Naked Documentation

Release 0.1.31

Christopher Simpkins

Nov 04, 2017

Contents

CHAPTER 1

A Python Command Line Application Framework

Naked (source: PyPI, GitHub) is a MIT licensed command line application framework that provides a variety of features for Python application developers.

Note: The *QuickStart Guide* demonstrates how to go from an empty directory to a PyPI push of your first application release using tools provided by the Naked Framework.

Here is a sample of the framework features:

1.1 New Projects

- **New Project Generator** : Create a complete project directory structure and project file stubs with the naked executable:

```
naked make
```

1.2 Command Line Parser

- **Simple Command to Python Object Parser** :

```
# positional strings in the command are command object attributes
# short- (e.g. '-p') and long-form (e.g. '--print') options are tested with a method

# user enters: <executable> hello world --print
c = Naked.commandline.Command(sys.argv[0], sys.argv[1:])
if c.cmd == 'hello' and c.cmd2 == "world":
    if c.option('--print'):
        print('Hello World!')
```

- **Simple Command Argument Management** :

```
# argument testing and assignment by command object methods

# user enters: <executable> -l Python --framework Naked
if c.option_with_arg('-l') and c.option_with_arg('--framework'):
    language = c.arg('-l')
    framework = c.arg('--framework')
    print(framework + ' ' + language)    # prints 'Naked Python' to standard out
```

- **Simple Command Switch Management** :

```
# switch testing by command object method

if c.option('-s'):
    # do something
elif c.option('-l'):
    # do something else
```

See the *Command Line Parser* documentation for details.

1.3 State Data

- **Simple State Management** :

```
# assign your own attributes to the command object for later use in your coding logic

if c.option('--spam'):
    c.spam = True
    c.eggs = False
if c.option('--eggs'):
    c.spam = False
    c.eggs = True

# other stuff

if c.spam:
    print("yum")
elif c.eggs:
    print("yum"*2)
```

See the *Command Line Parser* documentation for details.

- **The StateObject** : a compendium of automatically generated user state information. It includes data such as the Python interpreter version, operating system, user directory path, current working directory, date, time, and more.

```
from Naked.toolshed.state import StateObject

state = StateObject() # collects state information at time of instantiation

working_directory = state.cwd
if state.py2:
    print("In the directory " + working_directory + " and using the Python 2␣
↪interpreter")
else:
    print("In the directory " + working_directory + " and using the Python 3␣
↪interpreter")
```

See the *Toolshed: state* documentation for details.

1.4 Networking

- GET and POST requests are as simple as:

```
from Naked.toolshed.network import HTTP

http = HTTP("http://www.google.com")
if http.get_status_ok():
    print(http.res.text)

http = HTTP("http://httpbin.org/post")
if http.post_status_ok():
    print(http.res.text)
```

Text and binary file writes from GET and POST requests are just as easy. See the *Toolshed: network* documentation for details.

1.5 File I/O

- Supports Unicode (UTF-8) reads and writes by default:

```
from Naked.toolshed.file import FileReader, FileWriter

fr = FileReader('myfile.txt')
u_txt = fr.read()

fw = FileWriter('newfile.txt')
fw.write(u_txt)
```

There are a number of I/O methods in the `FileReader` and `FileWriter` classes. See the *Toolshed: file* documentation for details.

1.6 Execution of System Executables and Scripts

- **System Command Execution** :

```
from Naked.toolshed.shell import execute

execute('curl http://www.naked-py.com')
```

- **Ruby Script Execution** :

```
from Naked.toolshed.shell import execute_rb

execute_rb('ruby/testscript.rb')
```

- **Node.js Script Execution** :

```
from Naked.toolshed.shell import execute_js

execute_js('node/testscript.js')
```

See the *Toolshed: shell* documentation for more information, including documentation of exit status code checks & standard output and error stream handling from the Python side using *Naked.toolshed.shell.muterun()*.

1.7 Explore Environment Variables

- **Access Each String in the User PATH**

```
from Naked.toolshed.shell import Environment

env = Environment()
if (env.is_var('PATH')):
      for i in env.get_split_var_list('PATH'):
          print(i)
```

See the *Toolshed: shell* documentation for details.

1.8 Function, Method, Class Extensions

- **The Naked toolshed types library** includes extensions of commonly used Python types:
  - **XString** extends the Python string
  - **XDict** extends the Python dictionary
  - **XList** extends the Python list
  - **XMaxHeap** a max heap priority queue that extends the Python heapq
  - **XMinHeap** a min heap priority queue that extends the Python heapq
  - **XSet** extends the Python set
  - **XQueue** extends the Python deque
- **Faster, compiled C versions of the library modules** with an *optional* post-install compile for those who need a jetpack.

See the *The Toolshed Library Overview* documentation for an overview and links to the respective parts of the toolshed library documentation.

1.9 Text Templates

- **The Ink Templating System** - a lightweight, flexible text templating system that allows you to define the replacement tag syntax in your template documents. Available in the `Naked.toolshed.ink` library module.

```
from Naked.toolshed.ink import Template, Renderer

template_string = "I like {{food}} and {{drink}}"
template = Template(template_string)
template_key = {'food': 'fries', 'drink': 'beer'}
```

```
renderer = Renderer(template, template_key)
rendered_text = renderer.render()
print(rendered_text)          # prints "I like fries and beer"
```

See the *Toolshed: ink* documentation for details.

1.10 Benchmarking

- Benchmarking decorators are available for your methods and functions. Insert a decorator above your function or method and get 10 trials of between 10 and 1 million repetitions of the code with comparison to a built-in test function. Comment it out and it's gone.

```
from Naked.toolshed.benchmarking import timer_trials_benchmark

@timer_trials_benchmark
def your_function(arg1, arg2):
    # your code
```

See the *Toolshed: benchmarking* documentation for details.

1.11 Profiling

- The `profiler.py` script is added to every project in the path `PROJECT/lib/profiler.py`. Insert your test code in the designated testing block and then run `naked profile` from any directory in your project. cProfile and pstats profiling is implemented with default report settings (which you can modify in the `profiler.py` file if you'd like).

Details are available in the naked executable profile documentation. An example is provided in the *QuickStart Guide*.

1.12 Testing

- Testing with the tox, nose, py.test, and the built-in Python unittest test runners can be run from any directory in your project with the `naked test` command. Use the included tests project directory for your unit test files.

Details are available in the naked executable test documentation. An example is provided in the *QuickStart Guide*.

1.13 Python Documentation

- Search the built-in Python documentation from the command line with the `pyh` naked executable command.

```
$ naked pyh dict

Help on class dict in module __builtin__:

class dict(object)
 |  dict() -> new empty dictionary
 |  dict(mapping) -> new dictionary initialized from a mapping object's
 |      (key, value) pairs
 |  dict(iterable) -> new dictionary initialized as if via:
```

```
|      d = {}
|      for k, v in iterable:
|          d[k] = v
|  dict(**kwargs) -> new dictionary initialized with the name=value pairs
|      in the keyword argument list.  For example:  dict(one=1, two=2)
|
|  Methods defined here:
|
|  __cmp__(...)
|      x.__cmp__(y) <==> cmp(x,y)
|
|  __contains__(...)
|      D.__contains__(k) -> True if D has a key k, else False

...
```

There is no need to enter the Python interactive interpreter.

1.14 Flexible and No Commitment

- **Every component of the framework is 100% optional**. You determine how much (if any) of the Naked source you need in your project. Building a project with the executable does not mandate use of the command parser, the automatically implemented help, usage, and version commands, or any part of the Naked toolshed library.

The goal is to help when you need it and get out of the way when you don't.

CHAPTER 2

Contents

2.1 Naked Resources

2.1.1 The Source Code

The source is available on GitHub and PyPI.

2.1.2 Issue Reports

Find a bug? Let's fix it. Please report it on the GitHub issue tracker.

2.2 Install Guide

Note: If you have an installed version of Naked and want to upgrade it, see the *Upgrade Guide*.

Use one of the following methods to install your first version of Naked.

2.2.1 Install with pip

To install Naked with pip, use the following command:

```
pip install Naked
```

2.2.2 Git Clone and Install

Navigate to the directory where you would like to pull the Naked source and then use git to clone the Naked repository with the following command:

```
git clone https://github.com/chrissimpkins/naked.git
```

Navigate to the top level of the source repository and run the following command:

```
python setup.py install
```

The cloned repository can be safely deleted after the install.

2.2.3 Download and Install

Download the zip or tar.gz source archive and decompress it locally. Navigate to the top level of the source directory and run the following command:

```
python setup.py install
```

The downloaded source file archive can be safely deleted after the install.

2.2.4 Confirm Install

To confirm your install, type `naked --version` on the command line. This will display the installed version of the Naked framework.

2.3 Upgrade Guide

2.3.1 Upgrade with pip

To upgrade Naked with pip, use the following command:

```
pip install --upgrade Naked
```

This will pull the most recent version from PyPI and install it on your system.

2.3.2 Git Clone and Upgrade

Navigate to the directory where you would like to pull the new version of the Naked source and then use git to clone the Naked repository with the following command:

```
git clone https://github.com/chrissimpkins/naked.git
```

Navigate to the top level of the source repository and run the following command:

```
python setup.py install
```

The cloned repository can be safely deleted after the upgrade.

2.3.3 Download and Upgrade

Download the new version of the zip or tar.gz source archive and decompress it locally. Navigate to the top level of the source directory and run the following command:

```
python setup.py install
```

The downloaded source file archive can be safely deleted after the upgrade.

2.3.4 Confirm Upgrade

Type `naked --version` to confirm that you have the latest version of Naked.

2.4 Definitions

Here are definitions of Naked framework specific terms that are commonly used in the documentation:

- **Naked executable** : the executable `naked` that is distributed with the Naked framework (*Naked Executable*)
- **Naked parser** : the command line command string to Python object parser that is distributed with the Naked framework (*Command Line Parser*)
- **Naked project** : the directory structure and automatically generated files that are created by the naked executable (*Naked Project Structure*)
- **Naked toolshed library** : a library of Python and C source files that are distributed with the Naked framework and designed for use by Python application developers (*The Toolshed Library Overview*)
- **StateObject** : an object that is instantiated with the Naked toolshed library. It includes numerous operating system, Python language, and application environment state attributes that can be used in the application logic. (*Toolshed: state*)

2.5 QuickStart Guide

2.5.1 Make Your Own Spam and Eggs

This guide will take you from an empty directory to a PyPI push of your first application release using features available in the Naked framework. You will learn how to:

1. Create a new `spameggs` project with the `naked make` command
2. Implement the command line logic for your `spameggs` executable with the Naked parser
3. Import part of the Naked toolshed library for use in the application
4. Perform unit tests across multiple Python versions with the `naked test` command
5. Perform profiling with cProfile and pstats using the `naked profile` command
6. Distribute your project to PyPI with the `naked dist` command.

Links are provided to other parts of the documentation where you can learn much more about how to incorporate the components of the Naked Framework into the stages of your development process.

2.5.2 Make a New Project with `naked make`

- Create a new directory and save a `naked.yaml` file in the directory that includes the following data:

```
application: spameggs
developer: Guido
license: MIT license
```

- Navigate to the directory in your terminal and run the command:

```
naked make
```

- You will receive the following prompt:

```
Detected a Naked project YAML setup file (naked.yaml).

Please confirm the information below:

------------------------------------------------------------
 spameggs
 Copyright 2016 Guido
 MIT license
------------------------------------------------------------

Is this correct? (y/n)
```

- Respond to the prompt with 'y'.
- naked displays the following information about your project structure:

```
spameggs was successfully built.

-----
Main application script:  spameggs/lib/spameggs/app.py
Settings file:  spameggs/lib/spameggs/settings.py
Commands directory:  spameggs/lib/spameggs/commands
setup.py file:  spameggs/setup.py
-----

Use 'python setup.py develop' from the top level of your project and you can begin␣
↪testing your application with the executable, spameggs
```

- Let's follow the instructions in the last statement so that we can begin using the application from the command line. Enter the following in the top level directory that contains your setup.py file:

```
python setup.py develop
```

Your application framework is all set for development. `spameggs` should be registered on your PATH so that you can use it.

Learn More

- The Naked Executable
- The Naked Make Command
- The naked.yaml file
- How to create a project without a naked.yaml file

- How Naked creates your LICENSE file

2.5.3 Test Your Application Version Command

- Let's make sure that it is working. `naked make` creates your version command for you. Give it a try:

```
$ spameggs --version
spameggs 0.1.0

$ spameggs -v
spameggs 0.1.0

$ spameggs version
spameggs 0.1.0
```

- The displayed text automatically changes when you increment your version number in the `spameggs/lib/spameggs/settings.py` file and the format of the displayed string can be modified to your liking. You can learn more with the links below.

Learn More

- The help, usage, and version commands
- How to set your version text
- How to remove the Naked implementation of the version command

2.5.4 Inspect Your Project Files

- Have a look through your project directory to familiarize yourself with what `naked` created for you.

Learn More

- Diagram of the Naked Project Structure
- Directories that are created in a Naked Framework project
- Files that are created in a Naked framework project

2.5.5 Create Your Application

`spameggs` is going to perform the extremely important task of printing 'Spam and Eggs' to the standard output stream. As with most academic exercises, this is going to be an extremely roundabout approach that is intended to be a demonstration of the capabilities of the framework rather than be the most efficient, or even correct (we are going to skip prints to std err and non-zero exit status returns for errors...), approach.

- Open your `spameggs/lib/spameggs/app.py` file in an editor and take a look through it. `main()` is where execution of your application script begins. `naked` included a few imports (the Python sys module, the Naked command line parser module, and the Naked state module for the StateObject). It created an instance of the Naked parser (named `c`) and also included the methods that handle help, usage, and version requests. We tested the version commands above and we'll look at the help and usage below. The last thing that `naked` inserts in this part of the file is a validation statement that confirms that the user entered a primary command (`c.command_suite_validates()`).

Note: If you are not making a command suite application with syntax like this: `<executable> <primary command> ...`, you can replace the `command_suite_validates()` method with the `app_validates_args()` method. The latter confirms that at least one argument, including short options (e.g. `-s`), long options (e.g. `--long`), and flags (e.g. `--flag=argument`), are included in the user's command. More information is available in the Syntax Validation documentation.

- Let's add a command that has the following syntax:

```
spameggs print [--meatsubstitute] <arg> [--overeasy] <arg>
```

- To do this, create a new module called `seprinter` in the path `spameggs/lib/spameggs/commands` with the following code:

```
#!/usr/bin/env python
# encoding: utf-8
# filename: seprinter.py

from Naked.toolshed.ink import Template, Renderer

class SpamPrinter:
    def __init__(self, the_meatsub, the_egg):
        self.meatsubstitute = the_meatsub
        self.egg = the_egg
        self.template_string = "{{spamtag}} and {{eggtag}}"

    def run(self):
        template = Template(self.template_string)
        r = Renderer(template, {'spamtag': self.meatsubstitute, 'eggtag': self.egg})
        spam_egg_string = r.render()
        print(spam_egg_string)


if __name__ == '__main__':
    pass
```

An instance of the SpamPrinter class is created with `the_meatsub` and `the_egg` arguments and these are used to define instance properties that we subsequently use in the run() method.

Note how we imported the Naked toolshed library code for the Ink templating system in the command module code. A Template instance is created from the template_string property on our SpamPrinter and it is rendered by passing a dictionary argument with keys that correspond to the strings inside your Template replacement tags. The dictionary values are used to replace the corresponding tags in the template. The opening `{{` and closing `}}` tags are the Ink template defaults.

Any component of the Naked toolshed library can be imported for use in your project with standard Python imports. Use the path, `Naked.toolshed.<MODULE>`, or for the compiled C versions of the library `Naked.toolshed.c.<MODULE>` (requires the C source files to be compiled first!).

Learn More

- The Naked toolshed library overview
- The Toolshed Ink Module
- The toolshed library documentation is in progress. Hold tight! It is coming soon...

2.5.6 Handle Command Line Arguments for Your Application

- Now let's implement the command line argument handling. Open the `spameggs/lib/spameggs/app.py` file in your editor and add the following to the PRIMARY COMMAND LOGIC code block:

```
elif c.cmd == 'print':
    if c.option('--meatsubstitute') and c.option('--overeasy'):
        from spameggs.commands.seprinter import SpamPrinter
        the_meat = c.arg('--meatsubstitute')
        the_eggs = c.arg('--overeasy')
        if state.py2:
            printer = SpamPrinter(the_meat, the_eggs)
        else:
            printer = SpamPrinter(the_meat.upper(), the_eggs.upper())
        printer.run()

    else:
        print("It would be extremely helpful if you enter '-- meatsubstitute Spam --
↪overeasy Eggs' for the example.")
```

Warning: Notice that we used 'elif' rather than if. This logic is in sequence with the help, usage, and version tests that were included in your script above this level. If you remove the Naked implementation of these commands and handle them yourself, make sure that you switch your first statement in the command tests to an 'if' statement.

Note how the Naked parser handles user entered arguments on the command line. The primary command becomes an attribute of the `c` Command object that was instantiated earlier in the script. `cmd` is the first positional argument to the executable (i.e. the primary command). See the link in the Learn More section below to view all of the available argument attributes and to learn how to use `naked args` to help plan your command logic tests with the Naked parser.

We begin by testing that the user entered the primary command 'print' (i.e. `spameggs print ...`). If it was submitted, then we test for the presence of both of the options that are required to prepare our string. The `option()` method returns a boolean for the question, "is the option argument that is passed to this method present?". If these tests return True, the SpamPrinter object that we just developed is imported from the commands directory. The arguments to these options that the user listed on the command line are retrieved with the `arg()` method of the Command object. In this case, we assign them to local variables for clarity.

Next, we meet another branch in the logic that demonstrates one of the features of the Naked toolshed library StateObject (the instance is named 'state') that was automatically generated by `naked` when the project was built. This object collects a number of user state attributes at instantiation, including the version of the Python interpreter that they are using which we test for in the `if state.py2:` statement. For Python 2 interpreters, we print the arguments to the `meatsubstitute` and `overeasy` options as is, and for Python 3 interpreters, we print them in all caps (with the `string.upper()` function).

Lastly, our run() method is called which executes the template replacements and prints the string to the standard output stream.

Let's give it a shot. Try the following from your command line:

```
spameggs print --meatsubstitute Spam --overeasy Eggs
```

If you are using Python 2.x, you should see `Spam and Eggs` in your terminal and if you are using Python 3.x, you should see `SPAM and EGGS`.

The following areas of the documentation are helpful if you would like to delve into more detailed treatment of the parser.

Learn More

- How the Command Parser Works
- How to Import the Command Parser
- How to Instantiate a Command Object
- How to Handle Primary and Secondary Commands
- How to Handle Options
- How to Retrieve the Values for Arguments to Options
- The List of All Command Object Attributes
- Get Help with Your Command Parsing Logic Using the naked args Command

2.5.7 Create Your Help Text

Now that we have an application, let's help our users out by providing some documentation when they request it with either `spameggs --help`, `spameggs -h`, or `spameggs help`. There is no need to add anything to the app.py file in order to handle these requests. The `naked make` build takes care of that for you.

Open your `spameggs/lib/spameggs/settings.py` file in an editor and locate the help variable. Add your help text like this:

```
help = """
--------------------------
spameggs
Copyright 2014 Guido
MIT license
--------------------------

ABOUT
  spameggs is a Python application that tells you about spam.  And it tells you about␣
↪eggs.  Pipe it to whatever application might find that to be useful.

USAGE
  spameggs [print] [--meatsubstitute] <arg> [--overeasy] <arg>

OPTIONS
   --meatsubstitute      Takes a delectable meat substitute as an argument
   --overeasy            Takes an avian object as an argument
"""
```

and then give it a try:

```
spameggs --help
```

Learn More

- The help, usage, and version commands
- How to Set Your Help Text
- How to Remove the Help Command Created by naked make

2.5.8 Create Your Usage Text

To set your usage text, locate the usage variable in the `spameggs/lib/spameggs/settings.py` file that we just used above. Let's add the usage string that we just used in the help text:

```
usage = """
Usage: spameggs [print] [--meatsubstitute] <arg> [--overeasy] <arg>
"""
```

Then confirm that it works with:

```
spameggs --usage
```

Learn More

- The help, usage, and version commands
- How to Set Your Usage Text
- How to Remove the Usage Command Created by naked make

2.5.9 Testing with `naked test`

Time to unit test. Let's set up a tox.ini file to test this in multiple versions of Python with the nose unit test runner. If you are following along, both of these applications need to be installed in order to run the tests. You can install them with pip:

```
$ pip install tox
$ pip install nose
```

In the top directory of your project (where your setup.py file is located), save the following in a file named `tox.ini`:

```
[tox]
envlist = py27,py33
[testenv]
deps=nose
commands=nosetests \
         "--where=tests"
```

This instructs tox to run the unit tests in our `tests` directory using the `nosetests` executable with our installed Python 2.7.x and Python 3.3.x versions (*Note*: both versions need to be installed locally to run these tests). Refer to the tox documentation for instructions on how to test with other Python versions (including pypy).

