Pythonic Design Patterns

Aaron Maxwell

aaron@powerfulpython.com

Contents

Chapter 1

Classes and Objects: Beyond The Basics

This chapter assumes you are familiar with Python's OOP basics: creating classes, defining methods, and using inheritance. We build on this.

As with any object-oriented language, it's useful to learn about **design patterns** - reusable solutions to common problems involving classes and objects. A LOT has been written about design patterns. Curiously, though, much of what's out there doesn't completely apply to Python - or, at least, it applies *differently*.

That's because many of these design-pattern books, articles, and blog posts are for languages like Java, C++ and C#. But as a language, Python is quite different. Its dynamic typing, first-class functions, and other additions all mean the "standard" design patterns just work differently.

So let's learn what Pythonic OOP is *really* about.

1.1 Quick Note on Python 2

This chapter uses Python 3 syntax. Python 2.7 is nearly the same, and I'll point out the few differences as we go along. But there is one *big* difference worth emphasizing here.

In modern Python, all classes need to inherit from a built-in base class called `object`. (It's lowercased, defying the normal convention.) This happens automatically for all classes in Python 3:

```
>>> # Python 3
... class Dog:
...     def speak(self):
...         return "woof"
...
>>> dog = Dog()
>>> isinstance(dog, object)
True
```

In Python 2, you must explicitly inherit your classes from `object`. Fail to do this, and your class builds on "old-style classes":

```
>>> # Python 2
... class DogFromObject(object):
...     def speak(self):
...         return "woof"
...
>>> class DogNotFromObject:
...     def speak(self):
...         return "woof"
...
>>> issubclass(DogFromObject, object)
True
>>> issubclass(DogNotFromObject, object)
False
```

If you don't already base your Python 2 classes on `object`, start today. Old-style classes are long obsolete, and removed in Python 3; they partially or completely break many important tools in Python's object system, like properties and `super()`. The rest of this chapter assumes you're inheriting from `object`.

1.2 Properties

In object-oriented programming, a *property* is a special sort of object attribute. It's almost a cross between a method and an attribute. The idea is that you can, when designing the class, create "attributes" whose reading, writing, and so on can be managed by special methods. In Python, you do this with a decorator named `property`. Here's an example:

```
class Person:
    def __init__(self, firstname, lastname):
        self.firstname = firstname
        self.lastname = lastname

    @property
    def fullname(self):
        return self.firstname + " " + self.lastname
```

By instantiating this, I can access `fullname` as a kind of virtual attribute:

```
>>> joe = Person("Joe", "Smith")
>>> joe.fullname
'Joe Smith'
```

Notice carefully the members here: there are two attributes called `firstname` and `lastname`, set in the constructor. There is also a method called `fullname`. But after creating the object, we reference `joe.fullname` as an attribute; we don't call `joe.fullname()` as a method.

This is all due to the `@property` decorator. When applied to a method, this decorator makes it inaccessible as a method. You must access it as an attribute. In fact, if you try to call it as a method, you get an error:

```
>>> joe.fullname()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object is not callable
```

As defined above, `fullname` is read-only. We can't modify it:

```
>>> joe.fullname = "Joseph Smith"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

In other words, Python properties are read-only by default. Another way of saying this is that `@property` automatically defines a *getter*, but not a *setter*. If you do want `fullname` to be writable, here is how you define the setter:

```
class Person:
    def __init__(self, firstname, lastname):
        self.firstname = firstname
        self.lastname = lastname

    @property
    def fullname(self):
        return self.firstname + " " + self.lastname

    @fullname.setter
    def fullname(self, value):
        self.firstname, self.lastname = value.split(" ", 1)
```

This lets us assign to `joe.fullname`:

```
>>> joe = Person("Joe", "Smith")
>>> joe.firstname
'Joe'
>>> joe.lastname
'Smith'
>>> joe.fullname = "Joseph Smith"
>>> joe.firstname
'Joseph'
>>> joe.lastname
'Smith'
```

The first time I saw this, I had all sorts of questions. "Wait, why is `fullname` defined twice? And why is the second decorator named `@fullname`, and what's this `setter` attribute? How on earth does this even compile?"

The code is actually correct, and designed to work this way. The `@property def fullname` must come first. That creates the property to begin with, and *also* creates the getter. By "create the property", I mean that an object named `fullname` exists *in the namespace of the class*, and it has a method named `fullname.setter`. This `fullname.setter` is a decorator that is applied to the next `def fullname`, christening it as the setter for the `fullname` property.

It's okay to not fully understand how this all works. A full explanation relies on understanding both implementing decorators, and Python's descriptor protocol, both of which are beyond the scope of what we want to focus on here. Fortunately, you don't have to understand *how* it works in order to use it.

(Besides getting and setting, you can handle the del operation for the object attribute by decorating with @fullname.deleter. You won't need this very often, but it's available when you do.)

What you see here with the Person class is one way properties are useful: magic attributes whose values are derived from other values. This denormalizes the object's data, and lets you access the property value as an attribute instead of as a method. You'll see a situation where that's extremely useful later.

Properties enable a useful collection of design patterns. One - as mentioned - is in creating read-only member variables. In Person, the fullname "member variable" is a dynamic attribute; it doesn't exist on its own, but instead calculates its value at run-time.

It's also common to have the property backed by a single, non-public member variable. That pattern looks like this:

```
class Coupon:
    def __init__(self, amount):
        self._amount = amount
    @property
    def amount(self):
        return self._amount
```

This allows the class itself to modify the value internally, but prevent outside code from doing so:

```
>>> coupon = Coupon(1.25)
>>> coupon.amount
1.25
>>> coupon.amount = 1.50
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

In Python, prefixing a member variable by a single underscore signals the variable is non-public, i.e. it should only be accessed internally, inside methods of that class, or its subclasses.[1] What this pattern says is "you can access this variable, but not change it".

Between "regular member variable" and "ready-only" is another pattern: allow changing the attribute, but validate it first. Suppose my event-management application has a Ticket class,

[1]This isn't enforced by Python itself. If your teammates don't already honor this widely-followed convention, you'll have to educate them.

representing tickets sold to concert-goers:

```
class Ticket:
    def __init__(self, price):
        self.price = price
    # And some other methods...
```

One day, we find a bug in our web UI, which lets some shifty customers adjust the price to a negative value... so we ended up actually *paying* them to go to the concert. Not good!

