mush Documentation

Release 1.3

Simplistix Ltd

December 14, 2015

Contents

Mush is a light weight type-based dependency injection framework aimed at enabling the easy testing and re-use of chunks of code that make up scripts.

CHAPTER 1

How Mush works

Note: This documentation explains how Mush works using fairly abstract examples. If you'd prefer more "real world" examples please see the Example Usage documentation.

1.1 Constructing runners

Mush works by assembling a number of callables into a *Runner*:

```
from mush import Runner

def func1():
    print('func1')

def func2():
    print('func2')

runner = Runner(func1, func2)
```

Once assembled, a runner can be called any number of times. Each time it is called, it will call each of its callables in turn:

```
>>> runner()
func1
func2
```

More callables can be added to a runner:

```
def func3():
    print('func3')

runner.add(func3)
```

If you want to add several callables in one go, you can use the runner's *extend()* method:

```
def func4():
    print('func4')

def func5():
    print('func5')

runner.extend(func4, func5)
```

Now, when called, the runner will call all five functions:

```
>>> runner()
func1
func2
func3
func4
func5
```

Runners can also be added together to create a new runner:

```
runner1 = Runner(func1)
runner2 = Runner(func2)
runner3 = runner1 + runner2
```

This addition does not modify the existing runners, but does give the result you'd expect:

```
>>> runner1()
func1
>>> runner2()
func2
>>> runner3()
func1
func2
```

This can also be done by passing runners in when creating a new runner or calling the extend method on a runner, for example:

```
runner1 = Runner(func1)
runner2 = Runner(func2)
runner4_1 = Runner(runner1, runner2)
runner4_2 = Runner()
runner4_2.extend(runner1, runner2)
```

In both cases, the results are as you would expect:

```
>>> runner4_1()
func1
func2
>>> runner4_2()
func1
func2
```

Finally, runners can be cloned, providing a way to encapsulate commonly used base runners that can then be extended for each specific use case:

```
runner5 = runner3.clone()
runner5.add(func4)
```

The existing runner is not modified, while the new runner behaves as expected:

```
>>> runner3()
func1
func2
>>> runner5()
func1
func2
func4
```

1.2 Configuring Resources

Where Mush becomes useful is when the callables in a runner either produce or require objects of a certain type. Given the right configuration, Mush will wire these together enabling you to write easily testable and reusable callables that encapsulate specific pieces of functionality. This configuration is done either imperatively, declaratively or using a combination of the two styles as described in the sections below.

For the examples, we'll assume we have three types of resources:

```
class Apple:
    def __str__(self):
        return 'an apple'
    __repr__ = __str__

class Orange:
    def __str__(self):
        return 'an orange'
    __repr__ = __str__

class Juice:
    def __str__(self):
        return 'a refreshing fruit beverage'
    __repr__ = __str__
```

1.2.1 Imperative configuration

Imperative configuration requires no decorators and is great when working with callables that come from another package or the standard library:

```
def apple_tree():
     print('I made an apple')
     return Apple()

def magician(fruit):
     print('I turned {0} into an orange'.format(fruit))
     return Orange()

def juicer(fruit1, fruit2):
     print('I made juice out of {0} and {1}'.format(fruit1, fruit2))
```

The requirements are passed to the `add()` method of a runner which can express requirements for both arguments and keyword parameters:

```
runner = Runner()
runner.add(apple_tree)
runner.add(magician, Apple)
runner.add(juicer, fruit1=Apple, fruit2=Orange)
```

Calling this runner will now manage the resources, collecting them and passing them in as configured:

```
>>> runner()
I made an apple
I turned an apple into an orange
I made juice out of an apple and an orange
```

1.2.2 Declarative configuration

This is done using the *requires()* decorator to mark the callables with their requirements, which can specify the types required for either arguments or keyword parameters:

```
from mush import requires

def apple_tree():
    print('I made an apple')
    return Apple()

@requires(Apple)
def magician(fruit):
    print('I turned {0} into an orange'.format(fruit))
    return Orange()

@requires(fruit1=Apple, fruit2=Orange)
def juicer(fruit1, fruit2):
    print('I made juice out of {0} and {1}'.format(fruit1, fruit2))
    return Juice()
```

