Picobox

Release 2.2.0

Dec 24, 2018

Contents

Picobox is opinionated dependency injection framework designed to be clean, pragmatic and with Python in mind. No complex graphs, no implicit injections, no type bindings, no XML configurations.

CHAPTER 1

Why?

Dependency Injection (DI) design pattern is intended to decouple various parts of an application from each other. So a class can be independent of how the objects it requires are created, and hence the way we create them may be different for production and tests.

One of the most easiest examples is to say that DI is essentially about writing

```
def do_something(my_service):
    return my_service.get_val() + 42

my_service = MyService(foo, bar)
do_something(my_service)
```

instead of

```
def do_something():
    my_service = MyService(foo, bar)
    return my_service.get_val() + 42

do_something()
```

because the latter is considered non-configurable and is harder to test.

In Python, however, dependency injection is not a big deal due to its dynamic nature and duck typing: anything could be defined anytime and passed anywhere. Due to that reason (and maybe some others) DI frameworks aren't popular among Python community, though they may be handy in some cases.

One of such cases is code decoupling when we want to create and use objects in different places, preserving clean interface and avoiding global variables. Having all these considerations in mind, Picobox was born.

CHAPTER 2

Quickstart

Picobox provides `Box` class that acts as a container for objects you want to deal with. You can put, you can get, you can pass them around.

```python
import picobox

box = picobox.Box()
box.put('foo', 42)

@box.pass_('foo')
def spam(foo):
    return foo

@box.pass_('foo', as_='bar')
def eggs(bar):
    return bar

print(box.get('foo'))    # 42

print(spam())            # 42
print(eggs())            # 42
```

One of the key principles is *not to break* existing code. That's why Picobox does not change function signature and injects dependencies as if they are defaults.

```python
print(spam())            # 42
print(spam(13))          # 13
print(spam(foo=99))      # 99
```

Another key principle is that `pass_()` resolves dependencies lazily which means you can inject them everywhere you need and define them much later. The only rule is to define them before calling the function.

```python
import picobox

box = picobox.Box()
```

(continues on next page)

(continued from previous page)

```

@box.pass_('foo')
def spam(foo):
    return foo

print(spam(13))          # 13
print(spam())            # KeyError: 'foo'

box.put('foo', 42)

print(spam())            # 42
```

The value to inject is not necessarily an object. You can pass a factory function which will be used to produce a dependency. A factory function has no arguments, and is assumed to have all the context it needs to work.

```
import picobox
import random

box = picobox.Box()
box.put('foo', factory=lambda: random.choice(['spam', 'eggs']))

@box.pass_('foo')
def spam(foo):
    return foo

print(spam())            # spam
print(spam())            # eggs
print(spam())            # eggs
print(spam())            # spam
print(spam())            # eggs
```

Whereas factories are enough to implement whatever creation policy you want, there's no good in repeating yourself again and again. That's why Picobox introduces *scope* concept. Scope is a way to say whether you want to share dependencies in some execution context or not.

For instance, you may want to share it globally (singleton) or create only one instance per thread (threadlocal).

```
import picobox
import random
import threading

box = picobox.Box()
box.put('foo', factory=random.random, scope=picobox.threadlocal)
box.put('bar', factory=random.random, scope=picobox.singleton)

@box.pass_('foo')
def spam(foo):
    print(foo)

@box.pass_('bar')
def eggs(bar):
    print(bar)

# prints
# > 0.9464005851114538
# > 0.8585111290081737
for _ in range(2):
```

(continues on next page)

(continued from previous page)

```
    threading.Thread(target=spam).start()

# prints
# > 0.5333214411659912
# > 0.5333214411659912
for _ in range(2):
    threading.Thread(target=eggs).start()
```

But the cherry on the cake is a so called Picobox's stack interface. `Box` is great to manage dependencies but it requires to be created before using. In practice it usually means you need to create it globally to get access from various places. The stack interface is called to solve this by providing general methods that will be applied to latest active box instance.

```
import picobox

@picobox.pass_('foo')
def spam(foo):
    return foo

box_a = picobox.Box()
box_a.put('foo', 13)

box_b = picobox.Box()
box_b.put('foo', 42)

with picobox.push(box_a):
    print(spam())               # 13

    with picobox.push(box_b):
        print(spam())           # 42

    print(spam())               # 13

spam()                          # RuntimeError: no boxes on the stack
```

