phyles Documentation

Release 0.2.0b4

James C. Stroud

February 20, 2013

CONTENTS

Phyles is a set of somewhat eclectic functions that makes the implementation of utilities (little programs that can be controlled by config files) easier. It started as a mass of boilerplate that I would copy into almost every utility I wrote. I finally decided to consolidate this code into a package and add some schema-based validation of config files and to document it fully.

Phyles provides support for YAML-based config files as well as a means for validating the config files. Phyles also provides several facilities for making utilities more user friendly, including automatically generated banners, automatically documented configuration templates, and graceful recovery from configuration errors.

The accompanying tutorial shows how phyles assists in turning one-off python scripts into robust packages worthy of distribution, or at least worthy of a permanent place in one's work-flow.

Phyles can be obtained at https://pypi.python.org/pypi/phyles/.

CHAPTER

ONE

INTRODUCTION TO PHYLES

About 90% of the convenience that phyles offers can be summarized by a few lines of code. From the example utility in the tutorial:

```
1  spec = phyles.package_spec(phyles.Undefined, "barbecue",
2                             "schema", "barbecue-time.yml")
3  converters = {'celsius to farenheit':
4                barbecue.celsius_to_farenheit}
5  setup = phyles.set_up(__program__, __version__,
6                                    spec, converters)
7  phyles.run_main(main, setup['config'],
8                  catchall=barbecue.BarbecueError)
```

These few lines find a schema specification from the package contents (line 1), parses command line arguments (line 5), validates a config file (lines 3 & 5), overrides configuration settings therein (line 5), and runs the main function of the utility in a try-except block that ensures graceful exit in the event that an anticipated exception is raised (line 5).

Schema are specified in YAML, terse, and hopefully intuitive. Following is the example from the tutorial:

```
!!omap
- dish :
   - - vegetable kabobs
     - smoked salmon
     - brisket
   - smoked salmon
   - Dish to cook
- doneness :
   - rare : 200
     medium : 350
     well-done : 500
   - medium
   - How much to cook the dish
- temperature :
   - celsius to farenheit
   - 105
   - Cooking temperature in °C
   - 105
- width :
   - int
   - 70
   - width of report
   - 70
```

Phyles will automatically generate a documented sample config files for users if they run the utility with the `--template` (or `-t`) command line option.

As one final example, a valid config file for this schema is:

```
dish : smoked salmon
doneness : medium
temperature : 107
width : 70
```

CHAPTER

TWO

TUTORIAL

For the sake of a tutorial, lets assume that we need to calculate cooking times for our barbecue. For this daunting task, we decide we want a utility: a program that we can run from the command line whenever we get the urge for a taste of smokey goodness.

2.1 The Calculation

Note: This section isn't as much about using phyles as it is about framing the purpose of the example utility quantitatively (*i.e.* into numbers) and in a way that can be formulated as a python function. Hopefully the example is not too complicated. However, I have tried to make it just complicated enough to warrant a full-fledged utility.

Let's assume that we have three dishes we usually barbecue:

Dish	Difficulty
vegetable kabobs	1 dc
smoked salmon	2 dc
brisket	3 dc

The numbers to the right of each dish gives the **difficulty** of cooking (which we'll abbreviate as "dc" to express its units).

As an example of difficulty, cooking a typical batch of vegetable kabobs for 1 hour at some some temperature (say 200 °F) is equivalent to cooking a typical brisket for 3 hours at that temperature. Or stated another way, brisket cooks about three times as slow as vegetable kabobs.

For the sake of this tutorial, cooking difficulty applies to temperature as well. For instance, cooking a typical batch of vegetable kabobs at 200 °F for 2 hours is equivalent to cooking a typical brisket at 600 °F for 2 hours:

$$\frac{200\,^\circ\mathrm{F} \times 2\,\mathrm{hr}}{1\,\mathrm{dc}} = \frac{600\,^\circ\mathrm{F} \times 2\,\mathrm{hr}}{3\,\mathrm{dc}}$$

Note how we divided both sides of the equation by the difficulty of cooking for the respective dish (1 dc for vegetable kabobs; 3 dc for brisket). This calculation shows that we can quantify how much we cook something by calculating what we'll call the "doneness". Taking this example for brisket:

$$400\,\mathrm{doneness} = \frac{600\,^\circ\mathrm{F} \times 2\,\mathrm{hr}}{3\,\mathrm{dc}}$$

Or, as a mathematical formula:

$$\mathrm{doneness} = \frac{\mathrm{temperature} \times \mathrm{time}}{\mathrm{difficulty}}$$

Using algebra [1], we can rearrange this equation to calculate cooking times:

$$\text{time} = \frac{\text{doneness} \times \text{difficulty}}{\text{temperature}}$$

In other words, if we know the amount we need to cook a dish (doneness), how difficult the dish is to cook (difficulty), and the temperature that we can achieve with our grill, then we can calculate the cooking time.

So, how much do we want to cook an dish? This table quantifies doneness for several common cooking terms:

Term	Doneness
Rare	200
Medium	350
Well-Done	500

Let's try a calculation for smoked salmon (difficulty of 2) cooked medium (doneness of 350) at 225 °F (which is about 107 °C):

$$\text{time} = \frac{350\,\text{doneness} \times 2\,\text{dc}}{225\,^\circ\text{F}} \approx 3.11\,\text{hr}$$

Thus it takes about 3.11 hr to cook a smoked salmon to medium at 225 °F.

As a python function, this calculation might take the form:

```
def cooking_time(doneness, difficulty, temperature):
  """
  Return then cooking time given the desired 'doneness'
  cooking, the 'difficulty' of cooking,
  and the 'temperature'.

   Args:
     - 'doneness': desired doneness (hr•°F/dc)
     - 'difficulty': difficulty of cooking for the dish (dc)
     - 'temperature': cooking temperature (°F)

   Returns: cooking time in hours (''float'')

   Raises: ''ValueError'' if the 'temperature' is <= 0 °F

   >>> round(cooking_time(350, 2, 225), 2)
   3.11
   """
   if T <= 120:
     msg = "%s °F is too cold to cook!" % T
     raise ValueError(msg)
   return float(doneness * difficulty) / T
```

2.2 The Config Format

Assuming that we have a utility that calulates cooking times based on a config file, the file for this example might take the following form:

```
dish : smoked salmon
doneness : medium
temperature : 225
```

[1] The rule of algebra used here can be stated like this: if a quantity is on top of the fraction on one side of the equals sign, then it can be moved to the bottom of the fraction on the other side of the equals sign, and vice versa.

This config format is convenient for a user who doesn't care that smoked salmon has a difficulty of 1 dc or that medium corresponds to a doneness of 350. However, it places a burden on the programmer to read the file, ensure that "smoked salmon" and "medium" are spelled correctly, and convert these string values into numbers.