These can now be combined into a runner and executed. The runner will extract the requirements stored by the decorator and will use them to map the parameters as appropriate:

```
>>> runner = Runner(apple_tree, magician, juicer)
>>> runner()
I made an apple
I turned an apple into an orange
I made juice out of an apple and an orange
```

1.2.3 Hybrid configuration

The two styles of configuration are entirely interchangeable, with declarative requirements being inspected whenever a callable is added to a runner, and imperative requirements being taken whenever they are passed via the *add()* method:

```
@requires(Juice)
def packager(juice):
    print('I put {0} in a bottle'.format(juice))

def orange_tree():
    print('I made an orange')
    return Orange()

trees = Runner(apple_tree, orange_tree)
runner = trees.clone()
runner.extend(juicer, packager)
```

This runner now ends up with bottled juice:

```
>>> runner()
I made an apple
I made an orange
I made juice out of an apple and an orange
I put a refreshing fruit beverage in a bottle
```

It's useful to note that imperative configuration will be used in preference to declarative configuration where both are present:

```
runner = trees.clone()
runner.add(juicer, Orange, Apple)
```

This runner will give us juice made in a different order:

```
>>> runner()
I made an apple
I made an orange
I made juice out of an orange and an apple
```

1.3 Resource usage periods

It can be important for a callable to have either first access or last access to a particular resource. For this reason, configuration can specify one of three periods during which a callable needs to be called with a particular resource; the default is *normal* and a decorator can be used to indicate either *first* or *last*. Within these periods, callables are called in the order they are added to the runner.

As an example, consider a ring and some things that can be done to it:

```
class Ring:
    def __str__(self):
        return 'a ring'

def forge():
    return Ring()

def polish(ring):
    print('polishing {0}'.format(ring))

def more_polish(ring):
    print('polishing {0} again'.format(ring))

def engrave(ring):
    print('engraving {0}'.format(ring))

def package(ring):
    print('packaging {0}'.format(ring))
```

These can now be added to a runner with configuration expressing the correct periods:

```
from mush import Runner, first, last

runner = Runner(forge)
runner.add(package, last(Ring))
runner.add(polish, first(Ring))
runner.add(more_polish, first(Ring))
runner.add(engrave, Ring)
```

Even though the callables were added out order, they will be executed correctly:

```
>>> runner()
polishing a ring
polishing a ring again
engraving a ring
packaging a ring
```

The configuration of periods works identically with both the imperative and declarative forms.

1.3.1 Waiting for a resource

Sometimes, a callable needs to wait for some other callable to do its work but does not need or cannot accept objects of the type returned by that callable. For example, and miss-using the fruit types from above:

```
def func1():
    return Apple()

def func2(apple):
    print('func2 got {0}'.format(apple))
    return Orange()

def func3(apple):
    print('func3 got {0}'.format(apple))

def func4(orange):
    print('func4 processed {0}'.format(orange))
```

If we want `func3()` only to get called once `func4()` has processed the `Orange` but, for reasons of abstraction, we want to add the callables in the order defined above, the simplest runner will not give us what we want:

```
>>> runner = Runner(func1)
>>> runner.add(func2, Apple)
>>> runner.add(func3, Apple)
>>> runner.add(func4, Orange)
>>> runner()
func2 got an apple
func3 got an apple
func4 processed an orange
```

The problem is that the runner hasn't been told that `func3()` has a dependency on `Orange`. This can be done using the *`after()`* type wrapper to specify that `func2()` requires an `Orange` to exist, and for any other callables added to the runner that need an `Orange` to have been called first, but that it must not be passed that orange:

```
from mush import Runner, after

runner = Runner(func1)
runner.add(func2, Apple)
runner.add(func3, after(Orange), Apple)
runner.add(func4, Orange)
```

Now, even though we've added the callables in the order we want, we get the order of calling that we need:

```
>>> runner()
func2 got an apple
func4 processed an orange
func3 got an apple
```

1.4 Special return types

There are certain types that can be returned from a callable that have special meaning. While these are provided in case of their specific need, you should think twice when you find yourself wanting to use them as it is often a sign that your code can be better structured differently.

For the examples below, we'll use the fruit classes from above.

