Injector Documentation

Release 0.12.1

Alec Thomas

May 18, 2017

Contents

CHAPTER 1

Introduction

Dependency injection as a formal pattern is less useful in Python than in other languages, primarily due to its support for keyword arguments, the ease with which objects can be mocked, and its dynamic nature.

That said, a framework for assisting in this process can remove a lot of boiler-plate from larger applications. That's where Injector can help. It automatically and transitively provides keyword arguments with their values. As an added benefit, Injector encourages nicely compartmentalised code through the use of *modules*.

While being inspired by Guice, it does not slavishly replicate its API. Providing a Pythonic API trumps faithfulness.

Contents:

Injector Change Log

0.12.0

- Fixed binding inference in presence of * and ** arguments (previously Injector would generate extra arguments, now it just ignores them)
- Improved error reporting
- Fixed compatibility with newer typing versions (that includes the one bundled with Python 3.6)

Technically backwards incompatible:

- Forward references as PEP 484 understands them are being resolved now when Python 3-style annotations are used. See https://www.python.org/dev/peps/pep-0484/#forward-references for details.

  Optional parameters are treated as compulsory for the purpose of injection.

0.11.1

- 0.11.0 packages uploaded to PyPI are broken (can't be installed), this is a fix-only release.

0.11.0

- The following way to declare dependencies is introduced and recommended now:

```
class SomeClass:
    @inject
    def __init__(self, other: OtherClass):
        # ...
```

  The following ways are still supported but are deprecated and will be removed in the future:

```
# Python 2-compatible style
class SomeClass
    @inject(other=OtherClass)
    def __init__(self, other):
        # ...

# Python 3 style without @inject-decoration but with use_annotations
class SomeClass:
    def __init__(self, other: OtherClass):
        # ...

injector = Injector(use_annotations=True)
# ...
```

- The following way to declare Module provider methods is introduced and recommended now:

```
class MyModule(Module):
    @provider
    def provide_something(self, dependency: Dependency) -> Something:
        # ...
```

  @provider implies @inject.

  Previously it would look like this:

```
class MyModule(Module):
    @provides(Something)
    @inject
    def provide_something(self, dependency: Dependency):
        # ...
```

  The *provides()* decorator will be removed in the future.

- Added a *noninjectable()* decorator to mark parameters as not injectable (this serves as documentation and a way to avoid some runtime errors)

Backwards incompatible:

- Removed support for decorating classes with *@inject*. Previously:

```
@inject(something=Something)
class Class:
    pass
```

  Now:

```
class Class:
    @inject
```

```
    def __init__(self, something: Something):
        self.something = something
```

- Removed support for injecting partially applied functions, previously:

```
@inject(something=Something)
def some_function(something):
    pass


class Class:
    @inject(function=some_function)
    def __init__(self, function):
        # ...
```

  Now you need to move the function with injectable dependencies to a class.

- Removed support for getting `AssistedBuilder(callable=...)`
- Dropped Python 2.6 support
- Changed the way `AssistedBuilder` and *`ProviderOf`* are used. Previously:

```
builder1 = injector.get(AssistedBuilder(Something))
# or: builder1 = injector.get(AssistedBuilder(interface=Something))
builder2 = injector.get(AssistedBuilder(cls=SomethingElse))
provider = injector.get(ProviderOf(SomeOtherThing))
```

  Now:

```
builder1 = injector.get(AssistedBuilder[Something])
builder2 = injector.get(ClassAssistedBuilder[cls=SomethingElse])
provider = injector.get(ProviderOf[SomeOtherThing])
```

- Removed support for injecting into non-constructor methods

0.10.1

- Fixed a false positive bug in dependency cycle detection (AssistedBuilder can be used to break dependency cycles now)

0.10.0

- `injector.Provider.get()` now requires an *`injector.Injector`* instance as its parameter
- deprecated injecting arguments into modules (be it functions/callables, *`Module`* constructors or *`injector.Module.configure()`* methods)
- removed *extends* decorator
- few classes got useful __repr__ implementations
- fixed injecting ProviderOf and AssistedBuilders when *`injector.Injector`* auto_bind is set to False (previously would result in *UnsatisfiedRequirement* error)
- fixed crash occurring when Python 3-function annotation use is enabled and __init__ method has a return value annotation ("injector.UnknownProvider: couldn't determine provider for None to None"), should also apply to free functions as well

0.9.1

- Bug fix release.