The first priority is, of course, to fix the bug in the UI. But how do we modify our code to prevent this from ever happening again? Before reading further, look at the `Ticket` class and ponder - how could you use properties to make this kind of bug impossible in the future?

The answer: verify the new price is non-zero in the setter:

```
# Version 1...
class Ticket:
    def __init__(self, price):
        self._price = price
    @property
    def price(self):
        return self._price
    @price.setter
    def price(self, new_price):
        # Only allow positive prices.
        if new_price < 0:
            raise ValueError("Nice try")
        self._price = new_price
```

This lets the price be adjusted... but only to sensible values:

```
>>> t = Ticket(42)
>>> t.price = 24 # This is allowed.
>>> print(t.price)
24
>>> t.price = -1 # This is NOT.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 11, in price
ValueError: Nice try
```

However, there's a defect in this new `Ticket` class. Can you spot what it is? (And how to fix it?)

The problem is that while we can't *change* the price to a negative value, this first version lets us *create* a ticket with a negative price to begin with. That's because we write `self._price = price` in the constructor. The solution is to use the *setter* in the constructor instead:

```python
# Final version, with modified constructor. (Constructor
# is different; code for getter & setter is the same.)
class Ticket:
    def __init__(self, price):
        # instead of "self._price = price"
        self.price = price
    @property
    def price(self):
        return self._price
    @price.setter
    def price(self, new_price):
        # Only allow positive prices.
        if new_price < 0:
            raise ValueError("Nice try")
        self._price = new_price
```

Yes, you can reference `self.price` in methods of the class. When we write `self.price = price`, Python translates this to calling the `price` setter - i.e., the second `price()` method. This final version of `Ticket` centralizes all reads AND writes of `self._price` in the property. It's a useful encapsulation principle in general. The idea is you centralize any special behavior for that member variable in the getter and setter, even for the class's internal code. In practice, sometimes methods need to violate this rule; you simply reference `self._price` and move on. But avoid that where you can, and you will tend to benefit from higher quality code.

1.2.1 Properties and Refactoring

Properties are important in most languages today. Here's a situation that often plays out. Imagine writing a simple money class:

```
class Money:
    def __init__(self, dollars, cents):
        self.dollars = dollars
        self.cents = cents
    # And some other methods...
```

Suppose you put this class in a library, which many developers are using. People on your current team, perhaps developers on different teams. Or maybe you release it as open-source, so developers around the world use and rely on this class.

Now, one day you realize many of `Money`'s methods - which do calculations on the money amount - can be simpler and more straightforward if they operate on the total number of cents, rather than dollars and cents separately. So you refactor the internal state:

```
class Money:
    def __init__(self, dollars, cents):
        self.total_cents = dollars * 100 + cents
```

This minor change creates a MAJOR maintainability problem. Can you spot it?

Here's the trouble: your original `Money` has attributes named `dollars` and `cents`. And since many developers are using these, changing to `total_cents` breaks all their code!

```
money = Money(27, 12)
message = "I have {:d} dollars and {:d} cents."
# This line breaks, because there's no longer
# dollars or cents attributes.
print(message.format(money.dollars, money.cents))
```

If no one but you uses this class, there's no real problem - you can just refactor your own code. But if that's not the case, coordinating this change with everyone's different code bases is a nightmare. It becomes a barrier to improving your own code.

So, what do you do? Can you think of a way to handle this situation?

You get out of this mess is with properties. You want two things to happen:

1. Use `total_cents` internally, and
2. All code using `dollars` and `cents` continues to work, without modification.

You do this by replacing `dollars` and `cents` with `total_cents` internally, but also creating getters and setters for these attributes. Take a look:

```
class Money:
    def __init__(self, dollars, cents):
        self.total_cents = dollars * 100 + cents
    # Getter and setter for dollars...
    @property
    def dollars(self):
        return self.total_cents // 100
    @dollars.setter
    def dollars(self, new_dollars):
        self.total_cents = 100 * new_dollars + self.cents
    # And for cents.
    @property
    def cents(self):
        return self.total_cents % 100
    @cents.setter
    def cents(self, new_cents):
        self.total_cents = 100 * self.dollars + new_cents
```

Now, I can get and set `dollars` and `cents` all day:

```
>>> money = Money(27, 12)
>>> money.total_cents
2712
>>> money.cents
12
>>> money.dollars = 35
>>> money.total_cents
3512
```

Python's way of doing properties brings many benefits. In languages like Java, the following story often plays out:

1. A newbie developer starts writing Java classes. They want to expose some state, so create public member variables.

2. They use this class everywhere. Other developers use it too.

3. One day, they want to change the name or type of that member variable, or even do away with it entirely (like what we did with `Money`).

4. But that would break everyone's code. So they can't.

Because of this, Java developers quickly learn to make all their variables private by default - proactively creating getters and setters for *every* publicly exposed chunk of data. They realize this boilerplate is far less painful than the alternative, because if everyone must use the public getters and setters to begin with, you always have the freedom to make internal changes later.

This works well enough. But it *is* distracting, and just enough trouble that there's always the temptation to make that member variable public, and be done with it.

In Python, we have the best of both worlds. We make member variables public by default, refactoring them as properties if and when we ever need to. No one using our code even has to know.

1.3 The Factory Patterns

There are several design patterns with the word "factory" in their names. Their unifying idea is providing a handy, simplified way to create useful, potentially complex objects. The two most important forms are:

- Where the object's type is fixed, but we want to have several different ways to create it. This is called the *Simple Factory Pattern*.
- Where the factory dynamically chooses one of several different types. This is called the *Factory Method Pattern*.

Let's look at how you do these in Python.

1.3.1 Alternative Constructors: The Simple Factory

Imagine a simple `Money` class, suitable for currencies which have dollars and cents:

```python
class Money:
    def __init__(self, dollars, cents):
        self.dollars = dollars
        self.cents = cents
```

We looked at this in the previous section, refactoring its attributes - but let's roll back, and focus instead on the constructor's interface. This constructor is convenient when we have the dollars and cents as separate integer variables. But there are many other ways to specify an amount of money. Perhaps you're modeling a giant jar of pennies:

```
# Emptying the penny jar...
total_pennies = 3274
# // is integer division
dollars = total_pennies // 100
cents = total_pennies % 100
total_cash = Money(dollars, cents)
```

Suppose your code splits pennies into dollars and cents over and over, and you're tired of repeating this calculation. You could change the constructor, but that means refactoring all `Money`-creating code, and perhaps a lot of code fits the current constructor better anyway. Some languages let you define several constructors, but Python makes you pick one.

