bistring

Release 0.5.0

Tavian Barnes

Jul 27, 2022

CONTENTS:

The bistring library provides non-destructive versions of common string processing operations like normalization, case folding, and find/replace. Each bistring remembers the original string, and how its substrings map to substrings of the modified version.

CHAPTER

ONE

INTRODUCTION

Many operations commonly performed on text strings are destructive; that is, they lose some information about the original string. Systems that deal with text will commonly perform many of these operations on their input, whether it's changing case, performing unicode normalization, collapsing whitespace, stripping punctuation, etc. This helps systems behave in a more uniform manner regarding the many different ways you or I might express the same thing. But the consequence is that when handling parts of this processed text, it may be hard to know what exactly the user originally wrote. Sometimes those details can be very important to the user.

Consider an AI personal assistant, for example, that is helping a user send a text message to a friend. The user writes,

> send jane a text that says, "Hey! How are you? Haven't seen you in a while, what's up "

The system may perform some normalization on that text, such that it ends up looking like this, with casing and punctuation gone:

> send jane a text that says hey how are you havent seen you in a while whats up emoji

The AI may then identify that the body of the message should be:

> hey how are you havent seen you in a while whats up emoji

However, that message wouldn't make much sense as-is. If the assistant uses *bistring* though, it's easy for it to match that with the original text the user intended:

```
>>> from bistring import bistr
>>> query = bistr(
...     'send jane a text that says, '
...     '"Hey! How are you? Haven\'t seen you in a while, what\'s up "'
... )
```

```
>>> # Get rid of upper-/lower-case distinctions
>>> query = query.casefold()
>>> print(query.modified)
send jane a text that says, "hey! how are you? haven't seen you in a while, what's up "
```

```
>>> import regex
>>> # Remove all punctuation
>>> query = query.sub(regex.compile(r'\pP'), '')
>>> # Replace all symbols with 'emoji'
>>> query = query.sub(regex.compile(r'\pS'), 'emoji')
>>> print(query.modified)
send jane a text that says hey how are you havent seen you in a while whats up emoji
```

```
>>> # Extract the substring we care about, the message body
>>> message = query[27:84]
>>> print(message.modified)
hey how are you havent seen you in a while whats up emoji
>>> print(message.original)
Hey! How are you? Haven't seen you in a while, what's up
```

Every *bistr* keeps track of the original string it started with, and maintains a sequence alignment between the original and the modified strings. This alignment means that it knows exactly what substring of the original text is associated with every chunk of the modified text. So when you slice a *bistr*, you get the matching slice of original text automatically!

CHAPTER

TWO

FREQUENTLY ASKED QUESTIONS

2.1 What is a bistring, anyway?

Simply put, a *bistring* is a pair of strings, an original string and a modified one, along with information about how they align with each other. The *bistring.bistr* class has an API very similar to the built-in `str`, but all its operations keep track of the original string and the alignment for you.

```
>>> from bistring import bistr
>>> s = bistr('HELLO WORLD')
>>> print(s)
'HELLO WORLD'
>>> s = s.lower()
>>> print(s)
('HELLO WORLD'  'hello world')
>>> print(s[6:])
('WORLD'  'world')
```

2.2 Why am I getting more text than I expect when slicing?

When a bistring doesn't have precise enough alignment information to slice exactly, it will give you back the smallest string it knows for certain contains a match for the region you requested. In the worst case, that may be the entire string! This happens, for example, when you use the two-argument *bistr* constructor, which makes no effort to infer a granular alignment between the strings:

```
>>> s = bistr('color', 'colour')
>>> print(s[3:5])
('color'  'ou')
```

Instead, you should start from your original string as a *bistr*, and then transform it how you want:

```
>>> s = bistr('color')
>>> s = s.sub(r'(?<=col)o(?=r)', 'ou')
>>> print(s)
('color'  'colour')
>>> print(s[3:5])
('o'  'ou')
```

Alternatively, you can piece many smaller bistrings together to achieve the alignment you want manually:

```
>>> s = bistr('col') + bistr('o', 'ou') + bistr('r')
>>> print(s)
('color'  'colour')
>>> print(s[3:5])
('o'  'ou')
```

2.3 What if I don't know the alignment?

If at all possible, you should use *bistring* all the way through your text processing code, which will ensure an accurate alignment is tracked for you. If you don't control that code, or there are other reasons it won't work with *bistring*, you can still have us guess an alignment for you in simple cases with *bistring.bistr.infer()*.

```
>>> s = bistr.infer('color', 'colour')
>>> print(s[0:3])
'col'
>>> print(s[3:5])
('o'  'ou')
>>> print(s[5:6])
'r'
```

infer() is an expensive operation (`O(N*M)` in the length of the strings), so if you absolutely need it, try to use it only for short strings.

