Journeyman Python Documentation

Release 3.6

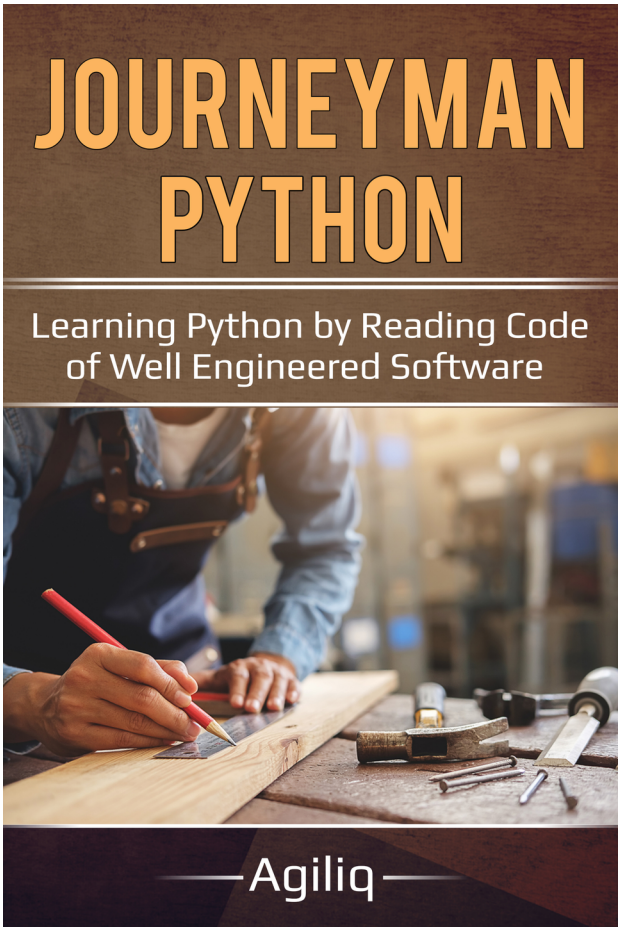
Agiliq and COntributors

Jul 22, 2018

Chapters:

CHAPTER 1

Learning Python by reading code of well engineered software



The best way to learn any programing language and technique is to read other people's code. What better way to learn that read through the code of some of the most well engineered open source Python libraries. In this book, we will read through selected code from Django, Flask and Pandas.

We will learn topics such as decorators, context managers, generators, iterators, itertool, and common design patterns used by Django, Flask and Pandas.

1.1 How Pandas uses first class functions

1.2 Iterators, slicing and generators in SQLAlchemy

1.3 How Django uses decorators to simplify apis

1.4 Understanding python magic methods by reading Django queryset source code.

1.4.1 What are magic methods?

Django querysets are amazing. We use them everyday, but rarely think about the wonderful API they give us. Just some of the amazing properties which queysets have

- You can get a slice `queryset[i:j]` out of them, only the needed objects are pulled from DB.
- You can lookup a specifc object `queryset[i]`, only the required object is pulled from DB.
- You can iterate over them, `for user in users_queryset`, as if they were a list.
- You can `AND` or `OR` them and they apply the criteria at the SQL level.
- You can use them like a boolean, `if users_queryset: users_queryset.update(first_name="Batman")`
- You can pickle and unpickle them, even when the individual istances may not be.
- You can get a useful representation of the queryset in python cli, or ipython. Even if the queryset consists of 1000s of records, only first 20 records will be printed and shown.

Querysets get all of these properties by implemnting the Python magic methods, aka the dunder methods. So why do you need these magic, dunder methods? **Because they make the api much cleaned to use.**

It is more intutive to say, `if users_queryset: users_queryset.do_something()` than `if users_queryset.as_boolean: users_queryset.do_something()`. It is more intutive to say `queryset_1 & queryset_2` rather than `queryse_1.do_and(queryset_2)`

Magic methods are metods implemented by classes which have a special meaning to the Python interpretor. They always start with a ___ and are sometimes called **dunder** method. (Dunder == double underscore).

Query and related classes implement the following methods to get the properies we listed above.

- `__getitem__`: For `queryset[i:j]` and `queryset[i]`
- `__iter__` for `for user in users_queryset`
- `__and__` and `__or__` for `queryset_1 & queryset_2` and `queryset_1 | queryset_2`
- `__bool__` to use them like a boolean
- `__getstate__` and `__setstate__` to pickle and unpickle them
- `__repr__` to get a useful representation and to limit the DB hit

We will look at how Django 2.0 does it.