0.9.0

- Child *Injector* can rebind dependancies bound in parent Injector (that changes *Provider* semantics), thanks to Ilya Orlov
- *CallableProvider* callables can be injected into, thanks to Ilya Strukov
- One can request *ProviderOf* (Interface) and get a `BoundProvider` which can be used to get an implementation of Interface when needed

0.8.0

- Binding annotations are removed. Use *Key()* to create unique types instead.

0.7.9

- Fixed regression with injecting unbound key resulting in None instead of raising an exception

0.7.8

- Exception is raised when *Injector* can't install itself into a class instance due to __slots__ presence
- Some of exception messages are now more detailed to make debugging easier when injection fails
- You can inject functions now - *Injector* provides a wrapper that takes care of injecting dependencies into the original function

0.7.7

- Made `AssistedBuilder` behave more explicitly: it can build either innstance of a concrete class (`AssistedBuilder(cls=Class)`) or it will follow Injector bindings (if exist) and construct instance of a class pointed by an interface (`AssistedBuilder(interface=Interface)`). `AssistedBuilder(X)` behaviour remains the same, it's equivalent to `AssistedBuilder(interface=X)`

0.7.6

- Auto-convert README.md to RST for PyPi.

0.7.5

- Added a ChangeLog!
- Added support for using Python3 annotations as binding types.

Terminology

At its heart, Injector is simply a dictionary for mapping types to things that create instances of those types. This could be as simple as:

```
{str: 'an instance of a string'}
```

For those new to dependency-injection and/or Guice, though, some of the terminology used may not be obvious.

Provider

A means of providing an instance of a type. Built-in providers include:

- *ClassProvider* - creates a new instance from a class
- *InstanceProvider* - returns an existing instance directly
- *CallableProvider* - provides an instance by calling a function

In order to create custom provider you need to subclass *Provider* and override its `get()` method.

Scope

By default, providers are executed each time an instance is required. Scopes allow this behaviour to be customised. For example, *SingletonScope* (typically used through the class decorator *singleton*), can be used to always provide the same instance of a class.

Other examples of where scopes might be a threading scope, where instances are provided per-thread, or a request scope, where instances are provided per-HTTP-request.

The default scope is `NoScope`.

See also:

Scopes

Keys

Key may be used to create unique types as necessary:

```
>>> from injector import Key
>>> Name = Key('name')
>>> Description = Key('description')
```

Binding

A binding is the mapping of a unique binding key to a corresponding provider. For example:

```
>>> from injector import InstanceProvider
>>> bindings = {
...     (Name, None): InstanceProvider('Sherlock'),
...     (Description, None): InstanceProvider('A man of astounding insight'),
... }
```

Binder

The *Binder* is simply a convenient wrapper around the dictionary that maps types to providers. It provides methods that make declaring bindings easier.

Module

A *Module* configures bindings. It provides methods that simplify the process of binding a key to a provider. For example the above bindings would be created with:

```
>>> from injector import Module
>>> class MyModule(Module):
...     def configure(self, binder):
...         binder.bind(Name, to='Sherlock')
...         binder.bind(Description, to='A man of astounding insight')
```

For more complex instance construction, methods decorated with *@provider* will be called to resolve binding keys:

```
>>> from injector import provider
>>> class MyModule(Module):
...     def configure(self, binder):
...         binder.bind(Name, to='Sherlock')
...
...     @provider
...     def describe(self) -> Description:
...         return 'A man of astounding insight (at %s)' % time.time()
```

Injection

Injection is the process of providing an instance of a type, to a method that uses that instance. It is achieved with the *inject* decorator. Keyword arguments to inject define which arguments in its decorated method should be injected, and with what.