When only partial overriding is necessary, you can chain pushed box so any missed lookups will be proxied to the box one level down the stack.

```
import picobox

@picobox.pass_('foo')
@picobox.pass_('bar')
def spam(foo, bar):
    return foo + bar

box_a = picobox.Box()
box_a.put('foo', 13)
box_a.put('bar', 42)

box_b = picobox.Box()
box_b.put('bar', 0)

with picobox.push(box_a):
    with picobox.push(box_b, chain=True):
        print(spam())           # 13
```

The stack interface is recommended way to use Picobox because it allows to switch between DI containers (boxes) on the fly. This is also the only way to test your application because patching (mocking) globally defined boxes is not a solution.

```
def test_spam():
    with picobox.push(picobox.Box(), chain=True) as box:
        box.put('foo', 42)
        assert spam() == 42
```

`picobox.push()` can also be used as a regular function, not only as a context manager.

```
def test_spam():
    box = picobox.push(picobox.Box(), chain=True)
    box.put('foo', 42)
    assert spam() == 42
    picobox.pop()
```

Every call to `picobox.push()` should eventually be followed by a corresponding call to `picobox.pop()` to remove the box from the top of the stack, when you are done with it.

Note: Dependency Injection is usually used in applications, not libraries, to wire things together. Occasionally such need may come in libraries too, so picobox provides a *`picobox.Stack`* class to create an independent non overlapping stack with boxes suitable to be used in such cases.

Just create a global instance of stack (globals themeselves aren't bad), and use it as you'd use picobox stacked interface:

```
import picobox

stack = picobox.Stack()

@stack.pass_('a', as_='b')
def mysum(a, b):
    return a + b

with stack.push(picobox.Box()) as box:
    box.put('a', 42)
    assert mysum(13) == 55
```

CHAPTER 3

API reference

3.1 Box

class picobox.**Box**

Box is a dependency injection (DI) container.

DI container is an object that contains any amount of factories, one for each dependency apart. Dependency, on the other hand, is an ordinary instance or value the container needs to provide on demand.

Thanks to scopes, the class keeps track of produced dependencies and knows exactly when to reuse them or when to create new ones. That is to say each scope defines a set of rules for when to reuse dependencies.

Here's a minimal example of how a Box instance can be used:

```
import picobox

box = picobox.Box()
box.put('magic', 42)

@box.pass_('magic')
def do(magic):
    return magic + 1

assert box.get('magic') == 42
assert do(13) == 14
assert do() == 43
```

get(*key*, *default=<optional>*)

Retrieve a dependency (aka service) out of the box instance.

The process involves creation of requested dependency by calling an associated *factory* function, and then returning result back to the caller code. If a dependency is *scoped*, there's a chance for an existing instance to be returned instead.

Parameters

- **key** – A key to retrieve a dependency. Must be the one used when calling `put()` method.

- **default** – (optional) A fallback value to be returned if there's no *key* in the box. If not passed, *KeyError* is raised.

Raises **KeyError** – If no dependencies saved under *key* in the box.

pass_(*key*, *as_=<optional>*)
Pass a dependency to a function if nothing explicitly passed.

The decorator implements late binding which means it does not require to have a dependency instance in the box before applying. The instance will be looked up when a decorated function is called. Other important property is that it doesn't change a signature of decorated function preserving a way to explicitly pass arguments ignoring injections.

Parameters

- **key** – A key to retrieve a dependency. Must be the one used when calling `put()` method.
- **as_** – (optional) Bind a dependency associated with *key* to a function argument named *as_*. If not passed, the same as *key*.

Raises **KeyError** – If no dependencies saved under *key* in the box.

put(*key*, *value=<optional>*, *factory=<optional>*, *scope=<optional>*)
Define a dependency (aka service) within the box instance.