1.4.1 Returning multiple resources

A callable can return multiple resources by returning either a list or a tuple:

```
def orchard():
    return Apple(), Orange()
```

This can be used to provide both types of fruit:

```
>>> runner = Runner(orchard, juicer)
>>> runner()
I made juice out of an apple and an orange
```

1.4.2 Overriding the type of a resource

Sometimes you may need to force a returned resource to be of a particular type. When this is the case, a callable can return a dictionary mapping the forced type to the required type:

```
class Pear:
    def __str__(self):
        return 'a pear'

def desperation():
    print('oh well, a pear will have to do')
    return {Apple: Pear()}
```

We can now make juice even though we don't have apples:

```
>>> runner = Runner(orange_tree, desperation, juicer)
>>> runner()
I made an orange
oh well, a pear will have to do
I made juice out of a pear and an orange
```

If you have no control over a callable that returns an object of the 'wrong' type, you have two options. You can either decorate it:

```
from mush import returns

@returns(Apple)
def make_pears():
    print('I made a pear')
    return Pear()
```

Now you can use the pear maker to get juice:

```
>>> runner = Runner(orange_tree, make_pears, juicer)
>>> runner()
I made an orange
I made a pear
I made juice out of a pear and an orange
```

If you can't even decorate the callable, then you can also imperatively indicate that the return type should be overridden:

```
from mush import returns

def someone_elses_pears():
    print('I made a pear')
    return Pear()
```

Now you can use the pear maker to get juice:

```
>>> runner = Runner(orange_tree, juicer)
>>> runner.add_returning(someone_elses_pears, returns=Apple)
>>> runner()
I made an orange
I made a pear
I made juice out of a pear and an orange
```

1.4.3 Returning marker types

In some circumstances, you may need to ensure that some callables are used only after another callable has done its work, even though that work does not return a resource. This can be achieved using a marker type as follows:

```
from mush import nothing, marker

@returns(marker('SetupComplete'))
def setup():
    print('setting things up')

@requires(after(marker('SetupComplete')))
def body():
    print('doing stuff')
```

Note that the `body()` callable does not take any arguments or parameters; because *`after()`* is used, it is not passed and is used purely to affect the order of calling:

```
>>> runner = Runner(body, setup)
>>> runner()
setting things up
doing stuff
```

1.5 Using parts of a resource

When pieces of functionality use settings provided either by command line arguments or from configuration files, it's often cleaner to structure that code to recieve the specific setting value rather than the setting's container. It's certainly easier to test. Mush can take care of the needed wiring when configured to do so using the *`attr`* and *`item`* helpers:

```
from mush import Runner, attr, item, requires

class Config(dict): pass

class Args(object):
    fruit = 'apple'
    tree = dict(fruit='pear')

def parse_args():
    return Args()

def read_config():
    return Config(fruit='orange')
```

```
@requires(attr(Args, 'fruit'),
          item(Config, 'fruit'),
          item(attr(Args, 'tree'), 'fruit'))
def pick(fruit1, fruit2, fruit3):
    print('I picked {0}, {1} and {2}'.format(fruit1, fruit2, fruit3))
    picked = []
    for fruit in fruit1, fruit2:
        if fruit=='apple':
            picked.append(Apple())
        elif fruit=='orange':
            picked.append(Orange())
        else:
            raise TypeError('You have made a poor fruit choice')
    return picked
```

While the `pick()` function remains usable and testable on its own:

```
>>> pick('apple', 'orange', 'pear')
I picked apple, orange and pear
[an apple, an orange]
```

It can also be added to a runner with the other necessary functions and Mush will do the hard work:

```
>>> runner = Runner(parse_args, read_config, pick, juicer)
>>> runner()
I picked apple, orange and pear
I made juice out of an apple and an orange
```

1.6 Context manager resources

A frequent requirement when writing scripts is to make sure that when unexpected things happen they are logged, transactions are aborted, and other necessary cleanup is done. Mush supports this pattern by allowing context managers to be added as callables:

```
from mush import Runner, requires

class Transactions(object):

    def __enter__(self):
        print('starting transaction')

    def __exit__(self, type, obj, tb):
        if type:
            print(obj)
            print('aborting transaction')
        else:
            print('committing transaction')
        return True

def a_func():
    print('doing my thing')

def good_func():
    print('I have done my thing')
```

```
def bad_func():
    raise Exception("I don't want to do my thing")
```

The context manager is wrapped around all callables that are called after it:

```
>>> runner = Runner(Transactions, a_func, good_func)
>>> runner()
starting transaction
doing my thing
I have done my thing
committing transaction
```

This gives it a chance to clear up when things go wrong:

```
>>> runner = Runner(Transactions, a_func, bad_func)
>>> runner()
starting transaction
doing my thing
I don't want to do my thing
aborting transaction
```

1.7 Debugging

Mush makes some heuristic decisions about the order in which to call objects added to a runner. If your expectations of the call order don't match that used by Mush, it can be confusing to figure out where the difference comes from.

