bfa Documentation

Release

Mark Williams

Feb 25, 2018

Contents:

`bfa` implements the builder pattern for attrs -decorated classes.

CHAPTER 1

Why?

Python programmers rarely use the builder pattern because class initializers can express multiple creation strategies with keyword and default arguments:

```
>>> class ComplexCreation(object):
...     def __init__(self, value1=None, value2=None, value3=None):
...         self.value1 = value1
...         self.value2 = value2
...         self.value3 = value3
...     def __repr__(self):
...         fmt = "ComplexCreation(value1={}, value2={}, value3={})"
...         return fmt.format(self.value1, self.value2, self.value3)
>>> ComplexCreation(value3=3)
ComplexCreation(value1=None, value2=None, value3=3)
>>> ComplexCreation(value1=1, value3=3)
ComplexCreation(value1=1, value2=None, value3=3)
```

Languages without these features must use something like the builder pattern to achieve equally flexiblity.

Some classes, however, don't work well with Python's function signatures and calling convention.

1.1 Many Complex Arguments

It's best to keep the number of arguments to a class initializer small and simple. But some classes have to model irreducibly complicated things. X.509 certificates, for example, contain many differently-typed fields and support arbitrary extensions. The cryptography library encapsulates the combinatoric complexity of certificate creation within its x509.CertificateBuilder. Each setter method, like not_valid_before, validates and converts its argument in isolation, resulting in an interface that's both clearer for users and easier to test.

1.2 Immutable Classes and Incomplete Data

Immutability eliminates bugs by preventing values from changing unexpectedly.

Asynchronous network programming eliminates bugs by making concurrency explicit instead of implicit.

Unfortunately, the two can be hard to mix. A network protocol message might be best represented by a frozen attrs class because so that downstream code can't accidently change any of its values. At the same time, the data necessary to create that class may not arrive at the same time. One way to deal with this is to create a temporary `dict` to store initializer arguments as they become available:

```
import attr

@attr.s(frozen=True)
class Message(object):
    key = attr.ib()
    value = attr.ib()
    ttl = attr.ib()
    owner = attr.ib()


class Protocol(object):
    def __init__(self, received):
        self.received = received

    def connectionMade(self):
        self._arguments = {}
        self._keys = []

    def dataReceived(self, data):
        type, value = parse(data)
        if type == Types.STOP:
            message = Message(**self._arguments)
            self.received.callback(message)
            self.transport.loseConnection()
        elif type == Types.KEY:
            self._keys.append(value)
        elif type == Types.VALUE:
            key = self._keys.pop(0)
            self._arguments[key] = value
        elif type == Types.TTL:
            ...
```

While the downstream code waits on the `received` Deferred benefits from `Message`'s mutability, the parsing code in `Protocol` does not, even though it's liable to be full of details that hide bugs.

CHAPTER 2

How?

`bfa` works with attrs to make the builder pattern as painless as it is powerful. Imagine a network protocol `Message` with even more fields, so that our program suffers from both issues that make immutability hard:

```
import attr

@attr.s(frozen=True)
class Message(object):
    key = attr.ib()
    value = attr.ib()
    ttl = attr.ib()
    owner = attr.ib()
    address = attr.ib()
    kitchen = attr.ib()
    sink = attr.ib()
    ...
```

The protocol itself can use `bfa.builder` to incrementally construct a `Message`:

```
import bfa

class Protocol(object):
    def __init__(self, received):
        self.received = received

    def connectionMade(self):
        self._builder = bfa.builder(for_class=Message)
        self._keys = []

    def dataReceived(self, data):
        type, value = parse(data)
        if type == Types.STOP:
            message = self._builder.builder()
            self.received.callback(message)
            self.transport.loseConnection()
        elif type == Types.KEY:
```

```
            self._keys.append(value)
        elif type == Types.VALUE:
            key = self._keys.pop(0)
            self._builder.key(key)
        elif type == Types.TTL:
            ...
```

See `bfa.builder()` for the details.

CHAPTER 3

Where?

`bfa` is developed on GitHub.

Contributors adhere to the project's Code of Conduct.

3.1 src

3.1.1 bfa package

Module contents

CHAPTER 4

Indices

- genindex
- modindex
- search