mush Documentation

Release 2.7.2

Simplistix Ltd

Sep 20, 2017

Contents

1 How Mush works 3
- 1.1 Constructing runners 3
- 1.2 Configuring Resources 5
- 1.3 Labels 11
- 1.4 Plugs 13
- 1.5 Context manager resources 16
- 1.6 Testing 17
- 1.7 Debugging 18

2 Example Usage 21
- 2.1 Encapsulating the re-usable parts of scripts 23
- 2.2 Writing the specific parts of your script 24
- 2.3 Assembling the components into a script 25
- 2.4 Testing 26

3 API Reference 27

4 Installation Instructions 31

5 Development 33
- 5.1 Setting up a virtualenv 33
- 5.2 Running the tests 33
- 5.3 Building the documentation 34
- 5.4 Making a release 34

6 Changes 35
- 6.1 2.7.2 (20 September 2017) 35
- 6.2 2.7.1 (7 September 2017) 35
- 6.3 2.7.0 (7 September 2017) 35
- 6.4 2.6.0 (6 February 2017) 35
- 6.5 2.5.0 (23 November 2016) 36
- 6.6 2.4.0 (17 November 2016) 36
- 6.7 2.3 (24 June 2016) 36
- 6.8 2.2 (2 January 2016) 36
- 6.9 2.1 (14 December 2015) 36
- 6.10 2.0 (11 December 2015) 36
- 6.11 1.3 (21 October 2015) 37

6.12 1.2 (11 December 2013) 37
6.13 1.1 (27 November 2013) 37
6.14 1.0 (29 October 2013) 37

7 License **39**

8 Indices and tables **41**

Python Module Index **43**

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.

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
```

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__
```

Specifying requirements

When callables take parameters, Mush can be configured to pass objects of the correct type that have been returned from previous callables in the runner. For example, consider the following functions:

```
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 specified by passing the required type in the *requires* parameter when adding the callable to the runner using `add()`. If more complex requirements need to be specified, a `requires` instance can be passed which accepts both positional and keyword parameters that specify the types required by the callable being added:

```
from mush import Runner, requires

runner = Runner()
runner.add(apple_tree)
runner.add(magician, requires=Apple)
runner.add(juicer, requires(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
```

Optional requirements

It may be that, while a callable needs an object of a particular type, a default can be used if no such object is present. Runners can be configured to take this into account. Take the following function:

```
def greet(name='stranger'):
    print('Hello ' + name + '!')
```

If a name is not always be available, it can be added to a runner as follows:

```
from mush import Runner, optional

runner = Runner()
runner.add(greet, requires=optional(str))
```

Now, when this runner is called, the default will be used:

```
>>> runner()
Hello stranger!
```

The same callable can be added to a runner where the required strings is available:

```
from mush import Runner, optional

def my_name_is():
   return 'Slim Shady'

runner = Runner(my_name_is)
runner.add(greet, requires=optional(str))
```

In this case, the string returned will be used:

```
>>> runner()
Hello Slim Shady!
```

Using parts of a resource

Resources can have attributes or items that are directly required by callables. For example, consider these two functions that return such resources:

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

def some_attributes():
    return Stuff()

def some_items():
    return dict(fruit='orange')
```

Also consider this function:

```
def pick(fruit1, fruit2, fruit3):
    print('I picked {0}, {1} and {2}'.format(fruit1, fruit2, fruit3))
```

All three can be added to a runner such that mush will pass the correct parts of the returns resources through to the `pick()` function:

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

runner = Runner(some_attributes, some_items)
runner.add(pick, requires(fruit1=attr(Stuff, 'fruit'),
                          fruit2=item(dict, 'fruit'),
                          fruit3=item(attr(Stuff, 'tree'), 'fruit')))
```

So now we can pick fruit from some interesting places:

```
>>> runner()
I picked apple, orange and pear
```

The `pick()` function, however, remains usable and testable on its own:

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

Specifying returned resources

As seen above, Mush will track resources returned by callables based on the type of any object returned. This is usually what you want, but in some cases you may want to specify something different. For example, if you have a callable that returns a sequence of resources, this would be added to a runner as follows:

```
from mush import Runner, returns_sequence

def all_fruit():
    print('I made fruit')
    return Apple(), Orange()


runner = Runner()
runner.add(all_fruit, returns=returns_sequence())
runner.add(juicer, requires(Apple, Orange))
```

Now, the juicer will use all the fruit returned:

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

In rarer circumstances, you may need to override the types returned by a callable. This can be done as follows:

```
from mush import Runner, returns

class Tomato:
    def __str__(self):
        return 'a tomato'

class Cucumber:
    def __str__(self):
        return 'a cucumber'

def vegetables():
    print('I made vegetables')
    return Tomato(), Cucumber()

runner = Runner()
runner.add(vegetables, returns=returns(Apple, Orange))
runner.add(juicer, requires(Apple, Orange))
```

Now, even when a callable requires fruit, we can force it to be happy with vegetables:

```
>>> runner()
I made vegetables
I made juice out of a tomato and a cucumber
```

The *returns* indicator can be used even if a single object is returned.

Another way that the type used to track the resource can be different from the type of the resource itself is if a callable returns a mapping and Mush is configured to use the types from that mapping:

```
from mush import Runner, returns_mapping

def desperation():
    print('I sold vegetables as fruit')
    return {Apple: Tomato(), Orange: Cucumber()}

runner = Runner()
runner.add(desperation, returns=returns_mapping())
runner.add(juicer, requires(Apple, Orange))
```

Once again, we can happily make juice out of vegetables:

```
>>> runner()
I sold vegetables as fruit
I made juice out of a tomato and a cucumber
```

Finally, if you have a callable that returns results that you wish to ignore, you can do so using *nothing*:

```
from mush import Runner, nothing

def spam():
    return 'spam'
```

```
runner = Runner()
runner.add(spam, returns=nothing)
```

Named resources

Sometimes the types of resources are too common for them to uniquely identify a resource:

```
def age():
    return 37

def meaning():
    return 42
```

A callable such as the following cannot be configured to require the correct resource from these two functions by type alone:

```
def profound(age, it):
    print('by the age of %s I realised the meaning of life was %s' % (
        age, it
    ))
```

For these situations, Mush supports the ability to name a resource using a string:

```
runner = Runner()
runner.add(age, returns='age')
runner.add(meaning, returns='meaning')
runner.add(profound, requires('age', it='meaning'))
```

Anywhere that a type can be used, a string name can be used instead:

```
>>> runner()
by the age of 37 I realised the meaning of life was 42
```

Using Type Annotations

While the imperative configuration used so far means that callables do not need to be modified, Python's type annotations can also be used to specify requirements and returned resources:

```
from mush import requires

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

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

def juicer(fruit1: Apple, fruit2: 'citrus'):
    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 from the type annotations 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
a refreshing fruit beverage
```

Declarative configuration

When type annotations are not available, either because they're being used for something else or because of the version of Python being used, the helpers for specifying requirements and return types can also be used as decorators:

```
from mush import requires

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

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

@requires(fruit1=Apple, fruit2='citrus')
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
a refreshing fruit beverage
```

Default configuration

If no declarations are made using either decorators or type annotations, then arguments that are needed by a callable will be looked up based on the name of the argument:

```
from mush import requires

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

def magician(apple) -> 'citrus':
    print('I turned {0} into an orange'.format(apple))
    return Orange()
```

```
def juicer(apple, citrus):
    print('I made juice out of {0} and {1}'.format(apple, citrus))
    return Juice()
```

These can now be combined into a runner and executed. The runner will guess the requirements base on the names of the arguments for each function 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
a refreshing fruit beverage
```

If an argument has a default, then the requirement will be made *optional*.

Configuration Precedence

The four styles of configuration are entirely interchangeable and you can use any combination that suites your requirements.

In terms of precedence, requirements and returned resource specifications will be used in the following order, with the first one found being the one that is used:

- Imperative configuration.
- Declarative configuration.
- Type annotations.
- Default configuration.

The default configuration for requirements is described *above*.

The default configuration for return values is that a callable's return value is used as a resource and will be registered against the type of the object returned. The *returns_result_type* declaration encapsulates this behaviour.

Labels

One of the motivating reasons for Mush to be created was the ability to insert callables at a point in a runner other than the end. This allows abstraction of common sequences of calls without the risks of extracting them into a base class.

