Python developers suggest not using this feature at all.

1.19.3 References

- Python Language Reference - else Clauses on Loops

1.20 Not using explicit unpacking

When you see multiple variables being defined followed by an assignment to a list (e.g. `elem0, elem1, elem2 = elems`, where `elem0`, `elem1`, and `elem2` are variables and `elems` is a list), Python will automatically iterate through the list and assign `elems[0]` to `elem0`, `elems[1]` to `elem1`, and so on.

1.20.1 Anti-pattern

The code below manually creates multiple variables to access the items in a list. This code is error-prone and unnecessarily verbose, as well as tedious to write.

```
elems = [4, 7, 18]

elem0 = elems[0]
elem1 = elems[1]
elem2 = elems[2]
```

1.20.2 Best practice

Use unpacking

The modified code below is functionally equivalent to the original code, but this code is more concise and less prone to error.

```
elems = [4, 7, 18]

elem0, elem1, elem2 = elems
```

1.21 Not using `get()` to return a default value from a dict

Frequently you will see code create a variable, assign a default value to the variable, and then check a dict for a certain key. If the key exists, then the value of the key is copied into the value for the variable. While there is nothing wrong this, it is more concise to use the built-in method `dict.get(key[, default])` from the Python Standard Library. If the key exists in the dict, then the value for that key is returned. If it does not exist, then the default value specified as the second argument to `get()` is returned. Note that the default value defaults to `None` if a second argument is not provided.

1.21.1 Anti-pattern

The code below initializes a variable called `data` to an empty string. Then it checks if a certain key called `message` exists in a dict called `dictionary`. If the key exists, then the value of that key is copied into the `data` variable.

Although there is nothing wrong with this code, it is verbose and inefficient because it queries the dictionary twice. The solution below demonstrates how to express the same idea in a more concise manner by using `dict.get(key[, default])`.

```
dictionary = {"message": "Hello, World!"}

data = ""

if "message" in dictionary:
    data = dictionary["message"]

print(data)  # Hello, World!
```

1.21.2 Best practice

Use `dict.get(key[, default])` to assign default values

The code below is functionally equivalent to the original code above, but this solution is more concise.

When `get()` is called, Python checks if the specified key exists in the dict. If it does, then `get()` returns the value of that key. If the key does not exist, then `get()` returns the value specified in the second argument to `get()`.

```
dictionary = {"message": "Hello, World!"}

data = dictionary.get("message", "")

print(data)  # Hello, World!
```

1.21.3 References

- Python Standard Library - dict.get

1.22 Not using `setdefault()` to initialize a dictionary

When initializing a dictionary, it is common to see a code check for the existence of a key and then create the key if it does not exist. Although there is nothing wrong with this, the exact same idea can be accomplished more concisely by using the built-in dictionary method `setdefault()`.

1.22.1 Anti-pattern

The code below checks if a key named `list` exists in a dictionary called `dictionary`. If it does not exist, then the code creates the key and then sets its value to an empty list. The code then proceeds to append a value to the list.

Although there is nothing wrong with this code, it is unnecessarily verbose. Later you will see how you can use `setdefault()` to accomplish the same idea more concisely.

```
dictionary = {}

if "list" not in dictionary:
    dictionary["list"] = []

dictionary["list"].append("list_item")
```

1.22.2 Best practice

Use `setdefault()` to initialize a dictionary

The modified code below uses `setdefault()` to initialize the dictionary. When `setdefault()` is called, it will check if the key already exists. If it does exist, then `setdefault()` does nothing. If the key does not exist, then `setdefault()` creates it and sets it to the value specified in the second argument.

```
dictionary = {}

dictionary.setdefault("list", []).append("list_item")
```

1.22.3 References

- Stack Overflow - Use cases for the setdefault dict method

2 Maintainability

A program is **maintainable** if it is easy to understand and modify the code even for someone that is unfamiliar with the code base.

Avoid the following anti-patterns to increase maintainability and avoid creating *spaghetti code*.

2.1 using wildcard imports (*from . . . import **)

When an import statement in the pattern of `from MODULE import *` is used it may become difficult for a Python validator to detect undefined names in the program that imported the module. Furthermore, as a general best practice, import statements should be as specific as possible and should only import what they need.

2.1.1 Anti-pattern

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

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

2.1.2 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
```

Import the whole module

There are some cases where making the `import` statement specific is not a good solution:

- It may be unpractical or cumbersome to create or maintain the list of objects to be imported from a module
- A direct import would bind to the same name as that of another object (e.g. from asyncio import TimeoutError)
- The module that the object is imported from would provide valuable contextual information if it is right next to the object when it's used.

In these cases, use one of these idioms:

```
import math
x = math.ceil(y)

# or

import multiprocessing as mp
pool = mp.Pool(8)
```

2.1.3 References

- Stack Overflow - Importing Modules
- Stack Overflow - 'import module' or 'from module import'

2.2 Not using `with` to open files

In Python 2.5, the `file` class was equipped with special methods that are automatically called whenever a file is opened via a `with` statement (e.g. `with open("file.txt", "r") as file`). These special methods ensure that the file is properly and safely opened and closed.

2.2.1 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()
```