1.4.2 Implementing `__getitem__`

The code looks like this:

```
def __getitem__(self, k):
    """Retrieve an item or slice from the set of results."""
    if not isinstance(k, (int, slice)):
        raise TypeError
    assert ((not isinstance(k, slice) and (k >= 0)) or
            (isinstance(k, slice) and (k.start is None or k.start >= 0) and
             (k.stop is None or k.stop >= 0))), \
        "Negative indexing is not supported."

    if self._result_cache is not None:
        return self._result_cache[k]

    if isinstance(k, slice):
        qs = self._chain()
        if k.start is not None:
            start = int(k.start)
        else:
            start = None
        if k.stop is not None:
            stop = int(k.stop)
        else:
            stop = None
        qs.query.set_limits(start, stop)
        return list(qs)[::k.step] if k.step else qs
```

There is a lot going on here, but each `if` block is straightforward.

- In the first of block, we ensure slice has reaonable value.
- In second block, if `_result_cache` is filled, aka the queryset has been evaluated, we return the slice from the cache and skip hitting the db again.
- If the `_result_cache` is not filled, we `qs.query.set_limits(start, stop)` which sets the limit and offset in sql.

1.4.3 Implementing `__iter__`

```
def __iter__(self):
    # ...
    self._fetch_all()
    return iter(self._result_cache)
```

Pretty strightforward, we populate the data then use builtin `iter` to return an iterator.

It is also instructive to look at `FlatValuesListIterable.__iter__` which uses `yield` to implment `__iter__`.

```
class FlatValuesListIterable(BaseIterable):
    """
    Iterable returned by QuerySet.values_list(flat=True) that yields single
    values.
    """
```

(continues on next page)

(continued from previous page)

```
    def __iter__(self):
        queryset = self.queryset
        compiler = queryset.query.get_compiler(queryset.db)
        for row in compiler.results_iter(chunked_fetch=self.chunked_fetch, chunk_
↪size=self.chunk_size):
            yield row[0]
```

1.4.4 Implementing `__and__` and `__or__`

The code looks like this:

```
def __and__(self, other):
    self._merge_sanity_check(other)
    if isinstance(other, EmptyQuerySet):
        return other
    if isinstance(self, EmptyQuerySet):
        return self
    combined = self._chain()
    combined._merge_known_related_objects(other)
    combined.query.combine(other.query, sql.AND)
    return combined
```

We d some sanity checks on the querysets, return early if one of the querysets is empty then apply SQL or using `combined.query.combine(other.query, sql.AND)`. The `__or__` is essentially same except the SQL is changed using `combined.query.combine(other.query, sql.OR)`

1.4.5 Implementing `__bool__`

The code looks like this:

```
def __bool__(self):
    self._fetch_all()
    return bool(self._result_cache)
```

Pretty straightforward, `_fetch_all()` ensures that the queryset is evaluated, and `_result_cache` is filled. We then return the boolean equivalent of `_result_cache`, which means if there are any records, you will get a `True`.

1.4.6 Implementing `__getstate__` and `__setstate__`

`__getstate__` and `__setstate__` look like this:

```
def __getstate__(self):
    # Force the cache to be fully populated.
    self._fetch_all()
    return {**self.__dict__, DJANGO_VERSION_PICKLE_KEY: get_version()}

def __setstate__(self, state):
    msg = None
    pickled_version = state.get(DJANGO_VERSION_PICKLE_KEY)
    if pickled_version:
        current_version = get_version()
        if current_version != pickled_version:
```

(continues on next page)

(continued from previous page)

```
            msg = (
                "Pickled queryset instance's Django version %s does not "
                "match the current version %s." % (pickled_version, current_version)
            )
    else:
        msg = "Pickled queryset instance's Django version is not specified."

    if msg:
        warnings.warn(msg, RuntimeWarning, stacklevel=2)

    self.__dict__.update(state)
```

While pickling, we ensure data is populated, then use `self.__dict__` to get queryset representation, and return it along with Django version. While unpickling, `__setstate__` ensures that a warning is raised when pickled querysets are used across Django versions.

On a related note, `{**self.__dict__, DJANGO_VERSION_PICKLE_KEY: get_version()}`, shows why you should move to Python 3. This syntax for merging dictionaries doesn't work in Python2.