The points at which more callables can be inserted are created by specifying a label when adding a callable to the runner. This marks the point at which that callable is included so that it can be retrieved and appended to later. 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 engrave(ring):
    print('engraving {0}'.format(ring))
```

These might be added to a runner as follows:

```
from mush import Runner

runner = Runner()
runner.add(forge)
runner.add_label('forged')
runner.add(engrave, requires=Ring)
runner.add_label('engraved')
```

Now, suppose we want to polish the ring before it's engraved and then package it up when we're done:

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

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

We can insert these callables into the runner at the right points as follows:

```
runner['forged'].add(polish, requires=Ring)
runner.add(package, requires=Ring)
```

This results in the desired call order:

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

Now, suppose we want to polish the ring again after it's been engraved. We can insert another callable at the appropriate point:

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

runner['engraved'].add(more_polish, requires=Ring)
```

Mush will do the right thing when this runner is called:

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

When using labels, it's often good to be able to see exactly what is in a runner, what order it is in and where any labels point. For this reason, the representation of a runner gives all this information:

```
>>> runner
<Runner>
    <function forge ...> requires() returns_result_type()
    <function polish ...> requires(Ring) returns_result_type() <-- forged
    <function engrave ...> requires(Ring) returns_result_type()
    <function more_polish ...> requires(Ring) returns_result_type() <-- engraved
    <function package at ...> requires(Ring) returns_result_type()
</Runner>
```

As you can see above, when a callable is inserted at a label, the label moves to that callable. You may wish to keep track of the initial point that was labelled, so Mush supports multiple labels at each point:

```
>>> runner = Runner()
>>> point = runner.add(forge)
>>> point.add_label('before_polish')
>>> point.add_label('after_polish')
>>> runner
<Runner>
    <function forge ...> requires() returns_result_type() <-- after_polish, before_
↪polish
</Runner>
```

Now, when you add to a specific label, only that label is moved:

```
>>> point = runner['after_polish']
>>> point.add(polish)
>>> runner
<Runner>
    <function forge ...> requires() returns_result_type() <-- before_polish
    <function polish ...> requires('ring') returns_result_type() <-- after_polish
</Runner>
```

Of course, you can still add to the end of the runner:

```
>>> runner.add(package)
<mush.modifier.Modifier...>
>>> runner
<Runner>
    <function forge ...> requires() returns_result_type() <-- before_polish
    <function polish ...> requires('ring') returns_result_type() <-- after_polish
    <function package ...> requires('ring') returns_result_type()
</Runner>
```

However, the point modifier returned by getting a label from a runner will keep on moving the label as more callables are added using it:

```
>>> point.add(more_polish)
>>> runner
<Runner>
    <function forge ...> requires() returns_result_type() <-- before_polish
    <function polish ...> requires('ring') returns_result_type()
    <function more_polish ...> requires('ring') returns_result_type() <-- after_polish
    <function package ...> requires('ring') returns_result_type()
</Runner>
```

Plugs

You may run into situations where you wish to group callables together and add them in one go. Indeed, it might only make sense to add all callables in a group and would cause problems to add them individually.

For example, suppose we have this runner:

```
def prepare():
    print('cleaning kitchen table')
```

```
def wash(produce):
    print('washing '+str(produce))

def finished():
    print('service please!')

kitchen_runner = Runner()
kitchen_runner.add(prepare)
kitchen_runner.add_label('what')
kitchen_runner.add(wash, requires='produce')
kitchen_runner.add_label('how')
kitchen_runner.add(finished)
```

For any use of the kitchen, we need to specify both what to use and how we want to use it once it’s washed. This can neatly be done as follows:

```
from mush import Plug

class JuicePlug(Plug):

    def what(self) -> 'produce':
        return Apple()

    def how(self, produce):
        print('juicing '+str(produce))

runner = kitchen_runner.clone()
JuicePlug().add_to(runner)
```

The runner behaves as we require:

```
>>> runner()
cleaning kitchen table
washing an apple
juicing an apple
service please!
```

It may be that we want our plug to have helper methods, in which case they can either be named with a leading underscore, or the plug can be set up to only add explicitly marked methods, for example:

```
from mush.plug import Plug, insert

class JuicePlug(Plug):

    explicit = True

    def juice(self, produce):
        print('juicing '+str(produce))

    @insert()
    def what(self) -> 'produce':
        return Apple()

    @insert()
    def how(self, produce: 'produce'):
        self.juice(produce)

runner = kitchen_runner.clone()
```

```
JuicePlug().add_to(runner)
```

The runner behaves as before:

```
>>> runner()
cleaning kitchen table
washing an apple
juicing an apple
service please!
```