That's where phyles comes in!

2.3 Conversion

2.3.1 Dictionary-Based Conversion

We'll tackle these tasks in steps, first finding a way to convert specific strings into numbers. Python provides a convenient way to do this conversion using its `dict` class:

```
doneness_dict = {'rare': 200,
                 'medium': 350,
                 'well-done': 500}
```

Getting a value from a dictionary using a key is called "item-getting". Python item-getting raises a `KeyError` when it fails:

```
>>> doneness_dict['raw']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'raw'
```

As we'll see, phyles allows for the creation of a converter dictionary directly in the schema specification.

2.3.2 Type-Based Conversion

When a YAML config file is parsed by a YAML parser, literals like `225` evaluate to integers. However, a cooking temperature may often be more useful as a `float`, as when it serves in the denominator of a fraction, for example. In cases where a YAML literal evaluates to a python type (*e.g.* `int`, `float`, `str`) that is different from the type desired, the desired python type can be used to to perform the conversion:

```
>>> float(225)
225.0
```

Like the `dict` item-getting, python types provide error checking, raising exceptions upon failure:

```
>>> float([2, 2, 5])
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: float() argument must be a string or a number

>>> float("twotwentyfive")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: could not convert string to float: twotwentyfive
```

2.3.3 Conversion by User-Defined Functions

There is nothing particularly special about `dict.__getitem__()` or python types. They are simply functions (or more precisely, "callables") that take a single value as a parameter and return possibly different values. In cases where they fail, `dict.__getitem__()` and python types raise three kinds of exceptions:

Exception	Raised By
KeyError	:class‘dict‘ item-getting
TypeError	python types
ValueError	python types

Thus, any python function that takes one and only one parameter and raises either a `KeyError`, `TypeError`, or `ValueError` upon failure, can serve as a converter.

For example, say we want to release a European version of our barbecue utility, we could write a function to convert temperature in °C into temperature in °F:

```
1  def celsius_to_farenheit(c):
2    """
3    Returns the temperature in Farenheit given temperature
4    in Celsius.
5
6    Args:
7      - c: temperature in Celsius
8
9    Returns: temperature in Farenheit (float)
10
11   >>> celsius_to_farenheit(107.222)
12   224.9996
13   """
14   c = float(c)
15   if c < -273.15:
16     raise ValueError("Impossibly cold (%s °C)!" % c)
17   else:
18     return (1.8 * c) + 32
```

Notice that on line 16, `clesius_to_farenheit()` raises a `ValueError` if the temperature supplied to the function is lower than the thermodynamic legal limit of -273.15 °C.

2.3.4 Choices

In some cases, no conversion is required but it is desirable to check an option value against a list of choices. As shown below, phyles allows the creation of lists of choices within the schema specification. If choices are given in this way, phyles creates a sensible error message if the value for the option is not within the list of choices.

2.4 The Schema Specification

A schema in phyles (encapsulated by the `phyles.Schema` class), contains information to validate a configuration as well as produce a sample configuration, complete with documentation in comments. A schema is specified by a “schema specification”.

The schema specification (often shortened to “spec”) can take several forms, as fully explained in the documentation to the `phyles.load_schema()` function.

2.4.1 Example Schema Specification

For our barbecue example, we’ll use a schema specification written as a YAML omap:

```
1  !!omap
2  - dish :
3      - - vegetable kabobs
```

```
4         - smoked salmon
5         - brisket
6       - smoked salmon
7       - Dish to cook
8   - doneness :
9       - rare : 200
10        medium : 350
11        well-done : 500
12      - medium
13      - How much to cook the dish
14  - temperature :
15      - celsius to farenheit
16      - 105
17      - Cooking temperature in °C
18      - 105
```

2.4.2 Required Elements of a Specification

The sequence value for each parameter (*e.g.* `dish`, `doneness`, and `temperature`) in the schema specification has three required elements (and a fourth optional element, described below in the section titled Optional Default Values):

1. **converter as either**
   (a) a YAML string object of the name of the converter function as keyed in the *converters* argument to `phyles.Schema.load_schema()` (as with `temperature` above, and discussed in the section titled Dictionary of Converter Functions)
   (b) a YAML sequence object with a list of acceptable choices (as with `dish` above)
   (c) or a YAML mapping object that maps choices to converted values (as with `doneness` above)
2. an example value (for sample config files)
3. documentation (which can be set to `null` for no documentation; see YAML null)

For `temperature`, these elements are

1. converter: `celsius to farenheit`
2. example: `105`
3. documentation: `Cooking temperature in °C`

2.4.3 Dictionary of Converter Functions

One question is how a *name* that evaluates to a python `str` (*e.g.* `doneness`) translates into a converter, which must be a function. As explained more thoroughly in the discussion of schema below, this translation is achieved using a `dict`. For this barbecue example, we construct this `dict` in the following way:

```
converters = {'celsius to farenheit': celsius_to_farenheit}
```

We'll see exactly how to use the `converters` `dict` in the example utility.

Note: The `celsius_to_farenheit()` function is defined in the section titled Conversion by User-Defined Functions

---

2.4.4 Phyles Standard Converters

There are several python types for which it is not necessary to add entries to the `converters` `dict`. The reason is that phyles provides a set of built-in converters. For example, if a `float` converter were needed, then the following would be implicit and **not required** from the programmer:

```
converters = {'float': float}  #  <-- NOT necessary!!
```

These built-in converters provided by phyles are:

- **map**: `dict`
- **dict**: `dict` (encoded as a YAML dict)
- **omap**: `collections.OrderedDict`
- **odict**: `collections.OrderedDict` (alias for "omap")
- **pairs**: `list` of 2-`tuples`
- **set**: `set`
- **seq**: `list`
- **list**: `list` (encoded as a sequence, see `list()`)
- **tuple**: `tuple` (encoded as a sequence, see `tuple()`)
- **bool**: `bool`
- **float**: `float`
- **int**: `int`
- **long**: `long` (encoded as a YAML int)
- **complex**: `complex` (encoded as a sequence of 0 to 2, or as a string representation, e.g. `'3+2j'`; see `complex()`)
- **str**: `str`
- **unicode**: `unicode`
- **timestamp**: `datetime.datetime` (encoded as a YAML timestamp)
- **slice**: `slice` (encoded as a sequence of 1 to 3, see `slice()`)

Note: Except where indicated, these types are encoded according to the YAML types specification in a YAML representation of a config.

2.4.5 Optional Default Values

Additional to the three required elements of a specification parameter, an optional default value may be specified as a fourth element. In the example schema specification the default for the `temperature` parameter is `105`. If a default value is missing, as in `dish` and `doneness`, then the parameter is required in the config file.