Here is an example of injection on a module provider method, and on the constructor of a normal class:

```
from injector import inject

class User(object):
    @inject
    def __init__(self, name: Name, description: Description):
        self.name = name
        self.description = description


class UserModule(Module):
    def configure(self, binder):
       binder.bind(User)


class UserAttributeModule(Module):
    def configure(self, binder):
        binder.bind(Name, to='Sherlock')

    @provider
    def describe(self, name: Name) -> Description:
        return '%s is a man of astounding insight' % name
```

Injector

The *Injector* brings everything together. It takes a list of *Module* s, and configures them with a binder, effectively creating a dependency graph:

```
from injector import Injector
injector = Injector([UserModule(), UserAttributeModule()])
```

You can also pass classes instead of instances to *Injector*, it will instantiate them for you:

```
injector = Injector([UserModule, UserAttributeModule])
```

The injector can then be used to acquire instances of a type, either directly:

```
>>> injector.get(Name)
'Sherlock'
>>> injector.get(Description)
'Sherlock is a man of astounding insight'
```

Or transitively:

```
>>> user = injector.get(User)
>>> isinstance(user, User)
True
>>> user.name
'Sherlock'
>>> user.description
'Sherlock is a man of astounding insight'
```

Assisted injection

Sometimes there are classes that have injectable and non-injectable parameters in their constructors. Let's have for example:

```
class Database(object): pass


class User(object):
    def __init__(self, name):
        self.name = name


class UserUpdater(object):
    def __init__(self, db: Database, user):
        pass
```

You may want to have database connection *db* injected into *UserUpdater* constructor, but in the same time provide *user* object by yourself, and assuming that *user* object is a value object and there's many users in your application it doesn't make much sense to inject objects of class *User*.

In this situation there's technique called Assisted injection:

```
from injector import ClassAssistedBuilder
injector = Injector()
builder = injector.get(ClassAssistedBuilder[UserUpdater])
user = User('John')
user_updater = builder.build(user=user)
```

This way we don't get *UserUpdater* directly but rather a builder object. Such builder has *build(**kwargs)* method which takes non-injectable parameters, combines them with injectable dependencies of *UserUpdater* and calls *UserUpdater* initializer using all of them.

AssistedBuilder[T] and *ClassAssistedBuilder[T]* are injectable just as anything else, if you need instance of it you just ask for it like that:

```
class NeedsUserUpdater(object):
    @inject
    def __init__(self, builder: ClassAssistedBuilder[UserUpdater]):
        self.updater_builder = builder

    def method(self):
        updater = self.updater_builder.build(user=None)
```

ClassAssistedBuilder means it'll construct a concrete class and no bindings will be used.

If you want to follow bindings and construct class pointed to by a key you use *AssistedBuilder* and can do it like this:

```
>>> DB = Key('DB')
>>> class DBImplementation(object):
...     def __init__(self, uri):
...         pass
...
>>> def configure(binder):
...     binder.bind(DB, to=DBImplementation)
...
>>> injector = Injector(configure)
>>> builder = injector.get(AssistedBuilder[DB])
>>> isinstance(builder.build(uri='x'), DBImplementation)
True
```

More information on this topic:

- "How to use Google Guice to create objects that require parameters?" on Stack Overflow
- Google Guice assisted injection

Child injectors

Concept similar to Guice's child injectors is supported by *Injector*. This way you can have one injector that inherits bindings from other injector (parent) but these bindings can be overriden in it and it doesn't affect parent injector bindings:

```
>>> def configure_parent(binder):
...     binder.bind(str, to='asd')
...     binder.bind(int, to=42)
...
>>> def configure_child(binder):
...     binder.bind(str, to='qwe')
...
>>> parent = Injector(configure_parent)
```

```
>>> child = parent.create_child_injector(configure_child)
>>> parent.get(str), parent.get(int)
('asd', 42)
>>> child.get(str), child.get(int)
('qwe', 42)
```

Note: Default scopes are bound only to root injector. Binding them manually to child injectors will result in unexpected behaviour. **Note 2**: Once a binding key is present in parent injector scope (like *singleton* scope), provider saved there takes predecence when binding is overridden in child injector in the same scope. This behaviour is subject to change:

```
>>> def configure_parent(binder):
...     binder.bind(str, to='asd', scope=singleton)
...
>>> def configure_child(binder):
...     binder.bind(str, to='qwe', scope=singleton)
...
>>> parent = Injector(configure_parent)
>>> child = parent.create_child_injector(configure_child)
>>> child.get(str) # this behaves as expected
'qwe'
>>> parent.get(str) # wat
'qwe'
```

