The Sentinel Object Pattern ¶

A Python variation on the traditional Sentinel Value pattern

Verdict

> 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.

Contents:

Programming is easiest in problem domains where values are always specified: where everyone in the database is guaranteed to have a name, where we know the age of every employee, and where a datum was collected successfully for every second of the experiment.

But the world is rarely that simple, and so patterns are needed for those cases where object attributes or whole objects go missing. What simple mechanisms are available to distinguish useful data from placeholders that indicate data is absent?

Sentinel Value ¶

The traditional Sentinel Value pattern will be familiar to Python programmers from the `str.find()` method. While its alternative `str.index()` is more rigorous, raising an exception if it can't find the substring you're asking about, `find()` lets you skip writing an exception handler for a missing substring by returning the sentinel value `-1` when the substring is not found. This often saves a line of code and a bit of indentation:

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

```
# versus

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

This is a classic example of a sentinel value. The value `-1` is simply an integer, just like the function's other possible return values, but with a special meaning that has been agreed upon ahead of time — and woe betide the program that is returned `-1`, forgets to check, and tries using it as an index into the string! The result will not be what the programmer intended.

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

Sentinel values are particularly common today in the Go language, whose design encourages a programming style that always returns strings instead of mere references to them — forcing the programmer to choose some particular string, like the empty string or a special unique sentinel, to indicate that no data was collected or is present.

In Python, the Django framework is famous for contradicting several decades of database practice by recommending that you "Avoid using null on string-based fields" — with the frequent result that, as in Go, code becomes simpler; checks for the empty string replace checks for `None`; and the program no longer crashes when later code tries invoking a string method on what turns out to be `None`.

The Null Pointer Pattern ¶

The null pointer pattern is impossible in Python, but worth mentioning to outline the difference between Python and languages that complicate their data model with `nil` or `NULL` pointers.

Every name in Python either does not exist, or exists and refers to an object. You can remove a name with `del name`, or else you can assign a new object to it; Python offers no other alternatives. Behind the scenes, each name in Python is a pointer that stores the address of the object to which it currently refers. Even if the name points to an object as simple as the `None` object, it will contain a valid address.

This guarantee supports an interesting sentinel pattern down in the default C language implementation of Python. The C language lacks Python's guarantee that a name — which C calls a "pointer" — will always hold the address of a valid object. Taking advantage of this flexibility, C programmers use an address of zero to mean "this pointer currently doesn't point at anything" — which makes zero, or `NULL` as many C programs define it, a sentinel value. Pointers which might be `NULL` need to be checked before being used, or the program will die with a `segmentation fault`.

The fact that all Python values, even `None` and `False`, are real objects with non-zero addresses means that Python functions implemented in C have the value `NULL` available to mean something special: a `NULL` pointer means “this function did *not* complete and return a value; instead, it raised an exception.” This allows the C code beneath Python to avoid the two-value return pattern that pervades Go code:

```
# Go needs to separately represent “the return value”
# and “did this die with an error.”

byte_count, err := fmt.Print("Hello, world!")
if err != nil {
        ...
}
```

Instead, C language routines that call Python can distinguish legitimate return values from an exception using only the single return value supported by C functions:

```
# The pointer to a Python object instead means
# “an exception was raised” if its value is NULL.

PyObject *my_repr = PyObject_Repr(obj);
if (my_repr == NULL) {
      ...
}
```

The exception itself is stored elsewhere and can be retrieved using the Python C API.

The Null Object Pattern ¶

“Null objects” are real, valid objects that happen to represent a blank value or an item that does not exist. My attention was drawn to this pattern while reading the book Refactoring by Martin Fowler which credits Bobby Woolf for its explication. Note that this pattern has nothing to do with the “null pointer” explained in the previous section! Instead it describes a special kind of sentinel object.

Imagine a sequence of `Employee` objects which usually have another employee as their `manager` attribute but not always. The default Pythonic approach to represent “no manager” would be to assign `None` to the attribute.

A routine tasked with displaying an employee profile will have to check for the sentinel object `None` before trying to invoke any methods on the manager:

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

And this pattern will need to be repeated everywhere that code touches the manager attribute.

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')
```

Employee objects will now be assigned this `NO_MANAGER` object instead of `None`, and both kinds of code touching employee managers will benefit:

- Code that produces simple displays or summaries can simply print or tally the `NO_MANAGER` manager object as though it were a normal employee object. When code can run successfully against the Null Object, the need for a special `if` statement disappears.
- Code that does need to specially handle the case of an employee with no acting manager now becomes a bit more readable. Instead of using the generic `is None` it will perform the check with the specific `is NO_MANAGER` and will thereby gain a bit more readability.

While not appropriate in all situations — it can, for example, be difficult to design Null Objects that keep averages and other statistics valid — Null Objects appear even in the Python Standard Library: the `logging` module has a `NullHandler` which is a drop-in replacement for its other handlers but does no actual logging.

Sentinel Objects ¶

Finally we come to the Sentinel Object pattern itself.

The standard Python sentinel is the built-in `None` object, used wherever some alternative needs to be provided to an integer, float, string, or other meaningful value. For most programs it is entirely sufficient, and its presence can be infallibly tested with:

```
if other_object is None:
    ...
```

But there are two interesting circumstances where programs need an alternative to `None`.

First, 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.

As an example, the Python Standard Library's `functools.lru_cache()` uses the Sentinel Object pattern internally. Hidden inside of a closure is an utterly unique object that it creates separately for each separate instance of the cache:

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

By providing this sentinel object as the second argument to `dict.get()` — here aliased to the name `cache_get` in a closure-level private example of the Prebound Methods pattern — the cache can distinguish a function call whose result is already cached and happened to be `None` from a function call that has not yet been cached:

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

This pattern occurs several times in the Standard Library.

- As shown above, `functools.lru_cache()` uses a sentinel object internally.
- The `bz2` module has a global `_sentinel` object.
- The `configparser` module has a sentinal `_UNSET` also defined as a module global.

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.

An early description of using sentinels for parameter defaults was Fredrik Lundh's "Default Parameter Values in Python" which over the years was followed by posts from Ian Bicking "The Magic Sentinel" and Flavio Curella "Sentinel values in Python" who both worried about their sentinel objects' lack of a readable `repr()` and came up with various fixes.

But whatever the application, the core of the Sentinel Object pattern is that it is the object's identity — *not* its value — that lets the surrounding code recognize its significance. If you are using an equality operator to detect the sentinel, then you are merely using the Sentinel Value pattern described at the top of this page. The Sentinel Object is defined by its use of the Python `is` operator to detect whether the sentinel is present.
