dependency$_{i}njection.pyDocumentation$

Release 1.1.0

Gittip, LLC

Jun 01, 2019

Contents

1 Installation 3

2 What is Dependency Injection? 5

3 API Reference 7

Python Module Index 9

Index 11

This Python library defines a helper for building a dependency injection framework.

CHAPTER 1

Installation

dependency_injection is available on GitHub and on PyPI:

```
$ pip install dependency_injection
```

We test against Python 2.6, 2.7, 3.2, and 3.3.

dependency_injection is MIT-licensed.

CHAPTER 2

What is Dependency Injection?

When you define a function you specify its *parameters*, and when you call the function you pass in *arguments* for those parameters. Dependency injection means dynamically passing arguments to a function based on the parameters it defines. So if you define a function:

```
>>> def foo(bar, baz):
...     pass
```

Then you are advertising to a dependency injection framework that your function wants to have the `bar` and `baz` objects passed into it. What `bar` and `baz` resolve to depends on the dependency injection framework. This library provides a helper, *`resolve_dependencies`*, for building your own dependency injection framework. It doesn't provide such a framework itself, because that would take away all the fun.

CHAPTER 3

API Reference

exception dependency_injection.**CantUseThis**

dependency_injection.**get_signature**(*function*)

Given a function object, return a `namedtuple` representing the function signature.

Parameters **function** – a function object or other callable

Returns a `namedtuple` representing the function signature

This function is a helper for *resolve_dependencies*. It returns a `namedtuple` with these items:

0. `parameters` - a `tuple` of all parameters, in the order they were defined
1. `required` - a `tuple` of required parameters, in the order they were defined
2. `optional` - a `dict` of optional parameters mapped to their defaults

For example, if you have this function:

```
>>> def foo(bar, baz=1):
...     pass
...
```

Then *get_signature* will return:

```
>>> get_signature(foo)
Signature(parameters=('bar', 'baz'), required=('bar',), optional={'baz': 1})
```

Here are the kinds of callable objects we support (in this resolution order):

- functions
- methods (both bound and unbound)
- classes (both newstyle and oldstyle) with a user-defined `__new__`
- classes (both newstyle and oldstyle) with a user-defined `__init__`

- object instances with a `__call__` method

So you can do:

```
>>> class Foo(object):
...     def __init__(self, bar, baz=1):
...         pass
...
>>> get_signature(Foo)
Signature(parameters=('self', 'bar', 'baz'), required=('self', 'bar'), optional={
↪'baz': 1})
```

`dependency_injection.`**`resolve_dependencies`**(*function*, *available*)

Given a function object and a mapping of available dependencies, return a `namedtuple` that has arguments to suit the function's parameters.

Parameters

- **`function`** – a function object or other callable
- **`available`** – a `dict` mapping arbitrary names to objects

Returns a `namedtuple` representing the arguments to use in calling the function

The return value of this function is a `namedtuple` with these attributes:

0. `as_args` - a `tuple` of argument values
1. `as_kwargs` - a `dict` of keyword arguments
2. `signature` - a `namedtuple` as returned by *`get_signature`*

The `as_args` and `as_kwargs` arguments are functionally equivalent. You could call the function using either one and you'd get the same result, and which one you use depends on the needs of the dependency injection framework you're writing, and your personal preference.

This is the main function you want to use from this library. The idea is that in your dependency injection framework, you call this function to resolve the dependencies for a function, and then call the function. So here's a function:

```
>>> def foo(bar, baz):
...     return bar + baz
```

And here's the basics of a dependency injection framework:

```
>>> def inject_dependencies(func):
...     my_state = {'bar': 1, 'baz': 2, 'bloo': 'blee'}
...     dependencies = resolve_dependencies(func, my_state)
...     return func(*dependencies.as_args)
...
```

And here's what it looks like to call it:

```
>>> inject_dependencies(foo)
3
```

See *`get_signature`* for details on support for non-function callables.

Python Module Index

d

dependency_injection, **??**

Index

C

`CantUseThis`, 7

D

`dependency_injection` (*module*), 1

G

`get_signature()` (*in module dependency_injection*), 7

R

`resolve_dependencies()` (*in module dependency_injection*), 8