Testing with Injector

When you use unit test framework such as *unittest2* or *nose* you can also profit from *injector*. However, manually creating injectors and test classes can be quite annoying. There is, however, *with_injector* method decorator which has parameters just as *Injector* construtor and installes configured injector into class instance on the time of method call:

```
import unittest
from injector import Module, with_injector, inject

class UsernameModule(Module):
    def configure(self, binder):
        binder.bind(str, 'Maria')

class TestSomethingClass(unittest.TestCase):
    @with_injector(UsernameModule())
    def setUp(self):
        pass

    @inject(username=str)
    def test_username(self, username):
        self.assertEqual(username, 'Maria')
```

Each method call re-initializes *Injector* - if you want to you can also put *with_injector()* decorator on class constructor.

After such call all *inject()*-decorated methods will work just as you'd expect them to work.

Scopes

Singletons

Singletons are declared by binding them in the SingletonScope. This can be done in three ways:

1. Decorating the class with *@singleton*.
2. Decorating a *@provider* decorated Module method with *@singleton*.
3. Explicitly calling *binder.bind(X, scope=singleton)*.

A (redundant) example showing all three methods:

```
@singleton
class Thing(object): pass
class ThingModule(Module):
    def configure(self, binder):
        binder.bind(Thing, scope=singleton)
    @singleton
    @provider
    def provide_thing(self) -> Thing:
        return Thing()
```

Implementing new Scopes

In the above description of scopes, we glossed over a lot of detail. In particular, how one would go about implementing our own scopes.

Basically, there are two steps. First, subclass *Scope* and implement *Scope.get*:

```
from injector import Scope
class CustomScope(Scope):
    def get(self, key, provider):
        return provider
```

Then create a global instance of `ScopeDecorator` to allow classes to be easily annotated with your scope:

```
from injector import ScopeDecorator
customscope = ScopeDecorator(CustomScope)
```

This can be used like so:

```
@customscope
class MyClass(object):
    pass
```

Scopes are bound in modules with the `Binder.bind_scope()` method:

```
class MyModule(Module):
    def configure(self, binder):
        binder.bind_scope(CustomScope)
```

Scopes can be retrieved from the injector, as with any other instance. They are singletons across the life of the injector:

```
>>> injector = Injector([MyModule()])
>>> injector.get(CustomScope) is injector.get(CustomScope)
True
```

For scopes with a transient lifetime, such as those tied to HTTP requests, the usual solution is to use a thread or greenlet-local cache inside the scope. The scope is "entered" in some low-level code by calling a method on the scope instance that creates this cache. Once the request is complete, the scope is "left" and the cache cleared.

Logging

Injector uses standard `logging` module, the logger name is `injector`.

By default `injector` logger is not configured to print logs anywhere.

To enable `get()` tracing (and some other useful information) you need to set `injector` logger level to `DEBUG`. You can do that by executing:

```
import logging

logging.getLogger('injector').setLevel(logging.DEBUG)
```

or by configuring `logging` module in any other way.

Injector API reference

Note: Unless specified otherwise, instance methods are **not** thread safe.

The following functions are thread safe:

- `Injector.get()`
- injection provided by `inject()` decorator (please note, however, that it doesn't say anything about decorated function thread safety)

Injector - Python dependency injection framework, inspired by Guice

copyright

3. 2012 by Alec Thomas

license BSD

class `injector.`**`Binder`**(*injector*, *auto_bind=True*, *parent=None*)
Bases: `object`

Bind interfaces to implementations.

Note: This class is instantiated internally for you and there's no need to instantiate it on your own.

`bind`(*interface*, *to=None*, *scope=None*)
Bind an interface to an implementation.

Parameters

- **interface** – Interface or *Key()* to bind.
- **to** – Instance or class to bind to, or an explicit *Provider* subclass.
- **scope** – Optional *Scope* in which to bind.

bind_scope(*scope*)
Bind a Scope.

Parameters **scope** – Scope class.

install(*module*)
Install a module into this binder.

