Who am I?

William Gómez

Engineer at Huge

(Full stack wannabe web engineer at Huge)

Python Medellin meetups co-organizer

Google Cloud Certified Data Engineer

Freelancer

HUGE

Hello

Python Design Patterns - Part 1

September 30, 2020

Agenda.

1. Principles
2. Python Patterns

In software engineering the term pattern describes a proven solution to a common problem in a specific context.

1

Principles

1. Program to an interface not an implementation

Duck typing:

“If it looks like a duck and quacks like a duck, it's a duck!”

We don’t bother with the nature of the object, we don’t have to care what the object is; we just want to know if it’s able to do what we need (we are only interested in the interface of the object).

Call some method Quack on an object.

We program to the interface instead of the implementation

```
try:
    bird.quack()
except AttributeError:
    ...
```

2. Composition Over Inheritance

The subclass explosion problem:

Extending a base class to adding more functionalities, but when you need a combination of those functionalities you will end with at bunch of classes.

```
class Logger(object):
    def __init__(self, file):
        self.file = file

    def log(self, message):
        self.file.write(message + '\n')
        self.file.flush()
```

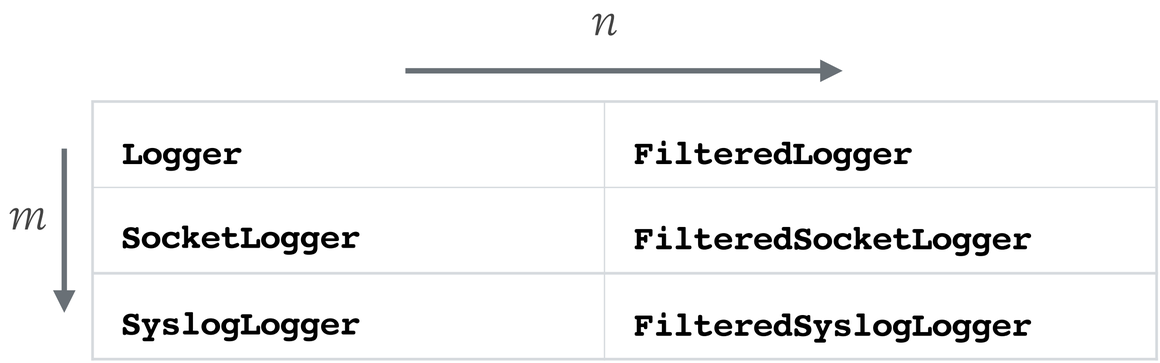
```
class SocketLogger(Logger):
    def __init__(self, sock):
        self.sock = sock

    def log(self, message):
        self.sock.sendall(message)

class SyslogLogger(Logger):
    def __init__(self, priority):
        self.priority = priority

    def log(self, message):
        syslog.syslog(self.priority, message)
```

```
class FilteredLogger(Logger):
    def __init__(self, pattern, file):
        self.pattern = pattern
        super().__init__(file)

    def log(self, message):
        if self.pattern in message:
            super().log(message)
```

A crucial weakness of inheritance as a design strategy is that a class often needs to be specialized along several different design axes at once

2. Composition Over Inheritance

Maybe a programmer will get lucky and no further combinations will be needed. But there is possibility of ending up with $3 \times 2 = 6$ classes:



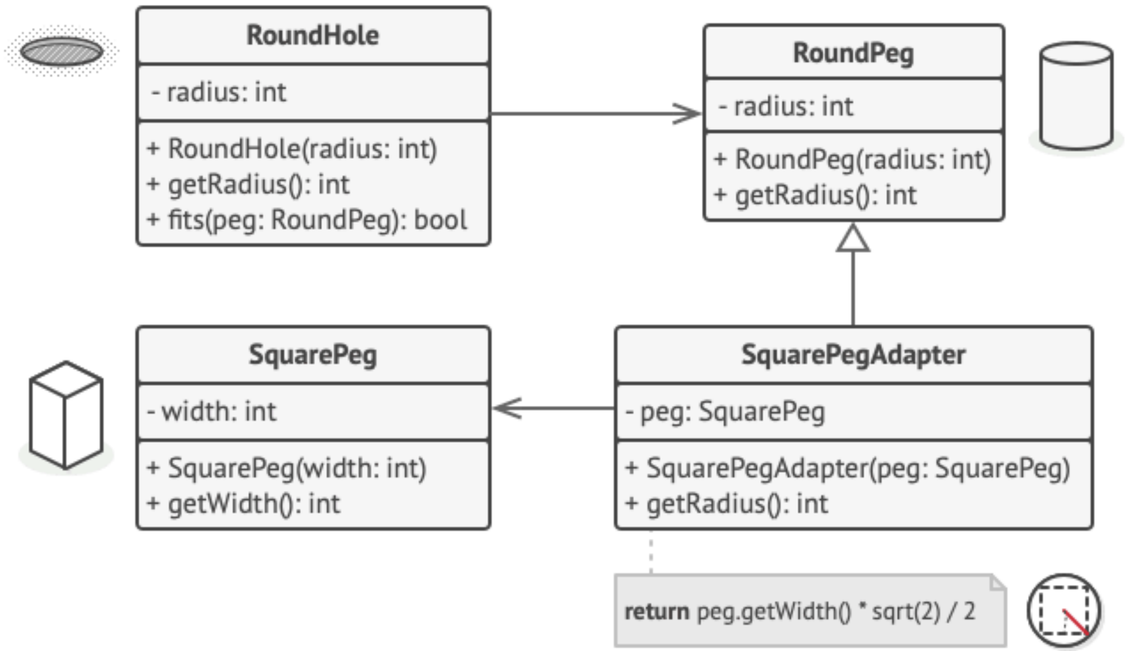
The total number of classes will increase geometrically if *m(outputs)* and *n* (functionalities such as filtering) continue to grow. This is the “proliferation of classes” and “explosion of subclasses”.