2.2.2 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 occurs
    1 / 0
```

2.2.3 References

effbot - Understanding Python's with statement

2.3 Returning more than one variable type from function call

If a function that is supposed to return a given type (e.g. list, tuple, dict) suddenly returns something else (e.g. `None`) the caller of that function will always need to check the type of the return value before proceeding. This makes for confusing and complex code. If the function is unable to produce the supposed return value it is better to raise an exception that can be caught by the caller instead.

2.3.1 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))
```

2.3.2 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.")
```

2.4 Using the `global` statement

Global variables are dangerous because they can be simultaneously accessed from multiple sections of a program. This frequently results in bugs. Most bugs involving global variables arise from one function reading and acting on the value of a global variable before another function has the chance to set it to an appropriate value.

Global variables also make code difficult to read, because they force you to search through multiple functions or even modules just to understand all the different locations where the global variable is used and modified.

2.4.1 Examples

The code below uses global variables and a function to compute the area and perimeter of a rectangle. As you can see, even with two functions it becomes difficult to keep track of how the global variables are used and modified.

```
WIDTH = 0 # global variable
HEIGHT = 0 # global variable

def area(w, h):
    global WIDTH # global statement
    global HEIGHT # global statement
    WIDTH = w
    HEIGHT = h
    return WIDTH * HEIGHT

def perimeter(w, h):
    global WIDTH # global statement
    global HEIGHT # global statement
    WIDTH = w
    HEIGHT = h
    return ((WIDTH * 2) + (HEIGHT * 2))

print("WIDTH:" , WIDTH) # "WIDTH: 0"
print("HEIGHT:" , HEIGHT) # "HEIGHT: 0"

print("area():" , area(3, 4)) # "area(): 12"

print("WIDTH:" , WIDTH) # "WIDTH: 3"
print("HEIGHT:" , HEIGHT) # "HEIGHT: 4"
```

2.4.2 Solutions

Encapsulate the global variables into objects

One common solution for avoiding global variables is to create a class and store related global variables as members of an instantiated object of that class. This results in more compact and safer code.

In the modified code below, the author eliminates the need for the global variables `WIDTH` and `HEIGHT` by encapsulating this data into a class called `Rectangle`.

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

(continues on next page)

(continued from previous page)

```
        self.height = height
    def area(self):
        return self.width * self.height
    def circumference(self):
        return ((self.width * 2) + (self.height * 2))

r = Rectangle(3, 4)
print("area():" , r.area())
```

2.4.3 References

- Cunningham & Cunningham, Inc. - Global Variables Are Bad
- PyLint - W0603, global-statement

2.5 Using single letter to name your variables

Sometimes you see programmers trying to shorten the amount of text needed to write a piece of code, but when this goes to extremes, it will result in extremely ugly and unreadable code.

2.5.1 Anti-pattern

```
d = {'data': [{'a': 'b'}, {'b': 'c'}, {'c': 'd'}], 'texts': ['a', 'b', 'c']}

for k, v in d.iteritems():
    if k == 'data':
        for i in v:
            # Do you know what are you iterating now?
            for k2, v2 in i.iteritems():
                print(k2, v2)
```

2.5.2 Best practice

Use more verbose names for your variables for clarity

It is much better to write more text and to be much more precise about what each variable means.

```
data_dict = {
    'data': [{'a': 'b'}, {'b': 'c'}, {'c': 'd'}],
    'texts': ['a', 'b', 'c']
}

for key, value in data_dict.iteritems():
    if key == 'data':
        for data_item in value:
            # Do you know what are you iterating now?
            for data_key, data_value in data_item.iteritems():
                print(data_key, data_value)
```

2.6 Dynamically creating variable/method/function names

Sometimes a programmer gets an idea to make his/her work easier by creating magically working code that uses `setattr()` and `getattr()` functions to set some variable. While this may look like a good idea, because there is no need to write all the methods by hand, you are asking for trouble down the road.

2.6.1 Example

Consider the following code. You have some data and want to update the *class* with all of the data. Of course you don't want to do this by hand, especially if there are tons of items in `data_dict`. However, when refactoring this kind of code after several years, and you'd like to know where some variable is added to this class, you'd usually use `grep` or `ack_grep` to find it. But when setting variables/methods/functions like this, you're screwed.

```
data_dict = {'var1': 'Data1', 'var2': 'Data2'}


class MyAwesomeClass:

    def __init__(self, data_dict):
        for key, value in data_dict.iteritems():
            setattr(self, key, value)
```

While previous example may look easy to find and debug, consider this:

```
data_list = ['dat1', 'dat2', 'dat3']
data_dict = {'dat1': [1, 2, 3],
             'dat2': [4, 5, 6],
             'dat3': [7, 8, 9],
             'dat4': [0, 4, 6]}

class MyAwesomeClass:

    def __init__(self, data_list, data_dict):
        counter = 0

        for key, value in data_dict.iteritems():
            if key in data_list:
                setattr(self, key, value)
            else:
                setattr(self, 'unknown' + str(counter), value)
                counter += 1
```

Now the class contains also `unknownX` variables indexed by their count. Well, what a nice mess we created here. Try to find a year later where these variables come from.

2.6.2 Solutions

Find another way

While the approach in the examples above may be the easiest to write, it is the worst to maintain later. You should always try to find another way to solve your problem.

Typical examples:

- Use function to parse incoming data
- Use the data dict/list itself without class

This however depends on the task at hand.

3 Readability

3.1 Asking for permission instead of forgiveness

The Python community uses an EAFP (easier to ask for forgiveness than permission) coding style. This coding style assumes that needed variables, files, etc. exist. Any problems are caught as exceptions. This results in a generally clean and concise style containing a lot of `try` and `except` statements.

3.1.1 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")
```