Next, create a unit test file named `test_spameggs.py` in the tests directory:

```
#!/usr/bin/env python
# coding=utf-8
# file: test_spameggs.py

import unittest
from spameggs.commands.seprinter import SpamPrinter

class SpamEggsTest(unittest.TestCase):

    def setUp(self):
        self.test_string = "{{spamtag}} and {{eggtag}}"
```

```
        self.template_string = SpamPrinter('Spam', 'Eggs').template_string

    def spam_eggs_test(self):
        """A test of spam, and of eggs"""
        self.assertEqual(self.test_string, self.template_string)
```

This test confirms that the template string is what we expect it to be and serves as a simple example. From any directory in your project, run the following:

```
naked test tox
```

This will launch tox and run the tests in Python 2.7 and 3.3 according to your specifications in the tox.ini file. Confirm that they both pass and then we'll move on.

The `test` command also works with py.test and the built-in Python unittest test runner. Click through the link below for more information.

Learn More

The Naked Test Command

2.5.10 Profiling with `naked profile`

Open the `spameggs/lib/profiler.py` file in your editor. The file is stubbed with all of the source that you need to profile with cProfile and pstats. The setup and profiled code blocks are indicated in the file. You can enter the code that you intend to profile in the block below the `pr.enable()` statement:

```
#!/usr/bin/env python
# encoding: utf-8

import cProfile, pstats, StringIO

def profile():
    #--------------------------------------------------------------------------
    # Setup a profile
    #--------------------------------------------------------------------------
    pr = cProfile.Profile()
    #--------------------------------------------------------------------------
    # Enter setup code below
    #--------------------------------------------------------------------------
    from spameggs.commands.seprinter import SpamPrinter


    #--------------------------------------------------------------------------
    # Start profiler
    #--------------------------------------------------------------------------
    pr.enable()

    #--------------------------------------------------------------------------
    # BEGIN profiled code block
    #--------------------------------------------------------------------------
    for x in range(10000):
        sp = SpamPrinter('Spam', 'Eggs')
        sp.run()
```

```
    #--------------------------------------------------------------------------
    # END profiled code block
    #--------------------------------------------------------------------------
    pr.disable()
    s = StringIO.StringIO()
    sortby = 'cumulative'
    ps = pstats.Stats(pr, stream=s).sort_stats(sortby)
    ps.strip_dirs().sort_stats("time").print_stats()
    print(s.getvalue())

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

Then use the following command from any directory in your project:

```
naked profile
```

Naked will run the profiler.py file script and your report will be displayed in the terminal.

Learn More

The Profiler Command

2.5.11 Distribution to PyPI with `naked dist`

Warning: The following set of instructions are intended to demonstrate how you would distribute this application to PyPI. If you run them, be aware that you will actually push spameggs to PyPI. While this will instantly improve your reputation in the Python community, it is likely not what you intend to do.

Complete Your setup.py File

For an application that you really intend to release, you will need to fill in the remainder of the fields in your `setup.py` file before you perform the next steps. Refer to the Python documentation for more information.

If you use the Naked toolshed library in your projects (including the command line parser), Naked should be listed as a dependency in your setup.py file with a line like this:

```
install_requires=['Naked'],
```

Verify Your Release Version Number

Confirm that the version number in your `spameggs/lib/spameggs/settings.py` file is set to the correct release. This is imported into your setup.py file as the release version number and then pushed to PyPI where it becomes the current release version for your project.

Complete Your README.rst File

The `spameggs/docs/README.rst` file is imported into your `setup.py` file as the long description for your project and then pushed to PyPI where it serves as the project description on your application page. In this project that

would be located at `http://pypi.python.org/pypi/spameggs`.

Fill in any details that you would like to display to potential users in this file. You can use reStructuredText in the file and this will be converted to valid HTML by the PyPI servers.

Register

To register your application on PyPI enter the following:

```
naked dist register
```

If you have not previously registered an account on PyPI, use the prompts to do so now. Otherwise, enter your account details. When this command completes, your application will be registered.

Push to PyPI

You can push versions of your application to PyPI with the `naked dist` command as well. There are secondary commands for various distribution types. Let's push both a Python wheel and source distribution:

```
naked dist swheel
```

See the `dist` command documentation link below for more information about the available release types. When the command completes, your release will be live in the remote PyPI repository and ready to be installed by the masses.

You can provide future users with install instructions using pip and non-pip approaches:

Install Instructions for Users WITH `pip`

```
pip install <executable>
```

This command pulls your project source from the PyPI repository (by default) and automatically installs it on the user's system.

Install Instructions for Users WITHOUT `pip`

Instruct your users to download your source code from your remote repository, unpack the source archive, and navigate to the top level directory of the project (where `setup.py` is located). Then provide them with instructions to enter the following:

```
python setup.py install
```

Learn More

- The Dist Command
- The Classify Command

There you have it. You started with an empty directory and ended with a push of your first release to PyPI. Now go create something great.

2.6 Naked Executable

The naked executable is a command line tool for application development, testing, profiling, and deployment. It is distributed with the Naked framework install.

The primary commands include:

- *The Args Command* - View parsed command strings and truth tests
- *The Build Command* - Compile Naked C library code
- *The Classify Command* - Search the PyPI application classifier list by keyword
- *The Dist Command* - Project deployment
- *The Locate Command* - Locate important project files
- *The Make Command* - Generate a new project
- *The Profile Command* - Project profiling
- *The pyh Command* - Help for built-in Python modules, classes, methods & functions
- *The Test Command* - Project unit testing

2.6.1 The Args Command

The `args` command will help you design your command syntax logic with the Naked parser. Pass a complete command example as an argument to the command and it will display every parsed attribute, the truth testing for options and flags, and the result of argument assignments to options and flags.

Args Command Usage

```
naked args 'testapp save somestring --unicode -s --name=file.txt'
```

You can see an example of the output in the Command Line Parser documentation.

Args Command Help

```
naked args help
```

2.6.2 The Build Command

Note: The build command requires an installed C compiler. Naked does not install a compiler or confirm that one is installed on the user's system.

The Naked C toolshed library can be compiled from the C source code files with the *build* command. Navigate to any level of your project directory and use the command:

Build Command Usage

```
naked build
```

This will compile the C library files in the `Naked.toolshed.c.<module> path. See the library documentation for more information about the available Naked toolshed modules.

Build Command Help

Help is available with:

```
naked build help
```

2.6.3 The Classify Command

The classify command attempts to match a user submitted keyword to classifiers in the PyPI application classifier list. These project classifiers categorize your project in the PyPI application listings and should be included in your `setup.py` file prior to distribution to PyPI.

Classify Command Usage

```
naked classify [keyword query]
```

The keyword query is optional. If you do not enter a query term, you will receive the entire classifier list. When you enter a query term, `naked` attempts to match items in the classifier list in a case-insensitive manner.

Classify Command Example

```
$ naked classify HTTP
•naked• Pulling the classifier list from python.org...
•naked• Performing a case insensitive search for 'HTTP'

Topic :: Internet :: WWW/HTTP
Topic :: Internet :: WWW/HTTP :: Browsers
Topic :: Internet :: WWW/HTTP :: Dynamic Content
Topic :: Internet :: WWW/HTTP :: Dynamic Content :: CGI Tools/Libraries
Topic :: Internet :: WWW/HTTP :: Dynamic Content :: Message Boards
Topic :: Internet :: WWW/HTTP :: Dynamic Content :: News/Diary
Topic :: Internet :: WWW/HTTP :: Dynamic Content :: Page Counters
Topic :: Internet :: WWW/HTTP :: HTTP Servers
Topic :: Internet :: WWW/HTTP :: Indexing/Search
Topic :: Internet :: WWW/HTTP :: Session
Topic :: Internet :: WWW/HTTP :: Site Management
Topic :: Internet :: WWW/HTTP :: Site Management :: Link Checking
Topic :: Internet :: WWW/HTTP :: WSGI
Topic :: Internet :: WWW/HTTP :: WSGI :: Application
Topic :: Internet :: WWW/HTTP :: WSGI :: Middleware
Topic :: Internet :: WWW/HTTP :: WSGI :: Server
```

2.6.4 The Dist Command

The dist command assists with distribution of your project to the Python Package Index (PyPI). This command can be used from any working directory in your Naked project.

The available secondary commands include:

all

The `all` secondary command builds a source distribution, wheel distribution, and Windows installer distribution by running the distutils command `python setup.py sdist bdist_wheel bdist_wininst upload`. It is run with the following command:

```
naked dist all
```

register

The `register` secondary command registers your Python project with PyPI. This is a mandatory first step to distribute your project through PyPI and should be the first dist secondary command that you use for new project releases. It is not necessary to run this again after the initial registration.

`register` runs the distutils command `python setup.py register` and is run with:

```
naked dist register
```

If you have not registered a project on PyPI from your local system before, you will receive prompts for your PyPI account information.

sdist

The `sdist` secondary command prepares a source distribution for your current release and pushes it to PyPI. This is performed by running the command `python setup.py sdist upload` and is run from the command line with:

```
naked dist sdist
```

swheel

The `swheel` secondary command prepares a source distribution and a wheel distribution for your current release and pushes it to PyPI. This is performed by running the command `python setup.py sdist bdist_wheel upload` and is run from the command line with:

```
naked dist swheel
```

wheel

The `wheel` secondary command prepares a wheel distribution for your current release and pushes it to PyPI. This is performed by running the command `python setup.py bdist_wheel upload` and is run from the command line with:

```
naked dist wheel
```

win

The `win` secondary command prepares a Windows installer for your current release and pushes it to PyPI. This is performed by running the command `python setup.py bdist_wininst upload` and is run from the command line with:

```
naked dist win
```

For more information about distutils and these release forms, please refer to the Python documentation.

Dist Command Help

Help is available for the dist command with:

```
naked dist help
```

2.6.5 The Locate Command

The locate command identifies several important file paths in your project. I forget. You forget. It's simply there to help you remember.

The secondary commands are:

main

The main secondary command displays the file path to the project `app.py` file where you main application script is located. You use the command like this:

```
naked locate main
```

setup

The setup secondary command displays the file path to the project `setup.py` file.

```
naked locate setup
```

settings

The settings secondary command displays the file path to the project `settings.py` file. This is where your Naked project settings are located.

```
naked locate settings
```

Locate Command Help

You can get help for the locate command with:

```
naked locate help
```

2.6.6 The Make Command

The *make* command builds the directory tree and project files for a new Naked project. You have the option to configure your project with a YAML settings file `naked.yaml` or via command line prompts.

The file and directory structure for command line parsing logic, command development, testing, profiling/benchmarking, licensing, application documentation, and deployment are included in a new Naked project. Help, version, and usage command handling is automatically implemented for you. Complete the strings that you intend to display to users (in the project `settings.py` file), and standard requests for help (e.g. `<executable> --help`), usage (e.g. `<executable> usage`), and version (e.g. `<executable> --version`) will display the corresponding text. For more information about these automatically generated commands, see *Help, Usage, and Version Commands*.

The goal is to allow you to click and begin coding your project without the tedious setup tasks that are common to many/most new projects.

naked.yaml Settings File Project Generation

The structure of a `naked.yaml` project settings file is:

```
application: <application-name>
developer: <developer-name>
license: <license-name>
```

Here is an example of the `naked.yaml` file for status:

```
application: status
developer: Christopher Simpkins
license: MIT License
```

Save your `naked.yaml` file in the top level of your new project directory and then run the following command in the same directory:

```
naked make
```

Naked will detect the settings file, prompt you to confirm your settings, and then use this information to build the new project. You will have the option to modify your project settings before the project writes to disk.

Command Line Prompt Project Generation

Use the following command syntax to initiate the command line prompts for a new Naked project:

```
naked make <application-name>
```

Naked will then prompt you to enter the developer or organization name and the license type.

Where the Information is Used

Your application name becomes the executable command that is used at the command line and is also the top level of your Python module directory structure for module imports. The information is also used to generate your main application module, LICENSE file, README file, and settings.py file.

You can examine the project file templates in the source repository to see all of the string replacement sites.

The Project License

Naked parses your license response and attempts to generate your project LICENSE file. This is performed with a case-insensitive attempt to match one of the following strings at *the beginning* of your response:

- Apache
- BSD
- GPL
- LGPL
- MIT
- Mozilla

If your license type is identified, the entire text of the license is populated in your LICENSE file with the copyright statement, year, and the developer/organization name that you submitted.

For more information on the structure of a generated Naked project, see *Naked Project Structure*.

Make Command Help

```
naked make help
```

2.6.7 The Profile Command

The profile command runs cProfile and pstats on the code that you enter in the test code block of your PROJECT/lib/profiler.py file.

Usage

```
naked profile
```

The Profile

The default profiler.py file sorts the pstats results with the 'time' argument. You can modify this default in the profiler.py file.

Identification of the profiler.py File

naked performs a bottom up search over up to 6 directory levels from the working directory to identify the `lib/profiler.py` path. Unless you have a deep project directory structure (and are in the bottom of one of these paths), this should allow you to run the command from any directory in your project. It is not necessary for lib to be your working directory.

Profile Command Help

Help is available for the profile command with:

```
naked profile help
```

2.6.8 The pyh Command

The pyh command displays built-in Python module, class, method, or function documentation for a query.

Usage

```
naked pyh <query>
```

Submit a built-in Python module, class, method, or function as the `query`.

Examples

Python Module

```
naked pyh sys
```

Python Class

```
naked pyh dict
```

Python Method

```
naked pyh dict.update
```

Python Function

```
naked pyh max
```

pyh Command Help

```
naked pyh help
```

2.6.9 The Test Command

The test command allows you to run unit tests with the built-in Python unittest module (v2, v3), nose, pytest, or tox. The commands can be run from any directory level in your project (when the tests are located in your PROJECT/tests directory).

Note: Please note that the testing application that you are attempting to use must be installed prior to using these commands. Naked does not confirm that they are present. Please refer to the respective application documentation for install instructions.

Usage

```
naked test <secondary command> [argument]
```

The available secondary commands include:

nose

Runs nosetests on your PROJECT/tests directory

```
naked test nose
```

pytest

Runs py.test on your PROJECT/tests directory

```
naked test pytest
```

tox

Runs tox on your PROJECT/tests directory. This uses your tox.ini file settings by default. To run a specific Python version, pass the **tox Python version argument** to the command (see examples below)

```
naked test tox        #runs tests with Python interpreter versions specified in tox.
↪ini
naked test tox py27   #runs tests with Python v2.7.x interpreter (must be installed)
naked test tox py33   #runs tests with Python v3.3.x interpreter (must be installed)
naked test tox pypy   #runs tests with pypy (installed version, must be installed)
```

unittest

Runs the built-in Python unittest module on the unit testing file that you specify as an argument to the command. The file path argument is mandatory. naked attempts to locate this test runner in your PROJECT/tests directory.

```
naked test unittest test_app.py
```

Identification of the tests Directory

A bottom up search is performed from the working directory over up to 6 directory levels to identify your tests directory. If naked is not able to locate your tests directory, or if your files are in a different location, you will receive a failure message.

Test Command Help

Help is available for the command with:

```
naked test help
```

2.7 Naked Project Structure

2.7.1 A Naked Project

A Naked project is generated with the `naked make` command (Make Command Docs). Here is the structure of the project.

Directory Structure

```
PROJECT---|
      |
      docs---|
      |      LICENSE
      |      |
      |      README.rst
      |
      tests---(__init__.py)
      |
      |
      lib----|
      |      PROJECT-------|
      |      |             commands---(__init__.py)
      |      |             |
      |      profiler.py   app.py
      |                    |
      |                    settings.py
      |
      MANIFEST.in
      |
      README.md
      |
      setup.cfg
      |
      setup.py
```

2.7.2 Directories

commands Directory

The commands directory is there to hold any command modules that you develop. You can import them into your main application script with:

```
import PROJECT.commands.MODULE
```

or

```
from PROJECT.commands.MODULE import OBJECT
```

docs Directory

This directory contains your LICENSE and README.rst files. The `naked` executable will make an attempt to generate a complete license file for you if you use the `naked make` command. See the naked executable documentation for details.

lib Directory

This directory is the top level of your application source code and the site within which your setup.py file is instructed to search for your Python source files. In order to establish your project name as the top level directory for Python imports, the project name is repeated as a sub-directory in the lib directory (with an associated `__init__.py` file). This allows you to perform imports with the syntax:

```
import <PROJECT>.commands.<COMMAND-MODULE>
```

or if you develop a library for other users, imports can be performed with the syntax:

```
from <PROJECT>.<directory>.<module> import <object>
```

The `lib/PROJECT` directory contains:

- **commands** directory : the location for your application command modules (see above)
- **app.py** : your main application script (see below)
- **settings.py** : a project settings script (see below). This is also the location of your help, usage, and version strings if you use the commands that the `naked` executable generates for you with `naked make` (Make Command Docs).

tests Directory

This is an "empty" directory (it actually includes an `__init__.py` file) for your unit tests if you choose to include them.

Note: The `naked test` command expects your unit tests and/or test runners to be in this directory in order to run the tests.

2.7.3 Files

app.py File

The app.py file is the main application script and the start of your script execution. Your application begins execution in the main() function in this module. It is pre-populated with module imports, the Naked command line parser, the Naked StateObject, and the necessary code to implement help, usage, and version commands if you use the `naked make` command to create your project.

LICENSE File

The LICENSE file is generated in the docs directory and the text of the license is automatically inserted if you use `naked make` with one of the supported open source licenses. More details are available in the naked executable documentation.

MANIFEST.in File

A distutils source distribution file include spec file (MANIFEST documentation).

profiler.py File

The profiler.py file is a profiling runner script. Insert the code that you would like to profile in the designated code block and run it with the command `naked profile`. cProfile and pstats are implemented for you.

README.md File

This Markdown file is populated with the name of your project. It is simply there in case you choose to use GitHub as a source repository and would like to display a message that differs from the one in your README.rst file (which ends up as your project description on PyPI). It is safe to remove this file if you do not need it.

README.rst File

The reStructuredText file README.rst is used as the long description of your project in the setup.py file. This text gets pushed to PyPI as your project description on your PyPI project page. You can use the reStructuredText syntax in this file for formatting on PyPI.

Note: Your README.rst file is used to generate the long description of your project in the setup.py file. This becomes the description that is displayed on your PyPI application page if you push to the PyPI repository. You can use reStructuredText in this document.

---

settings.py File

The settings.py file contains project-specific settings. This includes the string variables for your help, usage, and version implementations if you use the default Naked project that is generated with `naked make`.

setup.cfg File

A Python setup configuration file (config documentation).

setup.py File

A Python project distribution & install settings file. `naked make` populates this file with your project name, developer/organization name, and license type. (setup documentation)

2.8 Help, Usage, and Version Commands

2.8.1 Implementation of Commands in a Naked Project

When you create a new Naked project with the `naked make` command, the command line parsing logic for help, usage, and version information requests is implemented for you. This allows users to request this information from your application with any of the following:

Help

```
<executable> -h
```

```
<executable> --help
```

```
<executable> help
```

Usage

```
<executable> --usage
```

```
<executable> usage
```

Version

```
<executable> -v
```

```
<executable> --version
```

```
<executable> version
```

2.8.2 How to Set Your Help Text

The help text is assigned to the `help` variable in the `PROJECT/lib/PROJECT/settings.py` file. Modify this string and save the file. It will be published to the standard output stream for users when they request application help with the above syntax.

Here is an example that includes the initial part of the `naked` executable help text (full version of file):

```
help = """
------------------------------------------------
 Naked
```

```
 A Python command line application framework
 Copyright 2014 Christopher Simpkins
 MIT license
--------------------------------------------------

ABOUT

The Naked framework includes the "naked" executable and the Python toolshed library. ␣
↪The naked executable is a command line tool for application development, testing,␣
↪profiling, and deployment.  The toolshed library contains numerous useful tools for␣
↪application development that can be used through standard Python module imports. ␣
↪These features are detailed in the documentation (link below).