Adapter Pattern

Adapter Pattern (Structural Pattern)

This is a special object that converts the interface of one object so that another object can understand it.

An adapter wraps one of the objects to hide the complexity of conversion happening behind the scenes. The wrapped object isn't even aware of the adapter.



Adapting square pegs to round holes.

Adapter Pattern (Structural Pattern)

```
class RoundPeg:
    def __init__(self, radius):
        self.radius = radius

    def getRadius(self):
        return self.radius
```

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

    def getWidth(self):
        return self.width
```

```
class RoundHole:
    def __init__(self, radius):
        self.radius = radius

    def getRadius(self):
        return self.radius

    def fits(self, peg):
        return self.getRadius() >= peg.getRadius()
```

```
class SquarePegAdapter:
    def __init__(self, peg):
        self.peg = peg

    def getRadius(self):
        return self.peg.getWidth() * math.sqrt(2) / 2
```

Solution to logger problem

The original logger class doesn't need to be improved, because any mechanism for outputting messages can be wrapped up to look like the file object that the logger is expecting.

```
class Logger(object):
    def __init__(self, file):
        self.file = file

    def log(self, message):
        self.file.write(message + '\n')
        self.file.flush()
```

Solution to logger problem

We adapt each destination to the behavior of a file and then pass the adapter to a Logger as its output file. We “adapted” (imitated) the “output file”.

```
class FileLikeSocket:
    def __init__(self, sock):
        self.sock = sock

    def write(self, message_and_newline):
        self.sock.sendall(message_and_newline.encode('ascii'))

    def flush(self):
        pass
```

```
class FileLikeSyslog:
    def __init__(self, priority):
        self.priority = priority

    def write(self, message_and_newline):
        message = message_and_newline.rstrip('\n')
        syslog.syslog(self.priority, message)

    def flush(self):
        pass
```

Solution to logger problem

Python encourages duck typing, so an adapter's only responsibility is to offer the right methods.

```
class Logger(object):
    def __init__(self, file):
        self.file = file

    def log(self, message):
        self.file.write(message + '\n')
        self.file.flush()
```

```
class FilteredLogger(Logger):
    def __init__(self, pattern, file):
        self.pattern = pattern
        super().__init__(file)

    def log(self, message):
        if self.pattern in message:
            super().log(message)
```

```
sock1, sock2 = socket.socketpair()

fs = FileLikeSocket(sock1)
logger = FilteredLogger('Error', fs)
logger.log('Warning: message number one')
logger.log('Error: message number two')