For example, the following config will fail vailidation by a schema from the example schema specification because the specification requires a value for `doneness` (by virtue of the specification's missing a default value for `doneness`):

```
dish : smoked salmon
temperature : 107
```

2.5 The Schema

2.5.1 Loading Schema

To validate a config file, the information in the schema specification must be converted into a functional schema, a conversion accomplished by the `phyles.load_schema()` function.

2.5.2 Validating Configs

Although the `phyles.set_up()` function automates these steps, it is useful to see how a schema is constructed from a specification and further how the schema validates a config. Using the running example (*i.e.* with `converters` defined in the section titled Dictionary of Converter Functions):

```
import phyles
import yaml
spec = """
       !!omap
       - dish :
           - - vegetable kabobs
             - smoked salmon
             - brisket
           - smoked salmon
           - Dish to cook
       - doneness :
           - rare : 200
             medium : 350
             well-done : 500
           - medium
           - How much to cook the dish
       - temperature :
           - celsius to farenheit
           - 105
           - Cooking temperature in °C
           - 105
       """

cfg = yaml.load("""
                dish : smoked salmon
                doneness : medium
                temperature : 107
                """)

schema = phyles.load_schema(spec, converters)
config = schema.validate_config(cfg)
```

The behavior of the resulting `config`, which is an instance of `phyles.Configuration`, will be discussed in more detail in the section titled The Configuration.

Note: The `cfg` could have just as easily been created directly as a `dict`:

```
cfg = {"dish": "smoked salmon",
       "doneness": "medium",
       "temperature": 107}
```

However, YAML is used here for consistency with earlier parts of this example and to emphasize the point that the files wherein configurations are stored are YAML files. Phyles facilitates using YAML files for configurations. For

example the opening, reading, and validating of whiche are automated by the `phyles.Schema.read_config()` function.

2.5.3 Creating a Config Sample

An instance of `phyles.Schema` is capable of producing a sample config file using the `phyles.Schema.sample_config()`. For example given the `schema` we just created:

```
>>> print schema.sample_config()
%YAML 1.2
---

# Dish to cook
# One of: vegetable kabobs, smoked salmon, brisket
dish : smoked salmon

# How much to cook the dish
# One of: well-done, medium, rare
doneness : medium

# Cooking temperature in °C
temperature : 105
```

2.6 The Configuration

Instances of `phyles.Configuration` are simply ordered mappings. By virtue of their `original` attribute, `phyles.Configuration` objects also retain memory of the configuration before conversion (as with the temperature, which was converted from Celsius to Farenheit):

```
>>> for i in config.items():
...   print i
('dish', 'smoked salmon')
('doneness', 350)
('temperature', 107.0)
>>> config['temperature']
225.0
>>> config.original['temperature']
107
```

Instances of `phyles.Configuration` are useful inside a utility, potentially being the sole parameter that needs to be passed to functions. The following example assumes that the function `cooking_time()` is defined as in the section titled The Calculation:

```
difficulties = {'vegetable kabobs': 1,
                'smoked salmon': 2,
                'brisket': 3}

def report_cooking(config):
  t = cooking_time(config['doneness'], config['difficulty'],
                   config['temperature'])
  message = "Cooking time is %5.2f hr." % t
  message = message.center(70)
  config['outlet'](message)

```

```
12 def main(config):
13   ...
14   config['difficulty'] = difficulties[config['dish']]
15   config['outlet'] = lambda s: sys.stdout.write(s + "\n")
16   report_cooking(config)
```

While bundling arguments within a `Configuration` may seem a little combersome at first, it facilitates the adding of new configuration-based behaviors deep within a utility and without the need to modify functions to accommodate additional parameters.

Note: Not all functions of the utility need to take a `Configuration` object as an argument. Here `cooking_time()` still takes three distinct arguments, but the "higher-level" `report_cooking()` function takes config. Such design considerations are left to the programmer.

As an example of the utility of a `Configuration` object, notice that the message width above is hard-coded to 70 in line 9 above. In principle, this width could be user-configurable:

```
spec = """
       !!omap
       - dish :
           - - vegetable kabobs
             - smoked salmon
             - brisket
           - smoked salmon
           - Dish to cook
       - doneness :
           - rare : 200
             medium : 350
             well-done : 500
           - medium
           - How much to cook the dish
       - temperature :
           - celsius to farenheit
           - 105
           - Cooking temperature in °C
           - 105
       - width :
           - int
           - 70
           - width of messages
           - 70
       """

cfg = yaml.load("""
                dish : smoked salmon
                doneness : medium
                temperature : 107
                """)

schema = phyles.load_schema(spec, converters)
config = schema.validate_config(cfg)
```

Now, the message width needs not be hard-coded, which is a bane of maintenance:

```
def report_cooking(config):
  t = cooking_time(config['doneness'], config['difficulty'],
                   config['temperature'])
```

```
  message = "Cooking time is %s hr." % t
  message = message.center(config['width'])
  config['outlet'](message)
```

This enhanced functionality is essentially transparent to the user because a default value (70) is provided for the `width` option, rendering `width` optional in the config file.

2.7 Example Utility

We now have all of the pieces we need to make a utility package, complete with its own library module and scripts (also called "executable programs", or just "programs). As part of the phyles source, an example called "barbecue" is included in the directory called "examples".

2.7.1 Running the Example

Assuming phyles and its dependencies are installed, the barbecue example is fully functioning in-place. For example, try one of the following commands (depending on your shell) from the `examples/barbecue` directory:

bash-type shell:

```
PYTHONPATH=".:${PYTHONPATH}" bin/barbecue-time -t
```

csh/tcsh shell:

```
env PYTHONPATH=".:${PYTHONPATH}" bin/barbecue-time -t
```

Note: The part of the command that modifies `$PYTHONPATH` allows for running the `barbecue-time` executable in-place. Were the barbecue package installed as with `python setup.py install` this modificaiton of `$PYTHONPATH` would not be necessary.

These commands should produce the following output:

```
%YAML 1.2
---

# Dish to cook
# One of: vegetable kabobs, smoked salmon, brisket
dish : smoked salmon

# How much to cook the dish
# One of: well-done, medium, rare
doneness : medium

# Cooking temperature in °C
temperature : 105

# width of report
width : 70
```

Note: The example barbecue package can even be installed with `python setup.py install`, althought it isn't necessary.