For this reason, when constructing a *Runner* you can pass an optional debug parameter which can be either a boolean `True` or a file-like object. If passed, debug information will be generated whenever an object is added to the runner.

If `True`, this information will be written to `stderr`. If a file-like object is passed instead, the information will be written to that object.

As an example, consider this code:

```
from mush import Runner, requires

class T1(object): pass
class T2(object): pass
class T3(object): pass

def makes_t1():
    return T1()

@requires(T1)
def makes_t2(obj):
    return T2()

@requires(T2)
def makes_t3(obj):
    return T3()

@requires(T3, T1)
def user(obj1, obj2):
    m.user(type(obj1), type(obj2))
```

The debug information that would be written looks like this:

```
>>> import sys
>>> runner = Runner(makes_t1, makes_t2, makes_t3, user, debug=sys.stdout)
Added <function makes_t1 ...> to 'normal' period for <... 'NoneType'> with Requirements()
Current call order:
For <... 'NoneType'>:
  normal: <function makes_t1 ...> requires Requirements()

Added <function makes_t2 ...> to 'normal' period for <class 'T1'> with Requirements(T1)
Current call order:
For <... 'NoneType'>:
  normal: <function makes_t1 ...> requires Requirements()
For <class 'T1'>:
  normal: <function makes_t2 ...> requires Requirements(T1)

Added <function makes_t3 ...> to 'normal' period for <class 'T2'> with Requirements(T2)
Current call order:
For <... 'NoneType'>:
  normal: <function makes_t1 ...> requires Requirements()
For <class 'T1'>:
  normal: <function makes_t2 ...> requires Requirements(T1)
For <class 'T2'>:
  normal: <function makes_t3 ...> requires Requirements(T2)

Added <function user ...> to 'normal' period for <class 'T3'> with Requirements(T3, T1)
Current call order:
For <... 'NoneType'>:
  normal: <function makes_t1 ...> requires Requirements()
For <class 'T1'>:
  normal: <function makes_t2 ...> requires Requirements(T1)
For <class 'T2'>:
  normal: <function makes_t3 ...> requires Requirements(T2)
For <class 'T3'>:
  normal: <function user ...> requires Requirements(T3, T1)
```

CHAPTER 2

Example Usage

To show how Mush works from a more practical point of view, let's start by looking at a simple script that covers several common patterns:

```python
from argparse import ArgumentParser
from .configparser import RawConfigParser
import logging, os, sqlite3, sys

log = logging.getLogger()

def main():
    parser = ArgumentParser()
    parser.add_argument('config', help='Path to .ini file')
    parser.add_argument('--quiet', action='store_true',
                        help='Log less to the console')
    parser.add_argument('--verbose', action='store_true',
                        help='Log more to the console')
    parser.add_argument('path', help='Path to the file to process')

    args = parser.parse_args()

    config = RawConfigParser()
    config.read(args.config)

    handler = logging.FileHandler(config.get('main', 'log'))
    handler.setLevel(logging.DEBUG)
    log.addHandler(handler)

    if not args.quiet:
        handler = logging.StreamHandler(sys.stderr)
        handler.setLevel(logging.DEBUG if args.verbose else logging.INFO)
        log.addHandler(handler)

    conn = sqlite3.connect(config.get('main', 'db'))

    try:
        filename = os.path.basename(args.path)
        with open(args.path) as source:
            conn.execute('insert into notes values (?, ?)',
                         (filename, source.read()))
        conn.commit()
        log.info('Successfully added %r', filename)
    except:
        log.exception('Something went wrong')
```