print('The socket received: %r' % sock2.recv(512))
```

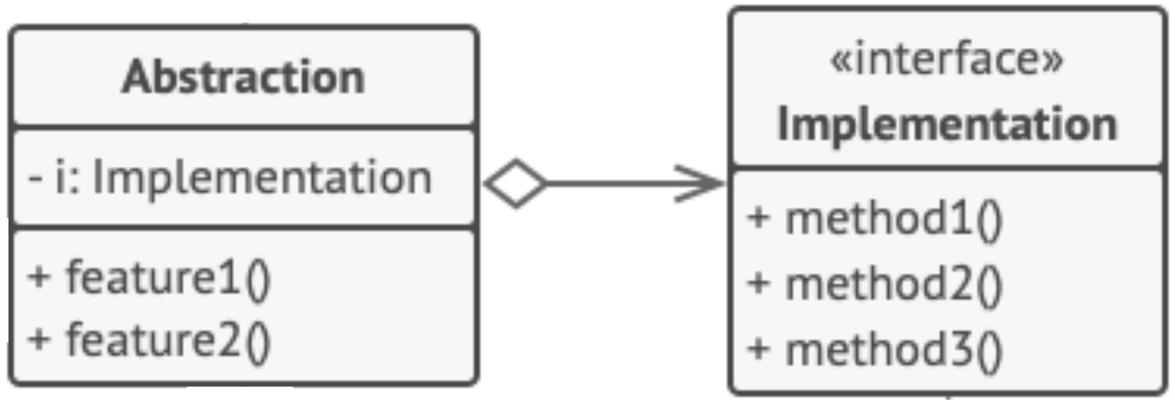
They are under no obligation to re-implement the full slate of more than a dozen methods that a real file offers. Just as it's not important that a duck can walk if all you need is a quack.

Bridge Pattern

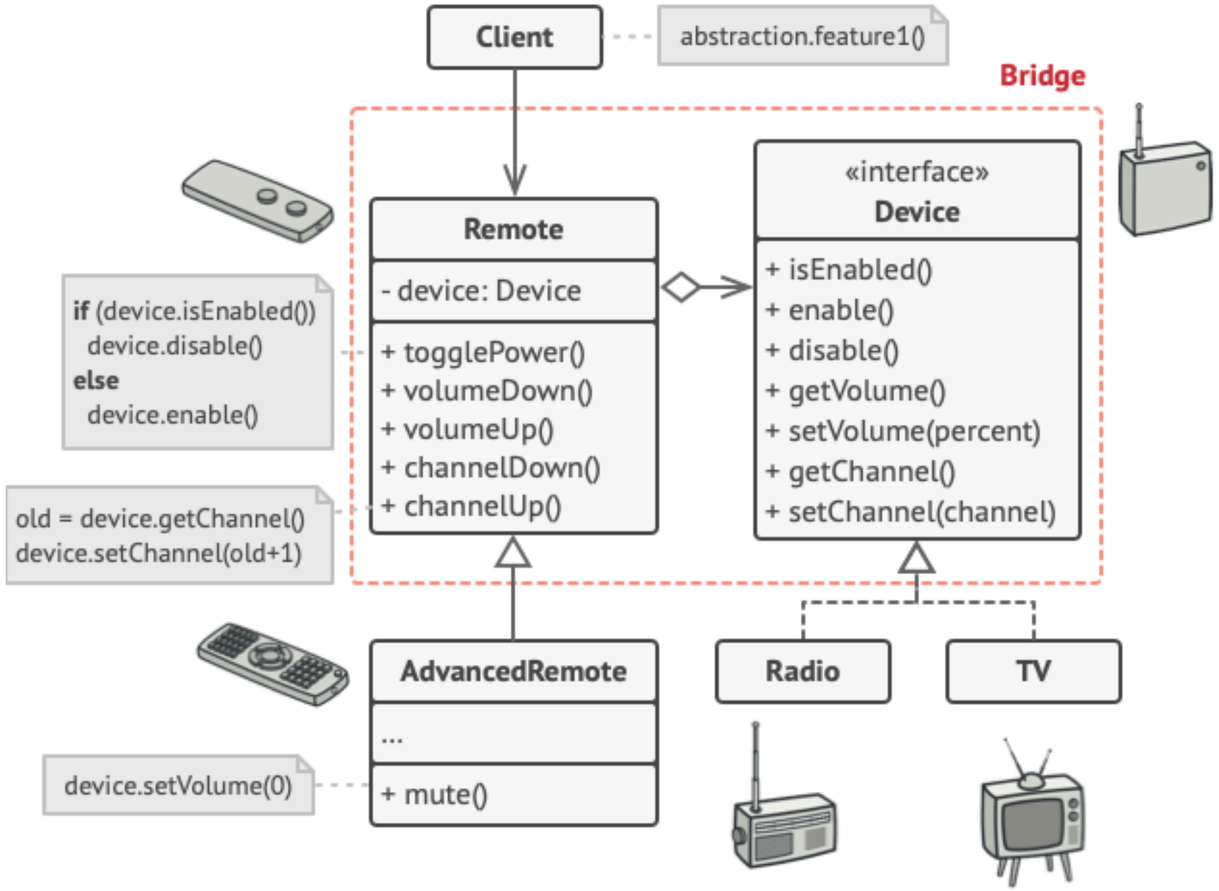
Bridge Pattern

Lets you split a large class or a set of closely related classes into two separate hierarchies:

- Abstraction (what the caller sees)
- Implementation (wrapped inside)



Bridge Pattern



Bridge Pattern: Logger solution

Let's make this decision:

- Filtering belongs out in the "abstraction" class.
- Output belongs in the "implementation" class: accept a raw message and emits it.

Abstraction (Callers)

```
class Logger(object):
    def __init__(self, handler):
        self.handler = handler

    def log(self, message):
        self.handler.emit(message)

class FilteredLogger(Logger):
    def __init__(self, pattern, handler):
        self.pattern = pattern
        super().__init__(handler)

    def log(self, message):
        if self.pattern in message:
            super().log(message)
```

Implementation (Actions)

```
class FileHandler:
    def __init__(self, file):
        self.file = file

    def emit(self, message):
        self.file.write(message + '\n')
        self.file.flush()
```

```
class SocketHandler:
    def __init__(self, sock):
        self.sock = sock

    def emit(self, message):
        self.sock.sendall((message + '\n')
          .encode('ascii'))
```