Within the `examples/barbecue/test-data` directory is also a config file called `time-config.yml`. This config file can be used without installing the barbecue package:

bash-type shell:

```
PYTHONPATH=".:${PYTHONPATH}" \
    bin/barbecue-time -c test-data/time-config.yml
```

csh/tcsh shell:

```
env PYTHONPATH=".:${PYTHONPATH}" \
   bin/barbecue-time -c test-data/time-config.yml
```

These commands should produce the following output:

```
======================================================================
                        barbecue-time v.0.1b1
======================================================================
                      Cooking time is  3.12 hr.
  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                       Done with smoked salmon!
  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

It is possible to override configuration settings on the command line with the `--override` (or `-o`) argument:

bash-type shell:

```
PYTHONPATH=".:${PYTHONPATH}" \
      bin/barbecue-time -c test-data/time-config.yml \
                        -o 'temperature : 120'
```

csh/tcsh shell:

```
env PYTHONPATH=".:${PYTHONPATH}" \
      bin/barbecue-time -c test-data/time-config.yml \
                        -o 'temperature : 120'
```

Here, the command line temperature of 120 °C (248 °F) overrides the temperature in the config (107 °C), reducing the cooking time. These commands should produce the following output:

```
======================================================================
                        barbecue-time v.0.1b1
======================================================================
                      Cooking time is  2.82 hr.
  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                       Done with smoked salmon!
  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Before looking deeper into the barbecue example, let's see how phyles gracefully handles an error that can not be found at the time when the config is validated because it potentially depends on the state of the system while the program is running:

bash-type shell:

```
PYTHONPATH=".:${PYTHONPATH}" \
      bin/barbecue-time -c test-data/time-config.yml \
                        -o 'width : 10000'
```

csh/tcsh shell:

```
env PYTHONPATH=".:${PYTHONPATH}" \
      bin/barbecue-time -c test-data/time-config.yml \
                        -o 'width : 10000'
```

Here, the a message width of 10000 overrides the config file width of 70. This width is much too large to be displayed on any normal terminal. The barbecue-time script uses the `phyles.get_terminal_size()` function to catch the problem and raise an exception that is itself caught, resulting in a sensible error message being sent to the user with a graceful exit from the program:

```
======================================================================
                        barbecue-time v.0.1b1
======================================================================

  ############################## ERROR ###############################
    Formatting 'width' (10000) bigger than window (78)
  ####################################################################
```

Inspection of the contents of the barbecue utility will reveal how these features of phyles can be used with a small amount of code.

2.7.2 Barbecue Example Directory Structure

The barbecue example is structured as a typical python package, serving as a template for most needs:

- **barbecue/** – top-level directory for package
  - ***CHANGES.txt*** contains version-by-version information about the evolution of the package [2]
  - ***LICENSE.txt*** contains the text of the license [2]
  - ***MANIFEST.in*** tells the setup script which extra files to include in a distribution [2]
  - ***README.rst*** contains broad information about the package [2]
  - **barbecue/** – package directory, holding the library code
    - ***__init__.py*** init module for the package
    - **_barbecue.py** module holding the library code
    - **schema/** – directory holding schema for configs
      - ***barbecue-time.yml*** the schema for the barbecue-time program
  - **bin/** – a directory holding executable programs
    - **barbecue-time** an example program that calculates cooking times
  - **setup.py** a script for distribution and installation
  - **test-data/** – directory that holds test-data
    - ***time-config.yml*** a test configuration file for the barbecue-time program

Let's look at some of the key files in the hierarchy and examine salient features of each, starting first with the barbecue-time program because it shows most directly how to use phyles.

2.7.3 barbecue-time

```
1  import sys
2  import phyles
3  import barbecue
4
5  __program__ = "barbecue-time"
```

[2] http://guide.python-distribute.org/creation.html

```
6  __version__ = "0.1b1"
7
8  def output_message(message, config):
9    console_width = phyles.get_terminal_size()[0]
10   if config['width'] > console_width:
11     tplt = "Formatting 'width' (%s) bigger than window (%s)"
12     message = tplt % (config['width'], console_width)
13     raise barbecue.FormatError(message)
14
15   message = message.center(config['width'])
16   config['outlet'](message)
17
18 def report_cooking(config):
19   t = barbecue.cooking_time(config['doneness'],
20                             config['difficulty'],
21                             config['temperature'])
22
23   message = "Cooking time is %5.2f hr." % t
24   output_message(message, config)
25
26 def finish_up(config):
27   hline = "~" * (config['width'] - 4)
28   output_message(hline, config)
29   message = "Done with %s!" % config['dish']
30   output_message(message, config)
31   output_message(hline, config)
32
33 def main(config):
34   config['difficulty'] = barbecue.difficulties[config['dish']]
35   config['outlet'] = lambda s: sys.stdout.write(s + "\n")
36   report_cooking(config)
37   finish_up(config)
38
39 if __name__ == "__main__":
40   spec = phyles.package_spec(phyles.Undefined, "barbecue",
41                              "schema", "barbecue-time.yml")
42   converters = {'celsius to farenheit':
43                     barbecue.celsius_to_farenheit}
44   setup = phyles.set_up(__program__, __version__, spec, converters)
45   phyles.run_main(main, setup['config'],
46                   catchall=barbecue.BarbecueError)
```

In terms of interacting with phyles, the most critical part of barbecue-time is in lines 40-46:

- **Lines 40-41** The `phyles.package_spec()` function is used to retrieve the schema from the package.
- **Lines 42-43** The converters `dict` is created as in the section title Dictionary of Converter Functions.
- **Line 44** The `phyles.set_up()` function is used to parse command line arguments, load the schema from the spec, validate the config, and override any config setting from the command line option `--override` (`-o`).
- **Lines 45-46** The `phyles.run_main()` function is used to run the main function inside a try-except block that catches any exceptions assigned by the `catchall` keyword argument, and exits gracefully if such exceptions arise.

Note: These few lines (40-46), along with specifying a schema, are all that is truely needed to interface with phyles and take advantage of the mose useful parts of its functionality.

Like any good program, barbecue-time has a `main()` function:

- **Lines 34-35** The `config` is used as a global state, defining new items called `'difficulty'` and `'outlet'`, that will be used in other parts of the program. Such use of a `phyles.Configuration` object is convenient, but left to the discretion of the programmer.

Using a `phyles.Configuration` object allows for abstraction of functionality that depends on the configuration.

- **Line 9** The `phyles.get_terminal_size()` function is used to determine the width of the console.
- **Lines 10-13** The message width from the config file (keyed by `'width'`) is checked against the console width. If the message width is to large, then a `FormatError` exception is raised. As we'll see upon inspection of the file _barbecue.py, `FormatError` is a subclass of `BarbecueError`, which is the catchall exception for graceful exit (see line 45).
- **Lines 26-31** The `finish_up()` function further demonstrates the utility of `Configuration` objects and the abstraction they allow. Note that the `output_message()` function does not care how the message is displayed–except that it is unfortunately tied to the console width. Even this dependencey can be remedied by further abstraction. For example, `config` could have the item:

  ```
  config['canvas_width'] = lambda: get_terminal_size()[0]
  ```

  And then `output_message()` could be changed accordingly:

  ```
  def output_message(message, config):
    max_width = config['get_canvas_width']()
    if config['width'] > max_width:
      tplt = "Formatting 'width' (%s) bigger than window (%s)"
      message = tplt % (config['width'], max_width)
      raise barbecue.FormatError(message)

    message = message.center(config['width'])
    config['outlet'](message)
  ```

  Now, since `config['get_canvas_width']` can be any function (or "callable"), the backend to which the message is sent can be anything, including a console or gui element like a `Tkinter.Label`.