3.1.2 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
```

3.1.3 References

- Python 2.7.8 - Glossary

3.2 Comparing things to *None* the wrong way

Per the PEP 8 Style Guide, the preferred way to compare something to `None` is the pattern `if Cond is None`. This is only a guideline. It can be ignored if needed. But the purpose of the PEP 8 style guidelines is to improve the readability of code.

3.2.1 Anti-pattern

The statement below uses the equality operator to compare a variable to `None`. This is not the PEP 8 preferred approach to comparing values to `None`.

```
number = None

if number == None:
    print("This works, but is not the preferred PEP 8 pattern")
```

3.2.2 Best practice

Compare values to `None` using the pattern `if cond is None`

The code below uses the PEP 8 preferred pattern of `if cond is None`.

```
number = None

if number is None:
    print("PEP 8 Style Guide prefers this pattern")
```

Here the identity operator `is` is used. It will check whether `number` is identical to `None`. `is` will return to `True` only if the two variables point to the same object.

3.2.3 References

- PEP 8 Style Guide - Programming Recommendations
- stackoverflow

3.3 Comparing things to *True* the wrong way

Per the PEP 8 Style Guide, the preferred ways to compare something to `True` are the patterns `if cond is True:` or `if cond:`. This is only a guideline. It can be ignored if needed. But the purpose of the PEP 8 Style Guide is to improve the readability of code.

3.3.1 Anti-pattern

The statement below uses the equality operator to compare a boolean variable to `True`. This is not the PEP 8 preferred approach to comparing values to `True`. For sure, it is an anti-pattern not only in Python but in almost every programming language.

```
flag = True

# Not PEP 8's preferred pattern
if flag == True:
    print("This works, but is not the preferred PEP 8 pattern")
```

3.3.2 Best practices

Evaluating conditions without comparing to `True`:

The code below uses the PEP 8 preferred pattern of `if condition:`. If the type of the `condition` is Boolean, it is obvious that comparing to `True` is redundant. But in Python, every *non-empty* value is treated as true in context of condition checking, see Python documentation:

> In the context of Boolean operations, and also when expressions are used by control flow statements, the following values are interpreted as false: `False`, `None`, numeric zero of all types, and empty strings and containers (including strings, tuples, lists, dictionaries, sets and frozensets). All other values are interpreted as true.

```
flag = True

if flag:
    print("PEP 8 Style Guide prefers this pattern")
```

Compare values to `True` using the pattern `if cond is True:`

The code below uses the pattern described in PEP 8 as *worse*:

```
flag = True

if flag is True:
    print("PEP 8 Style Guide abhors this pattern")
```

This pattern is useful, when you make actual distinction between `True` value and every other that could be treated as true. The same applies to `if cond is False`. This expression is true only if `cond` has actual value of `False` - not empty list, empty tuple, empty set, zero etc.

3.3.3 References

- PEP 8 Style Guide - Programming Recommendations

3.4 Using *type()* to compare types

The function `isinstance` is the best-equipped to handle type checking because it supports inheritance (e.g. an instance of a derived class is an instance of a base class, too). Therefore `isinstance` should be used whenever type comparison is required.

3.4.1 Anti-pattern

The `if` statement below uses the pattern `if type(OBJECT) is types.TYPE` to compare a `Rectangle` object to a built-in type (`ListType` in this example). This is not the preferred pattern for comparing types.

```
import types

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

r = Rectangle(3, 4)

# bad
if type(r) is types.ListType:
    print("object r is a list")
```

Note that the following situation will not raise the error, although it should.

```
import types

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

class Circle(object):
    def __init__(self, radius):
        self.radius = radius

c = Circle(2)
r = Rectangle(3, 4)

# bad
if type(r) is not type(c):
    print("object types do not match")
```

3.4.2 Best practice

Use `isinstance` to compare types

The preferred pattern for comparing types is the built-in function `isinstance`.

```
import types

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

(continues on next page)

(continued from previous page)

```
        self.height = height

r = Rectangle(3, 4)

# good
if isinstance(r, types.ListType):
    print("object r is a list")
```

3.4.3 References

- Stack Overflow: Differences between isinstance() and type() in Python

3.5 Not using dict comprehensions

You may encounter the old style of initializing a dict (passing an iterable of key-value pairs) in older Python code written before version 2.7. The new dict comprehension style is functionally equivalent and is much more readable. Consider refactoring the old-style code to use the new style (but only if you are using Python 2.7 or higher).