In this case, you can usefully create a *factory function* taking the arguments you want, creating and returning the object:

```
# Factory function taking a single argument, returning
# an appropriate Money instance.
def money_from_pennies(total_cents):
    dollars = total_cents // 100
    cents = total_cents % 100
    return Money(dollars, cents)
```

Imagine that, in the same code base, you also routinely need to parse a string like `"$140.75"`. Here's another factory function for that:

```
# Another factory, creating Money from a string amount.
import re
def money_from_string(amount):
    match = re.search(
        r'^\$(?P<dollars>\d+)\.(?P<cents>\d\d)$', amount)
    if match is None:
        raise ValueError("Invalid amount: " + repr(amount))
    dollars = int(match.group('dollars'))
    cents = int(match.group('cents'))
    return Money(dollars, cents)
```

These are effectively alternate constructors: callables we can use with different arguments, which are parsed and used to create the final object. But this approach has problems. First, it's awkward to have them as separate functions, defined outside of the class. But much more importantly: what happens if you subclass `Money`? Suddenly `money_from_string` and `money_from_pennies` are worthless. The base `Money` class is hard-coded.

Python solves these problems in unique way, absent from other languages: the `classmethod` decorator. Use it like this:

```
class Money:
    def __init__(self, dollars, cents):
        self.dollars = dollars
        self.cents = cents
    @classmethod
    def from_pennies(cls, total_cents):
        dollars = total_cents // 100
        cents = total_cents % 100
        return cls(dollars, cents)
```

The function `money_from_pennies` is now a method of the `Money` class, called `from_pennies`. But it has a new argument: `cls`. When applied to a method definition, `classmethod` modifies how that method is invoked and interpreted. The first argument is not `self`, which would be an *instance* of the class. The first argument is now *the class itself*. In the method body, `self` isn't mentioned at all; instead, `cls` is a variable holding the current class object - `Money` in this case. So the last line is creating a new instance of `Money`:

```
>>> piggie_bank_cash = Money.from_pennies(3217)
>>> type(piggie_bank_cash)
<class '__main__.Money'>
>>> piggie_bank_cash.dollars
32
>>> piggie_bank_cash.cents
17
```

Notice `from_pennies` is invoked off the class itself, not an instance of the class. This already is nicer code organization. But the real benefit is with inheritance:

```
>>> class TipMoney(Money):
...     pass
...
>>> tip = TipMoney.from_pennies(475)
>>> type(tip)
<class '__main__.TipMoney'>
```

This is the *real* benefit of class methods. You define it once on the base class, and all subclasses can leverage it, substituting their own type for cls. **This makes class methods perfect for the simple factory in Python.** The final line returns an instance of cls, using its regular constructor. And cls refers to whatever the current class is: Money, TipMoney, or some other subclass.

For the record, here's how we translate money_from_string:

```python
def from_string(cls, amount):
    match = re.search(
        r'^\$(?P<dollars>\d+)\.(?P<cents>\d\d)$', amount)
    if match is None:
        raise ValueError("Invalid amount: " + repr(amount))
    dollars = int(match.group('dollars'))
    cents = int(match.group('cents'))
    return cls(dollars, cents)
```

Class methods are a superior way to implement factories like this in Python. If we subclass Money, that subclass will have from_pennies and from_string methods that create objects of that subclass, without any extra work on our part. And if we change the name of the Money class, we only have to change it in one place, not three.

This form of the factory pattern is called "simple factory", a name I don't love. I prefer to call it "alternate constructor". Especially in the context of Python, it describes well what @classmethod is most useful for. And it suggests a general principle for designing your classes. Look at this complete code of the Money class, and I'll explain:

```
import re
class Money:
    def __init__(self, dollars, cents):
        self.dollars = dollars
        self.cents = cents
    @classmethod
    def from_pennies(cls, total_cents):
        dollars = total_cents // 100
        cents = total_cents % 100
        return cls(dollars, cents)
    @classmethod
    def from_string(cls, amount):
        match = re.search(
            r'^\$(?P<dollars>\d+)\.(?P<cents>\d\d)$', amount)
        if match is None:
            raise ValueError("Invalid amount: " + repr(amount))
        dollars = int(match.group('dollars'))
        cents = int(match.group('cents'))
        return cls(dollars, cents)
```

You can think of this class as having several constructors. As a general rule, you'll want to make `__init__` the most generic one, and implement the others as class methods. Sometimes, that means one of the class methods will be used more often than `__init__`.

When using a new class, most developer's intuition will be to reach for the default constructor first, without thinking to check the provided class methods - if they even know about that feature of Python in the first place. So in that situation, you may need to educate your teammates. (Hint: Good examples in the class's code docs go a long way.)

1.3.2 Dynamic Type: The Factory Method Pattern

This next factory pattern, called "Factory Method", is quite different. The idea is that the factory will create an object, but will choose its type from one of several possibilities, dynamically deciding at run-time based on some criteria. It's typically used when you have one base class, and are creating an object that can be one of several different derived classes.

Let's see an example. Imagine you are implementing an image processing library, creating classes to read the image from storage. So you create a base `ImageReader` class, and several derived types:

```python
import abc
class ImageReader(metaclass=abc.ABCMeta):
    def __init__(self, path):
        self.path = path
    @abc.abstractmethod
    def read(self):
        pass # Subclass must implement.
    def __repr__(self):
        return '{}({})'.format(self.__class__.__name__, self.path)

class GIFReader(ImageReader):
    def read(self):
        "Read a GIF"

class JPEGReader(ImageReader):
    def read(self):
        "Read a JPEG"

class PNGReader(ImageReader):
    def read(self):
        "Read a PNG"
```

The `ImageReader` class is marked abstract, requiring subclasses to implement the `read` method. So far, so good.

Now, when reading an image file, if its extension is ".gif", I want to use `GIFReader`. And if it is a JPEG image, I want to use `JPEGReader`. And so on. The logic is

- Analyze the file path name to get the extension,
- choose the correct reader class based on that,
- and finally create the appropriate reader object.