USAGE

The naked executable syntax is:

  naked <primary command> [secondary command] [option(s)] [argument(s)]

# more...
"""
```

Note the triple quote format that allows you to write multi-line strings in Python.

2.8.3 How to Set Your Usage Text

The usage text is assigned to the `usage` variable in the `PROJECT/lib/PROJECT/settings.py` file. Modify this string and save the file. It will be published to the standard output stream for users when they request application usage help with the above syntax.

Here is an example of the string from the `naked` executable (full version of file):

```
usage = """
Usage: naked <primary command> [secondary command] [option(s)] [argument(s)]
--- Use 'naked help' for detailed help ---
"""
```

2.8.4 How to Set Your Version Text

The version text is a concatenated string that is made from the `major_version`, `minor_version`, and `patch_version` strings in the `PROJECT/lib/PROJECT/settings.py` file. These should be set as Python strings by placing quotes around the numerals.

The `settings.py` version variables should look like the following:

```
#-----------------------------------------------------------------------------
# Version Number
#-----------------------------------------------------------------------------
major_version = "0"
minor_version = "1"
patch_version = "0"
```

By default, the version text for an application named 'testapp' is displayed like this:

```
$ testapp --version
  testapp 0.1.0
```

As you increment your version numbers with new releases, the new version will be displayed when a user requests it.

Note: The version number settings in the settings.py file are imported into your setup.py file on new installs and releases to PyPI with the `naked dist` commands. Make sure that they are correct for your release if you intend to use these features.

You can modify the displayed string format in this block of your app.py file:

```
elif c.version():
    from PROJECT.settings import app_name, major_version, minor_version, patch_version
    # ** modify the string below to change the version text that is displayed to the␣
↪user **
    version_display_string = app_name + ' ' + major_version + '.' + minor_version + '.
↪' + patch_version
    print(version_display_string)
    sys.exit(0)
```

For example, to remove the display of a patch version altogether, change the `version_display_string` assignment to:

```
version_display_string = app_name + ' ' + major_version + '.' + minor_version
```

2.8.5 How to Remove the Help, Version, & Usage Commands

These commands are completely optional and are implemented as a convenience. The parsing logic and standard output writes are removed by either commenting out or deleting the following blocks of code in your `app.py` file:

```
if c.help():
    from {{app_name}}.settings import help as {{app_name}}_help
    print({{app_name}}_help)
    sys.exit(0)
elif c.usage():
    from {{app_name}}.settings import usage as {{app_name}}_usage
    print({{app_name}}_usage)
    sys.exit(0)
elif c.version():
    from {{app_name}}.settings import app_name, major_version, minor_version, patch_
↪version
    version_display_string = app_name + ' ' + major_version + '.' + minor_version + '.
↪' + patch_version
    print(version_display_string)
    sys.exit(0)
```

In your project, the `{{app_name}}` template strings are replaced with your application name.

Warning: Since these code blocks are placed above your command logic, make sure that you change your first 'elif' to an 'if' statement if you have already started development below this level.

2.9 Command Line Parser

The Naked framework provides a command line parser that is intended to make the transition from a user command string to a Python object seamless and to make the generated Command object easy to use in your application logic.

2.9.1 How it Works

The command string is parsed into a series of positional and command line syntax specific arguments that are easily accessible through Command object attribute lookups or instance methods. If that statement was as clear as mud, here is an example that walks you through access to the commands, options, and their arguments.

Say you are developing a command suite application that expects a user to control the application with a command syntax like the following:

```
<executable> <primary command> [secondary command] [short option(s)] [long option(s)]␣
↪[argument to option]
```

Let's take a look at how you retrieve the information from the user input.

2.9.2 How to Import the Command Line Parser

The parser is available in the `Naked.commandline` module and can be imported into your `app.py` file (app.py file information in *Naked Project Structure*) with:

```
import sys
import Naked.commandline
```

If you created your project with the `naked make` command, this import is added to the generated `app.py` file for you.

2.9.3 How to Instantiate a Command Object

Create an instance of the command line parser with:

```
c = Naked.commandline.Command(sys.argv[0], sys.argv[1:])
```

The class is instantiated with two arguments. The name of your executable and the remainder of the command line string. The Python sys module takes care of both of these arguments for you.

Note: Import the Python sys module in your `app.py` file in order to pass the entire command line string to the Naked Command constructor (as shown above)

2.9.4 The Primary and Secondary Commands

The parser creates a new attribute from the first positional argument to the executable that is named `cmd` (for the primary command) and an attribute for the second positional argument that is named `cmd2` (for the secondary or sub-command). Assuming that you call your Command object instance, 'c', as I demonstrated above, these commands are accessible with the following attribute lookups:

```
primary_command = c.cmd
secondary_command = c.cmd2
```

And you can test for the presence of a specific command in the same fashion:

```
if c.cmd == "command1":
        # do something
elif c.cmd == "command2":
        # do something else
elif c.cmd == "command3":
        # do yet another thing
```

The secondary command can be inserted into the application logic for each primary command like so:

```
if c.cmd == "command1":
        if c.cmd2 == "sub_command1":
                # do command1 branch 1
        elif c.cmd2 == "sub_command2":
                # do command1 branch 2
        elif c.cmd2 == "sub_command3":
                # do command1 branch 3
```

2.9.5 Options

For the purposes of this discussion, I am going to call an option that looks like this `-s` a short option, one that looks like this `--long` a long option, and one that has the following appearance `--flag=argument` a flag.

The parser identifies options by the presence of the first '-' symbol in the string. You can test for the presence of these option forms with a Command object method.

For exclusive options:

```
if c.option('-s') or c.option('--something'):
        # the user indicated this option, handle it
elif c.option('-e') or c.option('--else'):
        # the user indicated this option, handle it
```

For non-exclusive, independent options:

```
if c.option('-s') or c.option('--something'):
        # the user indicated this option, handle it
if c.option('-e') or c.option('--else'):
        # the user indicated this option, handle it
```

For non-exclusive, dependent options:

```
if c.option('-s') or c.option('--something'):
        if c.option('-e') or c.option('--else'):
                # the user indicated both options, handle them
```

The presence of a flag (as you'll recall, an option that looks like this `--flag=argument`) is tested for with the `flag()` method:

```
if c.flag('--flag'):
        argument = c.flag_arg('--flag') # more information below on arguments!
```

Test for the Existence of Short and Long Options

To determine whether there were one or more options in the command that the user submitted, use either of the following tests that return a boolean:

Method Approach

```
if c.option_exists():
        # there is at least one short option, long option, or flag in command
else:
        # there are no options
```

Attribute Approach

```
if c.options:
        # there is at least one short option, long option, or flag in command
else:
        # there are no options
```

Test for the Existence of Flags

Flags are a subset of options. The above option tests will always return True if this test is True.

Method Approach

```
if c.flags_exists():
        # at least one flag was present in the command
else:
        # no flags were present in the command
```

Attribute Approach

```
if c.flags:
        # at least one flag was present in the command
else:
        # no flags were present in the command
```

2.9.6 Arguments to Options

Arguments to the options are retrieved with the `arg()` method for short and long options, and with the `flag_arg()` method for flags. These methods return a string that contains the n+1 positional argument relative to the option name that you enter as the method argument, or the string that immediately follows the '=' character for a flag. Here are examples:

For a short option:

```
# user enters '-l python' in the command
arg_value = c.arg('-l')
print(arg_value)  # prints 'python'
```

For a long option:

```
# user enters '--language python' in the command:
arg_value = c.arg('--language')
print(arg_value)  #prints 'python'
```

For a flag:

```
# user enters '--language=python' in the command:
arg_value = c.flag_arg('--language')
print(arg_value)  #prints python
```

2.9.7 Other Available Command Attributes

There is overlap in the naming of the Command object attributes in order to provide a flexible scheme that (hopefully) addresses most command line application needs. For instance, if you are developing an application that does not require primary or secondary commands, and instead takes up to one option after the executable:

```
<executable> [option]
```

then you could use an approach like the following:

```
# Example: <executable> --test
if c.options:
    if c.arg0 == '--test':
                # do something
else:
        # there are no options
```

or alternatively,

```
# Example: <executable> --test
if c.options:
        if c.first == '--test':
                # do something
else:
        # there are no options
```

Here is the list of all available Command object attributes

Attribute	Definition
obj.app	executable path
obj.argv	list of command arguments (excluding the executable)
obj.argc	number of command line arguments (excluding the executable)
obj.arg0	the first positional argument (excluding the executable)
obj.arg1	the second positional argument (excluding the executable)
obj.arg2	the third positional argument (excluding the executable)
obj.arg3	the fourth positional argument (excluding the executable)
obj.arg4	the fifth positional argument (excluding the executable)
obj.first	the first positional argument
obj.second	the second positional argument
obj.third	the third positional argument
obj.fourth	the fourth positional argument
obj.fifth	the fifth positional argument
obj.arglp	the last positional argument
obj.last	the last positional argument
obj.arg_to_exec	the first argument to the executable = obj.arg0
obj.arg_to_cmd	the first argument to a primary command = obj.arg1
obj.cmd	the primary command = first positional argument
obj.cmd2	the secondary command = second positional argument
obj.options	boolean for presence of one or more options
obj.flags	boolean for presence of one or more flags

2.9.8 The naked Executable Args Command

The `naked` executable `args` command will help you design your command syntax logic with the Naked parser. Just pass a complete command example as an argument and the `args` command will display every parsed attribute, the truth testing for options and flags, and the result of argument assignments to options and flags.

Here is an example of how it is used:

```
naked args 'testapp save something --unicode -s --name=file.txt'
```

and the output looks like this:

```
Application
-----------
c.app = testapp

Argument List Length
--------------------
c.argc = 5

Argument List Items
-------------------
c.argobj = ['save', 'something', '--unicode', '-s', '--name=file.txt']

Arguments by Zero Indexed Start Position
----------------------------------------
c.arg0 = save
c.arg1 = something
c.arg2 = --unicode
c.arg3 = -s
c.arg4 = --name=file.txt
```

```
Arguments by Named Position
---------------------------
c.first = save
c.second = something
c.third = --unicode
c.fourth = -s
c.fifth = --name=file.txt

Last Positional Argument
------------------------
c.arglp = --name=file.txt
c.last = --name=file.txt

Primary & Secondary Commands
----------------------------
c.cmd = save
c.cmd2 = something

Option Exists Tests
-------------------
c.option_exists() = True
c.options = True

Option Argument Assignment
--------------------------
c.arg("--unicode") = -s
c.arg("-s") = --name=file.txt

Flag Exists Tests
-----------------
c.flag_exists() = True
c.flags = True

Flag Argument Assignment
------------------------
c.flag_arg("--name") = file.txt
```

2.9.9 Syntax Validation

Two types of command syntax validation are available.

Validation of at Least One Argument

You can confirm that there is at least one argument (including options) passed to the executable with the following:

```
import sys
from Naked.commandline import app_validates_args

if not c.app_validates_args():
        # handle invalid syntax (e.g. print usage)
        sys.exit(1) # exit application with non-zero exit status
```

This test confirms that the argument list length is > 0 (i.e. obj.argc > 0) and returns a boolean value.

Validation of a Primary Command

You can also confirm that there is a primary command that is passed to the executable for command suite style applications. Use a test like this:

```
import sys
from Naked.commandline import command_suite_validates

if not c.command_suite_validates():
        # handle invalid syntax (e.g. print usage)
        sys.exit(1) # exit application with non-zero exit status
```

A primary command is defined as any non-option string (i.e. a string that does not begin with a '-' character). The method returns a boolean value for this test.

2.10 The Toolshed Library Overview

The toolshed library includes standard Python modules and C source files that can be compiled into binaries which Python will import with the standard dot syntax. The C source code is compiled with the `naked build` command (Build Command Documentation).

The library includes the following modules:

2.10.1 Benchmarking Module

Standard Module Import: `Naked.toolshed.benchmarking`

C Module Import: `Naked.toolshed.c.benchmarking`

The `benchmarking` module includes decorators for timed testing of methods and functions over 10 - 1 million repetitions. This includes a decorator that runs a benchmark built-in Python method in sequence with your function or method for comparison.

Documentation: *Toolshed: benchmarking*

2.10.2 Casts Module

Standard Module Import: `Naked.toolshed.casts`

C Module Import: `Naked.toolshed.c.casts`

The `casts` module includes functions that cast built-in Python types to Naked type extensions. This allows you to use the same type casting syntax that Python uses for the built-in types (e.g. the Python str() is xstr() for the Naked XString() type).

Documentation: coming soon...

2.10.3 File Module

Standard Module Import: `Naked.toolshed.file`

C Module Import: `Naked.toolshed.c.file`

The `file` module includes FileWriter and FileReader classes that perform I/O, including simple UTF-8 encoded reads and writes.

Documentation: *Toolshed: file*

2.10.4 Ink Module

Standard Module Import: `Naked.toolshed.ink`

C Module Import: `Naked.toolshed.c.ink`

The `ink` module includes text template and renderer classes to perform flexible text templating with the replacement tag syntax of your choice. A Python dictionary is used to map replacement strings to the replacement tags.

Documentation: *Toolshed: ink*

2.10.5 Network Module

Standard Module Import: `Naked.toolshed.network`

C Module Import: `Naked.toolshed.c.network`

The `network` module includes the HTTP class for simple GET and POST requests with text or binary data. It also supports simple text and binary file writes from GET or POST requests.

Documentation: *Toolshed: network*

2.10.6 Python Module

Standard Module Import: `Naked.toolshed.python`

C Module Import: `Naked.toolshed.c.python`

The `python` module includes Python interpreter version testing functions.

Documentation: *Toolshed: python*

2.10.7 Shell Module

Standard Module Import: `Naked.toolshed.shell`

C Module Import: `Naked.toolshed.c.shell`

The `shell` module includes external system, Ruby, & Node.js subprocess execution methods and environment variable testing methods.

Documentation: *Toolshed: shell*

2.10.8 State Module

Standard Module Import: `Naked.toolshed.state`

C Module Import: `Naked.toolshed.c.cstate`

The `state` (and `cstate` C module - note the change in the naming convention for this module) include the StateObject, an object that automatically generates operating system, user and working directory, Python interpreter, time, & date data on instantiation.

Documentation: *Toolshed: state*

2.10.9 System Module

Standard Module Import: `Naked.toolshed.system`

C Module Import: `Naked.toolshed.c.system`

The `system` module includes functions for file and directory paths, file and directory testing, file extension testing, file listings, file filters, file metadata, and decorators that insert file paths into function and method parameters. It also includes functions for simple printing to the standard output and standard error streams with exit code handling.

Documentation: *Toolshed: system*

2.10.10 Types Module

Standard Module Import: `Naked.toolshed.types`

C Module Import: `Naked.toolshed.c.types`

The `types` module includes extensions to built-in Python dictionary, list, set, frozenset, tuple, heapq, deque, and string classes. These extensions permit assignment of attributes to both mutable and immutable Python types with dictionary key to attribute name mapping in the constructor. Dictionary values are mapped to the attribute value. New methods for use with these common Python types are also available.

Documentation:

- NakedObject Documentation
- XDict Documentation
- XList Documentation
- XMaxHeap Documentation
- XMinHeap Documentation

2.11 Toolshed: `benchmarking`

2.11.1 Import Benchmarking Module

```
import Naked.toolshed.benchmarking
```

2.11.2 Import Benchmarking C Module

```
import Naked.toolshed.c.benchmarking
```

The C module must be compiled before you import it. See the naked build documentation for more information.

2.11.3 Description

The benchmarking module provides decorators for timed method and function testing. The `timer` decorators perform timed runs of a method or function over a specified number of repetitions. The `timer_trials_benchmark` decorators perform multiple timed trials of a method or function (over a specified number of repetitions per trial) and run an arbitrary timed Python method as a benchmark in sequence with the function or method tests.

The Python method that is used as a benchmark is an append of 10 integers to a Python list:

```
for j in range(repetitions):
    for i in range(10):
        L.append(i)
```

This is run over the same number of repetitions that the test method or function is run. Garbage collection is discontinued during all tests performed in this module.

The total duration of the timed run is displayed in the `timer` report.

The duration for each of the 10 trials, the mean duration across the 10 trials (with standard deviation if NumPy is installed), the duration per function or method run, the duration of the benchmark Python method, and a ratio of the duration of the test method or function to the benchmark Python method are displayed in the `timer_trials_benchmark` report.

The default *`timer()`* and *`timer_trials_benchmark()`* decorators perform 100,000 repetitions of the test method or function. Other decorators are available for between 10 and 1 million repetitions.

2.11.4 Classes

None

2.11.5 Decorators

`@Naked.toolshed.benchmarking.`**`timer`**
100,000 repetitions of the decorated function or method

`@Naked.toolshed.benchmarking.`**`timer_10`**
10 repetitions of the decorated function or method

`@Naked.toolshed.benchmarking.`**`timer_100`**
100 repetitions of the decorated function or method

`@Naked.toolshed.benchmarking.`**`timer_1k`**
1,000 repetitions of the decorated function or method

`@Naked.toolshed.benchmarking.`**`timer_10k`**
10,000 repetitions of the decorated function or method

`@Naked.toolshed.benchmarking.`**`timer_1m`**
1 million repetitions of the decorated function or method

`@Naked.toolshed.benchmarking.`**`timer_trials_benchmark`**
10 trials x 100,000 repetitions of the decorated function or method; 100,000 repetitions of the standard Python function

`@Naked.toolshed.benchmarking.`**`timer_trials_benchmark_10`**
10 trials x 10 repetitions of the decorated function or method; 10 repetitions of the standard Python function

`@Naked.toolshed.benchmarking.`**`timer_trials_benchmark_100`**
10 trials x 100 repetitions of the decorated function or method; 100 repetitions of the standard Python function

`@Naked.toolshed.benchmarking.`**`timer_trials_benchmark_1k`**
10 trials x 1,000 repetitions of the decorated function or method; 1,000 repetitions of the standard Python function

@Naked.toolshed.benchmarking.**timer_trials_benchmark_10k**
10 trials x 10,000 repetitions of the decorated function or method; 10,000 repetitions of the standard Python function

@Naked.toolshed.benchmarking.**timer_trials_benchmark_1m**
10 trials x 1 million repetitions of the decorated function or method; 1 million repetitions of the standard Python function

2.11.6 Examples

Timer Tests

```
from Naked.toolshed.benchmarking import timer

@timer
def test_func():
    test_list = []
    for x in range(100):
        test_list.append(x)
```

Example Result:

```
Starting 100000 repetitions of test_func()...
100000 repetitions of test_func : 1.12108016014 sec
```

Benchmark Timer Tests

```
from Naked.toolshed.benchmarking import timer_trials_benchmark

@timer_trials_benchmark
def test_func():
    test_list = []
    for x in range(100):
        test_list.append(x)
```

Example Result:

```
Starting timed trials of test_func().........
Trial 1:    1.12006998062
Trial 2:    1.09371995926
Trial 3:    1.09064292908
Trial 4:    1.09283995628
Trial 5:    1.09147787094
Trial 6:    1.1042740345
Trial 7:    1.10318899155
Trial 8:    1.10284495354
Trial 9:    1.10802388191
Trial 10:   1.10440087318
-------------------------------------------------
Mean for 100000 repetitions: 1.10114834309 sec
Standard Deviation: 0.00872229881241
Mean per repetition: 1.10114834309e-05 sec
Mean for 100000 of benchmark function:0.131036162376 sec
Ratio: 8.4033927972
```

2.12 Toolshed: `file`

2.12.1 Import File Module