In this context the module is one of the following:

•function taking the *Binder* as it's only parameter

```
def configure(binder):
    bind(str, to='s')

binder.install(configure)
```

•instance of *Module* (instance of it's subclass counts)

```
class MyModule(Module):
    def configure(self, binder):
        binder.bind(str, to='s')

binder.install(MyModule())
```

•subclass of *Module* - the subclass needs to be instantiable so if it expects any parameters they need to be injected

```
binder.install(MyModule)
```

multibind(*interface*, *to*, *scope=None*)
Creates or extends a multi-binding.

A multi-binding maps from a key to a sequence, where each element in the sequence is provided separately.

Parameters

- **interface** – *MappingKey()* or *SequenceKey()* to bind to.
- **to** – Instance, class to bind to, or an explicit *Provider* subclass. Must provide a sequence.
- **scope** – Optional Scope in which to bind.

class injector.**BoundKey**
Bases: tuple

A BoundKey provides a key to a type with pre-injected arguments.

```
>>> class A(object):
...   def __init__(self, a, b):
...     self.a = a
...     self.b = b
>>> InjectedA = BoundKey(A, a=InstanceProvider(1), b=InstanceProvider(2))
>>> injector = Injector()
>>> a = injector.get(InjectedA)
```

```
>>> a.a, a.b
(1, 2)
```

exception injector.**CallError**
Bases: *injector.Error*

Call to callable object fails.

class injector.**CallableProvider**(*callable*)
Bases: *injector.Provider*

Provides something using a callable.

The callable is called every time new value is requested from the provider.

```
>>> key = Key('key')
>>> def factory():
...     print('providing')
...     return []
...
>>> def configure(binder):
...     binder.bind(key, to=CallableProvider(factory))
...
>>> injector = Injector(configure)
>>> injector.get(key) is injector.get(key)
providing
providing
False
```

exception injector.**CircularDependency**
Bases: *injector.Error*

Circular dependency detected.

class injector.**ClassProvider**(*cls*)
Bases: *injector.Provider*

Provides instances from a given class, created using an Injector.

exception injector.**Error**
Bases: exceptions.Exception

Base exception.

class injector.**Injector**(*modules=None*, *auto_bind=True*, *parent=None*, *use_annotations=False*)
Bases: object

Parameters

- **modules** – Optional - a configuration module or iterable of configuration modules. Each module will be installed in current *Binder* using *Binder.install()*.

  Consult *Binder.install()* documentation for the details.
- **auto_bind** – Whether to automatically bind missing types.
- **parent** – Parent injector.
- **use_annotations** – Attempt to infer injected arguments using Python3 argument annotations.

If you use Python 3 you can make Injector use constructor parameter annotations to determine class dependencies. The following code:

```
class B(object):
    @inject(a=A):
    def __init__(self, a):
        self.a = a
```

can now be written as:

```
class B(object):
    def __init__(self, a:A):
        self.a = a
```

To enable Python 3 annotation support, instantiate your *Injector* with `use_annotations=True`.

New in version 0.7.5: `use_annotations` parameter

call_with_injection(*callable*, *self_=None*, *args=()*, *kwargs={}*)
Call a callable and provide it's dependencies if needed.

Parameters

- **self** – Instance of a class callable belongs to if it's a method, None otherwise.
- **args** (*tuple of objects*) – Arguments to pass to callable.
- **kwargs** (*dict of string -> object*) – Keyword arguments to pass to callable.

Returns Value returned by callable.

create_object(*cls*, *additional_kwargs=None*)
Create a new instance, satisfying any dependencies on cls.

get(*interface*, *scope=None*)
Get an instance of the given interface.

Note: Although this method is part of *Injector*'s public interface it's meant to be used in limited set of circumstances.

For example, to create some kind of root object (application object) of your application (note that only one *get* call is needed, inside the *Application* class and any of its dependencies *inject()* can and should be used):

```
class Application(object):

    @inject(dep1=Dep1, dep2=dep2)
    def __init__(self, dep1, dep2):
        self.dep1 = dep1
        self.dep2 = dep2

    def run(self):
        self.dep1.something()

injector = Injector(configuration)
application = injector.get(Application)
application.run()
```

Parameters

- **interface** – Interface whose implementation we want.

- **scope** – Class of the Scope in which to resolve.

Returns An implementation of interface.

install_into(*instance*)

Put injector reference in given object.