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

As you can see, the script above takes some command line arguments, loads some configuration from a file, sets up log handling and then loads a file into a database. While simple and effective, this script is hard to test. Even using the `TempDirectory` and `Replacer` helpers from the TestFixtures package, the only way to do so is to write one or more high level tests such as the following:

```
from testfixtures import TempDirectory, Replacer, OutputCapture
import sqlite3

class Tests(TestCase):

    def test_main(self):
        with TempDirectory() as d:
            # setup db
            db_path = d.getpath('sqlite.db')
            conn = sqlite3.connect(db_path)
            conn.execute('create table notes (filename varchar, text varchar)')
            conn.commit()
            # setup config
            config = d.write('config.ini', '''
[main]
db = %s
log = %s
''' % (db_path, d.getpath('script.log')), 'ascii')
            # setup file to read
            source = d.write('test.txt', 'some text', 'ascii')
            with Replacer() as r:
                r.replace('sys.argv', ['script.py', config, source, '--quiet'])
                main()
            # check results
            self.assertEqual(
                conn.execute('select * from notes').fetchall(),
                [('test.txt', 'some text')]
                )
```

The problem is that, in order to test the different paths through the small piece of logic at the end of the script, we have to work around all the set up and handling done by the rest of the script.

This also makes it hard to re-use parts of the script. It's common for a project to have several scripts, all of which get some config from the same file, have the same logging options and often use the same database connection.

2.1 Encapsulating the re-usable parts of scripts

So, let's start by looking at how these common sections of code can be extracted into re-usable functions that can be assembled by Mush into scripts:

```
from argparse import ArgumentParser, Namespace
from .configparser import RawConfigParser
from mush import Runner, requires, first, last, attr, item
import logging, os, sqlite3, sys

log = logging.getLogger()

@requires(ArgumentParser)
```

```
def base_options(parser):
    parser.add_argument('config', help='Path to .ini file')
    parser.add_argument('--quiet', action='store_true',
                        help='Log less to the console')
    parser.add_argument('--verbose', action='store_true',
                        help='Log more to the console')

@requires(last(ArgumentParser))
def parse_args(parser):
    return parser.parse_args()

class Config(dict): pass

@requires(first(Namespace))
def parse_config(args):
    config = RawConfigParser(dict_type=Config)
    config.read(args.config)
    return Config(config.items('main'))

def setup_logging(log_path, quiet=False, verbose=False):
    handler = logging.FileHandler(log_path)
    handler.setLevel(logging.DEBUG)
    log.addHandler(handler)
    if not quiet:
        handler = logging.StreamHandler(sys.stderr)
        handler.setLevel(logging.DEBUG if verbose else logging.INFO)
        log.addHandler(handler)

class DatabaseHandler:
    def __init__(self, db_path):
        self.conn = sqlite3.connect(db_path)
    def __enter__(self):
        return self
    def __exit__(self, type, obj, tb):
        if type:
            log.exception('Something went wrong')
            self.conn.rollback()
```

We start with a function that adds the options needed by all our scripts to an `argparse` parser. This uses the *`requires()`* decorator to tell Mush that it must be called with an `ArgumentParser` instance. See *Configuring Resources* for more details.

Next, we have a function that calls `parse_args()` on the parser and returns the resulting `Namespace`. It uses the *`last()`* decorator to indicate that it should be the last callable to use the `ArgumentParser` instance. See *Resource usage periods* for more details.

The `parse_config()` function uses the *`first()`* decorator to indicate that it needs first use of the command line arguments. It uses those to read the specified configuration and return the configuration as a dictionary. A sub-class of `dict` is used so that other callables can easily indicate that they require the configuration to be passed in.

Finally, there is a function that configures log handling and a context manager that provides a database connection and handles exceptions that occur by logging them and aborting the transaction. Context managers like this are handled by Mush in a specific way, see *Context manager resources* for more details.