As you can see, this might make for a lot of decorating if you only have one helper method. If that's the case, you can just tell the plug to ignore the helper:

```
from mush.plug import Plug, ignore

class JuicePlug(Plug):

    @ignore()
    def juice(self, produce):
        print('juicing '+str(produce))

    def what(self) -> 'produce':
        return Apple()

    def how(self, produce: 'produce'):
        self.juice(produce)

runner = kitchen_runner.clone()
JuicePlug().add_to(runner)
```

The runner still behaves as before:

```
>>> runner()
cleaning kitchen table
washing an apple
juicing an apple
service please!
```

It may be that it makes sense to give your method a name different to the label you wish to add it at. You may also wish to have a plug add a method to the end of the runner where there is no label. Both of these are supported:

```
from mush.plug import Plug, insert, append

class JuicePlug(Plug):

    @insert(label='what')
    def pick_fruit(self) -> 'produce':
        return Apple()

    def how(self, produce: 'produce'):
        print('juicing '+str(produce))

    @append()
    def relax(self):
        print('...and relax')

runner = kitchen_runner.clone()
JuicePlug().add_to(runner)
```

The runner now behaves as required:

```
>>> runner()
cleaning kitchen table
washing an apple
juicing an apple
service please!
...and relax
```

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
```

Testing

Mush has a couple of features to help with automated testing of runners. For example, if you wanted to test a runner that got configuration by calling a remote web service:

```
def load_config() -> 'config':
    return json.loads(urllib2.urlopen('...').read())

def do_stuff(username: item('config', 'username'),
             password: item('config', 'password')):
    print('doing stuff as ' + username + ' with '+ password)

runner = Runner(load_config, do_stuff)
```

When testing this runner, we may want to inject a hard-coded config. This can be done by cloning the original runner and replacing the `load_config()` callable:

```
>>> def test_config():
...     return dict(username='test', password='pw')
>>> test_runner = runner.clone()
>>> test_runner.replace(load_config, test_config)
>>> test_runner()
doing stuff as test with pw
```

If you have a base runner such as this:

```
from argparse import ArgumentParser, Namespace

def base_args(parser):
    parser.add_argument('config_url')

def parse_args(parser):
    return parser.parse_args()

def load_config():
    return json.loads(urllib2.urlopen('...').read())

def finalise_things():
    print('all done')

base_runner = Runner(ArgumentParser)
base_runner.add(base_args, requires=ArgumentParser, label='args')
base_runner.add(parse_args, requires=ArgumentParser)
point = base_runner.add(load_config, requires=attr(Namespace, 'config_url'),
                        returns='config')
point.add_label('body')
base_runner.add(finalise_things, label='ending')
```

That runner might be used for a specific script as follows:

```
def job_args(parser: ArgumentParser):
    parser.add_argument('--colour')
```

```
def do_stuff(username: item('config', 'username'),
             colour: attr(Namespace, 'colour')):
    print(username + ' is '+ colour)

runner = base_runner.clone()
runner['args'].add(job_args)
runner['body'].add(do_stuff)
```

To test this runner, we want to use a dummy configuration and command line and not have any finalisation take place. This can be achieved with a helper function such as the following:

```
def run_with(source_runner, config, argv):
    runner = Runner(ArgumentParser)
    runner.extend(source_runner.clone(added_using='args'))
    runner.add(lambda parser: parser.parse_args(argv),
               requires=ArgumentParser)
    runner.add(lambda: config, returns='config')
    runner.extend(source_runner.clone(added_using='body'))
    runner()
```

The helper can then be used as follows:

```
>>> run_with(runner,
...          config=dict(username='test', password='pw'),
...          argv=['--colour', 'red'])
test is red
```

Debugging

Mush has a couple of features to aid debugging of runners. The first of which is that the representation of a runner will show everything in it, in the order it will be called and what each callable has been declared as requiring and returning along with where any labels currently point.

For example, consider this runner:

```
from mush import Runner

def make_config() -> 'config':
    return {'foo': 'bar'}

def connect(foo: item('config', 'foo')):
    return 'connection'

def process(connection):
    print('using ' + repr(connection))

runner = Runner()
point = runner.add(make_config, label='config')
point.add(connect)
runner.add(process)
```

To see how the configuration panned out, we would look at the `repr()`:

```
>>> runner
<Runner>
    <function make_config ...> requires() returns('config')
    <function connect ...> requires(foo='config'['foo']) returns_result_type() <--␣
↪config
    <function process ...> requires('connection') returns_result_type()
</Runner>
```

As you can see, there is a problem with this configuration that will be exposed when it is run. To help make sense of these kinds of problems, Mush will add more context when a `TypeError` or *`ContextError`* is raised.

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:

```
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)
    log.setLevel(logging.DEBUG)

    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.