This method has, in general, two applications:

•Injector internal use (not documented here)

•**Making it possible to inject into methods of an object that wasn't created** using Injector. Usually it's because you either don't control the instantiation process, it'd generate unnecessary boilerplate or it's just easier this way.

For example, in application main script:

```
from injector import Injector

class Main(object):
    def __init__(self):
        def configure(binder):
            binder.bind(str, to='Hello!')

        injector = Injector(configure)
        injector.install_into(self)

    @inject(s=str)
    def run(self, s):
        print(s)

if __name__ == '__main__':
    main = Main()
    main.run()
```

Note: You don't need to use this method if the object is created using *Injector*.

Warning: Using *install_into* to install *Injector* reference into an object created by different *Injector* instance may very likely result in unexpected behaviour of that object immediately or in the future.

class injector.**InstanceProvider**(*instance*)

Bases: *injector.Provider*

Provide a specific instance.

```
>>> my_list = Key('my_list')
>>> def configure(binder):
...     binder.bind(my_list, to=InstanceProvider([]))
...
>>> injector = Injector(configure)
>>> injector.get(my_list) is injector.get(my_list)
True
>>> injector.get(my_list).append('x')
>>> injector.get(my_list)
['x']
```

injector.**Key**(*name*)
Create a new type key.

```
>>> Age = Key('Age')
>>> def configure(binder):
...   binder.bind(Age, to=90)
>>> Injector(configure).get(Age)
90
```

injector.**MappingKey**(*name*)
As for Key, but declares a multibind mapping.

class injector.**Module**
Bases: object

Configures injector and providers.

configure(*binder*)
Override to configure bindings.

class injector.**NoScope**(*injector=None*)
Bases: *injector.Scope*

An unscoped provider.

class injector.**Provider**
Bases: object

Provides class instances.

class injector.**ProviderOf**(*injector*, *interface*)
Bases: typing.Generic

Can be used to get a provider of an interface, for example:

```
>>> def provide_int():
...     print('providing')
...     return 123
>>>
>>> def configure(binder):
...     binder.bind(int, to=provide_int)
>>>
>>> injector = Injector(configure)
>>> provider = injector.get(ProviderOf[int])
>>> value = provider.get()
providing
>>> value
123
```

get()
Get an implementation for the specified interface.

class injector.**Scope**(*injector*)
Bases: object

A Scope looks up the Provider for a binding.

By default (ie. *NoScope*) this simply returns the default *Provider* .

configure()
Configure the scope.

get(*key*, *provider*)
Get a *Provider* for a key.

Parameters

- **key** – The key to return a provider for.
- **provider** – The default Provider associated with the key.

Returns A Provider instance that can provide an instance of key.

injector.**SequenceKey**(*name*)
As for Key, but declares a multibind sequence.

class injector.**SingletonScope**(*injector*)
Bases: *injector.Scope*

A *Scope* that returns a per-Injector instance for a key.

`singleton` can be used as a convenience class decorator.

```
>>> class A(object): pass
>>> injector = Injector()
>>> provider = ClassProvider(A)
>>> singleton = SingletonScope(injector)
>>> a = singleton.get(A, provider)
>>> b = singleton.get(A, provider)
>>> a is b
True
```

class injector.**ThreadLocalScope**(*injector*)
Bases: *injector.Scope*

A *Scope* that returns a per-thread instance for a key.

exception injector.**UnknownProvider**
Bases: *injector.Error*

Tried to bind to a type whose provider couldn't be determined.

exception injector.**UnsatisfiedRequirement**
Bases: *injector.Error*

Requirement could not be satisfied.

injector.**inject**(*function=None*, ***bindings*)
Decorator declaring parameters to be injected.

eg.

```
>>> Sizes = Key('sizes')
>>> Names = Key('names')
>>>
>>> # Recommended, Python 3+ style
>>> class A:
...     @inject
...     def __init__(self, number: int, name: str, sizes: Sizes):
...         print([number, name, sizes])
...
>>> # Or older, Python 2-compatible style
>>> class A(object):
...     @inject(number=int, name=str, sizes=Sizes)
...     def __init__(self, number, name, sizes):
```

```
...         print([number, name, sizes])
...
>>> def configure(binder):
...     binder.bind(A)
...     binder.bind(int, to=123)
...     binder.bind(str, to='Bob')
...     binder.bind(Sizes, to=[1, 2, 3])
```

Use the Injector to get a new instance of A:

```
>>> a = Injector(configure).get(A)
[123, 'Bob', [1, 2, 3]]
```

injector.**is_decorated_with_inject**(*function*)

See if given callable is declared to want some dependencies injected.