1.4.7 Implementing `__repr__`

The code for `__repr__`, look like this

```
def __repr__(self):
    data = list(self[:REPR_OUTPUT_SIZE + 1])
    if len(data) > REPR_OUTPUT_SIZE:
        data[-1] = "...(remaining elements truncated)..."
    return '<%s %r>' % (self.__class__.__name__, data)
```

This is straightforward, but has a few nice tricks worth looking at.

`self[:REPR_OUTPUT_SIZE + 1]` does slicing, which because we implemented `__getitem__`, does `... limit ... offset ...` query.

`REPR_OUTPUT_SIZE` ensures that we don't pull in the wholeyset to display data, but pulls up `REPR_OUTPUT_SIZE + 1` records. On next line `len(data) > REPR_OUTPUT_SIZE` allows us the check if there were more records without hitting the DB.

1.4.8 Final thoughts

Magic, dunder methods provide a clean straightforward way to provide a clean api to your classes. Unlike their name, they don't have any hidden magic and should be used where it makes sense.

1.5 Understanding Python context managers by reading Django source code

Django comes with a bunch of useful context managers. We will read their source code to find what context managers can do and how to implement them including some best parctices.

The three I use most are

- `transactions.atomic` - To get a atomic transaction block

- `TestCase.settings` - To change settings during a test run
- `connection.cursor` - TO get a raw cursor

`connection.cursor` Is generally implemented in the actual DB backends such a psycopg2, so we will focus on `transactions.atomic`, `TestCase.settings` and a few other contextmanagers.

1.5.1 What is a context manager?

Context managers are a code patterns for

- Step 1: Do something
- Step 2: Do something else
- Step 3: Final step, **this step must be guaranteed to run**.

For example when you say

```
with transaction.atomic():
    # This code executes inside a transaction.
    do_more_stuff()
```

What you really want is:

- create a savepoint
- `do_more_stuff()`
- Commit or rollback the savepoint

Similarly, when you say (Inside a `django.test.TestCase`)

```
with self.settings(LOGIN_URL='/other/login/'):
    response = self.client.get('/sekrit/')
```

What you want is

- Change `settings` to LOGIN_URL='/other/login/'
- `response = self.client.get('/sekrit/')`, assert something with on response *with the changed setting*.
- Change `settings` back to what existed at start.

A context manager povides a clean api to enforce this three step workflow.

1.5.2 Some non-Django context managers

The most common context manager is

```
with open('alice-in-wonderland.txt', 'rw') as infile:
    line = infile.readlines()
    do_something_more()
```

If you did not have `open` contextmanager, you would need to do the below everytime, because you need to ensure `do_something_more()` is called.

```
try:
    infile = open('alice-in-wonderland.txt', 'r')
    line = infile.readlines()
    do_something_more()
finally:
    infile.close()
```

Another common use is

```
a_lock = threading.Lock()

with a_lock:
    do_something_more()
```

And without a context manager, this would have been.

```
a_lock.acquire()
try:
    do_something_more()
finally:
    a_lock.release()
```

So at a high level, **context managers are syntactic sugar for "try: ... finally ..." block.** This is important, so I will repeat **context managers are syntactic sugar for "try: ... finally ..." block**

1.5.3 Implementing context managers

Context managers can be implemented as a class with two required methods and one optional `__init__`

- `__enter__`: what to do when the context starts
- `__exit__`: what to do when the context ends
- `__init__`: if your context manager requires arguments

Alternatively, you can use `contextlib.contextmanager` with yield statements to get a context manager. We will see an example in the next section.

1.5.4 A simple Django context manager

In `django/tests/backends/mysql/tests.py`, Django implements a very simple context manager.

```
@contextmanager
def get_connection():
    new_connection = connection.copy()
    yield new_connection
    new_connection.close()
```

And then uses it like this:

```
def test_setting_isolation_level(self):
    with get_connection() as new_connection:
        new_connection.settings_dict['OPTIONS']['isolation_level'] = self.other_
↪isolation_level
        self.assertEqual(
            self.get_isolation_level(new_connection),
```

(continues on next page)

(continued from previous page)

```
            self.isolation_values[self.other_isolation_level]
        )
```

There is some code here which doesn't immediately concern us, let us just focus on `with get_connection() as new_connection:`

Using `@contextmanager`, here is what happened:

- The part before yield `new_connection = connection.copy()` handles the context setup.
- The `yield new_connection` part allows using `new_connection` as `as new_connection`.
- The part after yield `new_connection.close()` handle context teardown.