3.5.1 Anti-pattern

The code below demonstrates the old syntax of dict initialization. Although there is nothing syntactically wrong with this code, it is somewhat hard to read.

```
numbers = [1,2,3]

# hard to read
my_dict = dict([(number,number*2) for number in numbers])

print(my_dict)  # {1: 2, 2: 4, 3: 6}
```

3.5.2 Best practice

The modified code below uses the new dict comprehension syntax which was introduced in Python 2.7.

```
numbers = [1, 2, 3]

my_dict = {number: number * 2 for number in numbers}

print(my_dict)  # {1: 2, 2: 4, 3: 6}
```

3.5.3 References

- Stack Overflow - Create a dictionary with list comprehesion

3.6 Not using dict keys when formatting strings

When formatting a string with values from a dictionary, you can use the dictionary keys instead of explicity defining all of the format parameters. Consider this dictionary that stores the name and age of a person.

```
person = {
    'first': 'Tobin',
    'age': 20
}
```

3.6.1 Anti-pattern

Here is an example of formatting the string with values from the person. This is bad! If we added another key-value pair to the person dictionary, we would have to change the string and the format arguments

```
person = {
    'first': 'Tobin',
    'age':20
}

print('{0} is {1} years old'.format(
    person['first'],
    person['age'])
)
# Output: Tobin is 20 years old

person = {
    'first': 'Tobin',
    'last': 'Brown',
    'age':20
}

# Bad: we have to change the replacement fields within
# our string, once we add new values
print('{0} {1} is {2} years old'.format(
    person['first'],
    person['last'],
    person['age'])
)  # bad
# Output: Tobin Brown is 20 years old
```

3.6.2 Best practice

By using the dictionary keys in the string we are formatting, the code is much more readable and explicit.

```
person = {
    'first': 'Tobin',
    'age':20
}

print('{first} is {age} years old'.format(**person))
# Output: Tobin is 20 years old

person = {
    'first':'Tobin',
    'last': 'Brown',
    'age':20
}
print('{first} {last} is {age} years old'.format(**person))
# Output: Tobin Brown is 20 years old
```

Going even further, the same result can be achieved with your own objects by using `obj.__dict__`.

```
class Person(object):

    def __init__(self, first, last, age):
        self.first = first
        self.last = last
        self.age = age

    def __str__(self):
        return '{first} {last} is {age} years old'.format(**self.__dict__)


person = Person('Tobin', 'Brown', 20)
print(person)
# Output: Tobin Brown is 20 years old
```

3.7 Not using `items()` to iterate over a dictionary

PEP 20 states "There should be one– and preferably only one –obvious way to do it." The preferred way to iterate over the key-value pairs of a dictionary is to declare two variables in a for loop, and then call `dictionary.items()`, where `dictionary` is the name of your variable representing a dictionary. For each loop iteration, Python will automatically assign the first variable as the key and the second variable as the value for that key.

3.7.1 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]))
```

3.7.2 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))
```

3.7.3 Difference Python 2 and Python 3

In python 2.x the above examples using `items` would return a list with tuples containing the copied key-value pairs of the dictionary. In order to not copy and with that load the whole dictionary's keys and values inside a list to the memory you should prefer the `iteritems` method which simply returns an iterator instead of a list. In Python 3.x the `iteritems` is removed and the `items` method returns view objects. The benefit of these view objects compared to the tuples containing copies is that every change made to the dictionary is reflected in the view objects.

3.7.4 References

- PEP 20 - The Zen of Python
- Python 2 dict.iteritems
- Python 3 dict.items

3.8 Not using named tuples when returning more than one value from a function

Named tuples can be used anywhere where normal tuples are acceptable, but their values can be accessed through their names in addition to their indexes. This makes the code more verbose and readable.

3.8.1 Anti-pattern

The code below returns a first name, middle name, and last name using a normal, unnamed tuple. After calling the tuple, each value can only be returned via an index. This code is difficult to use: the caller of the function has to know that the first element is the first name, the second is the middle name, and the third is the last name.

```
def get_name():
    return "Richard", "Xavier", "Jones"

name = get_name()

# no idea what these indexes map to!
print(name[0], name[1], name[2])
```

3.8.2 Best practice

Use named tuples to return multiple values

The modified code below uses named tuples to return multiple values. This code is easier to use and easier to read, as now the caller can access each piece of data via a straightforward name (like `name.first`).

```
from collections import namedtuple

def get_name():
    name = namedtuple("name", ["first", "middle", "last"])
    return name("Richard", "Xavier", "Jones")

name = get_name()

# much easier to read
print(name.first, name.middle, name.last)
```

3.8.3 References

- Python Standard Libary - collections.namedtuple

3.9 Not using unpacking for updating multiple values at once

In general, the Python programming community prefers concise code over verbose code. Using unpacking to update the values of multiple variables simultaneously is more concise than using assignments to update each variable individually.

3.9.1 Anti-pattern

The function below implements the classical Euclid algorithm for greatest common divisor. The updates of the variables `a` and `b` are made using variable `temp` and three lines of code.

```
def gcd(a, b):
    while b != 0:
        temp = b
        b = a % b
        a = temp
    return a
```

3.9.2 Best practice

Use unpacking to update multiple values simultaneously

The modified code below is functionally equivalent to the original code above, but this code is more concise.

```
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a
```

3.9.3 Gotchas

The unpacking can be sometimes quite misleading. Figure out what is the outcome of the code below.

```
b = "1984"
a = b, c = "AB"
print(a, b, c)
```

3.10 Not using `zip()` to iterate over a pair of lists

PEP 20 states "There should be one– and preferably only one –obvious way to do it." The preferred way to iterate through a pair of lists is to declare two variables in a loop expression, and then call `zip(list_one, list_two)`, where `list_one` and `list_two` are the two lists you wish to iterate through. For each loop iteration, Python will automatically assign the first variable as the next value in the first list, and the second variable as the next value in the second list.