2.7.4 _barbecue.py

The _barbecue.py file holds the main library code for the barbecue package.

```
1  #! /usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  difficulties = {'vegetable kabobs': 1,
5                  'smoked salmon': 2,
6                  'brisket': 3}
7
8  class BarbecueError(Exception):
9    pass
10
11 class FormatError(BarbecueError):
12   pass
13
14 class TemperatureError(BarbecueError):
15   pass
16
17 def cooking_time(doneness, difficulty, T):
18     """
```

```
19      Return then cooking time given the desired doneness
20      cooking, the difficulty of cooking, and the temperature.
21
22      Args:
23        - doneness: desired doneness (hr•°F/dc)
24        - difficulty: difficulty of cooking for the dish (dc)
25        - T: cooking temperature (°F)
26
27      Returns: cooking time in hours (float)
28
29      >>> round(cooking_time(350, 2, 225), 2)
30      3.11
31      """
32      if T <= 120:
33        msg = "%s °F is too cold to cook!" % T
34        raise TemperatureError(msg)
35      return float(doneness * difficulty) / T
36
37  def celsius_to_farenheit(c):
38    """
39    Returns the temperature in Farenheit given temperature
40    in Celsius.
41
42    Args:
43      - c: temperature in Celsius
44
45    Returns: temperature in Farenheit (float)
46
47    >>> celsius_to_farenheit(107.222)
48    224.9996
49    """
50    c = float(c)
51    if c < -273.15:
52      raise ValueError("Impossibly cold (%s °C)!" % c)
53    else:
54      return (1.8 * c) + 32
```

Most of _barbecue.py documents its functionality. However, it does have some key parts:

- **Line 2** This line designates the optional encoding for the file (see http://www.python.org/dev/peps/pep-0263/). The UTF-8 encoding allows for display of the ubiquitous units "°C" and "°F" in the docstrings and error messages.
- **Lines 6-8** Some data is kept in the module, namely the conversions from dish to cooking difficulty. If larger amounts of data are needed, then it is better to include these as so-called "package data" and use the `pkg_resources.resource_string()` function from the distribute package or, failing that, the `phyles.get_data_path()` function, which tries to find package data with every trick in the book.

  Note: With proper utilization of python eggs, a programmer should find that use of the `pkg_resources.resource_string()` function is failsafe.

- **Lines 10-17** As seen in the barbecue-time file (lines 45-46), the `BarbecueError` is used as a catchall for anticipated errors, allowing the program to exit gracefully if any are raised while executing the `main()` function.

  Created here are the `BarbecueError` and a couple of decendants, corresponding to problems with formatting and nonsensical cooking temperatures (lines 34-36). Since these exceptions are `BarbecueError` or inherit from it, then they fall under the catchall and trigger graceful exit.

2.7.5 setup.py

The setup.py script directs the distribution and installation of python packages. See http://guide.python-distribute.org/creation.html for a complete discussion.

Below is a partial (acutally, almost complete) listing of setup.py mainly to (1) show the minimal required keyword arguments and (2) show how to use the following keyword arguments of the `setup()` function:

- `packages`
- `include_package_data`
- `package_data`
- `scripts`

```
1  import os
2  import glob
3
4  from setuptools import setup, find_packages
5
6  setup(name='barbecue',
7        version='0.1b1',
8        author='James C. Stroud',
9        author_email='jstroud@mbi.ucla.edu',
10       description='Utilities for cooking barbecue.',
11       url='http://phyles.bravais.net/barbecue',
12       classifiers =[
13               'Programming Language :: Python :: 2',
14          ],
15       install_requires=["distribute", "phyles >= 0.2.0"],
16       license='LICENSE.txt',
17       long_description=open('README.rst').read(),
18       packages=find_packages(),
19       include_package_data=True,
20       package_data={'': ['*.yml']},
21       scripts=glob.glob(os.path.join('bin', '*')))
```

- **`packages` – line 18** It is notable that the keyword argument `package_dir` is not required, nor is it necessary to specify the package manually names because they are found automatically by `distribute.find_packages()` imported on line 4. Here, `find_packages()` evaluates to `['barbecue']`.
- **`include_package_data` – line 19** This keyword argument ensures that the files found by the `package_data` keyword argument will be included upon installation (not just packaging for distribution).
- **`package_data` – line 20** The empty string (`''`) means to include files that match the corresponding patterns (`['*yml']`) for **all** packages listed for the `packages` keyword argument. Here, these packages are found automatically. In this barbecue example `{'': ['*yml']}` matches `barbecue/schema/barbecue-time.yml`.
- **`scripts` – line 21** The value to `scripts` says to include all files (`'*'`) in the `bin` directory, using the `glob.glob()` function from the python standard library, imported on line 2. Here, `glob.glob(os.path.join('bin', '*'))` evaluates to `[barbecue-time]`.

CHAPTER

THREE

PHYLES API

3.1 API Overview

The phyles API has several functions and classes to facilitate the construction of medium-sized utilites. These functions and classes are divided into four categories:

- Classes for Configurations and Schemata
- Functions for Configurations and Schemata
- Functions for Files and Directories
- Functions for User Interaction
- Functions for a One-Size-Fits-All Runtime

3.2 Classes for Configurations and Schemata

- **`phyles.Schema`** encapsulates a schema and wraps phyles.sample_config, phyles.validate_config, and phyles.read_config for convenience
- **`phyles.Configuration`** encapsulates a configuration, remembering values before any conversion

3.3 Functions for Configurations and Schemata

- **phyles.read_schema** makes a schema from a specification in a YAML file
- **phyles.load_schema** makes a schema a specification in YAML text, a mapping, or a `list` of 2-`tuples` with unique keys
- **phyles.sample_config** produces a sample config from a schema
- **phyles.validate_config** validates a config file with a schema
- **phyles.read_config** reads a yaml config file and validates the config with a schema

3.4 Functions for Files and Directories

- **phyles.last_made** returns the most recently created file in a directory
- **phyles.get_home_dir** returns the users home directory in a representation native to the host OS

