arandomness Documentation

Release 0.2.0b1

Alex Hyer

Nov 01, 2018

Contents

Authors Alex Hyer

Date Nov 01, 2018

Version 0.2

Initializes arandomness package

CHAPTER 1

Introduction

arandomness is a package containing modules that I find myself re-writing for many programs. I've organized these modules into this amalgamative package as they are generally too diverse to fit into independent libraries. In general, I find the modules in arandomness to be very useful for many unrelated applications and wished to have them readily accessible in a unified, production-level library with proper unit tests. I hope you find something in this library useful!

CHAPTER 2

Installation

```
pip install arandomness
```

CHAPTER 3

Contents

3.1 argparse

3.1.1 Introduction

The `argparse` subpackage of arandomness contains scripts and actions to expand the utility of Python's argparse library.

3.1.2 CheckThreads

`CheckThreads` is an `argparse` `action`, as such, it is called as the value of the `action` argument in `argparse`. For example:

```
from arandomness.argparse import CheckThreads
import argparse
parser = argparse.ArgumentParser(description=__doc__)
parser.add_argument('-t', '--threads',
                    action=CheckThreads,
                    type=int,
                    default=1,
                    help='number of threads to use')
args = parser.parse_args()
```

When `-t` is parsed, the value is passed to `CheckThreads` which then checks that the value is between 1 and the maximum number of threads on the computer as per multiprocessing.cpu_count().

API Documentation

3.1.3 CopyRight

`CopyRight` is an `argparse action` that simply takes in text, strips it of leading and trailing whitespace, prints it, and exits the program. Its functionality is analogous to `argparse`'s version. The action can take in arbitrary text and is only named `CopyRight` for code readability.

```
from arandomness.argparse import CopyRight
import argparse
parser = argparse.ArgumentParser(description=__doc__)
parser.add_argument('--copyright',
                    action=CopyRight,
                    copyright_text='This is my copyright',
                    help='print copyright and exit')
args = parser.parse_args()
```

API Documentation

3.1.4 COpen

TODO: Add copen to files, move most of this there, and just reference it

`COpen` is an `argparse action` that seamlessly handles reading and writing compressed files using the gzip, bz2, and lzma libraries. To do this, `COpen` actually exposes the arguments of each to libraries `*File` function to the command line after automatically selecting the proper library based on the arguments it receives. Essentially, this `action` operates in a read mode and a write/append mode. In read mode, when mode is equal to any read mode supported by the appropriate library such as `r` or `rb`, `COpen` reads the first few bytes of the file to see what compression format the file uses and then opens the file with the corresponding in decompression algorithm. In write mode, basically when mode is set to anything else, `COpen` just checks the file extension and maps it to the corresponding compression algorithm. If `COpen` does not recognize the first few bytes of a file or a file extension, it defaults to reading and writing in plain text.

As aforementioned, `COpen` exposes the arguments of the underlying library. It does this by collecting arbitrary arguments, filtering them by the supported arguments of the `*File` functions, and only passing those arguments to the function. For example, `GzipFile` and `BZ2File` can control the level on compression via the argument `compresslevel` while `LZMAFile` uses `preset` to control compression levels. In order to use these arguments at the `argparse` level, simply add them as options to `COpen` as follows:

```
from arandomness.argparse import COpen
import argparse
parser = argparse.ArgumentParser(description=__doc__)
parser.add_argument('--gzip',
                    action=COpen,
                    mode='r',
                    type=str,
                    compresslevel=9,
                    help='compressed file to read')
parser.add_argument('--bz2',
                    action=COpen,
                    mode='w',
                    type=str,
                    compresslevel=9
                    help='compressed file to write')
parser.add_argument('--lzma',
```

(continues on next page)

(continued from previous page)

```
                    action=COpen,
                    mode='w',
                    type=str,
                    preset=9,
                    help='compressed file to write')
args = parser.parse_args(['-i', 'input.gz', '-o', 'output.xz'])
```

As stated, this works for any argument and arguments that aren't supported by the `*File` are silently ignored.

Common use example:

```
from arandomness.argparse import COpen
import argparse
parser = argparse.ArgumentParser(description=__doc__)
parser.add_argument('-i', '--input',
                    action=COpen,
                    mode='r',
                    type=str,
                    help='compressed file to read')
parser.add_argument('-o', '--output',
                    action=COpen,
                    mode='w',
                    type=str,
                    help='compressed file to write')
args = parser.parse_args(['-i', 'input.gz', '-o', 'output.xz'])
```

API Documentation

3.1.5 ParseSeparator

By default, `argparse` parses multiple arguments by spaces. While useful, it can sometimes be more practical, or at least easier to read, arguments parsed by commas or some other separator character when multiple arguments make use of nargs. `ParseSeparator` simply takes a string, splits it by the user-defined separator, and sets the resulting list as the value for the argument. For example:

```
from arandomness.argparse import ParseSeparator
import argparse
parser = argparse.ArgumentParser(description=__doc__)
parser.add_argument('-a', '--an_argument',
                    action=ParseSeparator,
                    type=str,
                    sep=',',
                    help='nargs using a string')
args = parser.parse_args(['hello,world'])
print(args.an_argument)
```

So the argument `hello,world` would be set as `['hello', 'world']` in args.