```
import Naked.toolshed.file
```

2.12.2 Import File C Module

```
import Naked.toolshed.c.file
```

The C module must be compiled before you import it. See the naked build documentation for more information.

2.12.3 Description

The `file` module includes the *`FileReader`* and *`FileWriter`* classes. These classes include a number of file I/O methods.

2.12.4 Classes

class `Naked.toolshed.file.`**`FileReader`**(*file_path*)

The `FileReader` class is used for local file reads. By default, the methods that deal with text return NFKD normalized UTF-8 encoded strings. In Python 2, these are of the type `unicode`, and in Python 3 they are of the type `string`. It is not necessary to close the file streams after you use these methods.

Parameters **`file_path`** (*`string`*) – The path to the file.

`read`()

Reads a text file with NFKD normalized UTF-8 string encoding. Returns a unicode string in Python 2 and a string in Python 3.

Returns Python 3 `string`, Python 2 `unicode`

`read_as`(*encoding*)

Reads a text file with a specified text encoding type. Use a Python codecs encoding type as the method argument (`encoding`).

Parameters **`encoding`** (*`string`*) – the Python codecs encoding type for the file text

Returns encoding specific Python string type

Raises **`RuntimeError`** – if `encoding` is not specified

`read_bin`()

Reads a binary file and returns a bytes string.

Returns bytes string

`read_gzip`([*encoding*])

Reads a gzip compressed file and returns a bytes string. The `encoding` parameter is optional. Include the parameter if the compressed file is Unicode text file and the method will attempt to decompress and read the file as a NFKD normalized UTF-8 encoded `bytes` string.

Parameters **`encoding`** (*`string`*) – accepts an optional 'utf-8' string parameter if the compressed file is a UTF-8 encoded text file

Returns bytes string

readlines()
Reads a text file by line with NFKD normalized UTF-8 encoding and returns a Python list containing each line of the file mapped to a list item. In Python 2, the lines are of the type unicode and in Python 3 the lines are of the type string.

Returns Python list with each line in the file mapped to a list item. List items are `unicode` in Python 2 and `string` in Python 3

readlines_as(*encoding*)
Reads a text file by line with a specified text encoding type. Returns a Python list containing each line of the file mapped to a list item. Use a Python codecs encoding type as the method argument (`encoding`). The list item types are dependent upon the encoding type that is passed as the parameter.

Parameters **encoding** (*string*) – the Python codecs encoding type for the file text

Returns encoding specific Python string type

class `Naked.toolshed.file.`**FileWriter**(*file_path*)
The `FileWriter` class is used for local file writes. It is not necessary to close the file streams after you use these methods.

Parameters **file_path** (*string*) – The path to the file.

append(*text*)
Append text to an existing text file at `file_path`. The existence of the file at `file_path` is confirmed before the write. If it does not exist, an `IOError` is raised. If the `text` string includes Unicode characters, the `append` method attempts to encode this as NFKD normalized UTF-8 text prior to the append.

Parameters **text** (*string*) – The text to be appended to the existing file string. Unicode encoded strings are acceptable.

Raises **IOError** – if the file located at the `file_path` parameter does not exist.

gzip(*data*[, *compression_level=6*])
Perform gzip compression of `data` with the zlib library and write to a file at `file_path`. The default compression level is 6 (integer range 0 - 9) in order to balance compression level and speed. In most use cases, this approaches maximal compression with a significant reduction in the duration of time necessary to compress the data for the file write compared with the maximal compression setting. Add a `compression_level` parameter to change this setting.

Parameters

- **data** (*string|bytes*) – the string or bytes string to compress and write to the file at `file_path`.
- **compression_level** (*integer*) – the integer value for the compression level. Range is 0=none to 9=maximal.

If the `file_path` string does not include it, `.gz` is added as the file extension to the `file_path` string.

safe_write(*text*)
Write `text` to a text file at `file_path` if `file_path` does not already exist. This method will not overwrite an existing file at the `file_path`. Use the *write()* method to permit overwrites. This method uses the system default encoding. If the `text` string includes Unicode text, the method will attempt to write with NFKD normalized UTF-8 encoding.

Parameters **text** (*string*) – The text to be written to the file at `file_path`.

Returns boolean for file write. `True` = new file write occurred; `False` = file exists and file write did not occur

safe_write_bin (*data*)
Write `data` to a binary file at `file_path` if `file_path` does not already exist. This method will not overwrite an existing file at `file_path`. Use the *write_bin()* method to permit overwrites.

Parameters **data** (*bytes*) – The data to be written to the file at `file_path`.

Returns boolean for file write. `True` = new file write occurred; `False` = file exists and file write did not occur

write (*text*)
Write `text` to a text file with the system default encoding. The `write` method will attempt to write with NFKD normalized UTF-8 encoding if the `text` string includes Unicode text.

Parameters **text** (*string*) – The text to be written to the file at `file_path`.

write_as (*text*, *encoding*)
Write `text` to a text file with the specified `encoding` type. Use a Python codecs encoding type as the second parameter to the method.

Parameters

- **text** (*string*) – the text that is to be written to the file at `file_path`.
- **encoding** (*string*) – the Python codecs string encoding type

Raises **RuntimeError** – if `encoding` is not specified

write_bin (*data*)
Write `data` to a binary file at `file_path`.

Parameters **data** (*bytes*) – The data to be written to the file at `file_path`.

2.12.5 Examples

Create an Instance of a FileReader

```
from Naked.toolshed.file import FileReader

fr = FileReader('textdir/file.txt')
```

Create an Instance of a FileWriter

```
from Naked.toolshed.file import FileWriter

fw = FileWriter('textdir/file.txt')
```

File Read with ASCII Text

```
from Naked.toolshed.file import FileReader

fr = FileReader('textdir/file.txt')
the_text = fr.read()
```

File Write with ASCII Text

```
from Naked.toolshed.file import FileWriter

fw = FileWriter('textdir/file.txt')
text = "A test string"
fw.write(text)
```

File Read with UTF-8 Encoded Unicode Text

```
from Naked.toolshed.file import FileReader

fr = FileReader('textdir/unicode.txt')
u_txt = fr.read()
```

`u_txt` is type `unicode` in Python 2 and type `string` in Python 3.

File Write with UTF-8 Encoded Unicode Text, Python 2

```
from Naked.toolshed.file import FileWriter

fw = FileWriter('textdir/unicode.txt')
u_txt = u'Here are some Tibetan characters  '
fw.write(u_txt)
```

File Write with UTF-8 Encoded Unicode Text, Python 3

```
from Naked.toolshed.file import FileWriter

fw = FileWriter('textdir/unicode.txt')
u_txt = 'Here are some Tibetan characters  '
fw.write(u_txt)
```

File Append with ASCII Text

```
from Naked.toolshed.file import FileWriter

fw = FileWriter('textdir/existingfile.txt')
text = 'And here is some more text for my file.'
fw.append(text)
```

File Append with UTF-8 Encoded Unicode Text, Python 2

```
from Naked.toolshed.file import FileWriter

fw = FileWriter('textdir/existingfile.txt')
u_txt = u'Here are some Tibetan characters  '
fw.append(u_txt)
```

File Append with UTF-8 Encoded Unicode Text, Python 3

```
from Naked.toolshed.file import FileWriter

fw = FileWriter('textdir/existingfile.txt')
u_txt = 'Here are some Tibetan characters  '
fw.append(u_txt)
```

Safe Write Text to a New File (Prevents File Overwrites)

```
from Naked.toolshed.file import FileWriter

fw = FileWriter('textdir/file.txt')
text = 'And here is some more text for my file.'
if fw.safe_write(text):
    # file write occurred
else:
    # file exists and write did not occur
```

File Read with Binary Data

```
from Naked.toolshed.file import FileReader

fr = FileReader('bindir/test.so')
data = fr.read_bin()
```

File Write with Binary Data

```
from Naked.toolshed.file import FileWriter, FileReader

fr = FileReader('bindir/test.so')
fw = FileWriter('otherbindir/test2.so')
data = fr.read_bin()
fw.write_bin(data)
```

Safe Write Binary Data to a New File (Prevents File Overwrites)

```
from Naked.toolshed.file import FileWriter, FileReader

fw = FileWriter('bindir/test.so')
fr = FileReader('otherbindir/test2.so')
data = fr.read_bin()
if fw.safe_write_bin(data):
    # file write occurred
else:
    # file exists and write did not occur
```

gzip Compression and File Write

```
from Naked.toolshed.file import FileWriter

fw = FileWriter('bindir/index.html.gz')
text = '<!DOCTYPE html><html lang="en"><body>Hi there, this is a test</body></html>'
fw.gzip(text)
```

Read gzip Compressed Data from File

```
from Naked.toolshed.file import FileReader

fr = FileReader('bindir/index.html.gz', encoding='utf-8')
data = fr.read_gzip()
```

2.13 Toolshed: `ink`

2.13.1 Import Ink Module

```
import Naked.toolshed.ink
```

2.13.2 Import Ink C Module

```
import Naked.toolshed.c.ink
```

The C module must be compiled before you import it. See the naked build documentation for more information.

2.13.3 Description

The Ink module contains two classes for text templating. The Ink templating syntax is very flexible, allowing you to assign the replacement tag delimiters that you would like to use. The default opening delimiter is `{{` and the default closing delimiter is `}}`.

This allows you to perform replacements in a string such as:

```
template_string = "{{name}} is {{attribute}}"
```

with values from a Python dictionary where the dictionary keys are mapped to the tag name inside the opening and closing delimiters:

```
replacement_dict = {'name': 'Naked', 'attribute': 'neat'}
```

Dictionary values are replaced at every position where a matching replacement tag is identified in the string.

The *Template* class is used to create instances of Ink template strings and the *Renderer* class is used to execute the text replacements in a *Template* instance.

2.13.4 Classes

class Naked.toolshed.ink.**Template**(*template_text [, open_delimiter="{{"] [, close_delimiter="}}"] [, escape_regex=False]*)

The Template class is an extension of the Python string type and you can use any string method on a Template instance. An instance of the Template class is constructed with a string that contains the template text. You have the option to indicate different opening `open_delimiter` and closing `close_delimiter` delimiters as arguments to the constructor if your template uses different characters. If you use special regular expression characters as delimiters, include an `escape_regex=True` argument.

Note: The need to escape special regular expression characters slows the construction of each instance of a Template. This will not significantly influence the running time of your application if you are creating a relatively small number of Template instances. Perform testing to confirm that this does not become significant if you are generating a large number of Template instances with special regular expression character delimiters. This does not apply to the default Ink template delimiters.

There are no public methods for the `Template` class.

class Naked.toolshed.ink.**Renderer**(*Template, key*[, *html_entities=False*])

The Renderer class takes a *Template* argument and a Python dictionary `key` argument. You have the option to escape HTML entities in the replacement text (i.e. the values contained in the Python dictionary `key`) by setting `html_entities=True` on construction of a new Renderer instance.

Dictionary keys are mapped to the replacement tag names (i.e. the string between the replacement delimiters) in the Template and the dictionary values are the strings that are used for text replacements at *every* matching replacement tag position in the Template.

render()

The render() method executes text replacements in the Template instance that was passed as an argument to the Renderer constructor using the key:value mapping in the dictionary that was passed as an argument to the Renderer constructor (see example below).

2.13.5 Examples

Create an Ink Template with Default Delimiters

```
from Naked.toolshed.ink import Template

template_string = "I like {{food}} and {{drink}}"
template = Template(template_string)
```

Create an Ink Template and Specify New Delimiters

```
from Naked.toolshed.ink import Template

template_string = "I like [[food]] and [[drink]]"
template = Template(template_string, open_delimiter="[[", close_delimiter="]]",␣
↪escape_regex=True)
```

Render an Ink Template

```
from Naked.toolshed.ink import Template, Renderer

template_string = "I like {{food}} and {{drink}}"
template = Template(template_string)
template_key = {'food': 'fries', 'drink': 'beer'}
renderer = Renderer(template, template_key)
rendered_text = renderer.render()
print(rendered_text)          # prints "I like fries and beer"
```

2.14 Toolshed: network

2.14.1 Import Network Module

```
import Naked.toolshed.network
```

2.14.2 Import Network C Module

```
import Naked.toolshed.c.network
```

The C module must be compiled before you import it. See the naked build documentation for more information.

2.14.3 Description

The network module contains a HTTP class that supports GET and POST requests. This module is built with the fantastic Python requests library.

2.14.4 Classes

class Naked.toolshed.network.**HTTP**(*url*[, *request_timeout=10*])

The HTTP class is instantiated with a URL string and has a default request timeout of 10 seconds. The protocol (`http://` or `https://`) must be included in the URL string. If the protocol is not included, a `requests.exceptions.MissingSchema` exception will be raised that can be caught and handled in your code. Include a second argument to the constructor with the time in seconds in order to change this request timeout duration.

Parameters

- **url** (*str*) – the URL for the HTTP method request
- **request_timeout** (*int*) – the duration of the request method timeout in seconds (default=10 seconds)

get()
Perform a GET request for text. This method returns the contents of the response as a string that is encoded with the `HTTP.res.encoding` encoding type. The requests library attempts to determine this encoding according to the encoding header in the response. Returns False on connection error (e.g. non-existent URL path).

get_bin()
Perform a GET request for binary data. This method returns the contents of the response as a bytes string type. Returns False on connection error (e.g. non-existent URL path)

get_bin_write_file(*[filepath] [, suppress_output=False] [, overwrite_existing=False]*)
Perform a GET request for binary data and write to disk. Returns True on successful file write. Returns False on unsuccessful writes, including on connection errors (e.g. non-existent URL path)

Parameters

- **filepath** (*str*) – the output file path for the binary file write. If the filepath is not specified, the current working directory path is prepended to the filename in the URL. This path is then used for the file write.
- **suppress_output** (*boolean*) – suppress standard output stream status updates during download (default=False)
- **overwrite_existing** (*boolean*) – overwrite an existing file at the same output filepath (default=False)

get_status_ok()
Perform a GET request and return a boolean value for the statement, "the response status code is in the 200 range". The returned text data can be retrieved from the `HTTP.res.text` attribute and returned binary data can be retrieved from `HTTP.res.content`.

get_txt_write_file(*[filepath] [, suppress_output=False] [, overwrite_existing=False]*)
Perform a GET request for text and write a UTF-8 encoded text file to disk. Returns True on successful file write. Returns False on unsuccessful writes, including on connection errors (e.g. non-existent URL path)

Parameters

- **filepath** (*str*) – the output file path for the text file write. If the filepath is not specified, the current working directory path is prepended to the filename in the URL. This path is then used for the file write.
- **suppress_output** (*boolean*) – suppress standard output stream status updates during download (default=False)
- **overwrite_existing** (*boolean*) – overwrite an existing file at the same output filepath (default=False)

post()
Perform a POST request for text. This method returns the contents of the response as a string that is encoded according to the `HTTP.res.encoding` response type. The requests library attempts to determine this encoding according to the encoding header in the response. Returns False on connection errors (e.g. non-existent URL path).

post_bin()
Perform a POST request for binary data. This method returns the contents of the response as a bytes string type. Returns False on connection errors (e.g. non-existent URL path)

post_bin_write_file(*[filepath] [, suppress_output=False] [, overwrite_existing=False]*)
Perform a POST request for binary data and write to disk. This method returns True on successful file write and returns False on unsuccesful write, including on connection errors (e.g. non-existent URL path)

Parameters

- **filepath** (*str*) – the output file path for the binary file write. If the filepath is not specified, the current working directory path is prepended to the filename in the URL. This path is then used for the file write.
- **suppress_output** (*boolean*) – suppress standard output stream status updates during download (default=False)
- **overwrite_existing** (*boolean*) – overwrite an existing file at the same output filepath (default=False)

post_txt_write_file(*[filepath] [, suppress_output=False] [, overwrite_existing=False]*)
Perform a POST request for text data and write a UTF-8 encoded text file to disk. This method returns True on successful file write and returns False on unsuccesful writes, including on connection errors (e.g. non-existent URL path)

Parameters

- **filepath** (*str*) – the output file path for the text file write. If the filepath is not specified, the current working directory path is prepended to the filename in the URL. This path is then used for the file write.
- **suppress_output** (*boolean*) – suppress standard output stream status updates during download (default=False)
- **overwrite_existing** (*boolean*) – overwrite an existing file at the same output filepath (default=False)

post_status_ok()
Perform a POST request and return a boolean value for the statement, "the response status code is in the 200 range". The returned text data can be retrieved from the `HTTP.res.text` attribute and returned binary data can be retrieved from `HTTP.res.content`.

response()
Return the response object following a HTTP GET or POST request. This is the same object that defines the `HTTP.res` attribute following one of these request types. See The Response Object section below for details about the response object attributes.

2.14.5 The Response Object

The response object `HTTP.res` is available after a successful GET or POST request for a response from a server. The response object and its attributes are accessible either directly with dot syntax or as the return value from the *`HTTP.response()`* method. The attributes of the response object include:

- `content` - bytes string of the returned data
- `encoding` - string encoding applied to the returned text in the text attribute
- `headers` - dictionary of the response headers
- `status_code` - integer response status code
- `text` - text string encoded with the encoding type defined in the encoding attribute

2.14.6 Examples

Instantiation of Default HTTP Object

```
from Naked.toolshed.network import HTTP

http = HTTP('http://httpbin.org/status/200')
```

Instantation of HTTP Object with Adjustment of Request Timeout

```
from Naked.toolshed.network import HTTP

http = HTTP('http://httpbin.org/status/200', request_timeout=20)
```

GET Request for Text

```
from Naked.toolshed.network import HTTP

http = HTTP('https://raw.github.com/chrissimpkins/naked/master/README.md')
if http.get_status_ok():
    text = http.res.text
    # do something with the text
```

GET Request for Text, Alternate Approach

```
from Naked.toolshed.network import HTTP

http = HTTP('https://raw.github.com/chrissimpkins/naked/master/README.md')
resp = http.get()
if resp:
    # do something with the `resp` text
```

GET Request for Binary Data

```
from Naked.toolshed.network import HTTP

http = HTTP('https://github.com/chrissimpkins/naked/tarball/master')
if http.get_status_ok():
    data = http.res.content
    # do something with the data
```

GET Request for Binary Data, Alternate Approach

```
from Naked.toolshed.network import HTTP

http = HTTP('https://github.com/chrissimpkins/naked/tarball/master')
resp = http.get_bin()
if resp:
    # do something with the `resp` data
```

POST Request for Text

```
from Naked.toolshed.network import HTTP

http = HTTP('http://httpbin.org/post')
if http.post_status_ok():
    text = http.res.text
    # do something with the text
```

POST Request for Text, Alternate Approach

```
from Naked.toolshed.network import HTTP

http = HTTP('http://httpbin.org/post')
resp = http.post()
if resp:
    # do something with the `resp` text
```

POST Request for Binary Data

```
from Naked.toolshed.network import HTTP

http = HTTP('http://httpbin.org/post')
if http.post_status_ok():
    data = http.res.content
    # do something with the data
```

POST Request for Binary Data, Alternate Approach

```
from Naked.toolshed.network import HTTP

http = HTTP('http://httpbin.org/post')
resp = http.post_bin()
if resp:
    # do something with the `resp` data
```

Write Text File from GET Request

```
import os
from Naked.toolshed.network import HTTP

filepath = os.path.join('test', 'naked_README.md')
http = HTTP('https://raw.github.com/chrissimpkins/naked/master/README.md')
if http.get_txt_write_file(filepath):
    # file write was successful
else:
    # file write was not successful
```

Write Binary File from GET Request

```
import os
from Naked.toolshed.network import HTTP

filepath = os.path.join('test', 'naked.tar.gz')
http = HTTP('https://github.com/chrissimpkins/naked/tarball/master')
if http.get_bin_write_file(filepath):
    # file write was successful
else:
    # file write was not successful
```

Write Text File from GET Request, Overwrite Existing File

```
import os
from Naked.toolshed.network import HTTP

filepath = os.path.join('test', 'naked_README.md')
http = HTTP('https://raw.github.com/chrissimpkins/naked/master/README.md')
```

```
if http.get_txt_write_file(filepath, overwrite_existing=True):
    # file write was successful
else:
    # file write was not successful
```

Write Binary File from GET Request, Overwrite Existing File

```
import os
from Naked.toolshed.network import HTTP

filepath = os.path.join('test', 'naked.tar.gz')
http = HTTP('https://github.com/chrissimpkins/naked/tarball/master')
if http.get_bin_write_file(filepath, overwrite_existing=True):
    # file write was successful
else:
    # file write was not successful
```

Response Headers

```
from Naked.toolshed.network import HTTP

http = HTTP('http://httpbin.org/status/200')
if http.get_status_ok():
    headers = http.res.headers
```

Response Status Code, 200 Range

```
from Naked.toolshed.network import HTTP

http = HTTP('http://httpbin.org/status/200')
if http.get_status_ok():
    status = http.res.status_code
```

Response Status Code, Non-200 Range

```
from Naked.toolshed.network import HTTP

http = HTTP('http://httpbin.org/status/404')
if http.get_status_ok():
    status = http.res.status_code
else:
    fail_status = http.res.status_code
```

Response Content-Type

```
from Naked.toolshed.network import HTTP

http = HTTP('http://httpbin.org/status/200')
if http.get_status_ok():
    content_type = http.res.headers['content-type']
```

2.15 Toolshed: python

2.15.1 Import Python Module