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, attr, item
import logging, os, sqlite3, sys

log = logging.getLogger()

def base_options(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')

def parse_args(parser: ArgumentParser):
    return parser.parse_args()

def parse_config(args: Namespace) -> 'config':
    config = RawConfigParser()
    config.read(args.config)
    return dict(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`. The `parse_config()` function reads configuration from a file specified as a command line argument and so requires the `Namespace`. Since the config is a `dict`, we configure it as a named rather than typed resource. See *Named resources* for more details.

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:

```
class DatabaseHandlerTests(TestCase):

    def setUp(self):
        self.dir = TempDirectory()
        self.addCleanup(self.dir.cleanup)
        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()
        self.addCleanup(self.log.uninstall)

    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'))
```

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 writing just 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.

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.

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, attr, item
base_runner = Runner(ArgumentParser)
base_runner.add(base_options, label='args')
base_runner.extend(parse_args, parse_config)
base_runner.add(setup_logging, requires(
    log_path = item('config', 'log'),
    quiet = attr(Namespace, 'quiet'),
    verbose = attr(Namespace, 'verbose')
))
```

The code above shows how to label a point, *args* in this case, enabling callables to be inserted at that point at a later time. See *Labels* for full details. It also 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['args'].add(args, requires=ArgumentParser)
main.add(DatabaseHandler, requires=item('config', 'db'))
main.add(do,
         requires(attr(DatabaseHandler, 'conn'), attr(Namespace, 'path')))

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

Using a factory

This pattern is most useful when you have lots of 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, requires=ArgumentParser)
    runner.add(parse_args, requires=ArgumentParser)
    runner.add(parse_config, requires=Namespace)
    runner.add(setup_logging, requires(
        log_path = item('config', 'log'),
        quiet = attr(Namespace, 'quiet'),
        verbose = attr(Namespace, 'verbose')
    ))
    runner.add(DatabaseHandler, requires=item('config', 'db'))
    runner.add(
        do,
        requires(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 that are clones of a parent runner and having factories that take common callable patterns and return complete runners can be very powerful.

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 integration tests that exercise 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.

Some specific tools that Mush provides to aid automated testing are covered in *Testing*.

CHAPTER 3

API Reference

class `mush.`**`Runner`**(**objects*)

A chain of callables along with declarations of their required and returned resources along with tools to manage the order in which they will be called.

`__add__`(*other*)

Return a new `Runner` containing the contents of the two `Runner` instances being added together.

`__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.

`__getitem__`(*label*)

Retrieve a `Modifier` for a previous labelled point in the runner.

`add`(*obj*, *requires=None*, *returns=None*, *label=None*)

Add a callable to the runner.

Parameters

- **`obj`** – The callable to be added.
- **`requires`** – The resources to required as parameters when calling *obj*. These can be specified by passing a single type, a string name or a `requires` object.
- **`returns`** – The resources that *obj* will return. These can be specified as a single type, a string name or a `returns`, `returns_mapping`, `returns_sequence` object.
- **`label`** – If specified, this is a string that adds a label to the point where *obj* is added that can later be retrieved with `Runner.__getitem__()`.

`add_label`(*label*)

Add a label to the the point currently at the end of the runner.

clone(*start_label=None*, *end_label=None*, *include_start=False*, *include_end=False*, *added_using=None*)

Return a copy of this `Runner`.

Parameters

- **start_label** – An optional string specifying the point at which to start cloning.
- **end_label** – An optional string specifying the point at which to stop cloning.
- **include_start** – If `True`, the point specified in `start_label` will be included in the cloned runner.
- **include_end** – If `True`, the point specified in `end_label` will be included in the cloned runner.
- **added_using** – An optional string specifying that only points added using the label specified in this option should be cloned. This filtering is applied in addition to the above options.

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*, *requires=None*, *returns=None*)

Replace all instances of one callable with another.

No changes in requirements or call ordering will be made unless the replacements has been decorated with and requirements, or either `requires` or `returns` have been specified.

Parameters

- **requires** – The resources to required as parameters when calling *obj*. These can be specified by passing a single type, a string name or a `requires` object.
- **returns** – The resources that *obj* will return. These can be specified as a single type, a string name or a `returns`, `returns_mapping`, `returns_sequence` object.

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

Represents requirements for a particular callable.

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