- **phyles.get_data_path** returns the absolute path to a data directory within a package
- **phyles.package_spec** reads and returns the contents a schema specification somewhere in a package as YAML text
- **phyles.prune** recursively deletes files matching specified unix sytle patterns

3.5 Functions for User Interaction

- **phyles.wait_exec** waits for a command to execute via a system call and returns the output from stdout; slightly more convenient than `popen2.Popen3`
- **phyles.doyn** queries user for yes/no input from `raw_input()` and can execute an optional command with phyles.wait_exec
- **phyles.banner** prints a banner for the program to stdout
- **phyles.usage** uses `optparse.OptionParser` to print usage and can print an optional error message, if provided.
- **phyles.default_argparser** returns a default `argparse.ArgumentParser` with mutually exclusive template, config, and override arguments added
- **phyles.get_terminal_size** returns the terminal size as a (width, height) `tuple` (works with Linux, OS X, Windows, Cygwin)

3.6 Functions for a One-Size-Fits-All Runtime

- **phyles.set_up** sets up the runtime with an `argparse.ArgParser`, loads a schema and validates a config with config override, and prepares state for graceful recovery from user error
- **phyles.run_main** trivial try-except block for graceful recovery from anticipated types of user error

3.7 API Details

phyles: A package to simplify authoring utilites. Copyright (C) 2013 James C. Stroud All rights reserved.

class `phyles.`**`Schema`**(**args*, ***kwds*)

An `OrderedDict` subclass that provides identity for schemata.

A `Schema` has an implicit structure and is created by the `load_schema()` or `read_schema()` functions. See the documentation in `load_schema()` for a detailed explanation.

> **Warning:** Creating instances of `Schema` through its class constructor (i.e. `'__init__'`) is not yet advised or supported and may break forward compatibility.

`read_config`(**args*, ***kwargs*)

This is a wrapper for `read_config()` (see documentation therein).

Comparison of usage with `read_config()`:

```
phyles.read_config(schema, config)
schema.read_config(config)
```

sample_config(**args*, ***kwargs*)
This is a wrapper for `sample_config()` (see documentation therein).

Comparison of usage with `sample_config()`:

```
phyles.sample_config(schema)
schema.sample_config()
```

validate_config(**args*, ***kwargs*)
This is a wrapper for `validate_config()` (see documentation therein).

Comparison of usage with `validate_config()`:

```
phyles.validate_config(schema, config)
schema.validate_config(config)
```

class phyles.**Configuration**(*config=None*)
An `OrderedDict` subclass that encapsulates configurations and also remembers the original input.

A `Configuration` has a specific structure and is creatd by `validate_config()` or `read_config()` functions, usually by the latter. The `Configuration` class is exposed in the API purely for purposes of documentation.

Attributes:

***original*: the original config as an `OrderedDict`,** allowing the remembering of user input while also allowing conversion

```
>>> colors = {'red': 'ff0000',
...           'green': '00ff00',
...           'blue': '0000ff'}
>>> c = Configuration({'color': 'blue'})
>>> c['color'] = colors[c['color']]
>>> c['color']
>>> '0000ff'
>>> c.original['color']
'blue'
```

Warning: Creating instances of `Configuration` through its class constructor (i.e. with `phyles.Configuration()`) is not yet advised or supported and may break forward compatibility.

phyles.**read_schema**(*yaml_file*, *converters=None*)
Loads the schema specified in the file named in *yaml_file*. This function simply opens and reads the *yaml_file* before sending its contents to `load_schema()` to produce the `Schema`.

Args: *yaml_file*: name of a yaml file holding the specification

***converters*:** a `dict` of converters keyed by config entry names, as described in `load_schema()`

Returns: a `Schema` as described in `load_schema()`

phyles.**load_schema**(*spec*, *converters=None*)
Creates a `Schema` from the specification, *spec*.

Note: If the schema specification is in a YAML file, then use `phyles.read_schema()`, which is a convenience wrapper around `load_schema()`.

Args:

***spec*:** Can either be YAML text, a `list` of 2-`tuples` with unique first elements, or a mapping object (e.g. `dict`). If the schema is in a YAML file, then use `phyles.read_schema()`. The values of the items of *spec* are:

1. converter
2. example value
3. help string
4. default value (optional)

Example YAML specification as a complete YAML file:

```
%YAML 1.2
---
!!omap
- 'pdb model' : [str, my_model.pdb, null]
- 'reset b-facs' :
      - float
      - -1
      - "New B factor (-1 for no reset)"
      - -1
- 'cell dimensions' : [get_cell, [200, 200, 200], null]
```

The same example as a python specification via a `list` of 2-`tuples` with unique first elements:

```
[('pdb model',
  ['str', 'my_model.pdb', None]),
 ('reset b-facs',
  ['float', -1, 'New B factor (-1 for no reset)', -1]),
 ('cell dimensions',
  [get_cell, [200, 200, 200], None])]
```

For completeness, the same example as a `dict`:

```
{'pdb model': ['str', 'my_model.pdb', None],
 'reset b-facs':
   ['float', -1, 'New B factor (-1 for no reset)', -1],
 'cell dimensions': [get_cell, [200, 200, 200], None]}
```

Note: The python structure (`list` of 2-`tuples`) of this example specification is simply the result of parsing the YAML with the PyYAML parser. Because of isomorphism between a `list` of 2-`tuples` with unique first elements, `OrderedDicts`, `dicts`, and other mapping types, the specification may take any of these forms.

The following YAML representation of a config conforms to the preceding schema specification:

```
pdb model : model.pdb
reset b-facs : 20
cell dimensions : [59, 95, 159]
```

***converters*:** A `dict` of callables keyed by converter name (which must match the converter names in *spec*), The callables convert values from the actual config.

Converters that correspond to several native python types and YAML types do not need to be explicitly specified. The names that these converters take in a schema specification and the corresponding python types produced by these converters are:

- **map**: `dict`
- **dict**: `dict` (encoded as a YAML dict)
- **omap**: `collections.OrderedDict`
- **odict**: `collections.OrderedDict` (alias for "omap")
- **pairs**: `list` of 2-`tuples`
- **set**: `set`
- **seq**: `list`
- **list**: `list` (encoded as a sequence, see `list()`)
- **tuple**: `tuple` (encoded as a sequence, see `tuple()`)
- **bool**: `bool`
- **float**: `float`
- **int**: `int`
- **long**: `long` (encoded as a YAML int)
- **complex**: `complex` (encoded as a sequence of 0 to 2, or as a string representation, e.g. `'3+2j'`; see `complex()`)
- **str**: `str`
- **unicode**: `unicode`
- **timestamp**: `datetime.datetime` (encoded as a YAML timestamp)
- **slice**: `slice` (encoded as a sequence of 1 to 3, see `slice()`)

Note: Except where noted, these types are encoded according to the YAML types specification in a YAML representation of a config.