```
import Naked.toolshed.state
```

2.15.2 Import Python C Module

```
import Naked.toolshed.c.python
```

The C module must be compiled before you import it. See the naked build documentation for more information.

2.15.3 Description

The python module provides functions for Python version testing. It is used internally in the Naked Framework by the *Naked.toolshed.state.StateObject*. The functions are public if you would like to use them directly.

2.15.4 Classes

None

2.15.5 Functions

Naked.toolshed.python.**is_py2**()
Truth test for execution of script with Python version 2 interpreter

Return type boolean

Naked.toolshed.python.**is_py3**()
Truth test for execution of script with Python version 3 interpreter

Return type boolean

Naked.toolshed.python.**py_version**()
Full Python version tuple.

Return type tuple (major, minor, patch)

Naked.toolshed.python.**py_major_version**()
The major version of the Python interpreter

Return type int

Naked.toolshed.python.**py_minor_version**()
The minor version of the Python interpreter

Return type int

Naked.toolshed.python.**py_patch_version**()
The patch version of the Python interpreter

Return type int

2.16 Toolshed: shell

2.16.1 Import Shell Module

```
import Naked.toolshed.shell
```

2.16.2 Import Shell C Module

```
import Naked.toolshed.c.shell
```

The C module must be compiled before you import it. See the naked build documentation for more information.

2.16.3 Description

The shell module includes functions for the execution of system executables and scripts, and the *Environment* class for access to shell environment variables.

2.16.4 Functions for the Execution of System Commands

Naked.toolshed.shell.**execute**(*command*)

The execute function runs a shell command as a new process with the Python method subprocess.call(). The standard output or standard error stream data are displayed in the terminal immediately and **are not** returned to the calling function/method, rather the success or failure of the command execution is returned.

Parameters **command** (*string*) – the complete command that is to be executed by the shell

Return type (boolean) Defined as True = zero exit status code, False = non-zero exit status code

Naked.toolshed.shell.**muterun**(*command*)

The muterun function runs a shell command as a new process with the Python method subprocess.check_output(). There is no display of data in the terminal from the standard output or standard error streams of the executed command. Instead, the content of these streams is returned to the calling code as a generic NakedObject with the standard output stream data, standard error stream data, and exit code mapped to the attributes NakedObject.stdout, NakedObject.stderr, and NakedObject.exitcode, respectively. These can be accessed with standard Python dot syntax and handled in your own code.

Parameters **command** (*string*) – the complete command that is to be executed by the shell

Return type

NakedObject

NakedObject.**stdout**
(bytes string) The stdout attribute of the returned NakedObject contains the standard output stream data on success and an empty bytes string on failure of the executed command.

NakedObject.**stderr**
(bytes string) The stderr attribute of the returned NakedObject contains the standard error stream data on failure and an empty bytes string on success of the executed command

NakedObject.**exitcode**
(integer) The exitcode attribute of the returned NakedObject contains the exit status code that is returned from the executed command. This can be used to test for the success or failure of the command. See examples below.

Naked.toolshed.shell.**run**(*command*, *suppress_stdout=False*, *suppress_stderr=False*, *suppress_exit_status_call=True*)

The run function provides a flexible approach to the execution of a shell command. As with the execute() and muterun() functions, a complete command string is provided as the first parameter to the function. It differs from the other functions in that there are options to suppress prints of standard output and standard error streams prints to the terminal by the executed command, and to suppress the raise of a SystemExit on return of a non-zero exit status code from the executed command (or from the shell if the executable was absent). You can use different permutations of these parameter settings to determine how much of the executable output is displayed to the user. Furthermore, you can permit the executable to return a non-zero exit status code which will terminate execution of your Python script.

Parameters

- **command** (*string*) – the complete command that is to be executed by the shell
- **suppress_stdout** (*boolean*) – *optional*, suppress print of standard output to the terminal (default = False)
- **suppress_stderr** (*boolean*) – *optional*, suppress print of standard error to the terminal (default = False)
- **suppress_exit_status_call** (*boolean*) – *optional*, suppress raise of SystemExit for non-zero exit status codes from the executed command (default = True). When set to True, your Python script is able to continue execution despite failure of the shell command.

Return type (bytes string or boolean) returns string containing standard output stream data on command execution success (irrespective of suppress_stdout setting), False on non-zero exit status code returned by the shell command (irrespective of the suppress_stderr setting). The suppress_<stream> settings only affect the diplay of these data streams in the user's terminal.

2.16.5 JavaScript (Node.js) Execution Functions

Naked.toolshed.shell.**execute_js**(*file_path*, *arguments=""*)

The execute_js() function runs the *execute()* function on a Node.js script file. Instead of passing the command to be executed as the first parameter, pass a Node.js script filepath as the first parameter and any additional command arguments as the second parameter (*optional*). The executed command is concatenated from these strings with the following code:

```
if len(arguments) > 0:
    js_command = 'node ' + file_path + " " + arguments
else:
    js_command = 'node ' + file_path
```

Parameters

- **file_path** (*string*) – the filepath to the Node.js script that is to be executed by the shell
- **arguments** (*string*) – *optional*, any additional arguments to be used with your command as demonstrated above.

Naked.toolshed.shell.**muterun_js**(*file_path*, *arguments=""*)

The muterun_js() function runs the *muterun()* function on a Node.js script file. Instead of passing the command to be executed as the first parameter, pass a Node.js script filepath as the first parameter and any additional command arguments as the second parameter (*optional*). The executed command is concatenated from these strings as demonstrated in the *execute_js()* function description above.

Parameters

- **file_path** (*string*) – the filepath to the Node.js script that is to be executed by the shell
- **arguments** (*string*) – *optional*, any additional arguments to be used with your command as demonstrated above.

Naked.toolshed.shell.**run_js**(*file_path*, *arguments=""*)

The run_js() function runs the *run()* function on a Node.js script file. Instead of passing the command to be executed as the first parameter, pass a Node.js script filepath as the first parameter and any additional command arguments as the second parameter (*optional*). The executed command is concatenated from these strings as demonstrated in the *execute_js()* function description above.

Parameters

- **file_path** (*string*) – the filepath to the Node.js script that is to be executed by the shell
- **arguments** (*string*) – *optional*, any additional arguments to be used with your command as demonstrated above.

2.16.6 Ruby Script Execution Functions

Naked.toolshed.shell.**execute_rb**(*file_path*, *arguments=""*)

The execute_rb() function runs the *execute()* function on a Ruby script file. Instead of passing the command to be executed as the first parameter, pass a Ruby script filepath as the first parameter and any additional command arguments as the second parameter (*optional*). The executed command is concatenated from these strings with the following code:

```
if len(arguments) > 0:
    rb_command = 'ruby ' + file_path + " " + arguments
else:
    rb_command = 'ruby ' + file_path
```

Parameters

- **file_path** (*string*) – the filepath to the Ruby script that is to be executed by the shell
- **arguments** (*string*) – *optional*, any additional arguments to be used with your command as demonstrated above.

Naked.toolshed.shell.**muterun_rb**(*file_path*, *arguments=""*)

The muterun_js() function runs the *muterun()* function on a Ruby script file. Instead of passing the command to be executed as the first parameter, pass a Ruby script filepath as the first parameter and any additional command arguments as the second parameter (*optional*). The executed command is concatenated from these strings as demonstrated in the *execute_rb()* function description above.

Parameters

- **file_path** (*string*) – the filepath to the Ruby script that is to be executed by the shell
- **arguments** (*string*) – *optional*, any additional arguments to be used with your command as demonstrated above.

Naked.toolshed.shell.**run_rb**(*file_path*, *arguments=""*)

The run_rb() function runs the *run()* function on a Ruby script file. Instead of passing the command to be executed as the first parameter, pass a Ruby script filepath as the first parameter and any additional command

arguments as the second parameter (*optional*). The executed command is concatenated from these strings as demonstrated in the *execute_rb()* function description above.

Parameters

- **file_path** (*string*) – the filepath to the Ruby script that is to be executed by the shell
- **arguments** (*string*) – *optional*, any additional arguments to be used with your command as demonstrated above.

2.16.7 Shell Command Execution Function Examples

execute() Shell Command

```
from Naked.toolshed.shell import execute

success = execute('curl https://raw.github.com/chrissimpkins/naked/master/README.md')
if success:
    # the command was successful
else:
    # the command failed or the executable was not present)
```

muterun() Shell Command

```
from Naked.toolshed.shell import muterun

response = muterun('curl https://raw.github.com/chrissimpkins/naked/master/README.md')
if response.exitcode == 0:
    # the command was successful, handle the standard output
    standard_out = response.stdout
    print(standard_out)
else:
    # the command failed or the executable was not present, handle the standard error
    standard_err = response.stderr
    exit_code = response.exitcode
    print('Exit Status ' + exit_code + ': ' + standard_err)
```

run() Shell Command, Default

```
from Naked.toolshed.shell import run

success = run('curl https://raw.github.com/chrissimpkins/naked/master/README.md')
if success:
    # the command was successful, automatically prints to standard output
else:
    # the command failed or the executable was not present, automatically prints to␣
↪standard error
```

run() Shell Command, Suppress Standard Output and Standard Error

```
from Naked.toolshed.shell import run

success = run('curl https://raw.github.com/chrissimpkins/naked/master/README.md',␣
↪suppress_stdout=True, suppress_stderr=True)
if success:
    # the command was successful, success contains the data from standard output.
    # standard output is not printed to terminal
else:
```

```
    # the command failed or the executable was not present, success contains False
    # standard error is not printed to terminal
```

run() Shell Command, Permit SystemExit on Failure

```
from Naked.toolshed.shell import run

success = run('curl http://bogussite.io', suppress_exit_status_call=False)
if success:
    # if the command was successful, this block is executed
else:
    # this command fails (non-existent site), print to standard error and raise␣
↪SystemExit with non-zero exit status code
```

execute_rb() a Ruby Script

```
from Naked.toolshed.shell import execute_rb

success = execute_rb('testscript.rb')
if success:
    # the script run was successful, the standard output was automatically printed to␣
↪terminal
else:
    # the script run failed, the standard error was automatically printed to terminal
```

muterun_rb() a Ruby Script

```
from Naked.toolshed.shell import muterun_rb

response = muterun_rb('testscript.rb')
if response.exitcode == 0:
    # the command was successful, handle the standard output
    standard_out = response.stdout
    print(standard_out)
else:
    # the command failed or the executable was not present, handle the standard error
    standard_err = response.stderr
    exit_code = response.exitcode
    print('Exit Status ' + exit_code + ': ' + standard_err)
```

run_rb() a Ruby Script

```
from Naked.toolshed.shell import run_rb

success = run_rb('testscript.rb')
if success:
    # the script run was successful, standard output automatically printed to␣
↪terminal by default
else:
    # the script run failed, standard error automatically printed to terminal by␣
↪default
    # does not raise SystemExit by default
```

execute_js() a JavaScript (Node.js) Script

```
from Naked.toolshed.shell import execute_js

success = execute_js('testscript.js')
```

```
if success:
    # the script run was successful, the standard output was automatically printed to␣
↪terminal
else:
    # the script run failed, the standard error was automatically printed to terminal
```

muterun_js() a JavaScript (Node.js) Script

```
from Naked.toolshed.shell import muterun_js

response = muterun_js('testscript.js')
if response.exitcode == 0:
    # the command was successful, handle the standard output
    standard_out = response.stdout
    print(standard_out)
else:
    # the command failed or the executable was not present, handle the standard error
    standard_err = response.stderr
    exit_code = response.exitcode
    print('Exit Status ' + exit_code + ': ' + standard_err)
```

run_js() a JavaScript (Node.js) Script

```
from Naked.toolshed.shell import run_js

success = run_js('testscript.js')
if success:
    # the script run was successful, standard output automatically printed to␣
↪terminal by default
else:
    # the script run failed, standard error automatically printed to terminal by␣
↪default
    # does not raise SystemExit by default
```

2.16.8 Environment Class

class Naked.toolshed.shell.**Environment**

The Environment class contains methods that provide access to shell environment variables.

is_var (*variable_name*)

Determine the existence of a shell environment variable.

Parameters **variable_name** (*string*) – the name of the test environment variable

Return type (*boolean*) Boolean value for existence of the environment variable

get_var (*variable_name*)

Return the value of a shell environment variable

Parameters **variable_name** (*string*) – the name of the environment variable

Return type (*string*) Value of the environment variable

get_split_var_list (*variable_name*)

Returns a list of the strings in a shell environment variable assignment list (e.g. PATH).

Parameters **variable_name** (*string*) – the name of the environment variable

Return type (*list*) returns a list of strings that are split by the OS dependent separator symbol or an empty list if the variable is not present

2.16.9 Environment Examples

Create a New Environment Instance

```
from Naked.toolshed.shell import Environment

env = Environment()
```

Test for Environment Variable

```
from Naked.toolshed.shell import Environment

env = Environment()
if (env.is_var('PATH')):
    # the shell environment variable exists
else:
    # the shell environment variable does not exist
```

Get Value of Environment Variable

```
env = Environment()
if (env.is_var('PATH')):
    path_string = env.get_var('PATH')
    print(path_string)
```

Iterate Through List of Environment Variable Strings

```
env = Environment()
if (env.is_var('PATH')):
    for i in env.get_split_var_list('PATH'):
        print(i)
```

2.17 Toolshed: `state`

2.17.1 Import State Module

```
import Naked.toolshed.state
```

2.17.2 Import State C Module

```
import Naked.toolshed.c.cstate
```

Note: Note the difference in the name of the C source module `cstate` relative to other toolshed module imports which follow the same naming scheme as the standard Python version.

The C module must be compiled before you import it. See the naked build documentation for more information.

2.17.3 Description

The state module has a single class, the *StateObject*. This is an object that generates and maintains user state data that is current as of the time of the *StateObject* instantation.

2.17.4 Classes

class Naked.toolshed.state.**StateObject**

The StateObject is instantiated without parameters. Attributes can be accessed with standard Python dot syntax following instantiation (e.g. `state.py2`).

Attributes:

cwd
(*string*) User current working directory (from Python `os.getcwd()`)

day
(*int*) Local day of the calendar month (from `datetime.datetime.now().day`)

default_path
(*string*) Default user PATH string (from Python `os.defpath`)

file_encoding
(*string*) User system default file encoding (from Python `sys.getfilesystemencoding()`)

hour
(*int*) Local system hour of the day [24hr format] (from Python `datetime.datetime.now().hour`)

min
(*int*) Local system minute of the day (from Python `datetime.datetime.now().minute`)

month
(*int*) Local system month of the year (from Python `datetime.datetime.now().month`)

os
(*string*) User operating system (from Python `sys.platform`)

parent_dir
(*string*) User parent directory relative to current working directory (from Python `os.pardir`)

py2
(*boolean*) Truth test for Python 2 interpreter executing script on user system (test derived from Python `sys.version_info`)

py3
(*boolean*) Truth test for Python 3 interpreter executing script on user system (test derived from Python `sys.version_info`)

py_major
(*int*) The Python major version - **2** .7.6 - (from Python `sys.version_info`)

py_minor
(*int*) The Python minor version - 2. **7** .6 - (from Python `sys.version_info`)

py_patch
(*int*) The Python patch version - 2.7. **6** - (from Python `sys.version_info`)

second
(*int*) Local system seconds of the current minute (from Python `datetime.datetime.now().second`)

string_encoding
(*string*) User system string encoding (from Python `sys.getdefaultencoding()`)

user_path
(*string*) User's USER directory path (from Python `os.path.expanduser("~")`)

year
(*int*) Local system year string (from Python `datetime.datetime.now().year`)

2.17.5 Add Your Own Attributes to the StateObject

If you need to maintain additional information, simply add a new attribute to the StateObject:

```
from Naked.toolshed.state import StateObject

state = StateObject()
state.user_name = 'Guido'          # assign a new attribute
state.fav_food = 'spam and eggs'   # assign a new attribute

# do other things

print(state.user_name)    # prints 'Guido'
print(state.fav_food)     # prints 'spam and eggs'
```

There are no restrictions against overwriting an existing attribute in the StateObject if you would like to re-define it.

2.17.6 Examples

If you use `naked make` to generate your project, the *`StateObject`* is instantiated as an instance named `state` in your `app.py` file. If you create the instance of the StateObject in a different file, or implement this yourself in the `app.py` file, replace `state` in the following examples with the name of your instance. You can access the *`StateObject`* data with dot syntax.

Python 2 vs. 3 Test

```
if state.py2:
    # Python 2 code
else:
    # Python 3 code
```

Distinguish Python 2.6 from Python 2.7

```
if state.py2:
    if state.py_minor == 6:
        # Python 2.6 code
    elif state.py_minor == 7:
        # Python 2.7 code
```

Distinguish Python 3.2 from Python 3.3

```
if state.py3:
    if state.py_minor == 2:
        # Python 3.2 code
    elif state.py_minor == 3:
        # Python 3.3 code
```

Current Working Directory Lookup

```
curr_dir = state.cwd
# the current working directory path is now in `curr_dir`
```

User Operating System

```
opsys = state.os
# opsys contains the operating system name - see Python sys.platform documentation␣
↪for key
```

Print the Date

```
date_string = state.month + ' ' + state.day + ' ' + state.year
print(date_string)
```

Print the Time

```
time_string = state.hour + ':' + state.min + ':' + state.second
```

2.18 Toolshed: `system`

2.18.1 Import System Module

```
import Naked.toolshed.system
```

2.18.2 Import System C Module

```
import Naked.toolshed.c.system
```

The C module must be compiled before you import it. See the naked build documentation for more information.

2.18.3 Description

The `system` module includes functions for file and directory paths, file and directory testing, file extension testing, file listings, file filters, file metadata, and decorators that insert file paths into function and method parameters. It also includes functions for simple printing to the standard output and standard error streams with exit code handling.

File I/O methods are available in the `Naked.toolshed.file` and `Naked.toolshed.c.file` modules. Additional information is available in the *Toolshed: file* documentation.

2.18.4 File and Directory Path Functions

`Naked.toolshed.system.`**`cwd`**`()`
Returns the current working directory path.

Returns (string) current working directory path

`Naked.toolshed.system.`**`directory`**`(`*file_path*`)`
Returns the directory path to the file in `file_path`.

Parameters **file_path** (*string*) – the absolute or relative file path to a file

Returns (string) the directory path that contains the file in `file_path`

`Naked.toolshed.system.`**file_extension**(*file_path*)
Returns the file extension from a file path.

Parameters **file_path** (*string*) – the absolute or relative file path to a file

Returns (string) the file extension, including the period character

`Naked.toolshed.system.`**filename**(*file_path*)
Returns the base filename from an absolute `file_path`.

Parameters **file_path** (*string*) – the absolute or relative file path to a file

Returns (string) base file name including the file extension

`Naked.toolshed.system.`**fullpath**(*file_name*)
Returns the absolute path to a file that is in the current working directory, including the basename of the file.

Parameters **file_name** (*string*) – the name of a file in the current working directory

Returns (string) the absolute path to a file that is in the current working directory

`Naked.toolshed.system.`**make_path**(**path_strings*)
Returns the OS independent file path from path component parameters

Parameters ***path_strings** (*string*) – tuple of path component strings