```
class SyslogHandler:
    def __init__(self, priority):
        self.priority = priority

    def emit(self, message):
        syslog.syslog(self.priority, message)
```

Solution to logger problem

Abstraction objects and implementation objects can now be freely combined at runtime:

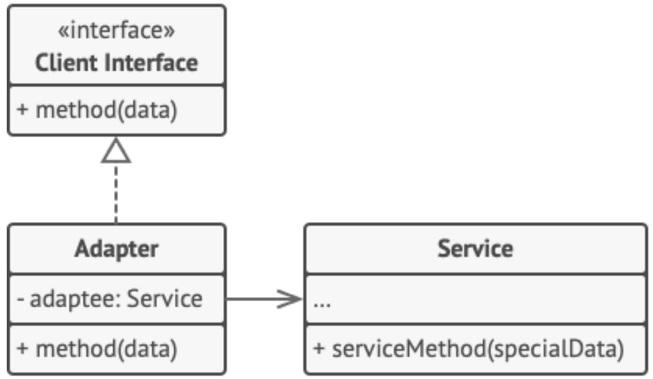
```
handler = FileHandler(sys.stdout)
logger = FilteredLogger('Error', handler)

logger.log('Ignored: this will not be logged')
logger.log('Error: this is important')
```

A functioning logger is now always built by composing an abstraction with an implementation.

Comparison

Adapter

A class is “adapted” to be used for a client.

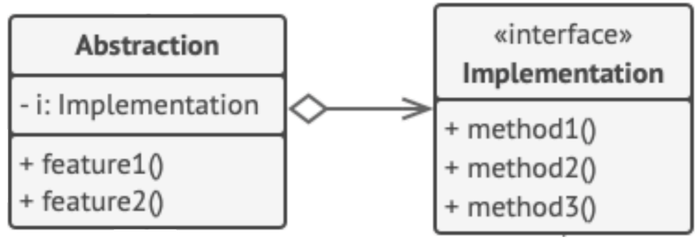


Uses:

- When you want to use some existing class, but its interface isn’t compatible with the rest of your code.

Bridge

A class is only focused on what is needed to do (features) and the other is focused on how to do it (methods)



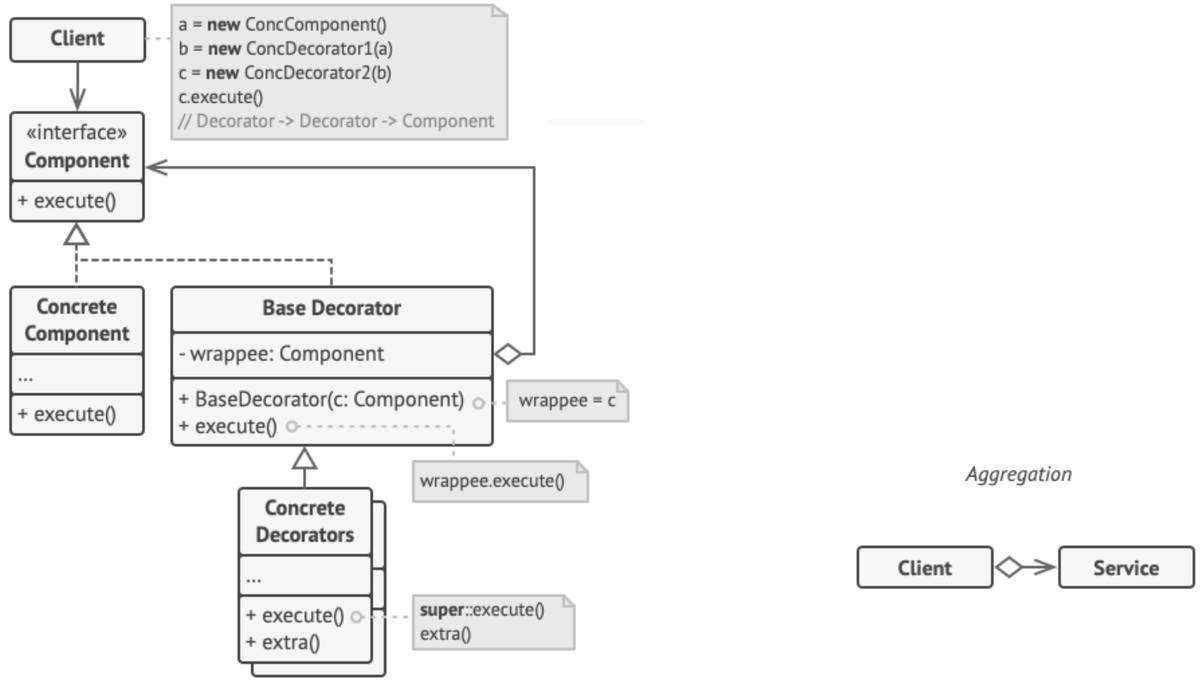
Uses:

- When you want to divide and organize a monolithic class that has several variants of some functionality.

Decorator Pattern

Decorator pattern

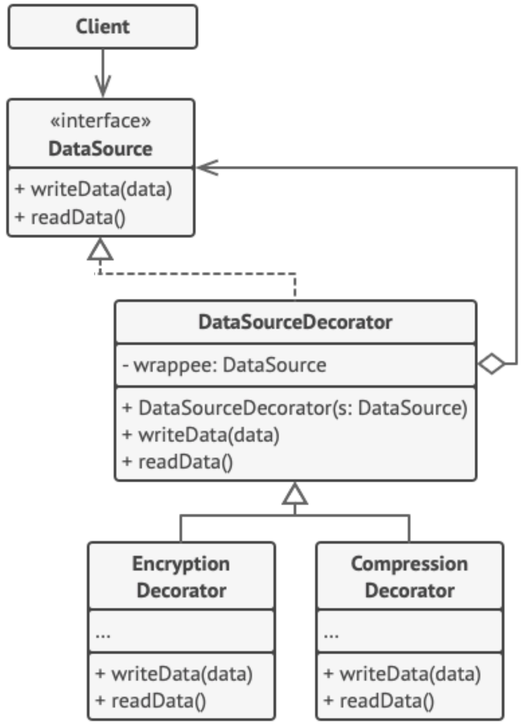
Lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.



"Wrapper" is the alternative nickname for the Decorator pattern that clearly expresses the main idea of the pattern.

Decorator pattern

Compress and encrypt sensitive data independently from the code that actually uses this data.



Both wrappers change the way the data is written to and read from the disk

Decorator pattern: Logger solution

What if we wanted to apply two different filters to the same log?

One filtering by priority and the other matching a keyword.

