retrie

Release 0.3.1

ddelange

Feb 22, 2024

CONTENTS:

CHAPTER

ONE

RETRIE

retrie offers fast methods to match and replace (sequences of) strings based on efficient Trie-based regex unions.

1.1 Trie

Instead of matching against a simple regex union, which becomes slow for large sets of words, a more efficient regex pattern can be compiled using a Trie structure:

```
from retrie.trie import Trie


trie = Trie()

trie.add("abc", "foo", "abs")
assert trie.pattern() == "(?:ab[cs]|foo)"  # equivalent to but faster than "(?
↪:abc|abs|foo)"

trie.add("absolute")
assert trie.pattern() == "(?:ab(?:c|s(?:olute)?)|foo)"

trie.add("abx")
assert trie.pattern() == "(?:ab(?:[cx]|s(?:olute)?)|foo)"

trie.add("abxy")
assert trie.pattern() == "(?:ab(?:c|s(?:olute)?|xy?)|foo)"
```

A `Trie` may be populated with zero or more strings at instantiation or via `Trie.add`, from which method chaining is possible. Two instances can be merged with the + (new instance) and += (in-place update) operators. Instances will compare equal if their data dictionaries are equal.

```
trie = Trie()
trie += Trie("abc")
assert (
    trie + Trie().add("foo")
    == Trie("abc", "foo")
    == Trie(*["abc", "foo"])
    == Trie().add(*["abc", "foo"])
    == Trie().add("abc", "foo")
```

(continues on next page)

(continued from previous page)

```
    == Trie().add("abc").add("foo")
)
```

1.2 Installation

This pure-Python, OS independent package is available on PyPI:

```
$ pip install retrie
```

1.3 Usage

For documentation, see retrie.readthedocs.io.

The following objects are all subclasses of `retrie.retrie.Retrie`, which handles filling the Trie and compiling the corresponding regex pattern.

1.3.1 Blacklist

The `Blacklist` object can be used to filter out bad occurences in a text or a sequence of strings:

```
from retrie.retrie import Blacklist

# check out docstrings and methods
help(Blacklist)

blacklist = Blacklist(["abc", "foo", "abs"], match_substrings=False)
blacklist.compiled
# re.compile(r'(?<=\b)(?:ab[cs]|foo)(?=\b)', re.IGNORECASE|re.UNICODE)
assert not blacklist.is_blacklisted("a foobar")
assert tuple(blacklist.filter(("good", "abc", "foobar"))) == ("good", "foobar")
assert blacklist.cleanse_text(("good abc foobar")) == "good  foobar"

blacklist = Blacklist(["abc", "foo", "abs"], match_substrings=True)
blacklist.compiled
# re.compile(r'(?:ab[cs]|foo)', re.IGNORECASE|re.UNICODE)
assert blacklist.is_blacklisted("a foobar")
assert tuple(blacklist.filter(("good", "abc", "foobar"))) == ("good",)
assert blacklist.cleanse_text(("good abc foobar")) == "good  bar"
```

1.3.2 Whitelist

Similar methods are available for the `Whitelist` object:

```
from retrie.retrie import Whitelist

# check out docstrings and methods
help(Whitelist)

whitelist = Whitelist(["abc", "foo", "abs"], match_substrings=False)
whitelist.compiled
# re.compile(r'(?<=\b)(?:ab[cs]|foo)(?=\b)', re.IGNORECASE|re.UNICODE)
assert not whitelist.is_whitelisted("a foobar")
assert tuple(whitelist.filter(("bad", "abc", "foobar"))) == ("abc",)
assert whitelist.cleanse_text(("bad abc foobar")) == "abc"

whitelist = Whitelist(["abc", "foo", "abs"], match_substrings=True)
whitelist.compiled
# re.compile(r'(?:ab[cs]|foo)', re.IGNORECASE|re.UNICODE)
assert whitelist.is_whitelisted("a foobar")
assert tuple(whitelist.filter(("bad", "abc", "foobar"))) == ("abc", "foobar")
assert whitelist.cleanse_text(("bad abc foobar")) == "abcfoo"
```