This is a prime candidate for automation. Let's define a little helper function:

```python
def extension_of(path):
    position_of_last_dot = path.rfind('.')
    return path[position_of_last_dot+1:]
```

With these pieces, we can now define the factory:

```python
# First version of get_image_reader().
def get_image_reader(path):
    image_type = extension_of(path)
    reader_class = None
    if image_type == 'gif':
        reader_class = GIFReader
    elif image_type == 'jpg':
        reader_class = JPEGReader
    elif image_type == 'png':
        reader_class = PNGReader
    assert reader_class is not None, \
        'Unknown extension: {}'.format(image_type)
    return reader_class(path)
```

Classes in Python can be put in variables, just like any other object. We take full advantage here, by storing the appropriate `ImageReader` subclass in `reader_class`. Once we decide on the proper value, the last line creates and returns the reader object.

This correctly-working code is already more concise, readable and maintainable than what some languages force you to go through. But in Python, we can do better. We can use the built-in dictionary type to make it even more readable and easy to maintain over time:

```python
READERS = {
    'gif' : GIFReader,
    'jpg' : JPEGReader,
    'png' : PNGReader,
    }
def get_image_reader(path):
    reader_class = READERS[extension_of(path)]
    return reader_class(path)
```

Here we have a global variable mapping filename extensions to `ImageReader` subclasses. This lets us readably implement `get_image_reader` in two lines. Finding the correct class is a simple dictionary lookup, and then we instantiate and return the object. And if we support new image formats in the future, we simply add a line in the `READERS` definition. (And, of course, define its reader class.)

What if we encounter an extension not in the mapping, like `tiff`? As written above, the code will raise a `KeyError`. That may be what we want. Or closely related, perhaps we want to catch that, and re-raise a different exception.

Alternatively, we may want to fall back on some default. Let's create a new reader class, meant as an all-purpose fallback:

```
class RawByteReader(ImageReader):
    def read(self):
        "Read raw bytes"
```

Then you can write the factory like:

```
def get_image_reader(path):
    try:
        reader_class = READERS[extension_of(path)]
    except KeyError:
        reader_class = RawByteReader
    return reader_class(path)
```

or more briefly

```
def get_image_reader(path):
    return READERS.get(extension_of(path), RawByteReader)
```

This design pattern is commonly called the "factory method" pattern, which wins my award for Worst Design Pattern Name In History. That name (which appears to originate from a Java implementation detail) fails to tell you anything about what it's actually *for*. I myself call it the "dynamic type" pattern, which I feel is much more descriptive and useful.

1.4 The Observer Pattern

The Observer pattern provides a "one to many" relationship. That's vague, so let's make it more specific.

In the observer pattern, there's one root object, called the *observable*. This object knows how to detect some kind of event of interest. It can literally be anything: a customer makes a new purchase; someone subscribes to an email list; or maybe it monitors a fleet of cloud instances, detecting when a machine's disk usage exceeds 75%. You use this pattern when the code to *reliably* detect the event of interest is at least slightly complicated; that detection code is encapsulated inside the observable.

Now, you also have other objects, called *observers*, which need to know when that event occurs, taking some action in response. You don't want to re-implement the robust detection algorithm in each, of course. Instead, these observers register themselves with the observable. The observable then notifies each observer - by calling a method on that observer - for each event. This separation of concerns is the heart of the observer pattern.

Now, I must tell you, I don't like the names of things in this pattern. The words "observable" and "observer" are a bit obscure, and sound confusingly similar - especially to those whose native tongue is not English. There is another terminology, however, which many developers find easier: *pub-sub*.

In this formulation, instead of "observable", you create a *publisher* object, which watches for events. And you have one or more *subscribers* who ask that publisher to notify them when the event happens. I've found the pattern is easier to reason about when looked at in this way, so that's the terminology I'm going to use.

Let's make this concrete, with code.

1.4.1 The Simple Observer

We'll start with the basic observer pattern, as it's often documented in design pattern books - except we'll translate it to Python. In this simple form, each subscriber must implement a method called `update`. Here's an example:

```python
class Subscriber:
    def __init__(self, name):
        self.name = name
    def update(self, message):
        print('{} got message "{}"'.format(
            self.name, message))
```

`update` takes a string. It's okay to define an `update` method taking other arguments, or even calling it something other than `update`; the publisher and subscriber just need to agree on the protocol. But we'll use a string.

Now, when a publisher detects an event, it notifies the subscriber by calling its `update` method. Here's what a basic `Publisher` class looks like:

```
class Publisher:
    def __init__(self):
        self.subscribers = set()
    def register(self, who):
        self.subscribers.add(who)
    def unregister(self, who):
        self.subscribers.discard(who)
    def dispatch(self, message):
        for subscriber in self.subscribers:
            subscriber.update(message)
    # Plus other methods, for detecting the event.
```

Let's step through:

- A publisher needs to keep track of its subscribers, right? We'll store them in a set object, named `self.subscribers`, created in the constructor.
- A subscriber is added with `register`. Its argument who is an instance of `Subscriber`. Who calls `register`? It could be anyone. The subscriber can register itself; or some external code can register a subscriber with a specific publisher.
- `unregister` is there in case a subscriber no longer needs to be notified of the events.
- When the event of interest occurs, the publisher notifies its subscribers by calling its `dispatch` method. Usually this will be invoked by the publisher itself, in some other method of the class (not shown) that implements the event-detection logic. It simply cycles through the subscribers, calling `.update()` on each.

Using these two classes in code is straightforward enough:

```
# Create a publisher and some subscribers.
pub = Publisher()
bob = Subscriber('Bob')
alice = Subscriber('Alice')
john = Subscriber('John')

# Register the subscribers, so they get notified.
pub.register(bob)
pub.register(alice)
pub.register(john)
```

Now, the publisher can dispatch messages:

```
# Send a message...
pub.dispatch("It's lunchtime!")
# John unsubscribes...
pub.unregister(john)
# ... and a new message is sent.
pub.dispatch("Time for dinner")
```

Here's the output from running the above:

```
John got message "It's lunchtime!"
Bob got message "It's lunchtime!"
Alice got message "It's lunchtime!"
Bob got message "Time for dinner"
Alice got message "Time for dinner"
```

This is the basic observer pattern, and pretty close to how you'd implement the idea in languages like Java, C#, and C++. But Python's feature set differs from those languages. That means we can do different things.

So let's explore that. If we leverage Pythonic features, what does that give us?

1.4.2 A Pythonic Refinement

Python's functions are first-class objects. That means you can store a function in a variable - not the value returned when you call a function, but store the function itself - as well as pass it as an argument to other functions and methods. Some languages support this too (or something like it, such as function pointers), but Python's strong support gives us a convenient opportunity for this design pattern.