```
class FileLogger:
    def __init__(self, file):
        self.file = file

    def log(self, message):
        self.file.write(message + '\n')
        self.file.flush()


class SocketLogger:
    def __init__(self, sock):
        self.sock = sock

    def log(self, message):
        self.sock.sendall((message + '\n')
          .encode('ascii'))
```

```
class SyslogLogger:
    def __init__(self, priority):
        self.priority = priority

    def log(self, message):
        syslog.syslog(self.priority, message)
```

```
class LogFilter:
    def __init__(self, pattern, logger):
        self.pattern = pattern
        self.logger = logger

    def log(self, message):
        if self.pattern in message:
            self.logger.log(message)
```

Decorator pattern

And because Decorator classes are symmetric — they offer exactly the same interface they wrap — we can now stack several different filters atop the same log!

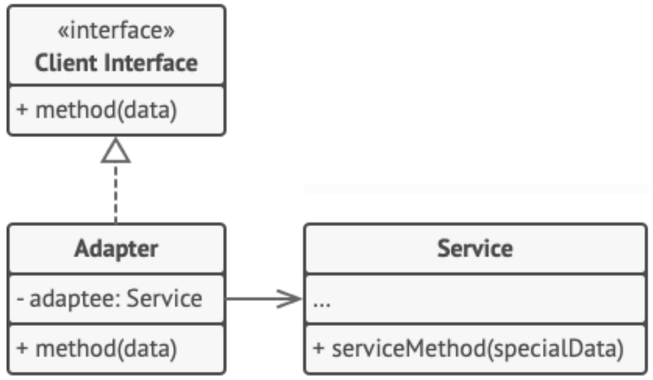
```
log1 = FileLogger(sys.stdout)
log2 = LogFilter('Error', log1)

log1.log('Noisy: this logger always produces output')

log2.log('Ignored: this will be filtered out')
log2.log('Error: this is important and gets printed')
```

```
log3 = LogFilter('severe', log2)

log3.log('Error: this is bad, but not that bad')
log3.log('Error: this is pretty severe')
```

But note the one place where the symmetry of this design breaks down: while filters can be stacked, output routines cannot be combined or stacked. Log messages can still only be written to one output.

Required eyebrow.

Adapter

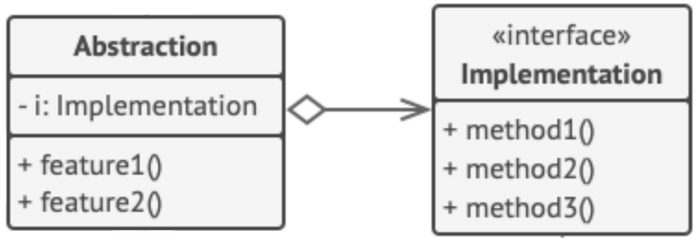


Uses:

- When you want to use some existing class, but its interface isn't compatible with the rest of your code.

Compatibility

Bridge



Uses:

- When you want to divide and organize a monolithic class that has several variants of some functionality.

Delegate responsibilities
High level logic (abstraction)
Platform details (implementation)

Decorator

Uses:

- When you need to be able to assign extra behaviors to objects at runtime without breaking the code that uses these objects.

Addition of functionalities

H

Beyond patterns

Python logging module implements its own Composition Over Inheritance pattern.

1. Filtering or output implementations are isolated. Python logging module maintains a list of filters and a list of handlers.

2. For each log message, the logger calls each of its filters. The message is discarded if any filter rejects it.

3. For each log message that's accepted by all the filters, the logger loops over its output handlers and asks every one of them to emit() the message.

Beyond patterns