Returns: A fully constructed schema in the form of a `Schema`. Most notably, the strings specifying the converters in the *spec* are replaced by the converters themselves.

```
>>> import phyles
>>> import textwrap
>>> def get_cell(cell):
      return [float(f) for f in cell]
>>> converters = {'get_cell' : get_cell}
>>> y = '''
        %YAML 1.2
        ---
        !!omap
        - 'pdb model' : [str, my_model.pdb, null]
        - 'reset b-facs' :
              - float
              - -1
              - "New B factor (-1 for no reset)"
              - -1
        - 'cell dimensions' : [get_cell, [200, 200, 200], null]
        '''
>>> phyles.load_schema(textwrap.dedent(y), converters=converters)
Schema([('pdb model',
         [<type 'str'>, 'my_model.pdb', None]),
```

```
            ('reset b-facs',
             [<type 'float'>, -1,
              'New B factor (-1 for no reset)', -1]),
            ('cell dimensions',
             [<function get_cell at 0x101d7cf50>,
              [200, 200, 200], None])])
```

phyles.**sample_config**(*schema*)

Creates a sample config specification (returned as a `str`) from the *schema*, as described in `read_schema()`.

Args: *schema*: a `Schema`

Returns: A `str` that is useful as a template config specification. Example values from the schema will be used. Additionally, the help strings will be inserted as reasonably formatted YAML comments.

```
>>> import phyles
>>> import textwrap
>>> def get_cell(cell):
      return [float(f) for f in cell]
>>> converters = {'get_cell' : get_cell}
>>> y = '''
        !!omap
        - 'pdb model' : [str, my_model.pdb, null]
        - 'reset b-facs' :
              - float
              - -1
              - "New B factor (-1 for no reset)"
              - -1
        - 'cell dimensions' : [get_cell, [200, 200, 200], null]
        '''
>>> schema = phyles.load_schema(textwrap.dedent(y),
                                converters=converters)
>>> print phyles.sample_config(schema)
%YAML 1.2
---

pdb model : my_model.pdb

# New B factor (-1 for no reset)
reset b-facs : -1

cell dimensions : [200, 200, 200]
```

phyles.**validate_config**(*schema*, *config*)

Takes a YAML specification for a configuration, *config*, and uses the *schema* (as described in `load_schema()`) for validation, which:

1. checks for required config entries, raising a `ConfigError` if any are missing
2. ensures that no unrecognized config entries are present, raising a `ConfigError` in any such entries are present
3. ensures, through the use of converters, that the values given in the config specification are of the appropriate types and within accepted limits (if applicable), raising a `ConfigError` if any fail to convert
4. uses the converters to turn values given in the configuration into values of the appropriate types (e.g. the YAML str `'1+4j'` is converted into the python `complex` number `(1+4j)` if the converter is `'complex'`)

Note: Why is conversion a part of validation? Conversion facilitates the end-user's working with a minimal subset of the YAML vocabulary. In the `complex` number example above, the end-user only needs to know how complex numbers are usually represented (e.g. `'1+4j'`) and not what gibbersh like `'!!python/object:__main__.SomeComplexClass'` means, where to put it, how to specify its attributes, etc.

Args:

***config*:** a mapping (e.g. `OrderedDict` or `dict`) of configuration entries

***schema*:** a `Schema` as described in `load_schema()`

Returns: The converted config as a `Configuration`.

Raises: `ConfigError`

```
>>> import phyles
>>> import textwrap
>>> import yaml
>>> def get_cell(cell):
      return [float(f) for f in cell]
>>> converters = {'get_cell' : get_cell}
>>> y = '''
        %YAML 1.2
        ---
        !!omap
        - 'pdb model' : [str, my_model.pdb, null]
        - 'reset b-facs' :
              - float
              - -1
              - "New B factor (-1 for no reset)"
              - -1
        - 'cell dimensions' : [get_cell, [200, 200, 200], null]
        '''
>>> schema = phyles.load_schema(textwrap.dedent(y),
                                converters=converters)
>>> y = '''
        pdb model : model.pdb
        reset b-facs : 20
        cell dimensions : [59, 95, 159]
        '''
>>> cfg = yaml.load(textwrap.dedent(y))
>>> cfg
{'cell dimensions': [59, 95, 159],
 'pdb model': 'model.pdb',
 'reset b-facs': 20}
>>> phyles.validate_config(schema, cfg)
Configuration([('pdb model', 'model.pdb'),
               ('reset b-facs', 20.0),
               ('cell dimensions', [59.0, 95.0, 159.0])])
```

`phyles.`**`read_config`**(*schema*, *config_file*)

Reads a YAML config file from the the file named *config_file* and returns the config validated by *schema*.

Args:

***config_file*:** YAML file holding the config, for example:

```
pdb model : model.pdb
reset b-facs : 20
cell dimensions : [59, 95, 159]
```

schema: a `Schema` as described in `load_schema()`

Returns: a `Configuration`

phyles.**last_made**(*dirpath='.'*, *suffix=None*)
Returns the most recently created file in *dirpath* If provided, the news of the files with the given suffix or suffices is returned.

The *suffix* parameter is either a single suffix (e.g. `'.txt'`) or a sequence of suffices (e.g. `['.txt', '.text']`).

phyles.**wait_exec**(*cmd*, *instr=None*)
Waits for *cmd* to execute and returns the output from stdout. The *cmd* follows the same rules as for python's `popen2.Popen3`. If *instr* is provided, this string is passed to the standard input of the child process.

Except for the convenience of passing *instr*, this funciton is somewhat redundant with pyhon's `subprocess.call()`.

phyles.**doyn**(*msg*, *cmd=None*, *exc=<built-in function system>*, *outfile=None*)
Uses the `raw_input()` builtin to query the user a yes or no question. If *cmd* is provided, then the function specified by *exc* (default `os.system()`) will be called with the argument *cmd*.

If a file name for *outfile* is provided, then stdout will be directed to a file of that name.

phyles.**banner**(*program*, *version*, *width=70*)
Uses the *program* and *version* to print a banner to stderr. The banner will be printed at *width* (default 70).

Args: *program*: `str`

version: `str`

width: `int`

phyles.**usage**(*parser*, *msg=None*, *width=70*, *pad=4*)
Uses the *parser* (`argparse.ArgumentParser`) to print the usage. If *msg* (which can be an `Exception`, `str`, etc.) is supplied then it will be printed as an error message, hopefully in a way that catches the user's eye. The usage message will be formatted at *width* (default 70). The *pad* is used to add some extra space to the beginning of the error lines to make them stand out (default 4).

phyles.**graceful**(*msg=None*, *width=70*, *pad=4*)
Gracefully exits the program with an error message.

