simplere Documentation

Release 1.2.12

Jonathan Eunice

May 31, 2017

Contents

A simplified interface to Python's regular expression (`re`) string search. Eliminates steps and provides simpler access to results. As a bonus, also provides compatible way to access Unix glob searches.

CHAPTER 1

Usage

Python regular expressions are powerful, but the language's lack of an *en passant* (in passing) assignment requires a preparatory motion and then a test:

```
import re

match = re.search(pattern, some_string)
if match:
    print match.group(1)
```

With `simplere`, you can do it in fewer steps:

```
from simplere import *

if match / re.search(pattern, some_string):
    print match[1]
```

In addition to its own classes, `from simplere import *` imports both the standard `re` module and the `match` object so you don't have to.

CHAPTER 2

Motivation

In the simple examples above, "fewer steps" seems like a small savings (3 lines to 2). While a 33% savings is a pretty good optimization, is it really worth using another module and a quirky *en passant* operator to get it?

In code this simple, maybe not. But real regex-based searching tends to have multiple, cascading searches, and to be tightly interwoven with complex pre-conditions, error-checking, and post-match formatting or actions. It gets complicated fast. When multiple `re` matches must be done, it consumes a lot of "vertical space" and often threatens to push the number of lines a programmer is viewing at any given moment beyond the number that can be easily held in working memory. In that case, it proves valuable to condense what is logically a single operation ("regular expression test") into a single line with its conditional `if`.

This is even more true for the "exploratory" phases of development, before a program's appropriate structure and best logical boundaries have been established. One can always "back out" the condensing *en passant* operation in later production code, if desired.

CHAPTER 3

Re Objects

`Re` objects are memoized for efficiency, so they compile their pattern just once, regardless of how many times they're mentioned in a program.

Note that the `in` test turns the sense of the matching around (compared to the standard `re` module). It asks "is the given string *in* the set of items this pattern describes?" To be fancy, the `Re` pattern is an intensionally defined set (namely "all strings matching the pattern"). This order often makes excellent sense whey you have a clear intent for the test. For example, "is the given string within the set of *all legitimate commands*?"

Second, the `in` test had the side effect of setting the underscore name `_` to the result. Python doesn't support *en passant* assignment–apparently, no matter how hard you try, or how much introspection you use. This makes it harder to both test and collect results in the same motion, even though that's often exactly appropriate. Collecting them in a class variable is a fallback strategy (see the *En Passant* section below for a slicker one).

If you prefer the more traditional `re` calls:

```
if Re(pattern).search(some_string):
    print Re._[1]
```

`Re` works even better with named pattern components, which are exposed as attributes of the returned object:

```
person = 'John Smith 48'
if person in Re(r'(?P<name>[\w\s]*)\s+(?P<age>\d+)'):
    print Re._.name, "is", Re._.age, "years old"
else:
    print "don't understand '{}'".format(person)
```

One trick being used here is that the returned object is not a pure `_sre.SRE_Match` that Python's `re` module returns. Nor is it a subclass. (That class appears to be unsubclassable.) Thus, regular expression matches return a proxy object that exposes the match object's numeric (positional) and named groups through indices and attributes. If a named group has the same name as a match object method or property, it takes precedence. Either change the name of the match group or access the underlying property thus: `x._match.property`

It's possible also to loop over the results:

```
for found in Re('pattern (\w+)').finditer('pattern is as pattern does'):
    print found[1]
```

Or collect them all in one fell swoop:

```
found = Re('pattern (\w+)').findall('pattern is as pattern does')
```

Pretty much all of the methods and properties one can access from the standard `re` module are available.

CHAPTER 4

Under the Covers

`ReMatch` objects wrap Python's native``_sre.SRE_Match`` objects (the things that `re` method calls return).:

```
match = re.match(r'(?P<word>th.s)', 'this is a string')
match = ReMatch(match)
if match:
    print match.group(1)     # still works
    print match[1]           # same thing
    print match.word         # same thing, with logical name
```

But that's a lot of boilerplate for a simple test, right? So `simplere` *en passant* operator redefining the division operation and proxies the `re` result on the fly to the pre-defined `match` object:

```
if match / re.search(r'(?P<word>th.s)', 'this is a string'):
    assert match[1] == 'this'
    assert match.word == 'this'
    assert match.group(1) == 'this'
```

If the `re` operation fails, the resulting object is guaranteed to have a `False`-like Boolean value, so that it will fall through conditional tests.