```
class Logger:
    def __init__(self, filters, handlers):
        self.filters = filters
        self.handlers = handlers

    def log(self, message):
        if all(f.match(message) for f in self.filters):
            for h in self.handlers:
                h.emit(message)
```

Filters

```
class TextFilter:
    def __init__(self, pattern):
        self.pattern = pattern

    def match(self, text):
        return self.pattern in text
```

- Reusable.
- Decoupled from logging.
- Easier to test and maintain.

All of the previous designs either hid filtering inside one of the logging classes itself, or saddled filters with additional duties beyond simply rendering a verdict.

Handlers

```
class FileHandler:
    def __init__(self, file):
        self.file = file

    def emit(self, message):
        self.file.write(message + '\n')
        self.file.flush()

class SocketHandler:
    def __init__(self, sock):
        self.sock = sock

    def emit(self, message):
        self.sock.sendall((message + '\n').encode('ascii'))

class SyslogHandler:
    def __init__(self, priority):
        self.priority = priority

    def emit(self, message):
        syslog.syslog(self.priority, message)
```

Final implementation

```
f = TextFilter('Error')
h = FileHandler(sys.stdout)
logger = Logger([f], [h])

logger.log('Ignored: this will not be logged')
logger.log('Error: this is important')
```

Lesson:

Design principles like Composition Over Inheritance are, in the end, more important than individual patterns like the Adapter or Decorator.

Always follow the principle. But don't always feel constrained to choose a pattern from an official list.

Avoid: If statements

When a new design requirement appears, does the typical Python programmer really go write a new class? No! “Simple is better than complex.”

Why add a class, when an if statement will work instead?

```
class Logger:
    def __init__(self, pattern=None, file=None, sock=None, priority=None):
        self.pattern = pattern
        self.file = file
        self.sock = sock
        self.priority = priority

    def log(self, message):
        if self.pattern is not None:
            if self.pattern not in message:
                return
        if self.file is not None:
            self.file.write(message + '\n')
            self.file.flush()
        if self.sock is not None:
            self.sock.sendall((message + '\n').encode('ascii'))
        if self.priority is not None:
            syslog.syslog(self.priority, message)
```

If statements problems

1. Locality. If you are tasked with improving or debugging one particular feature — say, the support for writing to a socket — you will find that you can't read its code all in one place.

2. Deletability. An underappreciated property of good design is that it makes deleting features easy. Can it be removed?

3. Testing. One of the strongest signals about code health that our tests provide is how many lines of irrelevant code have to run before reaching the line under test.

5. Efficiency. Even if you want a simple unfiltered log to a single file, every single message will be forced to run an if statement against every possible feature you could have enabled.

Avoid: Multiple inheritance

Controversial feature of the Python language: multiple inheritance.

```
class Logger(object):
    def __init__(self, file):
        self.file = file

    def log(self, message):
        self.file.write(message + '\n')
        self.file.flush()

class SocketLogger(Logger):
    def __init__(self, sock):
        self.sock = sock

    def log(self, message):
        self.sock.sendall((message + '\n').encode('ascii'))
```

```
class FilteredLogger(Logger):
    def __init__(self, pattern, file):
        self.pattern = pattern
        super().__init__(file)

    def log(self, message):
        if self.pattern in message:
            super().log(message)
```

```
class FilteredSocketLogger(FilteredLogger, SocketLogger):
    def __init__(self, pattern, sock):
        FilteredLogger.__init__(self, pattern, None)
        SocketLogger.__init__(self, sock)

logger = FilteredSocketLogger('Error', sock1)
logger.log('Warning: not that important')
logger.log('Error: this is important')
```

Avoid: Multiple inheritance

Similar to Decorator Pattern solution:

1. There's a logger class for each kind of output.
2. The message preserves the exact value provided by the caller (instead of our Adapter's habit of replacing it with a file-specific value by appending a newline).
3. The filter and loggers are symmetric in that they both implement the same method log().
4. The filter never tries to produce output on its own but, if a message survives filtering, defers the task of output to other code.

Multiple inheritance problems

The success of the Decorator example depends only on the public behaviors of each class. The public behavior of a FilteredLogger is that it offers a log() method that both filters and writes to a file.