The standard observer pattern requires the publisher hard-code a certain method - usually named `update` - that the subscriber must implement. But maybe you need to register a subscriber which doesn't have that method. What then? If it's your own class, you can probably just add it. Or if you are importing the subscriber class from another library (which you can't or don't want to modify), perhaps you can add the method by subclassing it.

Or sometimes you can't do any of those things. Or you *could*, but it's a lot of trouble, and you want to avoid it. What then?

Let's extend the traditional observer pattern, and make `register` more flexible. Suppose you have these subscribers:

```
# This subscriber uses the standard "update"
class SubscriberOne:
    def __init__(self, name):
        self.name = name
    def update(self, message):
        print('{} got message "{}"'.format(
            self.name, message))
# This one wants to use "receive"
class SubscriberTwo:
    def __init__(self, name):
        self.name = name
    def receive(self, message):
        print('{} got message "{}"'.format(
            self.name, message))
```

`SubscriberOne` is the same subscriber class we saw before. `SubscriberTwo` is almost the same: instead of `update`, it has a method named `receive`. Okay, let's modify `Publisher` so it can work with objects of either subscriber type:

```
class Publisher:
    def __init__(self):
        self.subscribers = dict()
    def register(self, who, callback=None):
        if callback is None:
            callback = who.update
        self.subscribers[who] = callback
    def dispatch(self, message):
        for callback in self.subscribers.values():
            callback(message)
    def unregister(self, who):
        del self.subscribers[who]
```

There's a lot going on here, so let's break it down. Look first at the constructor: it creates a `dict` instead of a set. You'll see why in a moment.

Now focus on `register`:

```
    def register(self, who, callback=None):
        if callback is None:
            callback = who.update
        self.subscribers[who] = callback
```

It can be called with one or *two* arguments. With one argument, who is a subscriber of some sort, and callback defaults to None. Inside, callback is set to who.update. Notice the lack of parentheses; who.update is a *method object*. It's just like a function object, except it happens to be tied to an instance. And just like a function object, you can store it in a variable, pass it as an argument to another function, and so on.[2] So we're storing it in a variable called callback.

What if register is called with 2 arguments? Here's how that might look:

```
pub = Publisher()
alice = SubscriberTwo('Alice')
pub.register(alice, alice.receive)
```

alice.receive is another method object; inside, this object is assigned to callback. Regardless of whether register is called with one argument or two, the last line inserts callback into the dictionary:

```
        self.subscribers[who] = callback
```

Take a moment to appreciate the remarkable flexibility of Python dictionaries. Here, you are using an arbitrary instance of either SubscriberOne or SubscriberTwo as a key. These two classes are unrelated by inheritance, so from Python's viewpoint they are completely distinct types. And for that key, you insert a *method object* as its value. Python does this seamlessly, without complaint! Many languages would make you jump through hoops to accomplish this.

Anyway, now it's clear why self.subscribers is a dict and not a set. Earlier, we only needed to keep track to who the subscribers were. Now, we also need to remember the callback for each subscriber. These are used in the dispatch method:

```
    def dispatch(self, message):
        for callback in self.subscribers.values():
            callback(message)
```

dispatch only needs to cycle through the values,[3] because it just needs to call each subscriber's update method (even if it's not called update). Notice we don't have to reference the subscriber object to call that method; the method object internally has a reference to its instance (i.e. its "self"), so callback(message) calls the right method on the right object. In fact, the only reason we keep track of subscribers at all is so we can unregister them.

Let's put this together with a few subscribers:

[2]This is all detailed in the "Advanced Functions" chapter.
[3]In Python 2: Remember, use .viewvalues() instead of .values().

```
pub = Publisher()
bob = SubscriberOne('Bob')
alice = SubscriberTwo('Alice')
john = SubscriberOne('John')

pub.register(bob, bob.update)
pub.register(alice, alice.receive)
pub.register(john)

pub.dispatch("It's lunchtime!")
pub.unregister(john)
pub.dispatch("Time for dinner")
```

Here's the output:

```
Bob got message "It's lunchtime!"
Alice got message "It's lunchtime!"
John got message "It's lunchtime!"
Bob got message "Time for dinner"
Alice got message "Time for dinner"
```

Now, pop quiz. Look at the `Publisher` class again:

```
class Publisher:
    def __init__(self):
        self.subscribers = dict()
    def register(self, who, callback=None):
        if callback is None:
            callback = who.update
        self.subscribers[who] = callback
    def dispatch(self, message):
        for callback in self.subscribers.values():
            callback(message)
```

Here's the question: does `callback` have to be a method of the subscriber? Or can it be a method of a different object, or something else? Think about this before you continue…

It turns out `callback` can be *any callable*, provided it has a signature compatible with how it's called in `dispatch`. That means it can be a method of some other object, or even a normal function. This lets you register subscriber objects without an update method at all:

```
# This subscriber doesn't have ANY suitable method!
class SubscriberThree:
    def __init__(self, name):
        self.name = name
# ... but we can define a function...
todd = SubscriberThree('Todd')
def todd_callback(message):
    print('Todd got message "{}"'.format(message))
# ... and pass it to register:
pub.register(todd, todd_callback)
# And then, dispatch a message:
pub.dispatch("Breakfast is Ready")
```

Sure enough, this works:

```
Todd got message "Breakfast is Ready"
```

1.4.3 Several Channels

So far, we've assumed the publisher watches for only one kind of event. But what if there are several? Can we create a publisher that knows how to detect all of them, and let subscribers decide which they want to know about?

To implement this, let's say a publisher has several *channels* that subscribers can subscribe to. Each channel notifies for a different event type. For example, if your program monitors a cluster of virtual machines, one channel signals when a certain machine's disk usage exceeds 75% (a warning sign, but not an immediate emergency); and another signals when disk usage goes over 90% (much more serious, and may begin to impact performance on that VM). Some subscribers will want to know when the 75% threshold is crossed; some, the 90% threshold; and some might want to be alerted for both. What's a good way to express this in Python code?