1.3.3 Replacer

The `Replacer` object does a fast single-pass search & replace for occurrences of `replacement_mapping.keys()` with corresponding values.

```
from retrie.retrie import Replacer

# check out docstrings and methods
help(Replacer)

replacement_mapping = dict(zip(["abc", "foo", "abs"], ["new1", "new2", "new3"]))

replacer = Replacer(replacement_mapping, match_substrings=True)
replacer.compiled
# re.compile(r'(?:ab[cs]|foo)', re.IGNORECASE|re.UNICODE)
assert replacer.replace("ABS ...foo... foobar") == "new3 ...new2... new2bar"

replacer = Replacer(replacement_mapping, match_substrings=False)
replacer.compiled
# re.compile(r'\b(?:ab[cs]|foo)\b', re.IGNORECASE|re.UNICODE)
assert replacer.replace("ABS ...foo... foobar") == "new3 ...new2... foobar"

replacer = Replacer(replacement_mapping, match_substrings=False, re_flags=None)
replacer.compiled  # on py3, re.UNICODE is always enabled
# re.compile(r'\b(?:ab[cs]|foo)\b')
assert replacer.replace("ABS ...foo... foobar") == "ABS ...new2... foobar"

replacer = Replacer(replacement_mapping, match_substrings=False, word_boundary=" ")
replacer.compiled
```

(continues on next page)

(continued from previous page)

```
# re.compile(r'(?<= )(?:ab[cs]|foo)(?= )', re.IGNORECASE|re.UNICODE)
assert replacer.replace(". ABS ...foo... foobar") == ". new3 ...foo... foobar"
```

1.4 Development

Run `make help` for options like installing for development, linting and testing.

CHAPTER

TWO

CODE REFERENCE

2.1 retrie package

2.1.1 Submodules

2.1.2 retrie.retrie module

Submodule containing the *Retrie* class, which handles filling the Trie and compiling the corresponding regex pattern, and its high-level wrappers.

The *Blacklist* class can be used to filter out bad occurences in a text or a sequence of strings:

```
from retrie.retrie import Blacklist

# check out docstrings and methods
help(Blacklist)

blacklist = Blacklist(["abc", "foo", "abs"], match_substrings=False)
blacklist.compiled
# re.compile(r'(?<=)(?:ab[cs]|foo)(?=)', re.IGNORECASE|re.UNICODE)
assert not blacklist.is_blacklisted("a foobar")
assert tuple(blacklist.filter(("good", "abc", "foobar"))) == ("good", "foobar")
assert blacklist.cleanse_text(("good abc foobar")) == "good  foobar"

blacklist = Blacklist(["abc", "foo", "abs"], match_substrings=True)
blacklist.compiled
# re.compile(r'(?:ab[cs]|foo)', re.IGNORECASE|re.UNICODE)
assert blacklist.is_blacklisted("a foobar")
assert tuple(blacklist.filter(("good", "abc", "foobar"))) == ("good",)
assert blacklist.cleanse_text(("good abc foobar")) == "good  bar"
```

Similar methods are available for the *Whitelist* class:

```
from retrie.retrie import Whitelist

# check out docstrings and methods
help(Whitelist)

whitelist = Whitelist(["abc", "foo", "abs"], match_substrings=False)
whitelist.compiled
# re.compile(r'(?<=)(?:ab[cs]|foo)(?=)', re.IGNORECASE|re.UNICODE)
```

(continues on next page)

(continued from previous page)

```
assert not whitelist.is_whitelisted("a foobar")
assert tuple(whitelist.filter(("bad", "abc", "foobar"))) == ("abc",)
assert whitelist.cleanse_text(("bad abc foobar")) == "abc"

whitelist = Whitelist(["abc", "foo", "abs"], match_substrings=True)
whitelist.compiled
# re.compile(r'(?:ab[cs]|foo)', re.IGNORECASE|re.UNICODE)
assert whitelist.is_whitelisted("a foobar")
assert tuple(whitelist.filter(("bad", "abc", "foobar"))) == ("abc", "foobar")
assert whitelist.cleanse_text(("bad abc foobar")) == "abcfoo"
```

The *Replacer* class does a fast single-pass search & replace for occurrences of `replacement_mapping.keys()` with corresponding values.

```
from retrie.retrie import Replacer

# check out docstrings and methods
help(Replacer)

replacement_mapping = dict(zip(["abc", "foo", "abs"], ["new1", "new2", "new3"]))

replacer = Replacer(replacement_mapping, match_substrings=True)
replacer.compiled
# re.compile(r'(?:ab[cs]|foo)', re.IGNORECASE|re.UNICODE)
assert replacer.replace("ABS ...foo... foobar") == "new3 ...new2... new2bar"

replacer = Replacer(replacement_mapping, match_substrings=False)
replacer.compiled
# re.compile(r'(?:ab[cs]|foo)', re.IGNORECASE|re.UNICODE)
assert replacer.replace("ABS ...foo... foobar") == "new3 ...new2... foobar"

replacer = Replacer(replacement_mapping, match_substrings=False, re_flags=None)
replacer.compiled  # on py3, re.UNICODE is always enabled
# re.compile(r'(?:ab[cs]|foo)')
assert replacer.replace("ABS ...foo... foobar") == "ABS ...new2... foobar"

replacer = Replacer(replacement_mapping, match_substrings=False, word_boundary=" ")
replacer.compiled
# re.compile(r'(?<= )(?:ab[cs]|foo)(?= )', re.IGNORECASE|re.UNICODE)
assert replacer.replace(". ABS ...foo... foobar") == ". new3 ...foo... foobar"
```

class retrie.retrie.**Blacklist**(*blacklisted*, *match_substrings=False*, *word_boundary='\\b'*, *re_flags=RegexFlag.IGNORECASE | UNICODE*)

Bases: *Checklist*

Mutate [sequences of] strings based on their match against blacklisted.

