• Home Page •

The Flyweight Pattern ¶

A "Structural Pattern" from the Gang of Four book

Verdict

> Flyweight objects are such a perfect fit for Python that the language itself uses the pattern for such popular values as identifiers, some integers, and Boolean true and false.

A perfect example of the Flyweight Pattern is the Python `intern()` function. It's a builtin in Python 2 which was moved into the `sys` module in Python 3. When you pass it a string, it returns an exactly equal string. Its advantage is that it saves space: no matter how many different string objects you pass it for a particular value like `'xyzzy'`, it returns the same `'xyzzy'` object each time.

It's used internally in Python to save memory. As Python parses your program it's likely to encounter a given identifier like `range` several times. Instead of storing each of them as a separate string object, it uses `intern()` so that all mentions of `range` in your code can share a single string object in memory that represents them all.

We can see its behavior by computing the string `'python'` two different ways (to make it likely that any given Python implementation will really give us two different strings) and pass them to the intern function:

```
>>> from sys import intern   # not necessary in Python 2
>>> a = intern('py' + 'thon')
>>> b = intern('PYTHON'.lower())
>>> a
'python'
>>> b
'python'
>>> a is b
True
```

Strings are natural candidates for the Flyweight Pattern because they have all three of the key properties of a flyweight object:

- Python strings are immutable, which makes them safe to share. Otherwise a routine that decided to change one of the string's characters would affect the single copy shared with everyone else.
- A Python string carries no context about how it is being used. If it needed to maintain a reference back to the list, dictionary, or other object that was using it,

then each string could only serve in one context at a time.

- Strings are important for their value, not their object identity. We compare them with `==` instead of with the `is` keyword. A well-written Python program will not even notice whether the string `"brandon"` used in one place as a directory name and somewhere else as a username are the same object or two different objects.

The Gang of Four describe these same requirements a bit differently when they require that, “Most object state can be made extrinsic.” They imagine starting with an object that’s a mix of what they call “extrinsic” and “intrinsic” state:

```
a1 = Glyph(width=6, ascent=9, descent=3, x=32, y=8)
a2 = Glyph(width=6, ascent=9, descent=3, x=8, y=60)
```

Given a typeface and size, each occurrence of a given letter — say, the letter *M* — will have the same width, the same ascent above the baseline, and the same descent below it. The Gang of Four call these attributes “intrinsic” to what it means to be the letter *M.* But each *M* on a page will have a different `x` and `y` coordinate; that state is “extrinsic” and varies from one occurrence of the letter to another.

Given an object that mixes intrinsic and extrinsic state, the Gang of Four arrives at the Flyweight by refactoring to separate the two kinds of state:

```
a = Glyph(width=6, ascent=9, descent=3)
a1 = DrawnGlyph(glyph=a, x=32, y=8)
a2 = DrawnGlyph(glyph=a, x=8, y=60)
```

Not only can the space savings from the Flyweight Pattern be considerable, but the original 1990 paper introducing Flyweights found that a document editor written using the pattern had considerably simpler code.

Factory or Constructor ¶

The Gang of Four only imagined using a factory function like `intern()` for managing a collection of flyweights, but Python often moves the logic into a class’s constructor instead.

The simplest example in Python is the `bool` type. It has exactly two instances. While they can be accessed through their builtin names `True` and `False`, they are also returned by their class when it is passed an object to test for truthiness or falsehood.

```
>>> bool(0)
False
>>> bool('')
False
>>> bool(12)
True
```

Another example is integers. As an implementation detail, the default C language version of Python treats the integers -5 through 256 as flyweights. Those integers are created ahead of time as the interpreter launches, and are returned when an integer with one of those values is needed. Computing any other integer value results in a unique object from each computation.

```
>>> 1 + 4 is 2 + 3
True
>>> 100 + 400 is 200 + 300
False
```

There are a few other flyweights hiding in the Standard Library for very common immutable objects, like the empty string and the empty tuple.

```
>>> str() is ''
True
>>> tuple([]) is ()
True
```

Note that not every object pre-built by the interpreter qualifies as a flyweight. The `None` object, for example, does not qualify: a class needs more than one instance to be a true Flyweight, but `None` is the only instance of `NoneType`.

Implementing ¶

The simplest flyweights are allocated ahead of time. A system for assigning letter grades might use flyweights for the grades themselves:

```
_grades = [letter + suffix
           for letter in 'ABCD'
           for suffix in ('+', '', '-')] + ['F']

def compute_grade(percent):
    percent = max(59, min(99, percent))
    return _grades[(99 - percent) * 3 // 10]

print(compute_grade(55))
print(compute_grade(89))
print(compute_grade(90))
```

```
F
B+
A-
```

Factories that need to build a flyweight population dynamically are more complicated: they'll need a dynamic data structure in which to enroll the flyweights and find them again later. A dictionary is a typical choice:

```
_strings = {}

def my_intern(string):
    s = _strings.setdefault(string, string)
    return s

a1 = my_intern('A')
b1 = my_intern('B')
```

```
a2 = my_intern('A')

print(a1 is b1)
print(a1 is a2)
```

```
False
True
```

One danger of dynamically allocated flyweights is the possibility of eventually exhausting memory, if the number of possible values is very large and callers might request a large number of unique values over a program's runtime. In such cases you might consider using a `WeakValueDictionary` from the `weakref` module.

Weak references wouldn't work in the simple example given above, because `my_intern` uses each interned string not only as a value but also as the corresponding key. But it should work fine in the more common case where the indexes are simple values but the keys are more complicated object instances.

The Gang of Four define the Flyweight Pattern as using a factory function, but Python provides another possibility: a class can implement the pattern right in its constructor, just like `bool()` and `int()`. Rewriting the above example as a class — and, for the sake of example, allocating objects on-demand instead of building them ahead of time — would produce something like:

```
class Grade(object):
    _instances = {}

    def __new__(cls, percent):
        percent = max(50, min(99, percent))
        letter = 'FDCBA'[(percent - 50) // 10]
        self = cls._instances.get(letter)
        if self is None:
            self = cls._instances[letter] = object.__new__(Grade)
            self.letter = letter
        return self

    def __repr__(self):
        return 'Grade {!r}'.format(self.letter)

print(Grade(55), Grade(85), Grade(95), Grade(100))
print(len(Grade._instances))     # number of instances
print(Grade(95) is Grade(100))   # ask for 'A' two more times
print(len(Grade._instances))     # number stayed the same?
```

```
Grade 'F' Grade 'B' Grade 'A' Grade 'A'
3
True
3
```

You can see that once a `Grade` object for *A* has been created, all further requests for it receive the same object; the instances dictionary doesn't continue to grow.

Note that we don't define `__init__()` in a class like this whose `__new__()` might return an existing object. That's because Python always calls `__init__()` on the object received back from `__new__()` (as long as the object is an instance of the class itself), which would be useful the first time we returned a new Flyweight object, but redundant on subsequent occasions when we returned the already-initialized object. So we instead do the work of initialization right in the middle of `__new__()`:

```
self.letter = letter
```

Having illustrated the possibility of hiding your Flyweight Pattern factory inside of `__new__()`, I recommend against it because it produces code whose behavior does not match its spelling. When a Python programmer sees `Grade(95)`, they are going to think "new object instance" along with all of the consequences, unless they are in on the secret that `__new__()` has been overridden and unless they always remember that fact when reading code.

A traditional Flyweight Pattern factory function will be less likely to trigger assumptions like "this code is building a new object" and in any case is simpler both to implement and debug.