Let's work with the mealtime-announcement code above. Rather than jumping right into the code, let's mock up the *interface* first. We want to create a publisher with two channels, like so:

```
# Two channels, named "lunch" and "dinner".
pub = Publisher(['lunch', 'dinner'])
```

So the constructor is different; it takes a list of channel names. Let's also pass the channel name to `register`, since each subscriber will register for one or more:

```
# Three subscribers, of the original type.
bob = Subscriber('Bob')
alice = Subscriber('Alice')
john = Subscriber('John')

# Two args: channel name & subscriber
pub.register("lunch", bob)
pub.register("dinner", alice)
pub.register("lunch", john)
pub.register("dinner", john)
```

Now, on dispatch, the publisher needs to specify the event type. So just like with `register`, we'll prepend a channel argument:

```
pub.dispatch("lunch", "It's lunchtime!")
pub.dispatch("dinner", "Dinner is served")
```

When correctly working, we'd expect this output:

```
Bob got message "It's lunchtime!"
John got message "It's lunchtime!"
Alice got message "Dinner is served"
John got message "Dinner is served"
```

Pop quiz (and if it's practical, pause here to write Python code): how would you implement this new, multi-channel `Publisher`?

There are several approaches, but the simplest I've found relies on creating a separate `subscribers` dictionary for each channel. One approach:

```python
class Publisher:
    def __init__(self, channels):
        # Create an empty subscribers dict
        # for every channel
        self.channels = { channel : dict()
                          for channel in channels }
    def register(self, channel, who, callback=None):
        if callback is None:
            callback = who.update
        subscribers = self.channels[channel]
        subscribers[who] = callback
    def dispatch(self, channel, message):
        subscribers = self.channels[channel]
        for subscriber, callback in subscribers.items():
            callback(message)
```

This `Publisher` has a dict called `self.channels`, which maps channel names (strings) to subscriber dictionaries. `register` and `dispatch` are not too different: they simply have an extra step, in which `subscribers` is looked up in `self.channels`. I use that variable just for readability, and I think it's well worth the extra line of code:

```python
# Works the same. But a bit less readable.
    def register(self, channel, who, callback=None):
        if callback is None:
            callback = who.update
        self.channels[channel][who] = callback
```

These are some variations of the general observer pattern, and I'm sure you can imagine more. What I want you to notice are the options available in Python when you leverage function objects, and other Pythonic features.

1.5 Magic Methods

Suppose we want to create a class to work with angles, in degrees. We want this class to help us with some standard bookkeeping:

- An angle will be at least zero, but less than 360.

- If we create an angle outside this range, it automatically wraps around to an equivalent, in-range value.
- In fact, we want the conversion to happen in a range of situations:
  - If we add 270º and 270º, it evaluates to 180º instead of 540º.
  - If we subtract 180º from 90º, it evaluates to 270º instead of -90º.
  - If we multiply an angle by a real number, it wraps the final value into the correct range.
- And while we're at it, we want to enable all the other behaviors we normally want with numbers: comparisons like "less than" and "greater or equal than" or "==" (i.e., equals); division (which doesn't normally require casting into a valid range, if you think about it); and so on.

Let's see how we might approach this, by creating a basic Angle class:

```
class Angle:
    def __init__(self, value):
        self.value = value % 360
```

The modular division in the constructor is kind of neat: if you reason through it with a few positive and negative values, you'll find the math works out correctly whether the angle is overshooting or undershooting. This meets one of our key criteria already: the angle is normalized to be from 0 up to 360. But how do we handle addition? We of course get an error if we try it directly:

```
>>> Angle(30) + Angle(45)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'Angle' and 'Angle'
>>>
```

We can easily define a method called `add` or something, which will let us write code like `angle3 = angle1.add(angle2)`. But it's better if we can reuse the familiar arithmetic operators everyone knows. Python lets us do that, through a collection of object hooks called *magic methods*. It lets you define classes so that their instances can be used with all of Python's standard operators. That includes arithmetic (+ - * / //), equality (==), inequality (!=), comparisons (< > >= <=), bit-shifting operations, and even concepts like exponentiation and absolute value.

Few classes will need all of these, but sometimes it's invaluable to have them available. Let's see how they can improve our `Angle` type.

1.5.1 Simple Math Magic

The pattern for each method is the same. For a given operation - say, addition - there is a special method name that starts and begins with double-underscores. For addition, it's `__add__` - the others also have sensible names. All you have to do is define that method, and instances of your class can be used with that operator. These are the magic methods.

When you discuss magic methods in face-to-face, verbal conversation, you'll find yourself saying things like "underscore underscore add underscore underscore" over and over. That's a lot of syllables, and you'll get tired of it fast. So people in the Python community use a kind of verbal abbreviation, with a word they invented: "dunder". That's not a real word; Python people made it up. When you say "dunder foo", it means "underscore underscore foo underscore underscore". This isn't used in writing, because it's not needed - you can just write `__foo__`. But at Python gatherings, you'll sometimes hear people say it. Use it; it saves you a lot of energy when talking.

Anyway, back to dunder add - I mean, `__add__`. For operations like addition - which take two values, and return a third - you write the method like this:

```
    def __add__(self, other):
        return Angle(self.value + other.value)
```

The first argument needs to be called "self", because this is Python. The second does not have to be called "other", but often is. This lets us use the normal addition operator for arithmetic:

```
>>> total = Angle(30) + Angle(45)
>>> total.value
75
```

There are similar operators for subtraction (`__sub__`), multiplication (`__mul__`), and so on:

`__add__`	a + b
`__sub__`	a - b
`__mul__`	a * b
`__truediv__`	a / b (floating-point division)
`__mod__`	a % b
`__pow__`	a ** b

Essentially, Python translates `a + b` to `a.__add__(b)`; `a % b` to `a.__mod__(b)`; and so on. You can also hook into bit-operation operators:

`__lshift__`	a << b
`__rshift__`	a >> b
`__and__`	a & b
`__xor__`	a ˆ b
`__or__`	a | b

So `a & b` translates to `a.__and__(b)`, for example. Since `__and__` corresponds to the bitwise-and operator (for expressions like `"foo & bar"`), you might wonder what the magic method is for *logical*-and (`"foo and bar"`), or logical-or (`"foo or bar"`). Sadly, there is none. For this reason, sometimes libraries will hijack the & and `|` operators to mean logical and/or instead of bitwise and/or, if the author feels the logical version is more important.

