Aspect Oriented Programming

REPORT

Aspect Oriented Programming in Python

By :
Mali Imre Gergely

Contents

1 Overview

In this report, we aim to present the aspect oriented programming paradigm from a pythonic point of view. In the following, we present AOP as a general concept, then we present Python, as a programming language, from the perspective of Aspect Oriented Programming. The next sections aim to describe Python's aspect oriented support libraries.

1.1 Aspect Oriented Programming

Aspect Oriented Programming is a horizontal programming paradigm whose main purpose is to add additional behavior to existing components without modifying the code itself, thus avoiding **code tangling**. With AOP we can apply some type of behavior to several classes that by default do not share the same vertical object-oriented inheritance.

AOP aims to increase modularity by implementing **cross-cutting concerns** with the help of **aspects**. Cross-cutting concerns are system-wide concerns that represent functionalities for secondary requirements of a system(like logging, caching etc.). These concerns are implemented in aspects and applied on **join points**. This is referred to as applying **advice**.

1.2 Python and AOP

Python is a general-purpose interpreted programming language that mainly focuses on code readability. Due to the dynamic mechanisms of the language, Python does not need a separate extension for being able to behave in an Aspect Oriented Manner. The only prerequisite is Python 3-s feature that allows one to annotate input parameters and return values of functions. Having this, an AOP library can make use of this to change the behavior of the program at "load time" already, not only at runtime.

There are several AOP libaries for Python, such as Aspyct, Pytilities, Spring Python's AOP module and Aspectlib. In the following, we will focus on the latter one.

2 Python Aspectlib

Python's aspectlib provides two core tools to do AOP: the aspect and the weaver. This library, unlike other AOP supports, targets another stage of the development: the maintenance.

2.1 Generators and Aspects

Generators in Python allow one to declare functions that can be used as iterators. Python has a generator function- and a generator expression support that can significantly simplify code.

An aspect can be generated by decorating a generator with an Aspect. By doing so, the generator will yield advises, that are simple behavior-changing instructions. The Aspect itself is a simple function decorator. When a generator function is decorated with the **@aspectlib.Aspect** decorator, it's behavior is changed according to the advices yielded by the involved generator.

2.2 The weaver

Aspectlib makes AOP possible via a mechanism called **monkey-patching**, that is the equivalent of weaving in other AOP languages. Yet this library does more than just patching: depending on the target object, besides patching it, it can create one or more subclasses and objects.

When instrumenting already written code for debugging or testing less flexible code, it's unfeasible to handle all the patching by hand. Aspectlib can handle all this gross patching mumbo-jumbo itself. Another solution would be to write wrappers. However, this process is repetitive and error-prone. Moreover, wrappers cannot be reused, while function decorators can be.

Aspectlib's weaver patches classes and functions with the given aspect. When it is used with a class, it patches all methods. Actually, these patched methods and functions are referred to as **point-cuts** in AOP parlance.

2.3 Advices

There are several kinds of advices that can be used on a function:

- **Proceed**: calls the wrapped function with it's default arguments
- **Proceed(*args,**kwargs)**: calls the wrapped function with the specified arguments
- **Return**: makes the function return None instead
- **Return (value)**: makes the function return the object specified by "value"
- **raise exception**:makes the wrapper raise an exception

3 Conclusion

Python is a fast-evolving, flexible language that given it's already existing functionalities and mechanisms, does not need a separate support for doing Aspect oriented programming, like other languages do. Yet, Python programmers working on long-running projects, doing mainly code maintenance feel the need of some of the advantages AOP might offer. Aspectlib is a library meant to address this issue.

However, we might emphasize again, that this refers mainly to the phase where code is already written, and it needs additional behavior in a hurry(" *aspectlib.debug.log is aspectlib's crown jewel*"). When a project is developed from scratch, and some Aspect Oriented behavior is desired, there are better tools than Aspectlib.

Bibliography

[1] Spring Python v1.2.1, Aspect Oriented Programming, *Spring Python v1.2.1 Final Documentation*, https://docs.spring.io/spring-python/1.2.x/sphinx/html/aop.html accessed on 2019-05-04

[2] Stack Overflow, *Cross-cutting concerns example*, 2018, available at: https://stackoverflow.com/questions/23700540/cross-cutting-concern-example, accessed on: 2019-04-05

[3] Stack Overflow, *Aspect Oriented Programming in Python*, 2017, available at: https://stackoverflow.com/questions/12356713/aspect-oriented-programming-aop-in-python/12356811, accessed on: 2019-04-05

[4] Python Documentation, *Python 3.7.3 Documentation* available at: https://docs.python.org/3/, accessed on: 2019-04-05

[5] Aspectlib Documentation, *Aspectlib 1.4.2 Documentation*, available at: https://python-aspectlib.readthedocs.io/en/latest/ accessed on 2019-04-05