Note: Although the Trie is case-sensitive, by default `re.IGNORECASE` is used for better performance. Pass `re_flags=None` to perform case-sensitive replacements.

Parameters

- **blacklisted** (*Sequence*) – Strings to build the Retrie from.
- **match_substrings** (*bool*) – Whether to override word_boundary with "".
- **word_boundary** (*str*) – Token to wrap the retrie to exclude certain matches.
- **re_flags** (*re.RegexFlag*) – Flags passed to regex engine.

cleanse_text(*term*)

Return text, removing all blacklisted terms.

Parameters
term (*str*) – The string to search.

Return type
str

filter(*sequence*)

Construct an iterator from those elements of sequence not blacklisted.

Parameters
sequence (*Sequence*) – The sequence of strings to filter.

Return type
Iterator[str]

is_blacklisted(*term*)

Return True if Pattern is found in term.

Parameters
term (*str*) – The string to search.

Return type
bool

re_flags

Regex flags passed to `re.compile()`.

trie

The underlying *retrie.trie.Trie*.

word_boundary

The boundary token to wrap the *retrie.trie.Trie* pattern in.

class retrie.retrie.**Checklist**(*keys*, *match_substrings=False*, *word_boundary='\\b'*, *re_flags=RegexFlag.IGNORECASE | UNICODE*)

Bases: *Retrie*

Check and mutate strings against a Retrie.

Note: Although the Trie is case-sensitive, by default `re.IGNORECASE` is used for better performance. Pass `re_flags=None` to perform case-sensitive replacements.

Parameters

- **keys** (*Sequence*) – Strings to build the Retrie from.
- **match_substrings** (*bool*) – Whether to override word_boundary with "".
- **word_boundary** (*str*) – Token to wrap the retrie to exclude certain matches.

- **re_flags** (*re.RegexFlag*) – Flags passed to regex engine.

property compiled: Pattern[str]

Compute and cache the compiled Pattern.

is_listed(*term*)

Return True if Pattern is found in term.

Parameters
term (*str*) – The string to search.

Return type
bool

not_listed(*term*)

Return True if Pattern is not found in term.

Parameters
term (*str*) – The string to search.

Return type
bool

re_flags

Regex flags passed to `re.compile()`.

trie

The underlying *retrie.trie.Trie*.

word_boundary

The boundary token to wrap the *retrie.trie.Trie* pattern in.

class retrie.retrie.**Replacer**(*replacement_mapping*, *match_substrings=False*, *word_boundary='\\b'*, *re_flags=RegexFlag.IGNORECASE | UNICODE*)

Bases: *Checklist*

Replace occurrences of `replacement_mapping.keys()` with corresponding values.

Note: Although the Trie is case-sensitive, by default `re.IGNORECASE` is used for better performance. Pass `re_flags=None` to perform case-sensitive replacements.

Parameters

- **replacement_mapping** (*Mapping*) – Mapping `{old: new}` to replace.
- **match_substrings** (*bool*) – Whether to override word_boundary with "".
- **word_boundary** (*str*) – Token to wrap the retrie to exclude certain matches.
- **re_flags** (*re.RegexFlag*) – Flags passed to regex engine.

replace(*text*, *count=0*)

Replace occurrences of replacement_mapping.keys() with corresponding values.

Parameters

- **text** (*str*) – String to search & replace.
- **count** (*int*) – Amount of occurences to replace. If 0 or omitted, replace all.

Returns
String with matches replaced.

Return type
str

`replacement_mapping`

`class retrie.retrie.Retrie(word_boundary='\\b', re_flags=RegexFlag.IGNORECASE | UNICODE)`

Bases: `object`

Wrap a *retrie.trie.Trie* to compile the corresponding regex pattern with word boundary and regex flags.

Note: Although the Trie is case-sensitive, by default `re.IGNORECASE` is used for better performance. Pass `re_flags=None` to perform case-sensitive replacements.