```
from Naked.toolshed.system import make_path

file_path = make_path('user', 'guido', 'python', 'file.txt')
```

Returns (string) the OS independent path from the path component parameters in `*path_strings`

File and Directory Path Examples are available below.

2.18.5 File Path Decorators

`@Naked.toolshed.system.`**currentdir_to_basefile**
Concatenates the absolute working directory path (*string*) to the basename of a file in the first parameter of the decorated function

`@Naked.toolshed.system.`**currentdir_firstparam**
Adds the current working directory path (*string*) as the first parameter of the decorated function

`@Naked.toolshed.system.`**currentdir_lastparam**
Adds the current working directory (*string*) as the last parameter of the decorated function

File Path Decorator Examples are available below.

2.18.6 File and Directory Testing Functions

`Naked.toolshed.system.`**dir_exists**(*dir_path*)
Test for the existence of a directory at the path `dir_path`. This function confirms that the path is a directory and not a symbolic link or file.

Parameters **dir_path** (*string*) – the path to be tested for the presence of a directory

Returns (boolean) `True` = directory exists at `dir_path`; `False` = directory does not exist at `dir_path`

`Naked.toolshed.system.`**`file_exists`**(*file_path*)
Test for the existence of a file at the path `file_path`. This function confirms that the path is a file and not a symbolic link or directory.

Parameters **`file_path`** (*`string`*) – the path to be tested for the presence of a file

Returns (boolean) `True` = file exists at `file_path`; `False` = file does not exist at `file_path`

`Naked.toolshed.system.`**`is_file`**(*file_path*)
Test whether a path resolves to an existing file.

Parameters **`file_path`** (*`string`*) – the path to be tested for a file

Returns (boolean) `True` = the `file_path` path is a file; `False` = the `file_path` path is not a file

`Naked.toolshed.system.`**`is_dir`**(*dir_path*)
Test whether a path resolves to an existing directory.

Parameters **`dir_path`** (*`string`*) – the path to be tested for a directory

Returns (boolean) `True` = the `dir_path` path is a directory; `False` = the `dir_path` path is not a directory

File and Directory Testing Examples are available below.

2.18.7 File Metadata Functions

`Naked.toolshed.system.`**`file_size`**(*file_path*)
Returns the size of the file at `file_path` in bytes.

Parameters **`file_path`** (*`string`*) – the path to the file

Returns (integer) the size of the file in bytes

`Naked.toolshed.system.`**`file_mod_time`**(*file_path*)
Returns the date and time of the last file modification

Parameters **`file_path`** (*`string`*) – the path to the file

Returns (string) The date and time of the last file modification with the format `'Wed Jan 29 23:49:04 2014'`.

File Metadata Examples are available below.

2.18.8 File Listings Functions

`Naked.toolshed.system.`**`list_all_files`**(*dir_path*)
List all files in the path `dir_path`.

Parameters **`dir_path`** (*`string`*) – the directory path containing the files of interest

Returns (list) Python list with each file path in `dir_path` mapped to a list item

`Naked.toolshed.system.`**`list_all_files_cwd`**()
List all files in the current working directory.

Returns (list) Python list with each file path in the current working directory mapped to a list item

Naked.toolshed.system.**list_filter_files**(*extension_filter*, *dir_path*)
List all files in the path `dir_path` that match the file extension filter `extension_filter`. Takes a file extension with or without the associated period character (e.g. `.py` or `py`).

Parameters

- **extension_filter** (*string*) – the file extension filter to be used for file selection
- **dir_path** (*string*) – the directory path containing the files of interest

Returns (list) Python list with each matching file path in `dir_path` mapped to a list item

Naked.toolshed.system.**list_filter_files_cwd**(*extension_filter*)
List all files in the current working directory that match the file extension filter `extension_filter`. Takes a file extension with or without the associated period character (e.g. `.py` or `py`).

Parameters **extension_filter** (*string*) – the file extension filter to be used for file selection

Returns (list) Python list with each matching file path in current working directory mapped to a list item

Naked.toolshed.system.**list_match_pattern**(*match_pattern*)
List all files that match a wildcard `match_pattern` parameter.

Parameters **match_pattern** (*string*) – the wildcard match pattern (e.g. `'/test/*.py'`)

Returns (list) Python list with each matching file path mapped to a list item

File Listings Examples are available below.

2.18.9 Directory Write Function

Naked.toolshed.system.**make_dirs**(*directory_path*)
Writes a new directory path to disk *if it does not already exist*. This function does not overwrite an existing directory path. Will perform a recursive directory tree write for multi-level directory structures. Returns `True` if the directory write is successful. Returns `False` if the directory write does not occur (e.g. requested directory already exists).

Parameters **directory_path** (*string*) – the directory path to be written to disk

Returns (boolean) `True` = successful directory path write; `False` = unsuccessful directory path write

Directory Write Examples are available below.

2.18.10 Symbolic Link Functions

Naked.toolshed.system.**is_link**(*link_path*)
Test for the presence of a symbolic link at `link_path`.

Parameters **link_path** (*string*) – the path to test for the presence of a symbolic link

Returns (boolean) `True` = the path `link_path` is a symbolic link; `False` = the path `link_path` is not a symbolic link

Naked.toolshed.system.**real_path**(*link_path*)
Return the real file path pointed to by the path `link_path`.

Parameters **link_path** (*string*) – the symbolic link path

Returns (string) the real file path pointed to by the symbolic link `link_path`

2.18.11 Data Stream Functions

Naked.toolshed.system.**stdout**(*text*)
Print the `text` string to the standard output stream with a newline character appended to the `text` string. Identical to the Python `print()` function.

Parameters **text** (*string*) – the string that will be printed to the standard output stream

Naked.toolshed.system.**stdout_iter**(*iter*)
Print the items in an iterable object (`iter`) to the standard output stream *with* a newline after each item.

Parameters **iter** (*object*) – An iterable object type in which all of the iterable items provide support for either the `__str__` or `__repr__` functions.

Naked.toolshed.system.**stdout_iter_xnl**(*iter*)
Print the items in an iterable object (`iter`) to the standard output stream *without* a newline character after each item. This prints the items in sequence on the same line of output.

Parameters **iter** (*object*) – An iterable object type in which all of the iterable items provide support for either the `__str__` or `__repr__` functions.

Naked.toolshed.system.**stdout_xnl**(*text*)
Print the `text` string to the standard output stream *without* a newline character appended to the `text` string.

Parameters **text** (*string*) – the string that will be printed to the standard output stream

Naked.toolshed.system.**stderr**(*text*[, *exit*])
Print the `text` string to the standard error stream with an optional non-zero exit status code. A newline character is appended to the `text` string. For non-zero `exit` integers, `SystemExit()` is raised with the exit status code. `SystemExit()` is not raised by default (or if `exit` is assigned a value of 0).

Parameters

- **text** (*string*) – the string that will be printed to the standard error stream
- **exit** (*integer*) – (*optional*) the exit status code

Naked.toolshed.system.**stderr_xnl**(*text*[, *exit*])
Print the `text` string to the standard error stream with an optional non-zero exit status code. This function does not append a newline character to the end of the `text` string before printing it to the standard error stream. If the exit status code is changed to a non-zero integer, `SystemExit()` is raised with the exit status code. `SystemExit()` is not raised by default (or if `exit` is assigned a value of 0).

Parameters

- **text** (*string*) – the string that will be printed to the standard error stream
- **exit** (*integer*) – (*optional*) the exit status code

Data Stream Examples are available below.

2.18.12 Application Exit Functions

Naked.toolshed.system.**exit_failure**()
Exit the application with exit status code 1.

Naked.toolshed.system.**exit_success**()
Exit the application with exit status code 0.

Naked.toolshed.system.**exit_with_status**(*exit_code*)
Exit the application with exit status code `exit_code`.

Parameters **exit_code** (*integer*) – the exit status code. By default, an exit status code of 0 is used.

Application Exit Examples are available below.

2.18.13 File and Directory Path Examples

Current Working Directory

```
from Naked.toolshed.system import cwd

curr_dir = cwd()
```

Make OS Independent Path String

```
from Naked.toolshed.system import make_path

file_path = make_path('path', 'to', 'test.txt')
print(file_path) # prints path with OS dependent path delimiters
```

Directory Path to File

```
from Naked.toolshed.system import directory, make_path

filepath = make_path('path', 'to', 'test.txt')
dir_path = directory(dir_path)
print(dir_path) # prints '/path/to/' with OS dependent delimiters
```

File Extension

```
from Naked.toolshed.system import file_extension, make_path

file_path = make_path('path', 'to', 'test.txt')
extension = file_extension(file_path)
print(extension) # prints '.txt'
```

Filename

```
from Naked.toolshed.system import filename, make_path

file_path = make_path('path', 'to', 'test.txt')
file_name = filename(file_path)
print(file_name) # prints 'test.txt'
```

Absolute File Path

```
from Naked.toolshed.system import fullpath, make_path

# file /usr/c/test/test.txt & current working directory is /usr/c/test/
absolute_path = fullpath('test.txt')
print(absolute_path) # prints '/usr/c/test/test.txt' with OS dependent delimiters
```

2.18.14 File Path Decorator Examples

Current Working Directory Path Concatenation to First Parameter

```
from Naked.toolshed.system import currentdir_to_basefile

@currentdir_to_basefile
def tester(path):
    print(path)

# when run as tester('test.txt') from /usr/c/test/, prints '/usr/c/test/test.txt'␣
↪with OS dependent delimiters
```

Current Working Directory Path as First Parameter

```
from Naked.toolshed.system import currentdir_firstparam

@currentdir_firstparam
def tester(path=''):
    print(path)

# when run as tester() from /usr/c/test/, prints '/usr/c/test/' with OS dependent␣
↪delimiters
```

Current Working Directory Path as Last Parameter

```
from Naked.toolshed.system import currentdir_lastparam

@currentdir_lastparam
def tester(file_name, dir_path=''):
    print(dir_path + file_name)

# when run as tester('test.txt') from /usr/c/test/, prints '/usr/c/test/test.txt'␣
↪with OS dependent delimiters
```

2.18.15 File and Directory Testing Examples

Directory Testing

```
from Naked.toolshed.system import dir_exists, make_path

# /usr/c/test does exist

dir_path = make_path('usr', 'c', 'test')
if dir_exists(dir_path):
    print('yep')  # prints 'yep'
```

File Testing

```
from Naked.toolshed.system import file_exists, make_path

# /usr/c/test/test.txt exists

file_path = make_path('usr', 'c', 'test', 'test.txt')
if file_exists('/usr/c/test/test.txt'):
    print('yep') # prints yep
```

2.18.16 File Metadata Examples

File Size

```
from Naked.toolshed.system import file_size

size = file_size('test.txt')
print(size) # prints size of 'test.txt' in current working directory in bytes
```

File Modification Time and Date

```
from Naked.toolshed.system import file_mod_time

m_time = file_mod_time('test.txt')
print(m_time) # prints 'Wed Jan 29 23:49:04 2014'
```

2.18.17 File Listings Examples

For the following examples, the test directory contains the files: 'test.txt', 'pytest.py', and 'rbtest.rb'

All Files in Current Working Directory

```
from Naked.toolshed.system import list_all_files_cwd

file_list = list_all_files_cwd()
for x in file_list:
    print(x)

# prints:
#   test.txt
#   pytest.py
#   rbtest.rb
```

All Files in Target Directory

```
from Naked.toolshed.system import list_all_files

dir_path = make_path('path', 'to', 'test')
file_list = list_all_files(dir_path)
for x in file_list:
    print(x)

# prints:
#   test.txt
#   pytest.py
#   rbtest.rb
```

Filter Files by File Extension in Current Working Directory

```
from Naked.toolshed.system import list_filter_files_cwd

file_list = list_filter_files_cwd('.py')
for x in file_list:
    print(x)

# prints:
#   pytest.py
```

Filter Files by Wildcard

```
from Naked.toolshed.system import list_match_pattern

file_list = list_match_pattern('./*.py')
for x in file_list:
    print(x)

# prints:
#   pytest.py
```

2.18.18 Directory Write Examples

Make Directory When it Does Not Exist

```
from Naked.toolshed.system import make_dirs

if make_dirs('test'):
    print('success') # prints 'success'
else:
    print('fail')
```

Make Directory When it Does Exist

```
from Naked.toolshed.system import make_dirs

if make_dirs('test'):
    print('success')
else:
    print('fail') # prints 'fail'
```

2.18.19 Data Stream Examples

Standard Output Stream Write, With Newline

```
from Naked.toolshed.system import stdout

stdout('This is a test string')

# prints 'This is a test string\n' to standard output with OS dependent newline␣
↪character(s)
```

Standard Output Stream Write, Without Newline

```
from Naked.toolshed.system import stdout_xnl

stdout_xnl('This is a test string')

# prints 'This is a test string' to standard output
```

Standard Output Stream Write with Iterable Object, With Newline

```
from Naked.toolshed.system import stdout_iter

the_list = ['test', 'this', 'string']
```

```
stdout_iter(the_list)

# prints to standard output:
#   test
#   this
#   string
```

Standard Output Stream Write with Iterable Object, Without Newline

```
from Naked.toolshed.system import stdout_iter_xnl

the_list = ['1', ' ' , '2', ' ' , '3']
stdout_iter_xnl(the_list)

# prints '1 2 3' to standard output
```

Standard Error Stream Write, With Newline, with Exit Status Code 1

```
from Naked.toolshed.system import stderr

stderr("Um, that was an error.", 1)

# prints 'Um, that was an error.\n' to standard output with OS dependent newline␣
↪character(s) and raises SystemExit(1)
```

Standard Error Stream Write, Without Newline, Without SystemExit

```
from Naked.toolshed.system import stderr_xnl

stderr_xnl("Um, that was an error.")

# prints 'Um, that was an error' to standard error and does not raise SystemExit()
```

2.18.20 Application Exit Examples

Exit with Zero Status Code

```
from Naked.toolshed.system import exit_success

# successful code here
exit_success() # raises SystemExit(0)
```

Exit with Status Code 1

```
from Naked.toolshed.system import exit_failure

# failing code here
exit_failure() # raises SystemExit(1)
```

Exit with Any Status Code

```
from Naked.toolshed.system import exit_with_status

# failing code here
exit_with_status(10) # raises SystemExit(10)
```

2.19 Toolshed: types : NakedObject

2.19.1 Import NakedObject

```
from Naked.toolshed.types import NakedObject
```

2.19.2 Import NakedObject from C Module

```
from Naked.toolshed.c.types import NakedObject
```

The C module must be compiled before you import it. See the naked build documentation for more information.

2.19.3 Description

A NakedObject is a generic object that supports equality testing based upon the contents of its attributes.

class Naked.toolshed.types.**NakedObject**([*attribute_dictionary*])

A NakedObject is instantiated with an optional Python dictionary parameter. If the parameter is included, the dictionary keys are mapped to NakedObject attributes and the corresponding dictionary values are used to define the attribute values.

Parameters **attribute_dictionary** (*dictionary*) – (*optional*) a Python dictionary that is used to define the attributes of a new instance of a NakedObject. Key names are mapped to attribute names and their corresponding values are mapped to the attribute values.

equals(*other_object*)

The equals() method performs equality testing between the NakedObject and another object. Equality is defined by the equality of type type(NakedObject()) == type(other_object) and equality of attribute names and values NakedObject().__dict__ == other_object.__dict__. This equality test will therefore fail if the other_object parameter:

- has a different type (e.g. comparison to a string type)
- has fewer attributes
- has more attributes
- has the same number of attributes with different names
- has the same number of attributes with the same names & different values
- has the same number of attributes with the same names, same values, but values are of different types (e.g. '1' vs. 1)

Parameters **other_object** (*object*) – a test object

Returns (boolean) True = the equality conditions are met; False = the equality conditions are not met

Note: The == and != operators can be used in place of the equals() method and the negation of the equals() method, respectively.

2.19.4 Examples

Create a New Instance of an Empty NakedObject

```
from Naked.toolshed.types import NakedObject

obj = NakedObject()
```

Create a New Instance of a NakedObject with Attributes

```
from Naked.toolshed.types import NakedObject

obj = NakedObject({'example': 'an example string'})
```

Determine Type of NakedObject

```
from Naked.toolshed.types import NakedObject

obj = NakedObject({'example': 'an example string'})
print(type(obj)) # prints <class 'Naked.toolshed.types.NakedObject'>
```

Set New Attribute on NakedObject

```
from Naked.toolshed.types import NakedObject

obj = NakedObject()
obj.example = 'an example string'
```

Get Attribute Value from NakedObject

```
from Naked.toolshed.types import NakedObject

obj = NakedObject({'example': 'an example string'})
the_value = obj.example
```

Delete Attribute from NakedObject

```
from Naked.toolshed.types import NakedObject

obj = NakedObject({'example': 'an example string'})
del obj.example
```

Test for Existence of an Attribute

```
from Naked.toolshed.types import NakedObject

obj = NakedObject({'example': 'an example string'})
if hasattr(obj, 'example'):
    # do something with the attribute
```

Equality Testing of NakedObjects

```
from Naked.toolshed.types import NakedObject

obj = NakedObject({'example': 'an example string'})
obj2 = NakedObject({'example': 'an example string'})
print(obj.equals(obj2)) # prints True
print(obj == obj2) #prints True
```

Equality Testing of NakedObjects, Failure on Different Attributes

```
from Naked.toolshed.types import NakedObject

obj = NakedObject({'example': 'an example string'})
obj2 = NakedObject({'different': 'an example string'})
print(obj.equals(obj2)) # prints False
print(obj == obj2) # prints False
```

Equality Testing of NakedObjects, Failure on Different Attribute Values

```
from Naked.toolshed.types import NakedObject

obj = NakedObject({'example': 'an example string'})
obj2 = NakedObject({'example': 'different'})
print(obj.equals(obj2)) # prints False
print(obj == obj2) # prints False
```

Equality Testing of NakedObjects, Failure on Different Attribute Number

```
from Naked.toolshed.types import NakedObject

obj = NakedObject({'example': 'an example string'})
obj2 = NakedObject({'example': 'an example string', 'example2': 'another string'})
print(obj.equals(obj2)) # prints False
print(obj == obj2) # prints False
```

Equality Testing of NakedObject, Failure on Different Type

```
from Naked.toolshed.types import NakedObject

obj = NakedObject({'example': 'an example string'})
obj2 = "an example string"
print(obj.equals(obj2)) # prints False
print(obj == obj2) # prints False
```

2.20 Toolshed: `types` : `XDict`

2.20.1 Import `XDict`

```
from Naked.toolshed.types import XDict
```

2.20.2 Import `XDict` from C Module