Example use:

```
>>> def fun(i: int) -> str:
...     return str(i)
```

```
>>> is_decorated_with_inject(fun)
False
>>>
>>> @inject
... def fun2(i: int) -> str:
...     return str(i)
```

```
>>> is_decorated_with_inject(fun2)
True
```

injector.**noninjectable**(**args*)

Mark some parameters as not injectable.

This serves as documentation for people reading the code and will prevent Injector from ever attempting to provide the parameters.

For example:

```
>>> class Service:
...     pass
...
>>> class SomeClass:
...     @inject
...     @noninjectable('user_id')
...     def __init__(self, service: Service, user_id: int):
...         # ...
...         pass
```

noninjectable() decorations can be stacked on top of each other and the order in which a function is decorated with *inject()* and *noninjectable()* doesn't matter.

injector.**provider**(*function*)

Decorator for *Module* methods, registering a provider of a type.

```
>>> class MyModule(Module):
...     @provider
```

```
...   def provide_name(self) -> str:
...       return 'Bob'
```

@provider-decoration implies @inject so you can omit it and things will work just the same:

```
>>> class MyModule2(Module):
...     def configure(self, binder):
...         binder.bind(int, to=654)
...
...     @provider
...     def provide_str(self, i: int) -> str:
...         return str(i)
...
>>> injector = Injector(MyModule2)
>>> injector.get(str)
'654'
```

Note: This function works only on Python 3

injector.**provides**(*interface*, *scope=None*, *eager=False*)

Decorator for *Module* methods, registering a provider of a type.

Warning: Deprecated, use *provider()* instead.

```
>>> class MyModule(Module):
...   @provides(str)
...   def provide_name(self):
...       return 'Bob'
```

Parameters

- **interface** – Interface to provide.
- **scope** – Optional scope of provided value.

injector.**with_injector**(**injector_args*, ***injector_kwargs*)

Decorator for a method. Installs Injector object which the method belongs to before the decorated method is executed.

Parameters are the same as for *Injector* constructor.

Frequently Asked Questions

If I use `inject()` or scope decorators on my classess will I be able to create instances of them without using Injector?

Yes. Scope decorators don't change the way you can construct your class instances without Injector interaction.

I'm calling this method (/function/class) but I'm getting "TypeError: XXX() takes exactly X arguments (Y given)"

Example code:

```
class X(object):
    @inject
    def __init__(self, s: str):
        self.s = s

def configure(binder):
    binder.bind(s, to='some string')

injector = Injector(configure)
x = X()
```

Result?

```
TypeError: __init__() takes exactly 2 arguments (1 given)
```

Reason? There's *no* global state that `Injector` modifies when it's instantiated and configured. Its whole knowledge about bindings etc. is stored in itself. Moreover `inject()` will *not* make dependencies appear out of thin air when you for example attempt to create an instance of a class manually (without `Injector`'s help) - there's no global state `@inject` decorated methods can access.

In order for `X` to be able to use bindings defined in `@inject` decoration `Injector` needs to be used (directly or indirectly) to create an instance of `X`. This means most of the time you want to just inject `X` where you need it, you can also use `Injector.get()` to obtain an instance of the class (see its documentation for usage notes).

Good and bad practices

Side effects

You should avoid creating side effects in your modules for two reasons:

- Side effects will make it more difficult to test a module if you want to do it
- Modules expose a way to acquire some resource but they don't provide any way to release it. If, for example, your module connects to a remote server while creating a service you have no way of closing that connection unless the service exposes it.

Injecting into constructors vs injecting into other methods

Note: Injector 0.11+ doesn't support injecting into non-constructor methods, this section is kept for historical reasons.

Note: Injector 0.11 deprecates using @inject with keyword arguments to declare bindings, this section remains unchanged for historical reasons.

