How to Write *Actually* Object Oriented Python

Per Fagrell » mango@spotify.com » perfa (github)

This presentation goes through what object-orientation is, why sticking to one coding paradigm is important and then presents a number of design principles your OO code should adhere to, with examples of how they look in code and how testing is simplified.

Procedural *vs* Object-oriented

- Procedural
  - The classic recipe style programs (scripts etc)
  - Simple to follow; linear
- Object Orientation
  - Write classes, but run objects
  - Classes are like blueprints, interaction descriptions
  - Maps mental model of program domain to code
    - server connection -> Connection(object)
    - config file -> Configuration(object)
    - URLs and identifiers -> URI(object)

Python Gives you great freedoms

- Allows both procedural and object-oriented code
- Allows for everything from scripts to large systems
- Doesn’t stop you from shooting yourself in the foot
  - You can put classes in functions in methods in …
- Consistency to one paradigm/style
  - Improves maintainability
  - Simplifies testing
  - Requires personal discipline
- Consistent paradigm simplifies communication

Dry - Don't repeat yourself

- Don’t repeat expressions or code
  - Refactor into variables and helper methods
- Don’t repeat method/function calls
  - Create loops
  - Make code data-driven
- Remove repetition in test-cases too
- Don’t repeat concepts
  - Create classes to encapsulate recurring concepts

```
value = remap((input + self.old_value) * 2.3)
if value < threshold:
    raise InputRangeError("Input out of range, adjusted input: %f",
                          (input + self.old_value) * 2.3)
log.info("Setting adjusted by %f", (input + self.old_value) * 2.3)
```

```
adjusted = (input + self.old_value) * 2.3
value = remap(adjusted)
if value < threshold:
    raise InputRangeError("Input out of range, adjusted input: %f",
                          adjusted)
log.info("Setting adjusted by %f", adjusted)
```

```
def load(self):
    with open(BASE_SETTINGS, 'r') as settings:
        try:
            load_base_settings(settings)
        except LoadError:
            log.error(“Failed to load %s”, BASE_SETTINGS)
    with open(PLUGIN_SETTINGS, 'r') as settings:
        try:
            load_plugin_settings(settings)
        except LoadError:
             log.error(“Failed to load %s”, PLUGIN_SETTINGS)
    with open(EXTENSION_SETTINGS, 'r') as settings:
        …
```

```
def load(self):
    try_to_load(BASE_SETTINGS, load_base_settings)
    try_to_load(PLUGIN_SETTINGS, load_plugin_settings)
    try_to_load(EXTENSION_SETTINGS, load_extension_settings)
```

```
CONFIG_LOAD_MAP = [(BASE_SETTINGS, load_base_settings),
                   (PLUGIN_SETTINGS, load_plugin_settings),
                   (EXTENSION_SETTINGS, load_extension_settings)]

def load(self):
    for settings_file, loader in CONFIG_LOAD_MAP:
        try_to_load(settings_file, loader)
```

```
def test_should_do_x(self):
    ...
    self.assertEqual(user, testobject.user)
    self.assertEqual(project, testobject.project)
    self.assertEqual(owner, testobject.owner)
def test_should_do_y(self):
    ...
    self.assertEqual(user, testobject.user)
    self.assertEqual(project, testobject.project)
    self.assertEqual(owner, testobject.owner)
```

```
def test_should_do_x(self):
    ...
    self.assertValidTestobject(testobject)
def test_should_do_y(self):
    ...
    self.assertValidTestobject(testobject)
```

Single Responsibility Principle

- Code should have one and *only* one reason to change
- ‘Responsibility’ is a very narrow set of functions
- Avoid adding methods from several domains
  - Business rules
  - Persistence
  - Data input
  - Et c.
- Avoid mixing object orchestration and doing work
  - Break out remaining code to new object
  - Original class strictly does orchestration

```
class Modem(object):
    def call(self, number): ...
    def disconnect(self): ...
    def send_data(self, data): ...
    def recv_data(self): ...
```

```
class ConnectionManager(object):
    def call(self, number): ...
    def disconnect(self): ...
class DataTransciever(object):
    def send_data(self, data):  ...
    def recv_data(self):  ...
```

```
class Person(object):
    def save(self): …
    def report_hours(self, hours): …
```

Opt 1

```
class Person(object):
    def report_hours(self, hours): …
class Persistor(object):
    def save(self, person): ...
```