Lets look at the `TestCase.settings` next, which uses the `__enter__` - `__exit__` protocol.

1.5.5 Implementing Testcase.settings

`Testcase.settings` is implemented as

```
def settings(self, **kwargs):
    """
    A context manager that temporarily sets a setting and reverts to the
    original value when exiting the context.
    """
    return override_settings(**kwargs)
```

There is a bit of class hierarchy to jup through which takes us from

`Testcase.settings` → `override_settings` → `TestContextDecorator`

Skipping the part we don't care about, we get

```
class TestContextDecorator:
    # ...
    def enable(self):
        raise NotImplementedError

    def disable(self):
        raise NotImplementedError

    def __enter__(self):
        return self.enable()

    def __exit__(self, exc_type, exc_value, traceback):
        self.disable()
```

And then `override_settings` implements `.enable` and `.disable`

```
class override_settings(TestContextDecorator):
    # ...
    def enable(self):
        # Keep this code at the beginning to leave the settings unchanged
        # in case it raises an exception because INSTALLED_APPS is invalid.
        if 'INSTALLED_APPS' in self.options:
            try:
                apps.set_installed_apps(self.options['INSTALLED_APPS'])
            except Exception:
```

(continues on next page)

(continued from previous page)

```
                apps.unset_installed_apps()
                raise
        override = UserSettingsHolder(settings._wrapped)
        for key, new_value in self.options.items():
            setattr(override, key, new_value)
        self.wrapped = settings._wrapped
        settings._wrapped = override
        for key, new_value in self.options.items():
            setting_changed.send(sender=settings._wrapped.__class__,
                                 setting=key, value=new_value, enter=True)

    def disable(self):
        if 'INSTALLED_APPS' in self.options:
            apps.unset_installed_apps()
        settings._wrapped = self.wrapped
        del self.wrapped
        for key in self.options:
            new_value = getattr(settings, key, None)
            setting_changed.send(sender=settings._wrapped.__class__,
                                 setting=key, value=new_value, enter=False)
```

There is a lot of boiler plate here which is interesting, but skipping the state management we see

```
class override_settings(TestContextDecorator):
    # ...
    def enable(self):
        # ...
        # This gets called by __enter__
        for key, new_value in self.options.items():
            setattr(override, key, new_value)
        self.wrapped = settings._wrapped
        settings._wrapped = override
        for key, new_value in self.options.items():
            setting_changed.send(sender=settings._wrapped.__class__,
                                 setting=key, value=new_value, enter=True)

    def disable(self):
        # ...
        # This gets called by __exit__
        for key in self.options:
            new_value = getattr(settings, key, None)
            setting_changed.send(sender=settings._wrapped.__class__,
                                 setting=key, value=new_value, enter=False)
```

1.5.6 Implmenting context manager to also be used as a decorator.

When you can say `with transaction.atomic():`, you can get the same effect by using it as a decorator.

```
@transaction.atomic
def do_something():
    # this must run in a transaction
    # ...
```

Implmenting a context manager to also be used as a decorator is a common pattern and Django does the same with atomic. `contextlib.ContextDecorator` makes this straightforward.

```
# class Atomic is implemented later
def atomic(using=None, savepoint=True):
    # Bare decorator: @atomic -- although the first argument is called
    # `using`, it's actually the function being decorated.
    if callable(using):
        return Atomic(DEFAULT_DB_ALIAS, savepoint)(using)
    # Decorator: @atomic(...) or context manager: with atomic(...): ...
    else:
        return Atomic(using, savepoint)

class Atomic(ContextDecorator):
    # There is a lot of complicated corner cases and error handling.
    # See the gory details in django/django/db/transaction.py
    def __init__(self, using, savepoint):
        self.using = using
        self.savepoint = savepoint

    def __enter__(self):
        connection = get_connection(self.using)
        # ...
        # sid = connection.savepoint()
        # connection.savepoint_ids.append(sid)

    def __exit__(self, exc_type, exc_value, traceback):
        # Skip the gory details
        # ...
        sid = connection.savepoint_ids.pop()
        if sid is not None:
            try:
                connection.savepoint_commit(sid)
            except DatabaseError:
                connection.savepoint_rollback(sid)
```

1.5.7 Final thoughts

Context managers provide a simple API for a powerfulo construct. Even though they are merely syntactic sugar, they make for an itutive API and in conjunction with the `contextlib` module are easy to implement.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search