Each of these components can be separately and thoroughly tested. The Mush decorations are inert to all but the Mush *`Runner`*, meaning that they can be used independently of Mush in whatever way is convenient. As an example, the following tests use a `TempDirectory` and a `LogCapture` to fully test the database-handling context manager:

```
    def setUp(self):
        self.dir = TempDirectory()
        self.db_path = self.dir.getpath('test.db')
        self.conn = sqlite3.connect(self.db_path)
        self.conn.execute('create table notes '
                          '(filename varchar, text varchar)')
        self.conn.commit()
        self.log = LogCapture()

    def tearDown(self):
        self.log.uninstall()
        self.dir.cleanup()

    def test_normal(self):
        with DatabaseHandler(self.db_path) as handler:
            handler.conn.execute('insert into notes values (?, ?)',
                                 ('test.txt', 'a note'))
            handler.conn.commit()
        # check the row was inserted and committed
        curs = self.conn.cursor()
        curs.execute('select * from notes')
        self.assertEqual(curs.fetchall(), [('test.txt', 'a note')])
        # check there was no logging
        self.log.check()

    def test_exception(self):
        with ShouldRaise(Exception('foo')):
            with DatabaseHandler(self.db_path) as handler:
                handler.conn.execute('insert into notes values (?, ?)',
                                     ('test.txt', 'a note'))
                raise Exception('foo')
        # check the row not inserted and the transaction was rolled back
        curs = handler.conn.cursor()
        curs.execute('select * from notes')
        self.assertEqual(curs.fetchall(), [])
        # check the error was logged
        self.log.check(('root', 'ERROR', 'Something went wrong'))
```

2.2 Writing the specific parts of your script

Now that all the re-usable parts of the script have been abstracted, writing a specific script becomes a case of just writing two functions:

```
def args(parser):
    parser.add_argument('path', help='Path to the file to process')

def do(conn, path):
    filename = os.path.basename(path)
    with open(path) as source:
        conn.execute('insert into notes values (?, ?)',
                     (filename, source.read()))
    conn.commit()
    log.info('Successfully added %r', filename)
```

As you can imagine, this much smaller set of code is simpler to test and easier to maintain.

2.3 Assembling the components into a script

So, we now have a library of re-usable components and the specific callables we require for the current script. All we need to do now is assemble these parts into the final script. The full details of this are covered in *Constructing runners* but two common patterns are covered below.

2.3.1 Cloning

With this pattern, a "base runner" is created, usually in the same place that other re-usable parts of the original script are located:

```
from mush import Runner, requires, first, last, attr, item

base_runner = Runner(ArgumentParser, base_options, parse_args, parse_config)
base_runner.add(setup_logging,
                log_path = item(first(Config), 'log'),
                quiet = attr(first(Namespace), 'quiet'),
                verbose = attr(first(Namespace), 'verbose'))
```

The code above shows some different ways of getting Mush to pass parts of an object returned from a previous callable to the parameters of another callable. See *Using parts of a resource* for full details.

Now, for each specific script, the base runner is cloned and the script-specific parts added to the clone leaving a callable that can be put in the usual block at the bottom of the script:

```
main = base_runner.clone()
main.add(args, ArgumentParser)
main.add(DatabaseHandler, item(Config, 'db'))
main.add(do,
         attr(DatabaseHandler, 'conn'),
         attr(Namespace, 'path'))

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

2.3.2 Using a factory

This pattern is most useful when you have several or more scripts that all follow a similar pattern when it comes to assembling the runner from the common parts and the specific parts. For example, if all your scripts take a path to a config file and a path to a file that needs processing, you can write a factory function that returns a runner based on the callable that does the work as follows:

```
def options(parser):
    parser.add_argument('config', help='Path to .ini file')
    parser.add_argument('--quiet', action='store_true',
                        help='Log less to the console')
    parser.add_argument('--verbose', action='store_true',
                        help='Log more to the console')
    parser.add_argument('path', help='Path to the file to process')

def make_runner(do):
    runner = Runner(ArgumentParser)
    runner.add(options, ArgumentParser)
    runner.add(parse_args, last(ArgumentParser))
    runner.add(parse_config, first(Namespace))
    runner.add(setup_logging,
               log_path = item(first(Config), 'log'),
               quiet = attr(first(Namespace), 'quiet'),
               verbose = attr(first(Namespace), 'verbose'))
    runner.add(DatabaseHandler, item(Config, 'db'))
    runner.add(do,
               attr(DatabaseHandler, 'conn'),
               attr(Namespace, 'path'))
    return runner
```

With this in place, the specific script becomes the `do()` function we abstracted above and a very short call to the factory:

```
main = make_runner(do)
```

A combination of the clone and factory patterns can also be used to get the best of both worlds. Setting up several base runners and are clones of a parent runner and having factories that take common callable patterns and return complete runners can be very powerful.

2.4 Testing

The examples above have shown how using Mush can make it easier to have smaller components that are easier to re-use and test, however care should still be taken with testing. In particular, it's a good idea to have some intergration tests that excercise the whole runner checking that it behaves as expected when all command line options are specified and when just the defaults are used.

When using the factory pattern, the factories themselves should be unit tested. It can also make tests easier to write by having a "testing runner" that sets up the required resources, such as database connections, while maybe doing some things differently such as not reading a configuration file from disk or using a `LogCapture` instead of file or stream log handlers.

2.5 Debugging

Mush does its best to call the things added to runners in the best order possible. If this doesn't match your expectations, passing the `debug` flag when the *`Runner`* is instantiated can give you more insight into what's going on. Please see *Debugging* for more information.

API Reference

class `mush.`**`Context`**

Stores requirements, callables and resources for a particular run.

`__iter__``()`

When iterated over, the context will yield tuples containing the requirements for a callable and the callable itself in the form `(requirements, object)`.

This can only be done once for a given context. A context that has been partially iterated over will remember where it had got to and pick up from there when iteration begins again.

`add``(`*it*, *type=None*`)`

Add a resource to the context.

Optionally specify the type to use for the object rather than the type of the object itself.

`get``(`*type*`)`

Get an object of the specified type from the context.

This will raise a `KeyError` if no object of that type can be located.

class `mush.`**`Marker`**

Type for Marker classes

class `mush.`**`Periods`**

A collection of lists used to store the callables that require a particular resource type.

`__iter__``()`

Yields callables in the order in which they require the resource this instance is used for.

`first` **= None**

The callables that require first use of a particular resource.

`last` **= None**

The callables that require last use of a particular resource.

`normal` **= None**

The callables that require use of a particular resource in the order to which they're added to the *`Runner`*.

class `mush.`**`Requirements`**`(`**args*, ***kw*`)`

Represents requirements for a particular callable

The passed in *args* and *kw* should map to the types, including any required *`when`* or *`how`*, for the matching arguments or keyword parameters the callable requires.

`__iter__``()`

When iterated over, yields tuples representing individual types required by arguments or keyword parameters in the form `(keyword_name, decorated_type)`.

If the keyword name is `None`, then the type is for a positional argument.

`returns`
An override for the type that this callable will return.

alias of `not_specified`

class `mush.`**`Runner`**(**objs*, ***debug*)
Used to run callables in the order in which they require particular resources and then, having taken that into account, in the order they are added to the runner.

Parameters

- **`objs`** – The callables to add to the runner as it is created.
- **`debug`** – If passed, debug information will be written whenever an object is added to the runner. If `True`, it will be written to `stderr`. A file-like object can also be passed, in which case the information will be written to that object.

`__add__`(*other*)
Concatenate two runners, returning a new runner.

The order of the new runner is as if the callables had been added in order from runner on the left-hand side of expression and then in order from the runner on the right-hand side of the expression.

`__call__`(*context=None*)
Execute the callables in this runner in the required order storing objects that are returned and providing them as arguments or keyword parameters when required.

A runner may be called multiple times. Each time a new *`Context`* will be created meaning that no required objects are kept between calls and all callables will be called each time.

Parameters **`context`** – Used for passing a context when context managers are used. You should never need to pass this parameter.

`add`(*obj*, **args*, ***kw*)
Add a callable to the runner.

If either `args` or `kw` are specified, they will be used to create the *`Requirements`* in this runner for the callable added in favour of any decoration done with *`requires`*.

`add_returning`(*obj*, *returns*, **args*, ***kw*)
Add a callable to the runner and specify that it should be treated as returning the type specified in `returns`, regardless of the actual type returned by calling `obj`.

If either `args` or `kw` are specified, they will be used to create the *`Requirements`* in this runner for the callable added in favour of any decoration done with *`requires`*.

`clone`()
Return a copy of this runner.

`extend`(**objs*)
Add the specified callables to this runner.

If any of the objects passed is a *`Runner`*, the contents of that runner will be added to this runner.

`replace`(*original*, *replacement*)
Replace all instances of one callable with another.

No changes in requirements or call ordering will be made.

`mush.`**`after`**(*type*)
A type wrapper that specifies the callable marked as requiring this type should not be passed an object of this type but should only be called once an object of that type is available, and should be done so in the `last` period.

class `mush.`**`attr`**(*type*, **names*)

A *`how`* that indicates the callable requires the named attribute from the decorated type.

class `mush.`**`first`**(*type=<class 'NoneType'>*)

A *`when`* that indicates the callable requires first use of the decorated type.

class `mush.`**`how`**(*type*, **names*)

The base class for type decorators that indicate which part of a resource is required by a particular callable.

Parameters

- **`type`** – The type to be decorated.
- **`name`** – The part of the type required by the callable.

class `mush.`**`ignore`**(*type*, **names*)

A *`how`* that indicates the callable should not be passed an object of the decorated type.

class `mush.`**`item`**(*type*, **names*)

A *`how`* that indicates the callable requires the named item from the decorated type.

class `mush.`**`last`**(*type=<class 'NoneType'>*)

A *`when`* that indicates the callable requires last use of the decorated type.

`mush.`**`marker`**(*name*)

Return a *`Marker`* for the given *name*, creating if needed.

`mush.`**`nothing`** **= Requirements()**

A singleton *`Requirements`* indicating that a callable requires no resources.

class `mush.`**`requires`**(**args*, ***kw*)

A decorator used for marking a callable with the *`Requirements`* it needs.

These are stored in an attribute called `__requires__` on the callable meaning that the callable can be used in its original form after decoration.

If you need to specify requirements for a callable that cannot have attributes added to it, then use the *`add()`* method to do so.

class `mush.`**`returns`**(*type*)

A decorator to indicate that a callable should be treated as returning the type passed to *`returns()`* rather than the type of the actual return value.

class `mush.`**`when`**(*type=<class 'NoneType'>*)

The base class for type decorators that indicate when a callable requires a particular type.

Parameters **`type`** – The type to be decorated.

For details of how to install the package or get involved in its development, please see the sections below:

CHAPTER 4

Installation Instructions

The easyiest way to install Mush is:

```
easy_install mush
```

Or, if you're using *zc.buildout*, just specify `mush` as a requirement.

Python version requirements

This package has been tested with Python 2.6, 2.7, 3.2 and 3.3 on Linux, Mac OS X and Windows.

CHAPTER 5

Development

This package is developed using continuous integration which can be found here:

https://travis-ci.org/Simplistix/mush

The latest development version of the documentation can be found here:

http://mush.readthedocs.org/en/latest/

If you wish to contribute to this project, then you should fork the repository found here:

https://github.com/Simplistix/mush/

Once that has been done and you have a checkout, you can follow these instructions to perform various development tasks:

5.1 Setting up a virtualenv

The recommended way to set up a development environment is to turn your checkout into a virtualenv and then install the package in editable form as follows:

```
$ virtualenv .
$ bin/pip install -U -e .[test,build]
```

5.2 Running the tests

Once you have a buildout, the tests can be run as follows:

```
$ bin/nosetests
```

5.3 Building the documentation

The Sphinx documentation is built by doing the following from the directory containg setup.py:

```
$ cd docs
$ make html
```

5.4 Making a release

To make a release, just update the version in `setup.py`, update the change log, tag it and push to https://github.com/Simplistix/mush and Travis CI should take care of the rest.

Once Travis CI is done, make sure to go to https://readthedocs.org/projects/mush/versions/ and make sure the new release is marked as an Active Version.

CHAPTER 6

Changes

6.1 1.3 (21 October 2015)

- Official support for Python 3.
- Drop official support for Windows, although things should still work.
- Move to Travis CI, Read The Docs and Coveralls for development.
- 'How' decorators like `attr()` and `item()` can now be nested as well as individually performing nested actions.
- Add `returns()` and `add_returning()` as new ways to override the type of a returned value.
- A better pattern for *marker types*.

6.2 1.2 (11 December 2013)

- Use `nothing` instead of `None` for marker return types, fixing a bug that occurred when a callable tried to type-map a result that was `None`.
- Add an `after()` type wrapper for callables that need to wait until after a resource is used but that can't accept that resource as a parameter.

6.3 1.1 (27 November 2013)

- Allow runners to be instantiated using other runners.
- Allow `Runner.extend()` to be passed `Runner` instances.
- Allow `requires()` decorations to be stacked.
- Add a `Runner.replace()` method to aid with testing assembled runners.

6.4 1.0 (29 October 2013)

- Initial Release

CHAPTER 7

License

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index