Parameters

- **word_boundary** (*str*) – Token to wrap the retrie to exclude certain matches.
- **re_flags** (*re.RegexFlag*) – Flags passed to regex engine.

`compile(word_boundary=None, re_flags=-1)`

Compile a `re.Pattern` for the current Trie.

Optionally the following args can be passed to temporarily override class attrs.

Parameters

- **word_boundary** (*str*) – Token to wrap the retrie to exclude certain matches.
- **re_flags** (*re.RegexFlag*) – Flags passed to regex engine.

Returns
Pattern capturing the Trie items enclosed by word_boundary.

Return type
re.Pattern

`classmethod parse_re_flags(re_flags)`

Convert re_flags to integer.

Parameters
re_flags (*re.RegexFlag | int | None*) – The flags to cast to integer.

Return type
int

`pattern()`

Build regex pattern for the current Trie.

Returns
Non-capturing regex representation.

Return type
str

`re_flags`

Regex flags passed to `re.compile()`.

trie

The underlying *retrie.trie.Trie*.

word_boundary

The boundary token to wrap the *retrie.trie.Trie* pattern in.

class retrie.retrie.**Whitelist**(*whitelisted*, *match_substrings=False*, *word_boundary='\\b'*, *re_flags=RegexFlag.IGNORECASE | UNICODE*)

Bases: *Checklist*

Mutate [sequences of] strings based on their match against whitelisted.

Note: Although the Trie is case-sensitive, by default `re.IGNORECASE` is used for better performance. Pass `re_flags=None` to perform case-sensitive replacements.

Parameters

- **whitelisted** (*Sequence*) – Strings to build the Retrie from.
- **match_substrings** (*bool*) – Whether to override word_boundary with "".
- **word_boundary** (*str*) – Token to wrap the retrie to exclude certain matches.
- **re_flags** (*re.RegexFlag*) – Flags passed to regex engine.

cleanse_text(*term*)

Return text, only keeping whitelisted terms.

Parameters
term (*str*) – The string to search.

Return type
str

filter(*sequence*)

Construct an iterator from whitelisted elements of sequence.

Parameters
sequence (*Sequence*) – The sequence of strings to filter.

Return type
Iterator[str]

is_whitelisted(*term*)

Return True if Pattern is found in term.

Parameters
term (*str*) – The string to search.

Return type
bool

re_flags

Regex flags passed to `re.compile()`.

trie

The underlying *retrie.trie.Trie*.

word_boundary

The boundary token to wrap the *retrie.trie.Trie* pattern in.

2.1.3 retrie.trie module

Submodule containing code to build a regex pattern from a trie of strings.

Standalone usage:

```
from retrie.trie import Trie

trie = Trie()

trie.add("abc", "foo", "abs")
assert trie.pattern() == "(?:ab[cs]|foo)"  # equivalent to but faster than "(?
↪:abc|abs|foo)"

trie.add("absolute")
assert trie.pattern() == "(?:ab(?:c|s(?:olute)?)|foo)"

trie.add("abx")
assert trie.pattern() == "(?:ab(?:[cx]|s(?:olute)?)|foo)"

trie.add("abxy")
assert trie.pattern() == "(?:ab(?:c|s(?:olute)?|xy?)|foo)"
```

A *Trie* may be populated with zero or more strings at instantiation or via *Trie.add()*, from which method chaining is possible. Two instances can be merged with the + (new instance) and += (in-place update) operators. Instances will compare equal if their data dictionaries are equal.

```
trie = Trie()
trie += Trie("abc")
assert (
    trie + Trie().add("foo")
    == Trie("abc", "foo")
    == Trie(*["abc", "foo"])
    == Trie().add(*["abc", "foo"])
    == Trie().add("abc", "foo")
    == Trie().add("abc").add("foo")
)
```

class retrie.trie.**Trie**(**word*)

Bases: object

Create a Trie with zero or more words at instantiation or (later via *Trie.add()*).

The Trie can be exported to a Regex pattern via *Trie.pattern()*, which should match much faster than a simple Regex union. For best performance, pass the pattern to `re.compile()` and cache it to avoid recompiling for every search. See also *retrie.retrie.Checklist.compiled*.

Two instances can be merged with the + (new instance) and += (in-place update) operators. Instances will compare equal if their data dictionaries are equal.

Parameters
word (*str*) – A string to add to the Trie.

add(**word*)

Add one or more words to the current Trie.

Parameters
word (*str*) – A string to add to the Trie.

Return type
Trie

`data: Dict[str, Dict]`

`dump()`

Dump the current trie as dictionary.

Return type
Dict[str, *Dict*]

`pattern()`

Dump the current trie as regex string.

Return type
str

PYTHON MODULE INDEX

r

INDEX