3.10.1 Anti-pattern

The code below defines a variable `index` which serves as an index variable for iterating through two lists. Within the for loop the code accesses the corresponding value for each list by using the index variable. This is a common way for iterating through two lists, but it is not the preferred way in Python.

```
numbers = [1, 2, 3]
letters = ["A", "B", "C"]

for index in range(len(numbers)):
    print(numbers[index], letters[index])
```

3.10.2 Best-practice

Use `zip()` to iterate through a pair of lists

The updated code below demonstrates the Pythonic style for iterating through a pair of lists. When the code defines two variables in its `for` loop in conjunction with a call to `zip(numbers, letters)` on the pair of lists, Python automatically assigns the first variable as the next value in the first list, and the second variable as the next value in the second list.

```
numbers = [1, 2, 3]
letters = ["A", "B", "C"]

for numbers_value, letters_value in zip(numbers, letters):
    print(numbers_value, letters_value)
```

3.10.3 References

- PEP 20 - The Zen of Python
- Built-in Functions > zip(*iterables)

3.11 Putting type information in a variable name

Python is a duck-typed language. Just because a variable is described as an integer does not mean that it actually is an integer. This can be very dangerous for any programmer who acts on the variable assuming that it is an integer. Note that the practice of including type notation in variable names is also called Hungarian Notation.

3.11.1 Anti-pattern

The code below demonstrates the dangers of variables whose names include type notation. Just because a variable is called `n_int` does not mean that the variable is actually an integer.

```
n_int = "Hello, World!"

# mistakenly assuming that n_int is a number
4 / n_int
```

3.11.2 Best practice

Remove type notation

Although the modifed code below does not fix the underlying problem of attempting to divide a number by a string, the code is generally less misleading, because there is no misleading description in the variable name `n` that `n` is a number.

```
n = "Hello, World!"

# still a problem, but less misleading now
4 / n
```

3.11.3 References

- Stack Overflow - Hungarian Notation

3.12 Test for object identity should be `is`

Testing the identity of two objects can be achieved in python with a special operator called `is`. Most prominently it is used to check whether a variable points to `None`. But the operator can examine any kind of identity. This often leads to confusion because equality of two different objects will return `False`.

3.12.1 Anti-pattern

```
a = range(10)
b = range(10)

print((a is b))
```

This code snippet will print `False` even though `a` and `b` have equal values. This can occur because `a` and `b` are references that point to different objects which happen to have the same value. To verify the equality of two variables the == operator should be used.

3.12.2 Best practice

Only use the `is` operator if you want to check the exact identity of two references.

```
some_list = None

if some_list is None:
    do_somthing_with_the_list()
```

3.12.3 References

- PEP8 Style Guide - Programming Recommendations

3.13 Using an unpythonic loop

PEP 20 states "There should be one– and preferably only one –obvious way to do it." Creating a loop that uses an incrementing index to access each element of a list within the loop construct is not the preferred style for accessing each element in a list. The preferred style is to use `enumerate()` to simultaneously retrieve the index and list element.

3.13.1 Anti-pattern

The code below uses an index variable `i` in a `for` loop to iterate through the elements of a list. This is not the preferred style for iterating through a list in Python.

```
l = [1,2,3]

# creating index variable
for i in range(0,len(l)):
    # using index to access list
    le = l[i]
    print(i,le)
```

3.13.2 Best practice

Retrieve index and element when defining loop

The updated code below demonstrates the Pythonic style for iterating through a list. When you define two variables in a `for` loop in conjunction with a call to `enumerate()` on a list, Python automatically assigns the first variable as an index variable, and the second variable as the corresponding list element value for that index location in the list.

```
for i, le in enumerate(l):
    print(i, le)
```

3.13.3 References

- PEP 20 - The Zen of Python

3.14 Using `map()` or `filter()` where list comprehension is possible

For simple transformations that can be expressed as a list comprehension, use list comprehensions over `map()` or `filter()`. Use `map()` or `filter()` for expressions that are too long or complicated to express with a list comprehension. Although a `map()` or `filter()` expression may be functionally equivalent to a list comprehension, the list comprehension is generally more concise and easier to read.

3.14.1 Anti-pattern

The code below defines a list, and then uses `map()` to create a second list which is just the doubles of each value from the first list.

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

3.14.2 Best practice

Use list comprehension instead of `map()`

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

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

3.14.3 References

- PyLint - W0110, deprecated-lambda
- Oliver Fromme - List Comprehensions

3.15 Using *CamelCase* in function names

Per the PEP 8 Style Guide, function names should be lowercase, with words separated by underscores.

3.15.1 Anti-pattern

```
def someFunction():
    print("Is not the preferred PEP 8 pattern for function names")
```

3.15.2 Best practice

Using lowercase with underscores

The code below uses the PEP 8 preferred pattern of function names.

```
def some_function():
    print("PEP 8 Style Guide prefers this pattern")
```

3.15.3 References

- PEP8 Style Guide - Function names

4 Security

Python is a highly dynamic language that gives the programmer many ways to change the runtime behavior of his code and even dynamically execute new code. This is powerful but can be a security risk as well.