String names for resources must be used instead of types where the callable returning those resources is configured to return the named resource.

__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.

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

A `how` that indicates the callable requires the wrapped requirement only if it's present in the `Context`.

class `mush.`**returns_result_type**

Default declaration that indicates a callable's return value should be used as a resource based on the type of the object returned.

`None` is ignored as a return value.

class `mush.`**returns_mapping**

Declaration that indicates a callable returns a mapping of type or name to resource.

class `mush.`**`returns_sequence`**

Declaration that indicates a callable's returns a sequence of values that should be used as a resources based on the type of the object returned.

Any `None` values in the sequence are ignored.

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

Declaration that specified names for returned resources or overrides the type of a returned resource.

This declaration can be used to indicate the type or name of a single returned resource or, if multiple arguments are passed, that the callable will return a sequence of values where each one should be named or have its type overridden.

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

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

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

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

class `mush.`**`Plug`**

Base class for a 'plug' that can add to several points in a runner.

`add_to`(*runner*)

Add methods of the instance to the supplied runner. By default, all methods will be added and the name of the method will be used as the label in the runner at which the method will be added. If no such label exists, a `KeyError` will be raised.

If `explicit` is `True`, then only methods decorated with an *`insert`* will be added.

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

Stores resources for a particular run.

`add`(*it*, *type*)

Add a resource to the context.

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

exception `mush.context.`**`ContextError`**(*text*, *point=None*, *context=None*)

Errors likely caused by incorrect building of a runner.

class `mush.modifier.`**`Modifier`**(*runner*, *callpoint*, *label*)

Used to make changes at a particular point in a runner. These are returned by *`Runner.add()`* and *`Runner.__getitem__()`*.

`add`(*obj*, *requires=None*, *returns=None*, *label=None*)

Parameters

- **`obj`** – The callable to be added.
- **`requires`** – The resources to required as parameters when calling *obj*. These can be specified by passing a single type, a string name or a *`requires`* object.
- **`returns`** – The resources that *obj* will return. These can be specified as a single type, a string name or a *`returns`*, *`returns_mapping`*, *`returns_sequence`* object.
- **`label`** – If specified, this is a string that adds a label to the point where *obj* is added that can later be retrieved with *`Runner.__getitem__()`*.

If no label is specified but the point which this *`Modifier`* represents has any labels, those labels will be moved to the newly inserted point.

`add_label`(*label*, *callpoint=None*)

Add a label to the point represented by this *`Modifier`*.

Parameters **`callpoint`** – For internal use only.

class `mush.declarations.`**`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 resource type to be decorated.
- **`names`** – Used to identify the part of the resource to extract.

`process`(*o*)
Extract the required part of the object passed in. `missing` should be returned if the required part cannot be extracted. `missing` may be passed in and is usually be handled by returning `missing` immediately.

`mush.declarations.`**`nothing`** **= requires()**
A singleton that be used as a *`requires`* to indicate that a callable has no required arguments or as a *`returns`* to indicate that anything returned from a callable should be ignored.

`mush.declarations.`**`result_type`** **= returns_result_type()**
A singleton indicating that a callable's return value should be stored based on the type of that return value.

`mush.declarations.`**`update_wrapper`**(*wrapper*, *wrapped*, *assigned=('__module__', '__name__', '__qualname__', '__doc__', '__annotations__', '__mush__requires__', '__mush_returns__')*, *updated=('__dict__',)*)
An extended version of `functools.update_wrapper()` that also preserves Mush's annotations.

class `mush.plug.`**`insert`**(*label=None*)
A decorator to explicitly mark that a method of a *`Plug`* should be added to a runner by *`add_to()`*. The *label* parameter can be used to indicate a different label at which to add the method, instead of using the name of the method.

class `mush.plug.`**`ignore`**
A decorator to explicitly mark that a method of a *`Plug`* should not be added to a runner by *`add_to()`*

class `mush.plug.`**`append`**
A decorator to mark that this method of a *`Plug`* should be added to the end of a runner by *`add_to()`*.

class `mush.plug.`**`Plug`**
Base class for a 'plug' that can add to several points in a runner.

`add_to`(*runner*)
Add methods of the instance to the supplied runner. By default, all methods will be added and the name of the method will be used as the label in the runner at which the method will be added. If no such label exists, a `KeyError` will be raised.

If *`explicit`* is `True`, then only methods decorated with an *`insert`* will be added.

`explicit` **= False**
Control whether methods need to be decorated with *`insert`* in order to be added by this *`Plug`*.

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

CHAPTER 4

Installation Instructions

The easiest way to install Mush is:

```
pip install mush
```

Python version requirements

This package has been tested with Python 2.7, 3.3+ on Linux, but is also expected to work on 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:

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]
```

Running the tests

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

```
$ bin/pytest
```

Building the documentation

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

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

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

2.7.2 (20 September 2017)

- Fix bugs relating to *default configuration* of `functools.partial()`.

2.7.1 (7 September 2017)

- Use the original's requirements when the replacement passed to `Runner.replace()` has no specified requirements of its own.

2.7.0 (7 September 2017)

- Move to pytest and sybil for testing.
- Drop support for Python 3.3.
- Add `update_wrapper()` helper.
- Add support for using Python 3 *type annotations* to specify requirements and returned resources.
- Add support for *arg names* being used as requirements when there is no other configuration.
- Add an explicit way of `ignoring` the return value of a callable.

2.6.0 (6 February 2017)

- Allow replacement of a callable to also supply new requirements.
- Officially support Python 3.6.

2.5.0 (23 November 2016)

- Allow `Plug` instances to be added directly using `Runner.add()` and friends.

2.4.0 (17 November 2016)

- Add support for cloning depending on what label was used to add callables.
- Add `Runner.add_label()` helper to just add a label at the end of the runner.
- Document and flesh out *Plugs*.
- Switch to full Semantic Versioning.

2.3 (24 June 2016)

- Stop catching `TypeError` and turning it into a `ContextError` when calling a `Runner`. This turns out to be massively unhelpful, especially when using Python 2.

2.2 (2 January 2016)

- Add `Plug` base class.

2.1 (14 December 2015)

- Typo fixes in documentation.
- Indicate that Python 2.6 is no longer supported.
- Raise exceptions when arguments to `requires()` and `returns()` are not either types or labels.
- Allow tuples are lists to be passed to `add()`, they will automatically be turned into a `requires()` or `returns()`.
- Better error messages when a requirement is not found in the `Context`.

Thanks to Dani Fortunov for the documentation review.

2.0 (11 December 2015)

- Re-write dropping all the heuristic callable ordering in favour of building up defined sequences of callables with labelled insertion points.

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".

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.

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.

1.0 (29 October 2013)

- Initial Release

CHAPTER 7

License

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

mush, 27
mush.context, 29
mush.declarations, 30
mush.modifier, 29
mush.plug, 30

Index

Symbols

__add__() (mush.Runner method), 27
__call__() (mush.Runner method), 27
__getitem__() (mush.Runner method), 27
__iter__() (mush.requires method), 28

A

add() (mush.context.Context method), 29
add() (mush.modifier.Modifier method), 29
add() (mush.Runner method), 27
add_label() (mush.modifier.Modifier method), 29
add_label() (mush.Runner method), 27
add_to() (mush.Plug method), 29
add_to() (mush.plug.Plug method), 30
append (class in mush.plug), 30
attr (class in mush), 29

C

clone() (mush.Runner method), 27
Context (class in mush.context), 29
ContextError, 29

E

explicit (mush.plug.Plug attribute), 30
extend() (mush.Runner method), 28

H

how (class in mush.declarations), 30

I

ignore (class in mush.plug), 30
insert (class in mush.plug), 30
item (class in mush), 29

M

Modifier (class in mush.modifier), 29
mush (module), 27
mush.context (module), 29
mush.declarations (module), 30
mush.modifier (module), 29
mush.plug (module), 30

N

nothing (in module mush.declarations), 30

O

optional (class in mush), 28

P

Plug (class in mush), 29
Plug (class in mush.plug), 30
process() (mush.declarations.how method), 30

R

replace() (mush.Runner method), 28
requires (class in mush), 28
result_type (in module mush.declarations), 30
returns (class in mush), 29
returns_mapping (class in mush), 28
returns_result_type (class in mush), 28
returns_sequence (class in mush), 28
Runner (class in mush), 27

U

update_wrapper() (in module mush.declarations), 30