CHAPTER 5

Options and Alternatives

If you prefer the look of the less-than (<) or less-than-or-equal (<=), as indicators that `match` takes the value of the following function call, they are experimentally supported as aliases of the division operation (`/`). You may define your own match objects, and can use them on memoized `Re` objects too. Putting a few of these optional things together:

```
answer = Match()    # need to do this just once

if answer < Re(r'(?P<word>th..)').search('and that goes there'):
    assert answer.word == 'that'
```

CHAPTER 6

Globs

Regular expressions are wonderfully powerful, but sometimes the simpler Unix glob is works just fine. As a bonus, `simplere` also provides simple glob access.:

```
if 'globtastic' in Glob('glob*'):
    print "Yes! It is!"
else:
    raise ValueError('YES IT IS')
```

If you want to search or test against multiple patterns at once, `Glob` objects take a variable number of patterns. A match is defined as *any* of the patterns matching.:

```
img_formats = Glob("*.png", "*.jpeg", "*.jpg", "*.gif")
if filename.lower() in img_formats:
    ... further processing ...
```

Alternatively, you can splat an existing list into the `Glob` constructor with Python's unary star syntax:

```
img_formats = "*.png *.jpeg *.jpg *.gif".split()
if filename.lower() in Glob(*img_formats):
    ... further processing ...
```

Case-insensitive glob searches are also available:

```
bg = InsensitiveGlob('b*')
if 'bubba' in bg:
    assert 'Bubba' in bg
```

Globs have their own syntax for case insensitive characters, but it can be a pain to use. It may be easier to use the `InsensitiveGlob` subclass. Or even alias the case-insensitive version as the main one:

```
from simplere import InsensitiveGlob as Glob
```

Note: Case folding / case-insensitive searches work well in the ASCII range, but Unicode characters and case folding is more intricate. Basic folding is provided out of the box. It's quite adequate for mapping against common filename

patterns, for example. Those needing more extensive Unicode case folding should consider normalizing strings, as described here. As the tests show, basic Unicode folding works fine everywhere. Using Unicode in glob patterns (not just strings to be matched) works *only* on Python 3.3 or above.

CHAPTER 7

Notes

- Automated multi-version testing managed with pytest and tox. Continuous integration testing with Travis-CI. Packaging linting with pyroma.
- Version 1.2.9 updates testing for early 2017 Python versions. Successfully packaged for, and tested against, all late-model versions of Python: 2.6, 2.7, 3.3, 3.4, 3.5, and 3.6, as well as PyPy 5.6.0 (based on 2.7.12) and PyPy3 5.5.0 (based on 3.3.5).
- The author, Jonathan Eunice or @jeunice on Twitter welcomes your comments and suggestions.

CHAPTER 8

Installation

To install or upgrade to the latest version:

```
pip install -U simplere
```

To install under a specific Python version (3.3 in this example):

```
python3.3 -m pip -U simplere
```

You may need to prefix these with `sudo` to authorize installation. In environments without super-user privileges, you may want to use `pip`'s `--user` option, to install only for a single user, rather than system-wide. If you use the standalone `pip` programs, you may also need to use `pip2` or `pip3` version-dependent variants, depending on your system configuration.

CHAPTER 9

Testing

To run the module tests, use one of these commands:

```
tox                # normal run - speed optimized
tox -e py27        # run for a specific version only (e.g. py27, py34)
tox -c toxcov.ini  # run full coverage tests
```

CHAPTER 10

Change Log

1.2.12 (May 31, 2017)

Updated Python 2 / 3 compatibility strategy to be Python 3 centric. Should be more future-proofed.

Updated testing.

1.2.11 (February 17, 2017)

Updates testing. Again. Python 3.2 support again dropped, for the last time, given failure on Travis CI. It's old, anyway. Time to upgrade!

1.2.10 (February 17, 2017)

Updates testing. Python 3.2 support re-established, given still supported on Travis CI. Docs tweaked.

1.2.9 (January 23, 2017)

Updates testing. Newly qualified under 2.7.13 and 3.6, as well as most recent builds of pypy and pypy3. Python 3.2 support withdrawn given obsolescence.

1.2.8 (August 26, 2015)

Substantial documentation reorg.

1.2.7 (August 23, 2015)

Starts automated measurement of test branch coverage. Inital runs show 100% branch coverage. Hooah!

1.2.6 (August 20, 2015)

Bumped to 100% test coverage.

1.2.5 (August 19, 2015)

Added automated measurement of test coverage. Line coverage started at 92%. Bumped to 97%.

1.2.0 (August 14, 2015)

Realized imports were overly restrictive, requiring clients of module to needlessly (and contra docs) manually import `re` and construct the `match` object. Fixed. Bumped minor version number to reflect de facto API change.

1.1.1 (August 14, 2015)

Simplified `setup.py` and packaging. Tweaked docs.

1.1.0

Adds multi-pattern and case insensitive Glob subclass. Added wheel packaging. Rearranged and extended testing structure. Updated setup and docs.

1.0.10

Added `bdist_wheel` package support. Extended testing matrix to 3.5 pre-release builds. Switched to Apache License.

1.0.5

In several dot-releases, have added support for Travis-CI cloud- based continuous integration testing, Sphinx-based documentation, and readthedocs.org hosted documentation. The Travis bit has required a separate Github repository be created. It is managed out of the same development directory, overlaying the existing Mercurial / Bitbucket repo. So far, that has caused no problems.

Documentation somewhat improved.

1.0.0

Cleaned up source for better PEP8 conformance

Bumped version number to 1.0 as part of move to semantic versioning, or at least enough of it so as to not screw up Python installation procedures (which don't seem to understand 0.401 is a lesser version that 0.5, because 401 > 5)