A dependency can be expressed either directly, by passing a concrete *value*, or via *factory* function. A *factory* may be accompanied by *scope* that defines a set of rules for when to create a new dependency instance and when to reuse existing one. If *scope* is not passed, no scope is assumed which means produce a new instance each time it's requested.

Parameters

- **key** – A key under which to put a dependency. Can be any hashable object, but string is recommended.
- **value** – A dependency to be stored within a box under *key* key. Can be any object. A syntax sugar for `factory=lambda: value`.
- **factory** – A factory function to produce a dependency when needed. Must be callable with no arguments.
- **scope** – A scope to keep track of produced dependencies. Must be a class that implements `Scope` interface.

Raises **ValueError** – If both *value* and *factory* are passed.

3.2 ChainBox

class picobox.**ChainBox**(**boxes*)
ChainBox groups multiple boxes together to create a single view.

ChainBox for boxes is essentially the same as `ChainMap` for mappings. It mimics `Box` interface and hence can substitute one but provides a way to look up dependencies in underlying boxes.

Here's a minimal example of how ChainBox instance can be used:

```
box_a = picobox.Box()
box_a.put('magic_a', 42)

box_b = picobox.Box()
```

(continues on next page)

(continued from previous page)

```
box_b.put('magic_a', factory=lambda: 10)
box_b.put('magic_b', factory=lambda: 13)

chainbox = picobox.ChainBox(box_a, box_b)

@chainbox.pass_('magic_a')
@chainbox.pass_('magic_b')
def do(magic_a, magic_b):
    return magic_a + magic_b

assert chainbox.get('magic_b') == 13
assert do() == 55
```

Parameters **`boxes`** – (optional) A list of boxes to lookup into. If no boxes are passed, an empty box is created and used as underlying box instead.

New in version 1.1.

`get` (*key*, *default=<optional>*)
Same as *`Box.get()`* but looks up for key in underlying boxes.

`put` (*key*, *value=<optional>*, *factory=<optional>*, *scope=<optional>*)
Same as *`Box.put()`* but applies to first underlying box.

3.3 Scopes

`class` `picobox.`**`Scope`**
Scope is an execution context based storage interface.

Execution context is a mechanism of storing and accessing data bound to a logical thread of execution. Thus, one may consider processes, threads, greenlets, coroutines, Flask requests to be examples of a logical thread.

The interface provides just two methods:

- *`set()`* - set execution context item
- *`get()`* - get execution context item

See corresponding methods for details below.

`get` (*key*)
Get *value* by *key* for current execution context.

`set` (*key*, *value*)
Bind *value* to *key* in current execution context.

`picobox.`**`singleton`**
Share instances across application.

`picobox.`**`threadlocal`**
Share instances across the same thread.

`picobox.`**`contextvars`**
Share instances across the same execution context (**PEP 567**).

Since asyncio does support context variables, the scope could be used in asynchronous applications to share dependencies between coroutines of the same `asyncio.Task`.

New in version 2.1.

picobox.**noscope**

Do not share instances, create them each time on demand.

picobox.contrib.flaskscopes.**application**

Share instances across the same Flask (HTTP) application.

In most cases can be used interchangeably with *picobox.singleton* scope. Comes around when you have multiple Flask applications and you want to have independent instances for each Flask application, despite the fact they are running in the same WSGI context.

New in version 2.2.

picobox.contrib.flaskscopes.**request**

Share instances across the same Flask (HTTP) request.

New in version 2.2.

3.4 Stacked API

class picobox.**Stack**(*name=None*)

Stack is a dependency injection (DI) container for containers (boxes).

While *Box* is a great way to manage dependencies, it has no means to override them. This might be handy most of all in tests, where you usually need to provide a special set of dependencies configured for test purposes. This is where *Stack* comes in. It provides the very same interface Box does, but proxies all calls to a box on the top.