API Documentation

3.2 string

3.2.1 Introduction

The `string` subpackage of arandomness contains a couple functions that analyze or manipulate strings in some way. That's about as specific as this subpackage gets. Enjoy!

3.2.2 autocorrect

The `autocorrect` function takes a single query string and a list of "correct" strings and identifies which string in the list the query most closely matches. There are many far more robust autocorrect algorithms written in Python than this one, but they all require a list of words organized by their frequency in a given language. Basically, these autocorrect algorithms are aimed at correcting words specific to a language and are thus better suited for use in language processing software, e.g. texting apps. This algorithm uses any list of strings and is order-agnostic. Thus, my `autocorrect` is better suited for attempting to match queries to small lists of arbitrary strings.

To help realize this concept, I have used this function in a program that presented data in a database about programs available on a given system. The query was the user's request and the possible strings was simply the list of program names in the database. Thus, if a user misspelled a program name, the program likely produced the proper entry.

API Documentation

3.2.3 max_substring

The `max_substring` function takes in a list of strings and finds the longest substring that they all share. By default, `max_substring` starts at the beginning of each string, but it can be optionally start at a later position as demonstrated in the docstring examples.

API Documentation

3.3 trees

Initializes trees package of arandomness

3.3.1 Introduction

The `argparse` subpackage of arandomness contains scripts and classes relating to trees.

3.3.2 OmniTree (Deprecated)

OmniTree is a class for creating a tree where each node can contain multiple children and multiple parents. I began writing this class because I could not find an extant tree library that supported this many-to-many paradigm. While binary and/or hierarchical tree with only a single parent are common, see anytree, trees like OmniTree are not. After a lot of R&D, I realized that these is such a structure, a graph. Basically, many-to-many trees don't exist because they cannot by definition. Thus, OmniTree is pointless as there are already amazing libraries for manipulating and managing graphs, such as NetworkX. As such, OmniTree is deprecated and is only included for archival purposes.

API Documentation

class arandomness.trees.**OmniTree**(*label=None*, *children=None*, *parents=None*)

A many-to-many tree for organizing and manipulating hierarchical data

label

unicode – optional, arbitrary name for node

__init__(*label=None*, *children=None*, *parents=None*)

Initialize node and inform connected nodes

__weakref__

list of weak references to the object (if defined)

add_children(*children*)

Adds new children nodes after filtering for duplicates

Parameters **children** (*list*) – list of OmniTree nodes to add as children

add_parents(*parents*)

Adds new parent nodes after filtering for duplicates

Parameters **parents** (*list*) – list of OmniTree nodes to add as parents

find_branches(*labels=False*, *unique=False*)

Recursively constructs a list of pointers of the tree's structure

Parameters

- **labels** (*bool*) – If True, returned lists consist of node labels. If False (default), lists consist of node pointers. This option is mostly intended for debugging purposes.
- **unique** (*bool*) – If True, return lists of all unique, linear branches of the tree. More accurately, it returns a list of lists where each list contains a single, unique, linear path from the calling node to the tree's leaf nodes. If False (default), a highly-nested list is returned where each nested list represents a branch point in the tree. See Examples for more.

Examples

```
>>> from arandomness.trees import OmniTree
>>> a = OmniTree(label='a')
>>> b = OmniTree(label='b', parents=[a])
```

(continues on next page)

(continued from previous page)

```
>>> c = OmniTree(label='c', parents=[b])
>>> d = OmniTree(label='d', parents=[b])
>>> e = OmniTree(label='e', parents=[c, d])
>>> a.find_branches(labels=True)
['a', ['b', ['c', ['e']], ['d', ['e']]]]
>>> a.find_branches(labels=True, unique=True)
[['a', 'b', 'c', 'e'], ['a', 'b', 'd', 'e']]
```

find_loops(*_path=None*)

Crappy function that finds a single loop in the tree

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

CHAPTER 5

Copyright

arandomness operates under the `GPLv3 License` and may be edited and redistributed as per that license.

Python Module Index

a

Index

Symbols

A

F

L

O