Use the following patterns to increase the security of your code.

4.1 use of exec

The `exec` statement enables you to dynamically execute arbitrary Python code which is stored in literal strings. Building a complex string of Python code and then passing that code to `exec` results in code that is hard to read and hard to test. Anytime the `Use of exec` error is encountered, you should go back to the code and check if there is a clearer, more direct way to accomplish the task.

4.1.1 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
```

4.1.2 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()
```

4.1.3 References

- PyLint - W0122, exec-used
- Python Language Reference - The exec statement
- Stack Overflow - Why should exec() and eval() be avoided?

5 Performance

In Python, large performance gains can be obtained by using appropriate functions and directives. Avoid the following anti-patterns to reduce overhead and make your code more performant.

5.1 Using `key in list` to check if key is contained in list

Using `key in list` to iterate through a list can potentially take `n` iterations to complete, where `n` is the number of items in the list. If possible, you should change the list to a set or dictionary instead, because Python can search for items in a set or dictionary by attempting to directly accessing them without iterations, which is much more efficient.

5.1.1 Anti-pattern

The code below defines a list `l` and then calls `if 3 in l` to check if the number 3 exists in the list. This is inefficient. Behind the scenes, Python iterates through the list until it finds the number or reaches the end of the list.

```
l = [1, 2, 3, 4]

# iterates over three elements in the list
if 3 in l:
    print("The number 3 is in the list.")
else:
    print("The number 3 is NOT in the list.")
```

5.1.2 Best practice

Use a set or dictionary instead of a list

In the modified code below, the list has been changed to a set. This is much more efficient behind the scenes, as Python can attempt to directly access the target number in the set, rather than iterate through every item in the list and compare every item to the target number.

```
s = set([1, 2, 3, 4])

if 3 in s:
    print("The number 3 is in the list.")
else:
    print("The number 3 is NOT in the list.")
```

5.2 Not using `iteritems()` to iterate over a large dictionary in Python 2

PEP 234 defines iteration interface for objects. It also states it has significant impact on performance of dict iteration.

Note: This anti-pattern only applies to Python versions 2.x. In Python 3.x `items()` returns an iterator (consequently, `iteritems()` and Python 2's iterative `range()` function, `xrange()`, have been removed from Python 3.x).

5.2.1 Anti-pattern

The code below defines one large dictionary (created with dictionary comprehension) that generates large amounts of data. When using `items()` method, the iteration needs to be completed and stored in-memory before `for` loop can begin iterating. The prefered way is to use `iteritems`. This uses (~1.6GB).

```
d = {i: i * 2 for i in xrange(10000000)}

# Slow and memory hungry.
for key, value in d.items():
    print("{0} = {1}".format(key, value))
```

5.2.2 Best-practice

Use `iteritems()` to iterate over large dictionary

The updated code below uses `iteritems()` instead of `items()` method. Note how the code is exactly the same, but memory usage is 50% less (~800MB). This is the preferred way to iterate over large dictionaries.

```
d = {i: i * 2 for i in xrange(10000000)}

# Memory efficient.
for key, value in d.iteritems():
    print("{0} = {1}".format(key, value))
```

5.2.3 References

- PEP 234 Iterators

6 Django

Django is a great framework to create fast and scalable web applications. To help you write great Django apps from the start, we started to compile a set of anti- and migration patterns. They'll help you to avoid common mistakes or to migrate to a new version faster. Some patterns are simply (more elaborate) explanations of tips and best practices that can be found in Django's docs. Others stem from our own experiences. Feel free to contribute your ideas or share your pattern via email.

6.1 Maintainability

Avoid the following anti-patterns to increase maintainability of your Django code base—for you, and for others.

6.1.1 Importing django.db.models.fields

In Django, models are defined in `django.db.models.fields`. However, for convenience they are imported into `django.db.models`. Django's standard convention is to use `from django.db import models` and refer to fields as `models<some>Field`. To improve readability and maintainability of your code, change your import statement and model definition.

Anti-pattern

```
from django.db.models import fields

class Person(models.Model):
    first_name = fields.CharField(max_length=30)
    last_name = fields.CharField(max_length=30)
```

Best practice

Stick to standard conventions and use `from django.db import models` instead.

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
```

References

- Django documentation - Model field reference

6.2 Security

Most Django applications contain a lot of proprietory or even confidential information. Hence, it is crucial to take all possible measures to take your Django application secure and to recude the possibility of being hacked.

Use the following patterns to increase the security of your code.

6.2.1 ALLOWED_HOSTS setting missing

In Django, you need to properly set the `ALLOWED_HOSTS` setting when `DEBUG = False`. This is a security mechanism. It prevents attackers from poisoning caches or password reset emails with links to malicious hosts by submitting requests with a fake HTTP Host header, which is possible even under many seemingly-safe web server configurations.

Anti-Pattern

`ALLOWED_HOSTS` not set or empty, when `DEBUG = False`.

```
""" settings.py """

DEBUG = False
# ...
ALLOWED_HOSTS = []
```

Best practice

Make sure, an appropriate host is set in *ALLOWED_HOSTS*, whenever *DEBUG* = *False*.