The default representation of an `Angle` object isn't very useful:

```
>>> Angle(30)
<__main__.Angle object at 0x106df9198>
```

It tells us the type, and the hex object ID, but we'd rather it tell us something about the value of the angle. There are two magic methods that can help. The first is `__str__`, which is used when printing a result:

```
    def __str__(self):
        return '{} degrees'.format(self.value)
```

The `print()` function uses this, as well as `str()`, and the string formatting operations:

```
>>> print(Angle(30))
30 degrees
>>> print('{}'.format(Angle(30) + Angle(45)))
75 degrees
>>> str(Angle(135))
'135 degrees'
```

Sometimes, you want a string representation that is more precise, which might be at odds with a human-friendly representation. Imagine you have several subclasses (e.g., `PitchAngle` and `YawAngle` in some kind of aircraft-related library), and want to easily log the exact type and arguments needed to recreate the object. Python provides a second magic method for this purpose, called `__repr__`:

```
    def __repr__(self):
        return 'Angle({})'.format(self.value)
```

You access this by calling either the `repr()` built-in function (think of it as working like `str()`, but invokes `__repr__` instead of `__str__`), or by passing the `!r` conversion to the formatting string:

```
>>> repr(Angle(30))
'Angle(30)'
>>> print('{!r}'.format(Angle(30) + Angle(45)))
Angle(75)
```

The official guideline is that the output of `__repr__` is something that can be passed to `eval()` to recreate the object exactly. It's not enforced by the language, and isn't always practical, or even possible. But when it is, doing so is useful for logging and debugging.

We also want to be able to compare two `Angle` objects. The most basic comparison is equality, provided by `__eq__`. It should return True or False:

```
    def __eq__(self, other):
        return self.value == other.value
```

If defined, this method is used by the == operator:

```
>>> Angle(3) == Angle(3)
True
>>> Angle(7) == Angle(1)
False
```

By default, the == operator for objects is based off the object ID. That's rarely useful:

```
>>> class BadAngle:
...     def __init__(self, value):
...         self.value = value
...
>>> BadAngle(3) == BadAngle(3)
False
```

The `!=` operator has its own magic method, `__ne__`. It works the same way:

```
    def __ne__(self, other):
        return self.value != other.value
```

What happens if you don't implement `__ne__`? In Python 3, if you define `__eq__` but not `__ne__`, then the `!=` operator will use `__eq__`, negating the output. Especially for simple classes like `Angle`, this default behavior is logically valid. So in this case, we don't need to define a `__ne__` method at all. For more complex types, the concepts of equality and inequality may have more subtle nuances, and you will need to implement both.

What's left are the fuzzier comparison operations; less than, greater than, and so on. Python's documentation calls these "rich comparison" methods, so you can feel wealthy when using them:

- `__lt__` for "less than" (<)
- `__le__` for "less than or equal" (<=)
- `__gt__` for "greater than" (>)
- `__ge__` for "greater than or equal" (>=)

For example:

```
    def __gt__(self, other):
        return self.value > other.value
```

Now the greater-than operator works correctly:

```
>>> Angle(100) > Angle(50)
True
```

Similar with `__ge__`, `__lt__`, etc. If you don't define these, you get an error, at least in Python 3:

```
>>> BadAngle(8) > BadAngle(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: BadAngle() > BadAngle()
```

`__gt__` and `__lt__` are reflections of each other. What that means is that, in many cases, you only have to define one of them. Suppose you implement `__gt__` but not `__lt__`, then do this:

```
>>> a1 = Angle(3)
>>> a2 = Angle(7)
>>> a1 < a2
True
```

This works thanks to some just-in-time introspection the Python runtime does. The `a1 < a2` is, semantically, equivalent to `a1.__lt__(a2)`. If `Angle.__lt__` is indeed defined, that semantic equivalent is executed, and the expression evaluates to its return value.

For normal scalar numbers, `n < m` is true if and only if `m > n`. For this reason, if `__lt__` does not exist, but `__gt__` does, then Python will rewrite the angle comparison: `a1.__lt__(a2)` becomes `a2.__gt__(a1)`. This is then evaluated, and the expression `a1 < a2` is set to its return value.

Note there are situations where this is actually *not* what you want. Imagine a `Point` type, for example, with two coordinates, x and y. You want `point1 < point2` to be `True` if and only if `point1.x < point2.x`, AND `point1.y < point2.y`. Similarly for `point1 > point2`. There are many points for which both `point1 < point2` and `point1 > point2` should both evaluate to `False`.

For types like this, you will want to implement both `__gt__` and `__lt__` (and `__ge__` and `__le__`.) You might also need to raise `NotImplemented` in the method. This built-in exception signals to the Python runtime that the operation is not supported, at least for these arguments.

1.5.1.1 Shortcut: functools.total_ordering

The `functools` module in the standard library defines a class decorator named `total_ordering`. In practice, for any class which needs to implement all the rich comparison operations, using this labor-saving decorator should be your first choice.

In essence: in your class, you define both `__eq__` and **one** of the comparison magic methods: `__lt__`, `__le__`, `__gt__`, or `__ge__`. (You can define more than one, but it's not necessary.) Then you apply the decorator to the *class*:

```
import functools
@functools.total_ordering
class Angle:
    # ...
    def __eq__(self, other):
        return self.value == other.value
    def __gt__(self, other):
        return self.value > other.value
```

When you do this, all missing rich comparison operators are supplied, defined in terms of `__eq__` and the one operator you defined. This can save you a fair amount of typing.

There are a few situations where you won't want to use `total_ordering`. One is if the comparison logic for the type is not well-behaved enough that each operator can be inferred from the other, via straightforward boolean logic. The `Point` class is an example of this, as might some types if what you are implementing boils down to some kind of abstract algebra engine.

The other reasons not to use it are (1) performance, and (2) the more complex stack traces it generates are more trouble than they are worth. Generally, I recommend you assume these are *not* a problem until proven otherwise. It's entirely possible you will never encounter one of the involved stack traces. And the relatively inefficient implementations that `total_ordering` provides are unlikely to be a problem unless deep inside some nested loop. Starting with `total_ordering` takes little effort, and you can always simply remove it and hand-code the other magic methods if you need to.