The *msg*, *width* and *pad* arguments are the same as for `usage()`.

phyles.**get_home_dir**()
Returns the home directory of the account under which the python program is executed. The home directory is represented in a manner that is comprehensible by the host operating system (e.g. `C:\something\` for Windows, etc.).

Adapted directly from K. S. Sreeram's approach, message 393819 on c.l.python (7/18/2006). I treat this code as public domain.

phyles.**get_data_path**(*env_var*, *package_name*, *data_dir*)
Returns the path to the data directory. First it looks for the directory specified in the *env_var* environment variable and if that directory does not exists, finds *data_dir* in one of the following places (in this order):

1. The package directory (i.e. where the `__init.py__` is for the package named by the *package_name* parameter)

2.If the package is a file, the directory holding the package file

3.If frozen, the directory holding the frozen executable

4.If frozen, the parent directory of the directory holding the frozen executable

5.If frozen, the first element of *sys.path*

Thus, if the package were at `/path/to/my_package`, (i.e. with `/path/to/my_package/__init__.py`), then a very reasonable place for the data directory would be `/path/to/my_package/package-data/`.

The anticipated use of this function is within the package with which the data files are associated. For this use, the package name can be found with the global variable __package__, which for this example would have the value `'my_package'`. E.g.:

```
pth = get_data_path('MYPACKAGEDATA', __package__, 'package-data')
```

This code is adapted from *_get_data_path()* from matplotlib `__init__.py`. Some parts of this code are most likely subject to the matplotlib license.

Note: The *env_var* argument can be ignored using `phyles.Undefined` because it's guaranteed not to be in `os.environ`:

```
pth = get_data_path(Undefined, __package__, 'package-data')
```

Warning: The use of `'__package__'` for *package_name* will fail in certain circumstances. For example, if the value of `__name__` is `'__main__'`, then `__package__` is usually `None`. In such cases, it is necessary to pass the package name explicitly.

```
pth = get_data_path(Undefined, 'my_package', 'package-data')
```

`phyles.`**`prune`**(*patterns*, *doit=False*)

Recursively deletes files matching the specified unix style *patterns*. The *doit* parameter must be explicitly set to `True` for the files to actually get deleted, otherwise, it just logs with `logging.info()` what *would* happen. Raises a `SystemExit` if deletion of any file is unsuccessful (only when *doit* is `True`).

Example:

```
prune(['*~', '*.pyc'], doit=True)
```

Args:

- *patterns*: `list` of unix style pathname patterns
- *doit*: `bool`

Returns: `None`

Raises: `SystemExit`

`phyles.`**`default_argparser`**()

Returns a default `argparse.ArgumentParser` with mutually exclusive template (`-t`, `--template`) and config (`-c`, `--config`) arguments added. It also has the override (`-o`, `--override`) option added, to override configuration items on the command line.

The argument to `--override` should be a valid YAML map (with the **single** exception that the outermost curly braces are optional for YAML flow style). Because YAML relies on syntactically meaningful whitespace, single quotes should surround the argument to `--override`.

The following examples execute a program called `program`, overriding `opt1` and `opt2` of the config in the file `config.yml` with `foo` and `bar`, respctively:

```
program -c config.yml -o 'opt1 : foo, opt2 : bar'

program -c config.yml -o '{opt1 : foo, opt2 : bar}'

program -c config.yml -o 'opt1 : foo\nopt2 : bar'
```

Note: The latter example illustrates how YAML block style can be used with `--override`: a single forward slash (`\`) escapes an `n`, which evaluates to a so-called "newline". In other words, the YAML that corresponds to this latter example is:

```
opt1 : foo
opt2 : bar
```

Similarly, other escape sequences can also be used with `--override`. For example, the following overrides an option called `sep`, setting its value to the tab character:

```
program -c config.yml -o 'sep : "\t"'
```

Returns: `argparse.ArgumentParser`

`phyles.`**`package_spec`**(*env_var*, *package_name*, *data_dir*, *specfile_name*)

Reads and returns the contents a schema specification somewhere in a package as YAML text (described in `load_schema()`).

This function pulls out all the stops to find the specification. It is best to try to give all of *env_var*, *package_name*, and *data_dir* if they are available to have the best chance of finding the path to the specification file. See `get_data_path()` for a full description.

Args: The arguments *env_var*, *package_name*, and *data_dir* are identical to those required in `get_data_path()`.

specfile_name: name of the schema specification file found within the package contents.

Returns: A YAML string specifying the schema.

`phyles.`**`set_up`**(*program*, *version*, *spec*, *converters=None*)

Given the name of the program (*program*), the *version* string, the specification for the schema (*spec*; described in `load_schema()`), and *converters* for the schema (also described in `load_schema()`), this function:

1. sets up the default argparser (see `default_argparser()`)
2. prints the template or banner as appropriate (see `template()` and `banner()`)
3. creates a schema and uses it to validate the config (see `load_config()` and `validate_config()`)
4. overrides items in the config according to the command line option `--override` or `-o` (see `default_argparser()` for a description of `--override`)
5. exits gracefully with `usage()` if any problems are found in the command line arguments of config
6. returns a `dict` of the `argparse.ArgumentParser`, the parsed command line arguments, the `Schema`, and the `Configuration` with keys `'argparser'`, `'args'`, `'schema'`, `'config'`, respectively

Args: *program*: the program name as a `str`

version: the program version as a `str`

spec: a schema specification as described in `load_schema()`

coverters: a `dict` of converters as described in `load_schema()`

Returns: a `dict` with the keys:

1. `'argparser'`: `argparse.ArgumentParser`
2. `'args'`: the parsed command lines arguments as a `argparse.Namespace`
3. `'schema'`: schema for the configuration as a `Schema`
4. `'config'`: the configuration as a `Configuration`

phyles.**run_main**(*main*, *config*, *catchall=<class 'phyles._phyles.DummyException'>*)

Trivial convenience function that runs a *main* function within a try-except block. The *main* function should take as its sole argument the *config*, which is a mapping object that holds the configuration (usually a `Configuration` object). The *catchall* is an `Exception` or a tuple of `Exceptions`, which if caught will result in a graceful exit of the program (see `graceful()`).

Args: *main*: a function, equivalent to the main function of a program

***config*: a mapping object, usually a `Configuration`** which is generally produced by `validate_config()` or `read_config()`

catchall: an `Exception` or a `tuple` of `Exceptions`

phyles.**get_terminal_size**()

return width and height of console; works on linux, os x,windows,cygwin(windows)

based on https://gist.github.com/jtriley/1108174 (originally retrieved from: http://goo.gl/CcPZh)

Returns: 2-`tuple` of `int`

CHAPTER

FOUR

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

p

INDEX