```
class SyslogLogger:
    def __init__(self, priority):
        self.priority = priority

    def log(self, message):
        syslog.syslog(self.priority, message)

class LogFilter:
    def __init__(self, pattern, logger):
        self.pattern = pattern
        self.logger = logger

    def log(self, message):
        if self.pattern in message:
            self.logger.log(message)
```

```
class Logger(object):
    def __init__(self, file):
        self.file = file

    def log(self, message):
        self.file.write(message + '\n')
        self.file.flush()

class FilteredLogger(Logger):
    def __init__(self, pattern, file):
        self.pattern = pattern
        super().__init__(file)

    def log(self, message):
        if self.pattern in message:
            super().log(message)
```

Multiple inheritance problems

Multiple inheritance has introduced a new _init_() method because neither base class's _init_() method accepts enough arguments for a combined filter and logger.

That new code needs to be tested, so at least one test will be necessary for every new subclass. (Using **kwargs solution will hurt readability).

Finally, it's possible that two classes work fine on their own, but have class or instance attributes with the same name that will collide when the classes are combined through multiple inheritance.

Avoid: Mixins

We can convert the FilteredLogger to a “mixin” that lives entirely outside the class hierarchy with which multiple inheritance will combine it.

```
class FilterMixin:  # No base class!
    pattern = ''

    def log(self, message):
        if self.pattern in message:
            super().log(message)

class FilteredLogger(FilterMixin, FileLogger):
    pass  # Again, the subclass needs no extra code.

logger = FilteredLogger(sys.stdout)
logger.pattern = 'Error'
logger.log('Warning: not that important')
logger.log('Error: this is important')
```

The mixin has no base class that might complicate method resolution order, so super() will always call the next base class listed in the class statement.

Avoid: Mixins

A mixin also has a simpler testing story than the equivalent subclass. Whereas the FilteredLogger would need tests that both run it standalone and also combine it with other classes, the FilterMixin only needs tests that combine it with a logger. Because the mixin is by itself incomplete, a test can't even be written that runs it standalone.

But all the other liabilities of multiple inheritance still apply. So while the mixin pattern does improve the readability and conceptual simplicity of multiple inheritance, it's not a complete solution for its problems.

1

Python Patterns

Global Object Pattern

Global Object Pattern

Python parses the outer level of each module as normal code.

Un-indented assignment statements, expressions, and even loops and conditionals will execute as the module is imported.

This presents an excellent opportunity to supplement a module's classes and functions with constants and data structures that callers will find useful.

Sadly, this also offers dangerous temptations: mutable global objects can wind up coupling distant code, and I/O operations impose import-time expense and side effects.

Every Python module is a separate namespace.

Constant pattern

Modules often assign useful numbers, strings, and other values to names in their global scope.

```
January = 1                      # calendar.py
WARNING = 30                     # logging.py
MAX_INTERPOLATION_DEPTH = 10     # configparser.py
SSL_HANDSHAKE_TIMEOUT = 60.0     # asyncio.constants.py
TICK = "'"                       # email.utils.py
CRLF = "\r\n"                    # smtplib.py
```

Constants are often introduced as a refactoring. We normally introduce a constant when they are appearing repeatedly in their code.

The constant's name now documents the value's meaning, improving the code's readability. It also provides a single location where the value can be edited

Constant pattern

Constants can be used also as an API.

For example: A constant like WARNING from the logging module offers the advantages of a constant to the caller: code will be more readable, and the constant's value could be adjusted later without every caller needing to edit their code.

Sometimes constants are introduced for efficiency, to avoid recomputing a value every time code is called.

```
ZIP_FILECOUNT_LIMIT = (1 << 16) - 1

COPY_BUFSIZE = 1024 * 1024 if _WINDOWS else 16 * 1024
```

Dunder constants

A special case of constants defined at a module's global level are "dunder" constants whose names start and end with double underscores. Several Module Global dunder constants are set by the language itself.

```
__name__ # If the source file is executed as the main program, the interpreter sets the __name__
variable to have a value "__main__". If this file is being imported from another module, __name__
will be set to the module's name.

__file__ # the full filesystem path to the module's Python file itself
__all__ # It's a list of public objects of that module, as interpreted by import *
```

An early reader probably misunderstood dunders, which really meant "special to the Python language runtime," as a vague indication that a value was module metadata rather than module code.

~~`__version__`~~
~~`__author__`~~

Global Object Pattern

A module instantiates an object at import time and assigns it a name in the module's global scope.

But the object does not simply serve as data; it is not merely an integer, string, or data structure. Instead, the object is made available for the sake of the methods it offers — for the actions it can perform.

A common example is a compiled regular expression:

```
escapesre = re.compile(r'[\\"]')        # email/utils.py
magic_check = re.compile('([*?[])')     # glob.py
commentclose = re.compile(r'--\s*>')    # html/parser.py
HAS_UTF8 = re.compile(b'[\x80-\xff]')   # json/encoder.py
```

Global Object Pattern tradeoffs

The cost of importing the module increases by the cost of compiling the regular expression (plus the tiny cost of assigning it to a global name).

The import-time cost is now borne by every program that imports the module.

The compiled regular expression is ready to start scanning a string immediately! If the regular expression is used frequently, like in the inner loop of a costly operation like parsing, the savings can be considerable.

The global name will make calling code more readable.

Global Objects that are mutable discussion:

https://python-patterns.guide/python/module-globals/#global-objects-that-are-mutable

The Prebound Method Pattern

Problem

Imagine that we want to offer a simple random number generator.

It returns, in an endless loop, the numbers 1 through 255 in a fixed pseudo-random order.

We also want to offer a simple set_seed() routine that resets the state of the generator to a known value — which is important both for tests that use random numbers, and for simulations driven by pseudo-randomness that want to offer reproducible results.

