Dependency injection and inversion of control in Python

Originally dependency injection pattern got popular in languages with static typing like Java. Dependency injection is a principle that helps to achieve an inversion of control. A dependency injection framework can significantly improve the flexibility of a language with static typing. Implementation of a dependency injection framework for a language with static typing is not something that one can do quickly. It will be a quite complex thing to be done well. And will take time.

Python is an interpreted language with dynamic typing. There is an opinion that dependency injection doesn't work for it as well as it does for Java. A lot of the flexibility is already built-in. Also, there is an opinion that a dependency injection framework is something that Python developer rarely needs. Python developers say that dependency injection can be implemented easily using language fundamentals.

This page describes the advantages of applying dependency injection in Python. It contains Python examples that show how to implement dependency injection. It demonstrates the usage of the `Dependency Injector` framework, its container, `Factory`, `Singleton`, and `Configuration` providers. The example shows how to use providers' overriding feature of `Dependency Injector` for testing or re-configuring a project in different environments and explains why it's better than monkey-patching.

What is dependency injection?

Let's see what the dependency injection is.

Dependency injection is a principle that helps to decrease coupling and increase cohesion.

What is coupling and cohesion?

Coupling and cohesion are about how tough the components are tied.

- **High coupling**. If the coupling is high it's like using superglue or welding. No easy way to disassemble.
- **High cohesion**. High cohesion is like using screws. Quite easy to disassemble and re-assemble in a different way. It is an opposite to high coupling.

Cohesion often correlates with coupling. Higher cohesion usually leads to lower coupling and vice versa.

Low coupling brings flexibility. Your code becomes easier to change and test.

How to implement the dependency injection?

Objects do not create each other anymore. They provide a way to inject the dependencies instead.

Before:

```
import os


class ApiClient:

    def __init__(self) -> None:
        self.api_key = os.getenv("API_KEY")  # <-- dependency
        self.timeout = int(os.getenv("TIMEOUT"))  # <-- dependency


class Service:

    def __init__(self) -> None:
        self.api_client = ApiClient()  # <-- dependency


def main() -> None:
    service = Service()  # <-- dependency
    ...


if __name__ == "__main__":
    main()
```

After:

```
import os


class ApiClient:

    def __init__(self, api_key: str, timeout: int) -> None:
        self.api_key = api_key  # <-- dependency is injected
        self.timeout = timeout  # <-- dependency is injected


class Service:

    def __init__(self, api_client: ApiClient) -> None:
        self.api_client = api_client  # <-- dependency is injected


def main(service: Service) -> None:  # <-- dependency is injected
    ...


if __name__ == "__main__":
    main(
        service=Service(
            api_client=ApiClient(
                api_key=os.getenv("API_KEY"),
                timeout=int(os.getenv("TIMEOUT")),
            ),
        ),
    )
```

`ApiClient` is decoupled from knowing where the options come from. You can read a key and a timeout from a configuration file or even get them from a database.

`Service` is decoupled from the `ApiClient`. It does not create it anymore. You can provide a stub or other compatible object.

Function `main()` is decoupled from `Service`. It receives it as an argument.

Flexibility comes with a price.

Now you need to assemble and inject the objects like this:

```
main(
    service=Service(
        api_client=ApiClient(
            api_key=os.getenv("API_KEY"),
            timeout=int(os.getenv("TIMEOUT")),
        ),
    ),
)
```

The assembly code might get duplicated and it'll become harder to change the application structure.

Here comes the `Dependency Injector`.

What does the Dependency Injector do?

With the dependency injection pattern, objects lose the responsibility of assembling the dependencies. The `Dependency Injector` absorbs that responsibility.

`Dependency Injector` helps to assemble and inject the dependencies.