Opt 2

```
class Person(object, DbMixin):
    def report_hours(self, hours): …
```

```
def process_frame(self):
    frame = self.input_processor.top()
    start_addr = frame.addr
    pow2_size = 1
    while pow2_size < frame.offs:
        pow2_size <<= 1
    ...
```

```
def process_frame(self):
    frame = self.input_processor.top()
    o_map = self.memory_mapper.map(frame)
    self.output_processor.flush(o_map)
```

Open/Closed Principle

- Code should be open to extension but closed to modification
- Extension means giving new features by changing and adding new classes to collaborate with
- Adding new functionality should not require *modifying* the original class
- Avoid direct references to concrete classes
  - e.g. creating new instances in the middle of a method
- Avoid use of *isinstance*
  - Duck-typing (assuming an interface) is OK
  - issubclass also OK

```
def validate_link(self, links):
    for link in links:
        if link.startswith("spotify:album:"):
            uri = Album(link)
        else:
            uri = Track(link)
        self.validate(uri)
```

```
def validate_link(self, links):
    for link in links:
        self.validate(uri_factory(link))
```

Liskov Substitutability Principle

- Anywhere you use a base class, you should be able to use a subclass and not know it
- Alternatives should have
  - same methods
  - same signatures
  - same intrinsic contracts
- Don’t surprise other developers

Interface Segregation Principle

- Don’t force clients to use interfaces they don’t need
- Keep classes and exposed methods minimal
- Don’t use more of other objects than you really need to
  - This avoids entangling them in your code
  - Frees them to change without breaking your code
- State your intent for your usage in the docstring
- Observe other module’s docstrings

Dependency Inversion Principle

- High-level modules shouldn’t rely on low level modules
  - Both should rely on abstractions
- A class should have 1 consistent level of abstraction
  - Business rules
  - Audio control
  - File IO
  - etc
- Split out low level functionality to new object
  - Makes object more flexible (net streaming instead of sound out)
  - Makes testing simpler (inject simple test class)

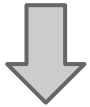
Tell, Don’t Ask

- Let objects you use handle their own data
- *Tell* the object to do the work, don’t *ask* it for data
- Keep responsibility localized
- Lets the object you’re using update implementation
- Object may know more about special cases etc

```
def calculate(self):
    cost = 0
    for line_item in self.bill.items:
        cost += line_item.cost
    ...
```

```
def calculate(self):
    cost = self.bill.total_cost()
    ...
```

```
def calculate(self, pos, vel):
    # Calculate amplitude of velocity
    abs_vel = math.sqrt(sum((vel.x**2,
                             vel.y**2,
                             vel.z**2))
    …
```



```
def calculate(self, position, velocity):
    vel = abs(velocity)
    …
```

Unit-testing

- Adhering to the design principles makes testing easier
  - Less set up
  - Smaller area to test
  - Fewer paths through your code
- Removing duplication in code can remove several times as much testing
- If objects only do work or coordinate objects setup is much simpler
  - less mocking
  - fewer explicit returns and pre-loaded values to set up
- Only concrete classes will need to import anything specific
  - e.g. 3rd party modules (DB orms, requests etc)

Think ‘Objects’

- Stop and ask yourself, “Why was this difficult?”
  - why did you need so much setup?
  - why were there so many mocks?
  - why was writing the test logic tricky?
- Follow up with “Am I missing an object?”
  - Taking lots of arguments → taking 1 new object
  - Code in a loop → looping over objects, calling method
  - Make domain concepts explicit
- Refactoring towards objects makes you program look like it’s written in a Domain Specific Language (DSL)

Plain code

```
if not valid_user(user):
   return -1

c = netpkg.open_connection("uri://server.path",
port=57100, flags=netpkg.KEEPALIVE)
if c is None:
   return -1

files = [str(f) for f in c.request(netpkg.DIRLIST)]
for source in files:
    local_path = "/home/%s/Downloads/%s" \
                    % (user_name, source)
    data = c.request(netpkg.DATA, source)
    with open(local_path, 'w') as local:
        local.write(data)
```

Refactored ‘DSL’

```
authenticate(user)
connection = connect(user, server)

files = RemoteDirectory(connection)
download = Downloader(files)

download.to(user.downloads_dir)
```