```
DEBUG = False
# ...
ALLOWED_HOSTS = ['djangoproject.com']
```

References

- Django documentation - Settings: The Basics
- Django documentation - Settings: ALLOWED_HOSTS

6.2.2 SECRET_KEY published

A secret key has to be be kept secret. Make sure it is only used in production, but nowhere else. Especially, avoid committing it to source control. This increases security and makes it less likely that an attacker may acquire the key.

Anti-pattern

This settings.py contains a SECRET_KEY. You should not do this!

```
""" settings.py """
SECRET_KEY = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'
```

Better Practices

Load key from environment variable

Instead of publishing your secret key, you can use an environment variable to set your secret key.

```
import os
SECRET_KEY = os.environ['SECRET_KEY']
```

Load secret key from file

Alternatively, you can read the secret key from a file.

```
with open('/etc/secret_key.txt') as f:
    SECRET_KEY = f.read().strip()
```

References

- Django

6.2.3 Same value for MEDIA_ROOT and STATIC_ROOT

According to Django's documentation, `MEDIA_ROOT` and `STATIC_ROOT` must have different values. Before STATIC_ROOT was introduced, `MEDIA_ROOT` was also used (as fallback) to also serve static files. As this can have serious security implications, Django has validation checks to prevent it.

Anti-pattern

`MEDIA_ROOT` and `STATIC_ROOT` point to the same folder.

```
""" settings.py """

# Media and static root are identical
STATIC_ROOT = '/path/to/my/static/files'
MEDIA_ROOT = '/path/to/my/static/files'
```

Best practice

Ensure, `STATIC_ROOT` and `MEDIA_ROOT` point to different folders.

```
""" settings.py """

STATIC_ROOT = '/path/to/my/static/files'
MEDIA_ROOT = '/path/to/my/media/files'
```

References

- Django documentation - Settings: MEDIA_ROOT

6.2.4 Same value for MEDIA_URL and STATIC_URL

According to Django's documentation, `MEDIA_URL` and `STATIC_URL` must have different values.

Anti-pattern

`MEDIA_URL` and `STATIC_URL` point to the same URL.

```
""" settings.py """

# Media and static root are identical
STATIC_URL = 'http://www.mysite.com/static'
MEDIA_URL = 'http://www.mysite.com/static'
```

Best practice

Ensure, *STATIC_URL* and *MEDIA_URL* point to different URL's.

```
""" settings.py """

STATIC_URL = 'http://www.mysite.com/static'
MEDIA_URL = 'http://www.mysite.com/media'
```

References

- Django documentation - Settings: MEDIA_URL
- Django documentation - Settings: MEDIA_ROOT

6.3 Correctness

6.3.1 Not using forward slashes

Django requires you to use forward slashes `/` whenever you indicate a path, even on Windows. In your settings, this is true for the following variables.

- `STATICFILES_DIRS`
- `TEMPLATE_DIRS`
- `DATABASES['<your database>'][NAME]`
- `FIXTURE_DIRS`

Anti-pattern

This pattern is exemplary for any of the above mentioned settings. It uses backslashes, instead of forward slashes.

```
""" settings.py """

STATICFILES_DIRS = [
    "\\path\\to\\my\\static\\files",
]
```

Best practice

Django requires you to use forward slashes `/`, even on Windows.

```
""" settings.py """

STATICFILES_DIRS = [
    "/path/to/my/static/files",
]
```

References

- Django documentation - Settings: TEMPLATE_DIRS
- Django documentation - Settings: FIXTURE_DIRS
- Django documentation - Settings: STATIC_FILES_DIRS
- Django documentation - Settings: HOST

6.3.2 Not using NullBooleanField

A `BooleanField` in Django accepts only the two values: `true` and `false`. If you need to accept `NULL` values, you have to use a `NullBooleanField`.

Anti-pattern

The following model uses a `BooleanField` with the option `null=True`, instead of using a `NullBooleanField`.

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
    activated = models.BooleanField(null=True)
```

Best practice

Use a `NullBooleanField` instead:

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
    # Using NullBooleanField instead
    activated = models.NullBooleanField()
```

Reference

- Django documentation - Model field reference: BooleanField
- Django documentation - Model field reference: NullBooleanField

6.4 Performance

Django has a lot of mechanisms built-in to build fast and efficient web applications. Still, there are several things to watch out for, especially when you start to scale your Django application. This chapter contains anti-patterns that can potentially harm the performance of your application and hence, should be avoided.

6.4.1 Inefficient database queries

Django's models make it easy for you, to filter the data of your application without using any SQL statements. This is a great thing, however, it sometimes hides that you are using object filters inefficiently. Unless you append `.values()` to your filter, your QuerySet will always query all columns within your database. This can be uncritical until you scale your application or once your tables grow bigger. Therefore, make sure you only retrieve the columns your really need within your program.

Anti-Pattern

Let's assume we have a an app `vehicle` which contains a model `Cars` to store plenty of information about a car:

```
""" models.py """

class Cars(models.Model):
    make = models.CharField(max_length=50)
    model = models.CharField(max_length=50)
    wheels = models.CharField(max_length=2)
    # ...
```

We import this model into one of your views to do something will make names within our database:

```
""" views.py """
from models import Cars

# ...

cars = Cars.objects.all()
for car in cars:
    do_something(car.make)
```

Even though this code works and looks harmless, it can kill you in production. You think, you are actually just accessing the `make` field, but you are actually retrieving ALL data from your database, once you start iterating over the retrieved QuerySet:

```
SELECT make, model, wheels, ... FROM vehicles_cars;
```

Especially, if you have many fields on your model and/or if you got millions of records in your table, this slows down the response time of your applications significantly. As QuerySets are cached upon evaluation, it will hit your database only once, but you'd better be carful.

Best practice

Use `.values()`

To avoid such a scenario, make sure you only query the data you really need for your program. Use `.values()` to restrict the underlying SQL query to required fields only.