It provides a container and providers that help you with the objects assembly. When you need an object you place a `Provide` marker as a default value of a function argument. When you call this function, framework assembles and injects the dependency.

```
from dependency_injector import containers, providers
from dependency_injector.wiring import Provide, inject


class Container(containers.DeclarativeContainer):

    config = providers.Configuration()

    api_client = providers.Singleton(
        ApiClient,
        api_key=config.api_key,
        timeout=config.timeout,
    )

    service = providers.Factory(
        Service,
        api_client=api_client,
    )


@inject
def main(service: Service = Provide[Container.service]) -> None:
    ...


if __name__ == "__main__":
    container = Container()
    container.config.api_key.from_env("API_KEY", required=True)
    container.config.timeout.from_env("TIMEOUT", as_=int, default=5)
    container.wire(modules=[__name__])

    main()  # <-- dependency is injected automatically

    with container.api_client.override(mock.Mock()):
        main()  # <-- overridden dependency is injected automatically
```

When you call the `main()` function the `Service` dependency is assembled and injected automatically.

When you do testing, you call the `container.api_client.override()` method to replace the real API client with a mock. When you call `main()`, the mock is injected.

You can override any provider with another provider.

It also helps you in a re-configuring project for different environments: replace an API client with a stub on the dev or stage.

Objects assembling is consolidated in a container. Dependency injections are defined explicitly. This makes it easier to understand and change how an application works.

Testing, Monkey-patching and dependency injection

The testability benefit is opposed to monkey-patching.

In Python, you can monkey-patch anything, anytime. The problem with monkey-patching is that it's too fragile. The cause of it is that when you monkey-patch you do something that wasn't intended to be done. You monkey-patch the implementation details. When implementation changes the monkey-patching is broken.

With dependency injection, you patch the interface, not an implementation. This is a way more stable approach.

Also, monkey-patching is way too dirty to be used outside of the testing code for re-configuring the project for the different environments.

Conclusion

Dependency injection provides you with three advantages:

- **Flexibility**. The components are loosely coupled. You can easily extend or change the functionality of a system by combining the components in a different way. You even can do it on the fly.
- **Testability**. Testing is easier because you can easily inject mocks instead of real objects that use API or database, etc.
- **Clearness and maintainability**. Dependency injection helps you reveal the dependencies. Implicit becomes explicit. And "Explicit is better than implicit" (PEP 20 - The Zen of Python). You have all the components and dependencies defined explicitly in a container. This provides an overview and control of the application structure. It is easier to understand and change it.

Is it worth applying dependency injection in Python?

It depends on what you build. The advantages above are not too important if you use Python as a scripting language. The picture is different when you use Python to create an application. The larger the application the more significant the benefits.

Is it worth using a framework for applying dependency injection?

The complexity of the dependency injection pattern implementation in Python is lower than in other languages but it's still in place. It doesn't mean you have to use a framework but using a framework is beneficial because the framework is:

- Already implemented
- Tested on all platforms and versions of Python
- Documented
- Supported
- Other engineers are familiar with it

An advice at last:

- **Give it a try**. Dependency injection is counter-intuitive. Our nature is that when we need something the first thought that comes to our mind is to go and get it. Dependency injection is just like "Wait, I need to state a need instead of getting something right away". It's like a little investment that will pay-off later. The advice is to just give it a try for two weeks. This time will be enough for getting your own impression. If you don't like it you won't lose too much.
- **Common sense first**. Use common sense when applying dependency injection. It is a good principle, but not a silver bullet. If you do it too much you will reveal too many of the implementation details. Experience comes with practice and time.

What's next?

Choose one of the following as a next step:

- Look at the application examples:
    - Application example (single container)
    - Application example (multiple containers)
    - Decoupled packages example (multiple containers)
    - Boto3 example
    - Django example
    - Flask example
    - Flask blueprints example
    - Aiohttp example
    - Sanic example
    - FastAPI example
    - FastAPI + Redis example
    - FastAPI + SQLAlchemy example
- Pass the tutorials:
    - Flask tutorial
    - Aiohttp tutorial
    - Asyncio daemon tutorial
    - CLI application tutorial
- Know more about the `Dependency Injector` Key features
- Know more about the Providers
- Know more about the Wiring
- Go to the Contents

Useful links

A few useful links related to a dependency injection design pattern for further reading:

- https://en.wikipedia.org/wiki/Dependency_injection
- https://martinfowler.com/articles/injection.html
- https://github.com/ets-labs/python-dependency-injector
- https://pypi.org/project/dependency-injector/