```
from Naked.toolshed.c.types import XDict
```

The C module must be compiled before you import it. See the naked build documentation for more information.

2.20.3 Description

The `XDict` class is an extension of the Python dictionary type. You can use all built-in Python dictionary methods with it. It extends the built-in Python dictionary type with operator overloads, metadata definitions on instantiation,

preservation of metadata on conversion to other types (with included XDict methods), and a number of additional dictionary methods.

The `XDict` supports equality testing based upon **both** the dictionary data as well as the supplemental metadata (if included). You can use the == and != operators to perform this testing (or alternatively, the *XDict.equals()* method).

class `Naked.toolshed.types.`**`XDict`**(*the_dictionary*[, *attribute_dictionary*])

A `XDict` is instantiated with a Python dictionary. You have the option to include a second Python dictionary to include additional metadata. The metadata are stored as attributes on the `XDict` with dictionary keys mapped to attribute names and dictionary values mapped to the corresponding attribute values.

Parameters

- **`the_dictionary`** (*dictionary*) – the data that are used to create an instance of a `XDict` dictionary.
- **`attribute_dictionary`** (*dictionary*) – (*optional*) a Python dictionary that is used to define the attributes of a new instance of a `XDict`. Key names are mapped to attribute names and their corresponding values are mapped to the attribute values.

Overloaded Operators

`__add__`(*other_dictionary*)

The + operator is overloaded to update the `XDict` with new key:value pairs from a Python dictionary or another `XDict`. An `XDict` must be the left sided operand in this statement as standard Python dictionaries do not support this form of dictionary combination. When used with a Python dictionary, the key:value pairs in the Python dictionary are added to the `XDict` dictionary. When used with another `XDict`, the key:value pairs from the `XDict` parameter are defined in the `XDict` dictionary **and** the attributes from the `XDict` parameter are defined in the `XDict`. The parameter dictionary definitions take precedence for key:value and attributes on the returned `XDict` when both objects contain the same dictionary key or attribute name.

Parameters **`other_dictionary`** (*dictionary*) – a Python dictionary or `XDict`

Returns (*XDict*) returns the original XDict updated with data in the `other_dictionary` as defined above

`__iadd__`(*other_dictionary*)

The += operator is overloaded to update the `XDict` operand on the left side of the operator with the Python dictionary or `XDict` on the right side of the operator. The update takes place as defined in the description of the *__add__()* method above.

Returns (*XDict*) returns a `XDict` that is updated with the data in the right sided operand.

`__eq__`(*other_dictionary*)

The == operator is overloaded to perform equality testing as defined for the *equals()* method below.

Returns (*boolean*) `True` = conditions for equality are met; `False` = conditions for equality are not met

`__neq__`(*other_dictionary*)

The != operator is overloaded to return the negation of the test for equality as it is defined in the *equals()* method below.

Returns (*boolean*) `True` = conditions for equality are not met; `False` = conditions for equality are met

Key Methods

difference(*other_dictionary*)
Returns the set of dictionary keys in the XDict that are not included in the other_dictionary parameter.

Parameters **other_dictionary** (*dictionary*) – a Python dictionary or XDict

Returns (*set*) Returns a set of dictionary key strings that meet this definition. Returns an empty set if there are no keys that meet the definition.

intersection(*other_dictionary*)
Returns the set of dictionary keys in the XDict that are also included in the other_dictionary parameter.

Parameters **other_dictionary** (*dictionary*) – a Python dictionary or XDict

Returns (*set*) Returns a set of dictionary key strings tha meet this definition. Returns an empty set if there are no keys that meet the definition.

key_xlist()
Returns a XList containing the keys in the XDict with preservation of the XDict attribute metadata in the returned XList.

Returns (*XList*) returns a XList that contains the XDict keys mapped to list items. The attribute data in the XDict is preserved in the returned XList.

Value Methods

conditional_map_to_vals(*conditional_func*, *map_func*)
Map a function parameter map_func to every XDict value that has a **key** that returns True when the key is passed as a parameter to the conditional_func function. Every XDict key is tested in the conditional_func.

Parameters

- **conditional_func** (*function*) – a function that accepts a XDict key as the first parameter and returns a boolean value. When the returned value is True, the value associated with this key is passed as the first parameter to the map_func.
- **map_func** (*function*) – a function that accepts a XDict value as the first parameter and returns the object that will be used to update the value definition for the key in the returned XDict.

Returns (*XDict*) returns the XDict with values that are updated as defined by the conditional_func and map_func processing. If the map_func does not return a value, the associated key is defined with None. If you intend to maintain the original value, return the value that was passed as the parameter to the function.

map_to_vals(*map_func*)
Maps a function parameter map_func to every value in the XDict. Every value in the XDict is passed to this function.

Parameters **map_func** (*function*) – a function that accepts a XDict value and returns the object that will be used to update the value definition for the key in the returned XDict

Returns (*XDict*) returns the XDict with values that are updated as defined by the returned values from the map_func. If the map_func does not return a value, the associated key is defined with None. If you intend to maintain the original value, return the value that was passed as the parameter to the function.

max_val()
Returns a 2-item tuple containing the maximum value and associated key as defined by the Python built-in max() function.

Returns (*tuple*) returns a 2-item tuple that includes (`max value`, `key`). The maximum numeric value is returned for numeric types. The value at the top of the reverse alphabetic order is returned for strings. For other types, the returned value is defined by the Python built-in `max()` function (if supported).

min_val()
Returns a 2-item tuple containing the minimum value and associated key as defined by the Python built-in `min()` function.

Returns (*tuple*) returns a 2-item tuple that includes (`min value`, `key`). The minimum numeric value is returned for numeric types. The value at the top of the alphabetic order is returned for strings. For other types, the returned value is as defined for the Python built-in `max()` function (if supported).

sum_vals()
Returns the sum of the values as determined by the Python built-in `sum()` function.

Returns (*numeric*) returns the sum as a numeric type defined by the input types

Raises `TypeError` for unsupported operand types encountered as values in the `XDict`

val_count(*the_value*)
Returns the count of `the_value` values in the `XDict`. Values are counted if they meet the criterion `XDict()[key] == the_value`.

Parameters **the_value** (*object*) – the value type and definition to be counted in the `XDict`

Returns (*integer*) returns the count of `the_value` in the `XDict` as an integer.

val_count_ci(*the_value*)
Returns the count of a case-insensitive test for `the_value` string in the `XDict` values. This method **can** be used with `XDict` that include value types that do not support the `string.lower()` method that is used in the case-insensitive testing.

Parameters **the_value** (*string*) – the string value that is to be used for a case-insensitive count across all `XDict` values

Returns (*integer*) returns the count of strings that match `the_value` in a case-insensitive test.

val_xlist()
Returns a `XList` that contains the `XDict` values mapped to list items.

Returns (*XList*) returns a `XList` that contains `XDict` values that are mapped to list items. Any attribute metadata from the original `XDict` is maintained in the returned `XList`.

Other Methods

equals(*other_object*)
The `equals()` method performs equality testing between a `XDict` and another object. The == operator can also be used to perform this test between the left (`XDict`) and right (`other_object`) sided operands. Equality testing is defined by meeting the criteria: (1) the type of the `XDict` and the `other_object` are the same; (2) the dictionary keys and values are the same in the `XDict` and the `other_object`; (3) the attribute metadata (if present) are the same in the `XDict` and the `other_object`.

Parameters **other_object** (*object*) – an object that is to be tested for equality

Returns (*boolean*) `True` = conditions for equality are met; `False` = conditions for equality are not met

random()
Returns a single, random key:value pair as a Python dictionary. The random pair is identified with the Python `random.sample()` method.

Returns (*dictionary*) a Python dictionary that contains a single key:value pair

random_sample(*number*)
Returns `number` random key:value pair(s) in a Python dictionary. The random pairs are identified with the Python `random.sample()` method. The random sampling is performed without replacement.

Parameters **number** (*integer*) – the number of random key:value pairs to return

Returns (*dictionary*) a Python dictionary that contains `number` key:value pairs

xitems()
A generator that yields 2-item tuples of key:value pairs from the `XDict`. This utilizes the `dict.iteritems()` generator when the Python 2 interpreter is used and the `dict.items()` generator when the Python 3 interpreter is used.

Returns (*tuple*) yields a 2-item tuple `(key, value)` on each iteration. Iteration ends when all `XDict` key:value pairs have been returned.

type()
Return the type of the `XDict` object.

Returns (type) returns the type of the `XDict`

2.20.4 Examples

Create a New Instance of XDict, No Metadata

```
from Naked.toolshed.types import XDict

xd = XDict({'name': 'Guido', 'language': 'python'})
```

Create a New Instance of XDict, With Metadata

```
from Naked.toolshed.types import XDict

xd = XDict({'name': 'Guido', 'language': 'python'}, {'dict_type': 'dev'})
```

Access XDict Value

```
from Naked.toolshed.types import XDict

xd = XDict({'name': 'Guido', 'language': 'python'}, {'dict_type': 'dev'})
print(xd['name']) # prints 'Guido'
print(xd['language']) # prints 'python'
```

Access XDict Attribute

```
from Naked.toolshed.types import XDict

xd = XDict({'name': 'Guido', 'language': 'python'}, {'dict_type': 'dev'})
print(xd.dict_type) # prints 'dev'
```

Compare XDict, Different Dictionaries

```
from Naked.toolshed.types import XDict

xd1 = XDict({'name': 'Guido', 'language': 'python'}, {'dict_type': 'dev'})
xd2 = XDict({'name': 'Yukihiro', 'language': 'ruby'}, {'dict_type': 'dev'})
```

```
print(xd1 == xd2) # prints False
print(xd != xd2) # prints True
```

Compare XDict, Different Attributes

```
from Naked.toolshed.types import XDict

xd1 = XDict({'name': 'Guido', 'language': 'python'}, {'dict_type': 'dev'})
xd2 = XDict({'name': 'Guido', 'language': 'python'}, {'rating': 1})
print(xd1 == xd2) # prints False
print(xd != xd2) # prints True
```

Update XDict with Dictionary

```
from Naked.toolshed.types import XDict

xd = XDict({'name': 'Guido', 'language': 'python'}, {'dict_type': 'dev'})
py_dict = {'year': 1991}
xd_with_year = xd + py_dict
print(xd_with_year) # prints {'name': 'Guido', 'language': 'python', 'year': 1991}
print(xd.dict_type) # prints 'dev'
```

Update XDict with XDict

```
from Naked.toolshed.types import XDict

xd1 = XDict({'name': 'Guido', 'language': 'python'}, {'dict_type': 'dev'})
xd2 = XDict({'year': 1991}, {'includes': 'year'})
xd3 = xd1 + xd2
print(xd3) # prints {'name': 'Guido', 'language': 'python', 'year': 1991}
print(xd3.dict_type) # prints 'dev'
print(xd3.includes) # prints 'year'
```

Update XDict with XDict, Alternate Approach with += Overload

```
from Naked.toolshed.types import XDict

xd1 = XDict({'name': 'Guido', 'language': 'python'}, {'dict_type': 'dev'})
xd2 = XDict({'year': 1991}, {'includes': 'year'})
xd1 += xd2
print(xd1) # prints {'name': 'Guido', 'language': 'python', 'year': 1991}
print(xd1.dict_type) # prints 'dev'
print(xd1.includes) # prints 'year'
```

Make XList from XDict Keys

```
from Naked.toolshed.types import XDict

xd = XDict({'name': 'Guido', 'language': 'python'}, {'dict_type': 'dev'})
xl = xd.key_xlist()
print(xl) # prints ['name', 'language']
print(xl.dict_type) # prints 'dev'
```

Make XList from XDict Values

```
from Naked.toolshed.types import XDict

xd = XDict({'name': 'Guido', 'language': 'python'}, {'dict_type': 'dev'})
```

```
xl = xd.val_xlist()
print(xl) # prints ['Guido', 'python']
print(xl.dict_type) # prints 'dev'
```

Conditional Mapping of a Function to XDict Values

```
from Naked.toolshed.types import XDict

def spam_corrector(the_argument):
    if the_argument == 'eggs':
        pass
    else:
        return 'eggs'

def comp_detector(the_argument):
    if the_argument == 'complements':
        return True
    else:
        return False


xd = XDict({'food': 'spam', 'complements': 'sausage'})
xd = xd.conditional_map_to_vals(comp_detector, spam_corrector)
print(xd) # prints {'food': 'spam', 'complements': 'eggs'}
```

2.21 Toolshed: `types` : `XList`

2.21.1 Import `XList`

```
from Naked.toolshed.types import XList
```

2.21.2 Import `XList` from C Module

```
from Naked.toolshed.c.types import XList
```

The C module must be compiled before you import it. See the naked build documentation for more information.

2.21.3 Description

The `XList` class is an extension of the Python list type. You can use all built-in Python list methods with it. It extends the built-in Python list type with operator overloads, metadata definitions on instantiation, preservation of metadata on conversion to other types (with included XList methods), and a number of additional list methods.

The `XList` supports equality testing based upon **both** the values of the list items as well as the supplemental XList metadata (if included). You can use the == and != operators to perform this testing (or alternatively, the *`XList.equals()`* method).

class `Naked.toolshed.types.`**`XList`**(*the_list*[, *attribute_dictionary*])

A `XList` is instantiated with any Python sequence type, including sets, tuples, and other lists. You have the option to include a Python dictionary as a second parameter to include additional metadata. The metadata are

stored as attributes on the XList with dictionary keys mapped to attribute names and dictionary values mapped to the corresponding attribute values.

Parameters

- **the_list** (*list*) – the data that are used to create an instance of a XList list. This can be of any Python sequence type, including sets, tuples, and other lists.
- **attribute_dictionary** (*dictionary*) – (*optional*) a Python dictionary that is used to define the attributes of a new instance of a XList. Key names are mapped to attribute names and their corresponding values are mapped to the attribute values.

Overloaded Operators

__add__ (**other_lists*)

The + operator is overloaded to extend the XList with one or more other XLists or lists. The XList must be the left sided operand in your statement to use this overloaded operator. When used with a Python list, the XList is extended with the items in the list. When used with another XList, the original XList is extended with the items *and the attributes* in the other XList. The right sided XList operand attribute values take precendence when the same attribute is included in both XLists.

Parameters **other_lists** (*list*) – one or more Python lists or XList (i.e. can add multiple XLists: xl = xl1 + xl2 + xl3)

Returns (*XList*) returns the original XList extended with data in the *other_lists as defined above

__iadd__ (*other_list*)

The += operator is overloaded to extend the XList with another XList or list. The XList must be the left sided operand in your statement to use this overloaded operator. When used with a Python list, the XList is extended with the items in the list. When used with another XList, the original XList is extended with the items *and the attributes* in the other XList. The right sided XList operand attribute values take precendence when the same attribute is included in both XLists.

Parameters **other_list** (*list*) – a Python list or XList

Returns (*XList*) returns the original XList extended with data in the other_list as defined above

__eq__ (*other_list*)

The == operator is overloaded to perform equality testing as defined for the *equals()* method below.

Returns (*boolean*) True = conditions for equality are met; False = conditions for equality are not met

__neq__ (*other_dictionary*)

The != operator is overloaded to return the negation of the test for equality as it is defined in the *equals()* method below.

Returns (*boolean*) True = conditions for equality are not met; False = conditions for equality are met

XList Methods

conditional_map_to_items (*conditional_func*, *map_func*)

Map a function map_func to items in a XList that meet a True condition in the function, conditional_func. See *map_to_items()* if you would like to map a function to every item in the list.

Parameters

- **conditional_func** (*function*) – a function that returns a boolean value where True means that the map_func should be executed on the item

- **map_func** (*function*) – the function that is conditionally executed with the `XList` item as a parameter. The return value is used as the replacement value in the `XList`. If the function does not return a value, the item is replaced with `None`.

Returns (*XList*) returns a `XList` with the above modifications

count_duplicates()

Count the number of duplicate items in the `XList`. See *remove_duplicates()* to remove the duplicated items.

Returns (*int*) returns the count of duplicate items

difference(*test_list*)

Return a set with the items in the `XList` that are not contained in the parameter `test_list`. Also see *intersection()*.

Parameters **test_list** (*list*) – a `XList` or list that is to be tested against

Returns (*set*) returns a Python set

equals(*other_object*)

The `equals()` method performs equality testing between a `XList` and another object. The == operator can also be used to perform this test between the left (`XList`) and right (`other_object`) sided operands. Equality testing is defined by meeting the criteria: (1) the type of the `XList` and the `other_object` are the same; (2) the list item values in the `XList` and the `other_object` are the same; (3) the attribute metadata (if present) are the same in the `XList` and the `other_object`.

Parameters **other_object** (*object*) – an object that is to be tested for equality

Returns (*boolean*) `True` = conditions for equality are met; `False` = conditions for equality are not met

intersection(*test_list*)

Return a set with the items in `XList` that are also contained in the parameter `test_list`. Also see *difference()*.

Parameters **test_list** (*list*) – a `XList` or list that is to be tested against

Returns (*set*) returns a Python set

join(*delimiter*)

Joins the string items in a `XList` with the `delimiter` string between each `XList` item and returns a string (or unicode) type.

Parameters **delimiter** (*string*) – the character or string to use as the delimiter between the items in the `XList` that are joined

Returns (*string*) returns a string or unicode type depending upon the types of the `XList` items, the `delimiter` character or string, and the Python interpreter version.

map_to_items(*map_func*)

Map a function to every item in the `XList`. To conditionally map a function to `XList` items (based upon conditions in a second function), see *conditional_map_to_items()*.

Parameters **map_func** (*function*) – the function that will take each item as a parameter and return the value for the replacement in the `XList`

Returns item and function dependent type. Items will be assigned a value of `None` if there is no return value from the function

max()

Returns the maximum item value in the `XList`. Also see *min()*.

Returns numeric type, dependent upon the type of the `XList` items

min()
Returns the minimum item value in the XList. Also see *max()*.

Returns numeric type, dependent upon the type of the XList items

postfix(*after_string*)
Appends a character or string suffix to each item in the XList. Also see *prefix()* and *surround()*.

Parameters **after_string** (*string*) – the character or string to append to each XList item

Returns (*XList*) returns a XList with the above modification to each item

prefix(*before_string*)
Prefixes a character or string to each item in the XList. Also see *postfix()* and *surround()*.

Parameters **before_string** (*string*) – the character or string to prefix on each item in the XList

Returns (*XList*) returns a XList with the above modification to each item

random()
Return a random item from the XList. The random selection is performed with the Python random.choice() method.

Returns random item from the XList

random_sample(*number_items*)
Return a random sample of items from the XList. Random sampling is performed with the Python random.sample() method. The number of items in the sample is defined with the number_items parameter. Random sampling is performed without replacement.

Parameters **number_items** (*integer*) – the number of items to include in the sample

Returns (*list*) returns a Python list containing number_items randomly sampled items from the XList.

remove_duplicates()
Removes the duplicate items in a XList and returns the XList. See *count_duplicates()* for duplicate counts.

Returns (*XList*) returns the modified XList with duplicates removed

shuffle()
Randomly shuffles the position of the items in the XList

Returns (*XList*) returns a XList with the above modification

sum()
Returns the sum of the item values in the XList. Not defined for non-numeric types.

Returns numeric type, dependent upon the type of the XList items

surround(*first_string*[, *second_string*])
Perform prefix and suffix string concatenation to every item in a XList. Also see *prefix()* and *postfix()*.

Parameters

- **first_string** (*string*) – character or string that is concatenated to the beginning of each XList item. If second_string is not specified, this character or string is also concatentated to the end of each XList item.

- **second_string** (*string*) – (*optional*) optional second character or string parameter that is appended to each `XList` item. If it is not specified, the `first_string` is concatenated to the beginning and end of each `XList` item.

Returns (*XList*) returns a `XList` with the above modifications to each item

wildcard_match(*wildcard*)

Match items in the `XList` by wildcard value and return a list that contains the matched items.

Parameters **wildcard** (*string*) – the wildcard value that is to be used for the match attempt

Returns (*list*) Python list containing the matched items. If there are no matched items, an empty list is returned.

Raises `TypeError` if the `XList` contains non-string items

multi_wildcard_match(*wildcard_sequence*)

Match items in the `XList` against more than one wildcard. Items are included in the returned list if they match any of the included wildcards.

Parameters **wildcard_sequence** (*string*) – a sequence of wildcards delimited by the `|` character (e.g. '*.py*|.pyc')

Returns (*list*) Python list containing the matched items. If there are no matched items, an empty list is returned.

Raises `TypeError` if the `XList` contains non-string items

XList Cast Methods

xset()

Cast a `XList` to a `XSet`.

Returns (*XSet*) returns a `XSet` with preservation of metadata

xfset()

Cast a `XList` to a `XFSet`.

Returns (*XFSet*) returns a `XFSet` with preservation of metadata

xtuple()

Cast a `XList` to a `XTuple`.

Returns (*XTuple*) returns a `XTuple` with preservation of metadata

2.21.4 Examples

Create a New Instance of XList, No Metadata

```
from Naked.toolshed.types import XList

xl = XList(['first', 'second', 'third'])
```

Create a New Instance of XList, With Metadata

```
from Naked.toolshed.types import XList

xl = XList(['first', 'second', 'third'], {'listtype': 'orderlist'})
```

Access XList Item

```
from Naked.toolshed.types import XList

xl = XList(['first', 'second', 'third'], {'listtype': 'orderlist'})
print(xl[0]) # prints 'first'
```

Access XList Attribute

```
from Naked.toolshed.types import XList

xl = XList(['first', 'second', 'third'], {'listtype': 'orderlist'})
print(xl.listtype) # prints 'orderlist'
```

Compare XList, Different List Items

```
from Naked.toolshed.types import XList

xl = XList(['first', 'second', 'third'], {'type': 'orderlist'})
xl2 = XList(['different', 'second', 'third'], {'type': 'orderlist'})
print(xl == xl2) # prints False
```

Compare XList, Different Attribute Metadata

```
from Naked.toolshed.types import XList

xl = XList(['first', 'second', 'third'], {'type': 'orderlist'})
xl2 = XList(['first', 'second', 'third'], {'type': 'another_orderlist'})
print(xl == xl2) # prints False
```

Extend the XList with Another List

```
from Naked.toolshed.types import XList

xl = XList(['first', 'second', 'third'], {'type': 'orderlist'})
a_list = ['fourth', 'fifth']
xl2 = xl + a_list
print(xl2) # prints ['first', 'second', 'third', 'fourth', 'fifth']
print(xl2.type) # prints 'orderlist'
```

Extend the XList with Another List, Alternate Approach with += Overload

```
from Naked.toolshed.types import XList

xl = XList(['first', 'second', 'third'], {'type': 'orderlist'})
a_list = ['fourth', 'fifth']
xl += a_list
print(xl) # prints ['first', 'second', 'third', 'fourth', 'fifth']
print(xl.type) # prints 'orderlist'
```

Comma Delimited String from XList

```
from Naked.toolshed.types import XList

xl = XList(['first', 'second', 'third'], {'type': 'orderlist'})
cd_string = xl.join(',')
print(cd_string) # prints 'first,second,third'
```

Wrap with Quotes

```
from Naked.toolshed.types import XList

xl = XList(['first', 'second', 'third'], {'type': 'orderlist'})
quote_list = xl.surround('"')
print(quote_list) # prints ['"first"', '"second"', '"third"']
```

Wrap with HTML Tags

```
from Naked.toolshed.types import XList

xl = XList(['paragraph one', 'paragraph two', 'paragraph three'], {'type': 'orderlist
↪'})
tag_list = xl.surround('<p class="naked">', '</p>')
for x in tag_list:
    print(x)

# prints:
#  '<p class="naked">paragraph one</p>'
#  '<p class="naked">paragraph two</p>'
#  '<p class="naked">paragraph three</p>'
```

Conditional Mapping of a Function to XList Items

```
from Naked.toolshed.types import XList

def true_a(xlist_item):
    return xlist_item.startswith('a')

def cap_val(xlist_item):
    return xlist_item.upper()

xl = XList(['another', 'one', 'many'], {'type': 'orderlist'})
new_list = xl.conditional_map_to_items(true_a, cap_val)
print(new_list)  # prints ['ANOTHER', 'one', 'many']
```

Multiple Wildcard Match