This basically means you can define injection points once, but change dependencies on the fly by changing DI containers (boxes) on the stack. Here's a minimal example of how a stack can be used:

```python
import picobox

stack = picobox.Stack()

@stack.pass_('magic')
def do(magic):
    return magic + 1

foobox = picobox.Box()
foobox.put('magic', 42)

barbox = picobox.Box()
barbox.put('magic', 13)

with stack.push(foobox):
    with stack.push(barbox):
        assert do() == 14
    assert do() == 43
```

Note: Usually you want to have only one stack instance to wire things up. That's why picobox comes with pre-created stack instance. You can work with that instance using *push()*, *pop()*, *put()*, *get()* and *pass_()* functions.

Parameters **name** – (optional) A name of the stack.

New in version 2.2.

get(*key*, *default=<optional>*)
The same as `Box.get()` but for a box at the top.

pass_(*key*, *as_=<optional>*)
The same as `Box.pass_()` but for a box at the top.

pop()
Pop the box from the top of the stack.

Should be called once for every corresponding call to `push()` in order to remove the box from the top of the stack, when a caller is done with it.

Note: Normally `push()` should be used a context manager, in which case the box on the top is removed automatically on exit from the block (i.e. no need to call `pop()` manually).

Returns a removed box

Raises `IndexError` – If the stack is empty and there's nothing to pop.

push(*box*, *chain=False*)
Push a `Box` instance to the top of the stack.

Returns a context manager, that will automatically pop the box from the top of the stack on exit. Can also be used as a regular function, in which case it's up to callers to perform a corresponding call to `pop()`, when they are done with the box.

Parameters

- **box** – A `Box` instance to push to the top of the stack.
- **chain** – (optional) Look up missed keys one level down the stack. To look up through multiple levels, each level must be created with this option set to `True`.

put(*key*, *value=<optional>*, *factory=<optional>*, *scope=<optional>*)
The same as `Box.put()` but for a box at the top.

picobox.**push**(*box*, *chain=False*)
The same as `Stack.push()` but for a shared stack instance.

New in version 1.1: `chain` parameter

picobox.**pop**()
The same as `Stack.pop()` but for a shared stack instance.

New in version 2.0.

picobox.**put**(*key*, *value=<optional>*, *factory=<optional>*, *scope=<optional>*)
The same as `Stack.put()` but for a shared stack instance.

picobox.**get**(*key*, *default=<optional>*)
The same as `Stack.get()` but for a shared stack instance.

picobox.**pass_**(*key*, *as_=<optional>*)
The same as `Stack.pass_()` but for a shared stack instance.

CHAPTER 4

Release Notes

Note: Picobox follows Semantic Versioning which means backward incompatible changes will be released along with bumping major version component.

4.1 2.2.0

- Fix `picobox.singleton`, `picobox.threadlocal` & `picobox.contextvars` scopes so they do not fail with unexpected exception when non-string formattable missing key is passed.
- Add `picobox.contrib.flaskscopes` module with *application* and *request* scopes for Flask web framework.
- Add `picobox.Stack` class to create stacks with boxes on demand. Might be useful for third-party developers who want to use picobox yet avoid collisions with main application developers.

4.2 2.1.0

- Add `picobox.contextvars` scope (python 3.7 and above) that can be used in asyncio applications to have a separate set of dependencies in all coroutines of the same task.
- Fix `picobox.threadlocal` issue when it was impossible to use any hashable key other than `str`.
- Nested `picobox.pass_` calls are now squashed into one in order to improve runtime performance.
- Add `Python 2.7` support.

4.3 2.0.0

Released on Mar 18, 2018.

- `picobox.push()` can now be used as a regular function as well, not only as a context manager. This is a breaking change because from now one a box is pushed on stack immediately when calling `picobox.push()`, no need to wait for `__enter__()` to be called.
- New `picobox.pop()` function, that pops the box from the top of the stack.
- Fixed a potential race condition on concurrent calls to `picobox.push()` that may occur in non-CPython implementations.

4.4 1.1.0

Released on Dec 19, 2017.

- New `ChainBox` class that can be used similar to `ChainMap` but for boxes. This basically means from now on you can group few boxes into one view, and use that view to look up dependencies.
- New `picobox.push()` argument called `chain` that can be used to look up keys down the stack on misses.

4.5 1.0.0

Released on Nov 25, 2017.

- First public release with initial bunch of features.

Python Module Index

Index