1.5.1.2 Python 2 != Python 3

Some magic methods operate a bit differently in Python 2. For example,, if `__eq__` is defined but `__ne__` is not, then `!=` does *not* use `__eq__`. Instead, it relies on the default comparison based on object ID:

```
# Python 2.
>>> class BadAngle(object):
...     def __init__(self, value):
...         self.value = value
...     def __eq__(self, other):
...         return self.value == other.value
...
>>> BadAngle(3) != BadAngle(3)
True
```

You will probably never actually want this behavior (which is why it was changed in Python 3). So for Python 2, if you do define `__eq__`, be sure to define `__ne__` also:

```
# A good default __ne__ for Python 2.
# This is basically what Python 3 does automatically.
    def __ne__(self, other):
        return not self.__eq__(other)
```

In Python 3, if you don't define `__lt__`, and then try to compare two objects with the < operator, you get a `TypeError`. And likewise for `__gt__` and the others. That's a *very* good thing. In Python 2, you instead get a default ordering based off the object ID. This can lead to truly infuriating bugs:

```
# Python 2.
>>> class BadAngle(object):
...     def __init__(self, value):
...         self.value = value
...
>>>
>>> BadAngle(6) < BadAngle(5)
True
>>> BadAngle(6) < BadAngle(5)
False
```

What the heck just happened? When parsing and running the first `BadAngle(6) < BadAngle(5)` line, the Python runtime created two `BadAngle` instances. It turns out the left-hand object was created with an ID whose value happens to be less than that of the right-hand object. So the expression evaluates as `True`. In the second line, the opposite happened: the right-hand object won the race, so to speak, so the expression evaluates as `False`. Watch out for this race condition if you employ magic methods in Python 2.

1.6 Rebelliously Misusing Magic Methods

Magic methods are interesting enough, and quite handy when you need them. A realistic currency type is a good example. But depending on the kind of applications you work on, it's not all that often you will need to define a class whose instances can be added, subtracted, or compared.

Things get much more interesting, though, when you don't follow the rules.

Here's a fascinating fact: methods like `__add__` are *supposed* to do addition. But it turns out Python doesn't require it. And methods like `__gt__` are *supposed* to return `True` or `False`. But if you write a `__gt__` which returns something that isn't a `bool`... Python won't complain at all.

This creates some *amazing* possibilities.

To illustrate, let me tell you about Pandas. As you may know, this is an excellent data-processing library. It's become extremely popular among data scientists who use Python (like some of you reading this). Pandas has a convenient data type called a `DataFrame`. It represents a two-dimensional collection of data, organized into rows, with labeled columns:

```
import pandas
df = pandas.DataFrame({
        'A': [-137, 22, -3, 4, 5],
        'B': [10, 11, 121, 13, 14],
        'C': [3, 6, 91, 12, 15],
    })
```

There are several ways to create a `DataFrame`; here I've chosen to use a dictionary. The keys are column names; the values are lists, which become that column's data. So you visually rotate each list 90 degrees:

```
>>> print(df)
     A    B   C
0 -137   10   3
1   22   11   6
2   -3  121  91
3    4   13  12
4    5   14  15
```

The rows are numbered for you, and the columns nicely labeled in a header. The `A` column, for example, has different positive and negative numbers.

Now, one of the many useful things you can do with a `DataFrame` is filter out rows meeting certain criteria. This doesn't change the original dataframe; instead, it creates a *new* dataframe, containing just the rows you want. For example, you can say "give me the rows of `df` in which the `A` column has a positive value":

```
>>> positive_a = df[df.A > 0]
>>> print(positive_a)
    A   B   C
1  22  11   6
3   4  13  12
4   5  14  15
```

All you have to do is pass in `"df > 0"` in the square brackets.

But there's something weird going on here. Take a look at the line in which `positive_a` is defined. Do you notice anything unusual there? Anything strange?

Here's what is odd: the expression `"df > 0"` ought to evaluate to either `True`, or `False`. Right? It's supposed to be a boolean value... with exactly *one bit* of information. But the source dataframe, `df`, has many rows. Realistic dataframes can easily have tens of thousands, even *millions* of rows of data. There's no way a boolean literal can express which of those rows to keep, and which to discard. **How does this even work?**

Well... turns out, it's not boolean at all:

```
>>> comparison = (df.A > 0)
>>> type(comparison)
<class 'pandas.core.series.Series'>
>>> print(comparison)
0    False
1     True
2    False
3     True
4     True
Name: A, dtype: bool
```

Yes, you can do that, thanks to Python's dynamic type system. Python translates `"df.A > 0"` into `"df.A.__gt__(0)"`. And that `__gt__` method doesn't have to return a bool. In fact, in Pandas, it returns a `Series` object (which is like a vector of data), containing `True` or `False` for each row. And when that's passed into `df[]` - the square brackets being handled by the `__getitem__` method - that `Series` object is used to filter rows.

To see what this looks like, let's re-invent part of the interface of Pandas. I'll create a library called `fakepandas`, which instead of `DataFrame` has a type called `Dataset`:

```
class Dataset:
    def __init__(self, data):
        self.data = data
        self.labels = sorted(data.keys())
    def __getattr__(self, label: str):
        "Makes references like df.A work."
        return Column(label)
    # Plus some other methods.
```

If I have a `Dataset` object named ds, with a column named `A`, the `__getattr__` method makes references like `ds.A` return a `Column` object:

```
import operator
class Column:
    def __init__(self, name):
        self.name = name
    def __gt__(self, value):
         return Comparison(self.name, value, operator.gt)
```

This Column class has a __gt__ method, which makes expressions like "ds.A > 0" return an instance of a class called Comparison. It represents a lazy calculation, for when the actual filtering happens later. Notice its constructor arguments: a column name, a threshold value, and a callable to implement the comparison. (The operator module has a function called gt, taking two arguments, expressing a greater-than comparison).

You can even support complex filtering criteria like ds[ds.C + 2 < ds.B]. It's all possible by leveraging magic methods in these unorthodox ways. If you care about the details, there's an article delving into that.[4] My goal here isn't to tell you how to re-invent the Pandas interface, so much as to get you to realize what's possible.

Have you ever implemented a compiler? If so, you know the parsing phase is a significant development challenge. Using Python magic methods in this manner does much of the hard work of lexing and parsing for you. And the best part is how natural and intuitive the result can be for end users. You are essentially implementing a mini-language on top of regular Python syntax, but consistently enough that people quickly become fluent and productive with its rules. And they often won't even think to ask why the rules seem to be bent; they won't notice "df.A > 0" isn't acting like a boolean. That's a clear sign of success. It means you designed your library so well, other developers become effortlessly productive.

[4]See https://powerfulpython.com/blog/rebellious-magic-methods-python-syntax/ . The article explains these ideas in richer detail, and includes the full code of fakepandas and its unit test suite.

Index