```
from Naked.toolshed.types import XList

xl = XList(['one', 'two', 'three'], {'type': 'orderlist'})
print(xl.multi_wildcard_match('o*|*hre*')) # prints ['one', 'three']
```

2.22 Toolshed: `types` : `XMaxHeap`

2.22.1 Import `XMaxHeap`

```
from Naked.toolshed.types import XMaxHeap
```

2.22.2 Import `XMaxHeap` from C Module

```
from Naked.toolshed.c.types import XMaxHeap
```

The C module must be compiled before you import it. See the naked build documentation for more information.

2.22.3 Description

The `XMaxHeap` class is a max heap priority queue that extends Python `heapq`. This class supports sorting of new items that are pushed to the queue by assigned priority and pop of the highest priority item (in contrast to the Python built-in heapq which returns the lowest priority item). It also supports the addition of attribute metadata on instantiation of the class.

class `Naked.toolshed.types.`**`XMaxHeap`**`(`[*attribute_dictionary*]`)`

Parameters **`attribute_dictionary`** (*`dict`*) – (*optional*) a Python dictionary that is used to define the attributes of a new instance of a `XMaxHeap`. Key names are mapped to attribute names and their corresponding values are mapped to the attribute values.

Function Overload

`__len__``()`

Returns (*int*) returns the number of items in the `XMaxHeap`. This allows you to use `len(XMaxHeap())` to determine the number of items in the priority queue.

XMaxHeap Methods

`length``()`

Returns (*int*) returns the number of items in the `XMaxHeap`

`pop``()`
Pops the highest priority item off of the queue.

Returns (*item type dependent*) returns the highest priority item which is defined as the item that has the highest `item_priority` value. If multiple items have the same value, they are returned on a first-in, first-out order (FIFO). If the queue is empty, returns `None`.

`push``(`*queue_item*, *item_priority*`)`
Pushes an item to the queue with the priority defined

Parameters

- **`queue_item`** (*`any`*) – an object that is added to the priority queue.
- **`item_priority`** (*`int`*) – an integer that represents the priority of the item from 1 (min) to x (max). It is possible to assign the same priority level to multiple items in the queue.

`pushpop``(`*queue_item*, *item_priority*`)`
Pushes an item to the queue and immediately pops and returns the highest priority item off of the queue.

Parameters

- **`queue_item`** (*`any`*) – an object that is added to the priority queue.
- **`item_priority`** (*`int`*) – an integer that represents the priority of the item from 1 (min) to x (max). It is possible to assign the same priority level to multiple items in the queue.

Returns (*item type dependent*) returns the highest priority item which is defined as the item that has the highest `item_priority` value. If multiple items have the same value, they are returned on a first-in, first-out order (FIFO). If the item that is pushed to the queue is the highest priority item, it is immediately returned. If the queue is empty, returns `None`.

2.22.4 Examples

Create a New Instance of XMaxHeap, No Metadata

```
from Naked.toolshed.types import XMaxHeap

xmh = XMaxHeap()
```

Create a New Instance of XMaxHeap, With Metadata

```
from Naked.toolshed.types import XMaxHeap

xmh = XMaxHeap({'heapnumber': 1})
```

Access XMaxHeap Attribute Data

```
from Naked.toolshed.types import XMaxHeap

xmh = XMaxHeap({'heapnumber': 1})
print(xmh.heapnumber) # prints 1
```

Push Items on to the XMaxHeap

```
from Naked.toolshed.types import XMaxHeap

xmh = XMaxHeap({'heapnumber': 1})
xmh.push('eat eggs', 1)
xmh.push('eat spam', 2)
```

Pop Items off of the XMaxHeap by Priority

```
from Naked.toolshed.types import XMaxHeap

xmh = XMaxHeap({'heapnumber': 1})
xmh.push('eat eggs', 1)
xmh.push('eat spam', 2)
print(xmh.pop()) # prints 'eat spam'
print(xmh.pop()) # prints 'eat eggs'
```

Priority Tie Handling with XMaxHeap

```
from Naked.toolshed.types import XMaxHeap

xmh = XMaxHeap({'heapnumber': 1})
xmh.push('eat eggs', 1)
xmh.push('eat spam', 1)  # same priority as above
print(xmh.pop()) # prints 'eat eggs' --> FIFO handling of ties
print(xmh.pop()) # prints 'eat spam'
```

Simultaneous Push and Pop with XMaxHeap

```
from Naked.toolshed.types import XMaxHeap

xmh = XMaxHeap({'heapnumber': 1})
xmh.push('eat eggs', 1)
xmh.push('eat spam', 2)
result = xmh.pushpop('buy Chris a coffee', 1)
print(result)      # prints 'eat spam'
print(xmh.pop())   # prints 'eat eggs'
print(xmh.pop())   # prints 'buy Chris a coffee' ;)
```

2.23 Toolshed: types : XMinHeap

2.23.1 Import XMinHeap

```
from Naked.toolshed.types import XMinHeap
```

2.23.2 Import XMinHeap from C Module

```
from Naked.toolshed.c.types import XMinHeap
```

The C module must be compiled before you import it. See the naked build documentation for more information.

2.23.3 Description

The XMinHeap class is a min heap priority queue that extends Python heapq. This class supports sorting of new items that are pushed to the queue by assigned priority and pop of the lowest priority item. It also supports the addition of attribute metadata on instantiation of the class.

class Naked.toolshed.types.**XMinHeap**([*attribute_dictionary*])

Parameters **attribute_dictionary** (*dict*) – (*optional*) a Python dictionary that is used to define the attributes of a new instance of a XMinHeap. Key names are mapped to attribute names and their corresponding values are mapped to the attribute values.

Function Overload

__len__()

Returns (*int*) returns the number of items in the XMinHeap. This allows you to use len(XMinHeap()) to determine the number of items in the priority queue.

XMinHeap Methods

length()
Returns (*int*) returns the number of items in the XMinHeap

pop()
Pops the lowest priority item off of the queue.
Returns (*item type dependent*) returns the lowest priority item which is defined as the item that has the lowest item_priority value. If multiple items have the same value, they are returned on a first-in, first-out order (FIFO). If the queue is empty, returns None.

push(*queue_item*, *item_priority*)
Pushes an item to the queue with the priority defined
Parameters
- **queue_item** (*any*) – an object that is added to the priority queue.
- **item_priority** (*int*) – an integer that represents the priority of the item from 1 (min) to x (max). It is possible to assign the same priority level to multiple items in the queue.

pushpop(*queue_item*, *item_priority*)
Pushes an item to the queue and immediately pops and returns the lowest priority item off of the queue.
Parameters
- **queue_item** (*any*) – an object that is added to the priority queue.

- **item_priority** (*int*) – an integer that represents the priority of the item from 1 (min) to x (max). It is possible to assign the same priority level to multiple items in the queue.

Returns (*item type dependent*) returns the lowest priority item which is defined as the item that has the lowest `item_priority` value. If multiple items have the same value, they are returned on a first-in, first-out order (FIFO). If the item that is pushed to the queue is the lowest priority item, it is immediately returned. If the queue is empty, returns `None`.

2.23.4 Examples

Create a New Instance of XMinHeap, No Metadata

```
from Naked.toolshed.types import XMinHeap

xmh = XMinHeap()
```

Create a New Instance of XMinHeap, With Metadata

```
from Naked.toolshed.types import XMinHeap

xmh = XMinHeap({'heapnumber': 1})
```

Access XMinHeap Attribute Data

```
from Naked.toolshed.types import XMinHeap

xmh = XMinHeap({'heapnumber': 1})
print(xmh.heapnumber) # prints 1
```

Push Items on to the XMinHeap

```
from Naked.toolshed.types import XMinHeap

xmh = XMinHeap({'heapnumber': 1})
xmh.push('eat eggs', 1)
xmh.push('eat spam', 2)
```

Pop Items off of the XMinHeap by Priority

```
from Naked.toolshed.types import XMinHeap

xmh = XMinHeap({'heapnumber': 1})
xmh.push('eat eggs', 1)
xmh.push('eat spam', 2)
print(xmh.pop()) # prints 'eat eggs'
print(xmh.pop()) # prints 'eat spam'
```

Priority Tie Handling with XMinHeap

```
from Naked.toolshed.types import XMinHeap

xmh = XMinHeap({'heapnumber': 1})
xmh.push('eat eggs', 1)
xmh.push('eat spam', 1)  # same priority as above
print(xmh.pop()) # prints 'eat eggs' --> FIFO handling of ties
print(xmh.pop()) # prints 'eat spam'
```

Simultaneous Push and Pop with XMinHeap

```
from Naked.toolshed.types import XMinHeap

xmh = XMinHeap({'heapnumber': 1})
xmh.push('eat eggs', 1)
xmh.push('eat spam', 2)
result = xmh.pushpop('buy Chris a coffee', 1)
print(result)       # prints 'eat eggs'
print(xmh.pop())    # prints 'buy Chris a coffee' ;)
print(xmh.pop())    # prints 'eat spam'
```

2.24 Changes

You can keep up with changes on the developer log that is available at http://devlog.naked-py.com.

2.25 Licenses

2.25.1 Naked Framework License

The **Naked framework** is licensed under the **MIT license**.

> The MIT License (MIT)
>
> Copyright (c) 2014 Chris Simpkins
>
> Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:
>
> The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.
>
> THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

2.25.2 Naked Documentation License

The **Naked documentation** is licensed under the **Creative Commons Attribution Share-Alike International 4.0 license**

> Creative Commons Attribution-ShareAlike 4.0 International Public License
>
> By exercising the Licensed Rights (defined below), You accept and agree to be bound by the terms and conditions of this Creative Commons Attribution-ShareAlike 4.0 International Public License ("Public

License"). To the extent this Public License may be interpreted as a contract, You are granted the Licensed Rights in consideration of Your acceptance of these terms and conditions, and the Licensor grants You such rights in consideration of benefits the Licensor receives from making the Licensed Material available under these terms and conditions.

Section 1 – Definitions.

Adapted Material means material subject to Copyright and Similar Rights that is derived from or based upon the Licensed Material and in which the Licensed Material is translated, altered, arranged, transformed, or otherwise modified in a manner requiring permission under the Copyright and Similar Rights held by the Licensor. For purposes of this Public License, where the Licensed Material is a musical work, performance, or sound recording, Adapted Material is always produced where the Licensed Material is synched in timed relation with a moving image. Adapter's License means the license You apply to Your Copyright and Similar Rights in Your contributions to Adapted Material in accordance with the terms and conditions of this Public License. BY-SA Compatible License means a license listed at creativecommons.org/compatiblelicenses, approved by Creative Commons as essentially the equivalent of this Public License. Copyright and Similar Rights means copyright and/or similar rights closely related to copyright including, without limitation, performance, broadcast, sound recording, and Sui Generis Database Rights, without regard to how the rights are labeled or categorized. For purposes of this Public License, the rights specified in Section 2(b)(1)-(2) are not Copyright and Similar Rights. Effective Technological Measures means those measures that, in the absence of proper authority, may not be circumvented under laws fulfilling obligations under Article 11 of the WIPO Copyright Treaty adopted on December 20, 1996, and/or similar international agreements. Exceptions and Limitations means fair use, fair dealing, and/or any other exception or limitation to Copyright and Similar Rights that applies to Your use of the Licensed Material. License Elements means the license attributes listed in the name of a Creative Commons Public License. The License Elements of this Public License are Attribution and ShareAlike. Licensed Material means the artistic or literary work, database, or other material to which the Licensor applied this Public License. Licensed Rights means the rights granted to You subject to the terms and conditions of this Public License, which are limited to all Copyright and Similar Rights that apply to Your use of the Licensed Material and that the Licensor has authority to license. Licensor means the individual(s) or entity(ies) granting rights under this Public License. Share means to provide material to the public by any means or process that requires permission under the Licensed Rights, such as reproduction, public display, public performance, distribution, dissemination, communication, or importation, and to make material available to the public including in ways that members of the public may access the material from a place and at a time individually chosen by them. Sui Generis Database Rights means rights other than copyright resulting from Directive 96/9/EC of the European Parliament and of the Council of 11 March 1996 on the legal protection of databases, as amended and/or succeeded, as well as other essentially equivalent rights anywhere in the world. You means the individual or entity exercising the Licensed Rights under this Public License. Your has a corresponding meaning. Section 2 – Scope.

License grant. Subject to the terms and conditions of this Public License, the Licensor hereby grants You a worldwide, royalty-free, non-sublicensable, non-exclusive, irrevocable license to exercise the Licensed Rights in the Licensed Material to: reproduce and Share the Licensed Material, in whole or in part; and produce, reproduce, and Share Adapted Material. Exceptions and Limitations. For the avoidance of doubt, where Exceptions and Limitations apply to Your use, this Public License does not apply, and You do not need to comply with its terms and conditions. Term. The term of this Public License is specified in Section 6(a). Media and formats; technical modifications allowed. The Licensor authorizes You to exercise the Licensed Rights in all media and formats whether now known or hereafter created, and to make technical modifications necessary to do so. The Licensor waives and/or agrees not to assert any right or authority to forbid You from making technical modifications necessary to exercise the Licensed Rights, including technical modifications necessary to circumvent Effective Technological Measures. For purposes of this Public License, simply making modifications authorized by this Section 2(a)(4) never produces Adapted Material. Downstream recipients. Offer from the Licensor – Licensed Material. Every recipient of the Licensed Material automatically receives an offer from the Licensor to exercise the Licensed Rights under the terms and conditions of this Public License. Additional offer from the Licensor –

Adapted Material. Every recipient of Adapted Material from You automatically receives an offer from the Licensor to exercise the Licensed Rights in the Adapted Material under the conditions of the Adapter's License You apply. No downstream restrictions. You may not offer or impose any additional or different terms or conditions on, or apply any Effective Technological Measures to, the Licensed Material if doing so restricts exercise of the Licensed Rights by any recipient of the Licensed Material. No endorsement. Nothing in this Public License constitutes or may be construed as permission to assert or imply that You are, or that Your use of the Licensed Material is, connected with, or sponsored, endorsed, or granted official status by, the Licensor or others designated to receive attribution as provided in Section 3(a)(1)(A)(i). Other rights.

Moral rights, such as the right of integrity, are not licensed under this Public License, nor are publicity, privacy, and/or other similar personality rights; however, to the extent possible, the Licensor waives and/or agrees not to assert any such rights held by the Licensor to the limited extent necessary to allow You to exercise the Licensed Rights, but not otherwise. Patent and trademark rights are not licensed under this Public License. To the extent possible, the Licensor waives any right to collect royalties from You for the exercise of the Licensed Rights, whether directly or through a collecting society under any voluntary or waivable statutory or compulsory licensing scheme. In all other cases the Licensor expressly reserves any right to collect such royalties. Section 3 – License Conditions.

Your exercise of the Licensed Rights is expressly made subject to the following conditions.

Attribution.

If You Share the Licensed Material (including in modified form), You must:

retain the following if it is supplied by the Licensor with the Licensed Material: identification of the creator(s) of the Licensed Material and any others designated to receive attribution, in any reasonable manner requested by the Licensor (including by pseudonym if designated); a copyright notice; a notice that refers to this Public License; a notice that refers to the disclaimer of warranties; a URI or hyperlink to the Licensed Material to the extent reasonably practicable; indicate if You modified the Licensed Material and retain an indication of any previous modifications; and indicate the Licensed Material is licensed under this Public License, and include the text of, or the URI or hyperlink to, this Public License. You may satisfy the conditions in Section 3(a)(1) in any reasonable manner based on the medium, means, and context in which You Share the Licensed Material. For example, it may be reasonable to satisfy the conditions by providing a URI or hyperlink to a resource that includes the required information. If requested by the Licensor, You must remove any of the information required by Section 3(a)(1)(A) to the extent reasonably practicable. ShareAlike. In addition to the conditions in Section 3(a), if You Share Adapted Material You produce, the following conditions also apply.

The Adapter's License You apply must be a Creative Commons license with the same License Elements, this version or later, or a BY-SA Compatible License. You must include the text of, or the URI or hyperlink to, the Adapter's License You apply. You may satisfy this condition in any reasonable manner based on the medium, means, and context in which You Share Adapted Material. You may not offer or impose any additional or different terms or conditions on, or apply any Effective Technological Measures to, Adapted Material that restrict exercise of the rights granted under the Adapter's License You apply. Section 4 – Sui Generis Database Rights.

Where the Licensed Rights include Sui Generis Database Rights that apply to Your use of the Licensed Material:

for the avoidance of doubt, Section 2(a)(1) grants You the right to extract, reuse, reproduce, and Share all or a substantial portion of the contents of the database; if You include all or a substantial portion of the database contents in a database in which You have Sui Generis Database Rights, then the database in which You have Sui Generis Database Rights (but not its individual contents) is Adapted Material, including for purposes of Section 3(b); and You must comply with the conditions in Section 3(a) if You Share all or a substantial portion of the contents of the database. For the avoidance of doubt, this Section 4 supplements and does not replace Your obligations under this Public License where the Licensed Rights

include other Copyright and Similar Rights. Section 5 – Disclaimer of Warranties and Limitation of Liability.

> Unless otherwise separately undertaken by the Licensor, to the extent possible, the Licensor offers the Licensed Material as-is and as-available, and makes no representations or warranties of any kind concerning the Licensed Material, whether express, implied, statutory, or other. This includes, without limitation, warranties of title, merchantability, fitness for a particular purpose, non-infringement, absence of latent or other defects, accuracy, or the presence or absence of errors, whether or not known or discoverable. Where disclaimers of warranties are not allowed in full or in part, this disclaimer may not apply to You. To the extent possible, in no event will the Licensor be liable to You on any legal theory (including, without limitation, negligence) or otherwise for any direct, special, indirect, incidental, consequential, punitive, exemplary, or other losses, costs, expenses, or damages arising out of this Public License or use of the Licensed Material, even if the Licensor has been advised of the possibility of such losses, costs, expenses, or damages. Where a limitation of liability is not allowed in full or in part, this limitation may not apply to You.

The disclaimer of warranties and limitation of liability provided above shall be interpreted in a manner that, to the extent possible, most closely approximates an absolute disclaimer and waiver of all liability. Section 6 – Term and Termination.

This Public License applies for the term of the Copyright and Similar Rights licensed here. However, if You fail to comply with this Public License, then Your rights under this Public License terminate automatically. Where Your right to use the Licensed Material has terminated under Section 6(a), it reinstates:

automatically as of the date the violation is cured, provided it is cured within 30 days of Your discovery of the violation; or upon express reinstatement by the Licensor. For the avoidance of doubt, this Section 6(b) does not affect any right the Licensor may have to seek remedies for Your violations of this Public License. For the avoidance of doubt, the Licensor may also offer the Licensed Material under separate terms or conditions or stop distributing the Licensed Material at any time; however, doing so will not terminate this Public License. Sections 1, 5, 6, 7, and 8 survive termination of this Public License. Section 7 – Other Terms and Conditions.

The Licensor shall not be bound by any additional or different terms or conditions communicated by You unless expressly agreed. Any arrangements, understandings, or agreements regarding the Licensed Material not stated herein are separate from and independent of the terms and conditions of this Public License. Section 8 – Interpretation.

For the avoidance of doubt, this Public License does not, and shall not be interpreted to, reduce, limit, restrict, or impose conditions on any use of the Licensed Material that could lawfully be made without permission under this Public License. To the extent possible, if any provision of this Public License is deemed unenforceable, it shall be automatically reformed to the minimum extent necessary to make it enforceable. If the provision cannot be reformed, it shall be severed from this Public License without affecting the enforceability of the remaining terms and conditions. No term or condition of this Public License will be waived and no failure to comply consented to unless expressly agreed to by the Licensor. Nothing in this Public License constitutes or may be interpreted as a limitation upon, or waiver of, any privileges and immunities that apply to the Licensor or You, including from the legal processes of any jurisdiction or authority. Creative Commons is not a party to its public licenses. Notwithstanding, Creative Commons may elect to apply one of its public licenses to material it publishes and in those instances will be considered the "Licensor." Except for the limited purpose of indicating that material is shared under a Creative Commons public license or as otherwise permitted by the Creative Commons policies published at creativecommons.org/policies, Creative Commons does not authorize the use of the trademark "Creative Commons" or any other trademark or logo of Creative Commons without its prior written consent including, without limitation, in connection with any unauthorized modifications to any of its public licenses or any other arrangements, understandings, or agreements concerning use of licensed material. For the avoidance of doubt, this paragraph does not form part of the public licenses.

Python Module Index

n

Index

Symbols

A

C

D

E

F

G

S

T

U

V

W

X

Y