```
""" views.py """
from cars.models import Cars

cars = Cars.objects.all().values('make')

# Print all makes
for car in cars:
    do_something(car['make'])
```

```
SELECT make from vehicles_cars;
```

Use `.values_list()`

Alternatively, you can use `.value_list()`. It is similar to `values()` except that instead of returning dictionaries, it returns tuples when you iterate over it.

```
""" views.py """
from cars.models import Cars

cars = Cars.objects.all().values_list('make', flat=True)

# Print all makes
for make in cars:
    do_something(make)
```

References

- Django documentation - Models: Querysets (values)
- Django documentation - Models: Querysets (values_list)

6.5 Migration to 1.8

Migrating to a new Django version can be time consuming. To make this process easier, this chapter lists deprecated features and shows potential migration patterns/pathes.

6.5.1 TEMPLATE_DIRS deprecated

This setting is deprecated since Django version 1.8. Set the `DIRS` option of a [*DjangoTemplates* backend](https://docs.djangoproject.com/en/1.8/topics/templates/#module-django.template.backends.django) instead.

Deprecated feature

Deprecated `TEMPLATE_DIRS` setting used.

```
""" settings.py """

TEMPLATE_DIRS = [
    "path/to/my/templates",
]
```

Migration path

As of Django 1.8 you should set `DIRS` option within `TEMPLATES` setting. It defines where the engine should look for template source files, in search order.

```
""" settings.py """

TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'APP_DIRS': True,
        'DIRS': '/path/to/my/templates',
    },
]
```

References

- Django documentation - Settings: TEMPLATES
- Django documentation - Settings: TEMPLATE_DIRS
- Django documentation - Templates: Built-in backends

6.5.2 TEMPLATE_DEBUG deprecated

This setting sets the output that the template system should use for invalid (e.g. misspelled) variables. The default value is an empty string `''`. This setting is deprecated since Django version 1.8. Set the *TEMPLATE_DEBUG* option in the `OPTIONS` of a `DjangoTemplates` backend instead.

Deprecated feature

Deprecated `TEMPLATE_DEBUG` setting used.

```
""" settings.py """

TEMPLATE_DEBUG = True
```

Migration path

As of Django 1.8 you should set `debug` option in the `OPTIONS` of a `DjangoTemplates` backend instead.

```
""" settings.py """

TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'APP_DIRS': True,
        'DIRS': '/path/to/my/templates',
        'OPTIONS': {
             'debug': True,
         }
    },
]
```

References

- Django documentation - Settings: TEMPLATE_DEBUG
- Django documentation - Settings: TEMPLATES
- Django documentation - Templates: Built-in backends

6.5.3 TEMPLATE_LOADERS deprecated

This setting is deprecated since Django version 1.8. Set the *LOADERS* option of a DjangoTemplates backend instead.

Deprecated feature

Deprecated `TEMPLATE_LOADERS` setting used.

```
""" settings.py """

TEMPLATE_LOADERS = (
        'django.template.loaders.filesystem.Loader',
        'django.template.loaders.app_directories.Loader',
)
```

Migration path

As of Django 1.8 you should set `loaders` option in the `TEMPLATES` setting. It defines where the engine should look for template source files, in search order.

```
""" settings.py """

TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'APP_DIRS': True,
        'DIRS': '/path/to/my/templates',
        'OPTIONS': {
             'loaders': (
                 'django.template.loaders.filesystem.Loader',
                 'django.template.loaders.app_directories.Loader',
              ),
         }
    },
]
```

References

- Django documentation - Settings: TEMPLATES]
- Django documentation - Settings: TEMPLATE_DIRS]
- Django documentation - Templates: Built-in backends]

6.5.4 TEMPLATE_STRING_IF_INVALID deprecated

This setting sets the output that the template system should use for invalid (e.g. misspelled) variables. The default value is an empty string `''`. This setting is deprecated since Django version 1.8. Set the *string_if_invalid* option in the *OPTIONS* of a *DjangoTemplates* backend instead.

Deprecated feature

Deprecated `TEMPLATE_STRING_IF_INVALID` setting used.

```
""" settings.py """

TEMPLATE_STRING_IF_INVALID = 'Invalid variable'
```

Migration path

As of Django 1.8 you should set `string_if_invalid` option in the `OPTIONS` of a `DjangoTemplates` backend instead.

```
""" settings.py """

TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'APP_DIRS': True,
        'DIRS': '/path/to/my/templates',
        'OPTIONS': {
             'string_if_invalid': 'Invalid varialbe!',
         }
    },
]
```

References

- Django documentation - Settings: TEMPLATES
- Django documentation - Settings: TEMPLATE_STRING_IF_INVALID
- Django documentation - Templates: Built-in backends
- Django documentation - Templates: How invalid variables are handled