In general you should prefer injecting into constructors to injecting into other methods because:

- it can expose potential issues earlier (at object construction time rather than at the method call)

- it exposes class' dependencies more openly. Constructor injection:

```
class Service1(object):
    @inject(http_client=HTTP)
    def __init__(self, http_client):
        self.http_client = http_client
        # some other code

    # tens or hundreds lines of code

    def method(self):
        # do something
        pass
```

Regular method injection:

```
class Service2(object):
    def __init__(self):
        # some other code

    # tens or hundreds lines of code

    @inject(http_client=HTTP)
    def method(self, http_client):
        # do something
        pass
```

In first case you know all the dependencies by looking at the class' constructor, in the second you don't know about `HTTP` dependency until you see the method definition.

Slightly different approach is suggested when it comes to Injector modules - in this case injecting into their constructors (or `configure` methods) would make the injection process dependent on the order of passing modules to Injector and therefore quite fragile. See this code sample:

```
A = Key('A')
B = Key('B')

class ModuleA(Module):
    @inject(a=A)
    def configure(self, binder, a):
        pass

class ModuleB(Module):
    @inject(b=B)
    def __init__(self, b):
        pass

class ModuleC(Module):
    def configure(self, binder):
        binder.bind(A, to='a')
        binder.bind(B, to='b')


# error, at the time of ModuleA processing A is unbound
Injector([ModuleA, ModuleC])

# error, at the time of ModuleB processing B is unbound
Injector([ModuleB, ModuleC])
```

```
# no error this time
Injector([ModuleC, ModuleA, ModuleB])
```

Doing too much in modules and/or providers

An implementation detail of Injector: Injector and accompanying classes are protected by a lock to make them thread safe. This has a downside though: in general only one thread can use dependency injection at any given moment.

In best case scenario you "only" slow other threads' dependency injection down. In worst case scenario (performing blocking calls without timeouts) you can **deadlock** whole application.

It is advised to avoid performing any IO, particularly without a timeout set, inside modules code.

As an illustration:

```
from threading import Thread
from time import sleep

from injector import inject, Injector, Key, Module, provider

SubA = Key('SubA')
A = Key('A')
B = Key('B')


class BadModule(Module):
    @provider
    def provide_a(self, suba: SubA) -> A:
        return suba

    @provider
    def provide_suba(self) -> SubA:
        print('Providing SubA...')
        while True:
            print('Sleeping...')
            sleep(1)

        # This never executes
        return 'suba'

    @provider
    def provide_b(self) -> B:
        return 'b'


injector = Injector([BadModule])

thread = Thread(target=lambda: injector.get(A))

# to make sure the thread doesn't keep the application alive
thread.daemon = True
thread.start()

# This will never finish
injector.get(B)
print('Got B')
```

Here's the output of the application:

```
Providing SubA...
Sleeping...
Sleeping...
Sleeping...
(...)
```

Injecting Injector and abusing Injector.get

Sometimes code like this is written:

```
class A(object):
    pass

class B(object):
    pass

class C(object):
    @inject
    def __init__(self, injector: Injector):
        self.a = injector.get(A)
        self.b = injector.get(B)
```

It is advised to use the following pattern instead:

```
class A(object):
    pass

class B(object):
    pass

class C(object):
    @inject
    def __init__(self, a: A, b: B):
        self.a = a
        self.b = b
```

The second form has the benefits of:

- expressing clearly what the dependencies of `C` are
- making testing of the `C` class easier - you can provide the dependencies (whether they are mocks or not) directly, instead of having to mock `Injector` and make the mock handle `Injector.get()` calls
- following the common practice and being easier to understand

Injecting dependencies only to pass them somewhere else

A pattern similar to the one below can emerge:

```
class A(object):
    pass

class B(object):
    def __init__(self, a):
        self.a = a
```

```
class C(object):
    @inject
    def __init__(self, a: A):
        self.b = B(a)
```

Class `C` in this example has the responsibility of gathering dependencies of class `B` and constructing an object of type `B`, there may be a valid reason for it but in general it defeats the purpose of using `Injector` and should be avoided.

The appropriate pattern is:

```
class A(object):
    pass

class B(object):
    @inject
    def __init__(self, a: A):
        self.a = a

class C(object):
    @inject
    def __init__(self, b: B):
        self.b = b
```

Python Module Index

i

Index