```
from datetime import datetime

_seed = datetime.now().microsecond % 255 + 1

def set_seed(value):
    global _seed
    _seed = value

def random():
    global _seed
    _seed, carry = divmod(_seed, 2)
    if carry:
        _seed ^= 0xb8
    return _seed
```

Problem

It is impossible to ever instantiate a second copy of this random number generator.

It is more difficult to decouple your random number generator tests from each other if the generator's state is global.

This approach abandons encapsulation

New solution

```
from datetime import datetime

class Random8(object):
    def __init__(self):
        self.set_seed(datetime.now().microsecond % 255 + 1)

    def set_seed(self, value):
        self.seed = value

    def random(self):
        self.seed, carry = divmod(self.seed, 2)
        if carry:
            self.seed ^= 0xb8
        return self.seed
```

This imposes an extra step on the caller: the object will have to be instantiated before the two methods can be called.

The Prebound Method Pattern

A powerful technique for offering callables at the top level of your module that share state through a common object.

There are occasions on which a Python module wants to offer several routines in the module's global namespace that will need to share state with each other at runtime.

The pattern

To offer your users a slate of Prebound Methods:

Instantiate your class at the top level of your module.

Consider assigning it a private name prefixed with an underscore _ that does not invite users to meddle with the object directly.

Finally, assign a bound copy of each of the object's methods to the module's global namespace.

The pattern

Users will now be able to invoke each method as though it were a stand-alone function. But the methods will secretly share state thanks to the common instance that they are bound to, without requiring the user to manage or pass that state explicitly.

```
from datetime import datetime

class Random8(object):
    def __init__(self):
        self.set_seed(datetime.now().microsecond % 255 + 1)

    def set_seed(self, value):
        self.seed = value

    def random(self):
        self.seed, carry = divmod(self.seed, 2)
        if carry:
            self.seed ^= 0xb8
        return self.seed

_instance = Random8()

random = _instance.random
set_seed = _instance.set_seed
```

The Prebound Method Pattern

Instantiating a random number generator requires a system call — in our case, asking for the date; for the Python random module, fetching bytes from the system entropy pool. If every module needing a random number had to instantiate its own Random object, then this cost would be incurred repeatedly.

Pseudo-random number generators are an interesting case of a resource whose behavior can be even more desirable when shared.

calendar.py

```
c = TextCalendar()

...

monthcalendar = c.monthdayscalendar
prweek = c.prweek
week = c.formatweek
weekheader = c.formatweekheader
prmonth = c.prmonth
month = c.formatmonth
calendar = c.formatyear
prcal = c.pryear
```

The Sentinel Object Pattern

The Sentinel Object Pattern

The Sentinel Object pattern is a standard Pythonic approach that's used both in the Standard Library and beyond.

The pattern most often uses Python's built-in None object, but in situations where None might be a useful value, a unique sentinel object() can be used instead to indicate missing or unspecified data.

Sentinel Value

Often saves a line of code and a bit of indentation

```
try:
    i = a.index(b)
except:
    return

# versus
# Sentinel value = -1 when the substring is not found

i = a.find(b)
if i == -1:
    return
```

If str.find() had been invented today, it would instead have used the Sentinel Object pattern that we will describe later by simply returning None for “not found”. That would have left no possibility of the return value being used accidentally as an index.

The Null Object Pattern

"Null objects" are real, valid objects that happen to represent a blank value or an item that does not exist.

```
for e in employees:
    if e.manager is None:
        m = 'no one'
    else:
        m = e.manager.display_name()
    print(e.name, '-', m)
```

Woolf offers the intriguing alternative of replacing all of the None manager values with an object specifically designed to represent the idea of "no one":

```
NO_MANAGER = Person(name='no acting manager')
```

Sentinel Object Pattern

There are two interesting circumstances where programs need an alternative to None:

1. A general purpose data store doesn't have the option of using None for missing data if users might themselves try to store the None object.

```
sentinel = object()  # unique object used to signal cache
                     misses

result = cache_get(key, sentinel)
if result is not sentinel:
    ...
```

Sentinel Object Pattern

2. The second interesting circumstance that calls for a sentinel is when a function or method wants to know whether a caller supplied an optional keyword argument or not.

Usually Python programmers give such an argument a default of None.

But if your code truly needs to know the difference, then a sentinel object will allow you to detect it.

```
class Logger:
    def __init__(self, pattern=None, file=None, sock=None, priority=None):
        self.pattern = pattern
        self.file = file
        self.sock = sock
        self.priority = priority
```

HUGE

Done.

Python Design Patterns
September 30, 2020