2.4 How do I get the actual indices, rather than just substrings?

Use *bistring.bistr.alignment*:

```
>>> s = bistr('The quick, brown ')
>>> s = s.replace(',', '')
>>> s = s.replace('', 'fox')
>>> print(s[16:19])
(''  'fox')
>>> s.alignment.original_bounds(16, 19)
(17, 18)
>>> s.alignment.modified_bounds(11, 16)
(10, 15)
>>> print(s[10:15])
'brown'
```

See *bistring.Alignment* for more details.

2.5 How do I perform case-insensitive operations?

Use *bistring.bistr.casefold()*. Do not use *lower()*, *upper()*, or any other method, as you will get wrong results for many non-English languages.

To check case-insensitive equality, you don't even need *bistring*:

```
>>> 'HELLO WORLD!'.casefold() == 'HeLlO wOrLd!'.casefold()
True
```

To search for a substring case-insensitively:

```
>>> s = bistr('Bundesstraße').casefold()
>>> s.find_bounds('STRASSE'.casefold())
(6, 13)
>>> print(s[6:13])
('straße'  'strasse')
```

2.6 Forget case insensitivity, how do I make sure that identical looking strings compare equal?

This is a hard problem with Unicode strings. To start with, you should at least perform some kind of Unicode normalization. That ensures that different ways of writing the semantically identical thing (e.g. with precomposed accented characters vs. combining accents) become actually identical:

```
>>> a = bistr('\u00EAtre')  # 'être' with a single character for the ê
>>> b = bistr('e\u0302tre') # 'etre' with an 'e' and a combining '^'
>>> a.normalize('NFC').modified == b.normalize('NFC').modified
True
>>> a.normalize('NFD').modified == b.normalize('NFD').modified
True
```

Normalization form NFC tries to keep precomposed characters together whenever possible, while NFD always decomposes them. In general, NFC is more convenient for people to work with, but NFD can be useful for things like removing accents and other combining marks from text.

2.7 What about similar-looking strings, that aren't necessarily identical?

Unicode contains things like ligatures, alternative scripts, and other oddities than can result in similar-looking strings that are represented very differently. Here is where the "compatibility" normalization forms, NFKC and NFKD, can help:

```
>>> s = bistr(' ')
>>> s = s.normalize('NFKC')
>>> print(s)
(' '  'Hello world')
>>> print(s[6:])
(''  'world')
```

2.8 How do I ensure I get the same results on every machine?

Always pass an explicit locale to any *bistr* method that takes one. Many of Python's string APIs implicitly use the system's default locale, which may be quite different than the one you developed with. While this may be the right behaviour if you're displaying strings to the current user, it's rarely the right behaviour if you're dealing with text that originated or will be displayed elsewhere, e.g. for cloud software. *bistr* always accepts a locale parameter in these APIs, to ensure reproducible and sensible results:

```
>>> # s will be 'I' in most locales, but 'İ' in Turkish locales!
>>> s = bistr('i').upper()
>>> # An English locale guarantees a dotless capital I
>>> print(bistr('i').upper('en_US'))
('i'  'I')
>>> # A Turkish locale gives a dotted capital İ
>>> print(bistr('i').upper('tr_TR'))
('i'  'İ')
```

2.9 Tokenization

2.9.1 How do I tokenize text in a reversible way?

bistring provides some convenient tokenization APIs that track string indices. To use Unicode word boundary rules, for example:

```
>>> from bistring import WordTokenizer
>>> tokenizer = WordTokenizer('en_US')
>>> tokens = tokenizer.tokenize('The quick, brown fox jumps over the lazy dog')
>>> print(tokens[1])
[4:9]='quick'
```

2.9.2 How do I find the whole substring of text for some tokens?

bistring.Tokenization.substring() gives the substring itself. *bistring.Tokenization.text_bounds()* gives the bounds of that substring.

```
>>> print(tokens.substring(1, 3))
'quick, brown'
>>> tokens.text_bounds(1, 3)
(4, 16)
```

2.9.3 How do I find the tokens for a substring of text?

bistring.Tokenization.bounds_for_text()

```
>>> tokens.bounds_for_text(4, 16)
(1, 3)
>>> print(tokens.substring(1, 3))
'quick, brown'
```

2.9.4 How to I snap a substring of text to the nearest token boundaries?

bistring.Tokenization.snap_text_bounds()

```
>>> print(tokens.text[6:14])
'ick, bro'
>>> tokens.snap_text_bounds(6, 14)
(4, 16)
>>> print(tokens.text[4:16])
'quick, brown'
```

2.9.5 What if I don't know the token positions?

If at all possible, you should use a *bistring.Tokenizer* or some other method that tokenizes with position information. If you can't, you can use *bistring.Tokenization.infer()* to guess the alignment for you:

```
>>> from bistring import Tokenization
>>> tokens = Tokenization.infer('hello, world!', ['hello', 'world'])
>>> print(tokens[0])
[0:5]='hello'
>>> print(tokens[1])
[7:12]='world'
```

CHAPTER

THREE

PYTHON

3.1 bistr

class bistring.**bistr**(*original: Union[str,* bistring._bistr.bistr*], modified: Optional[str] = None, alignment: Optional[*bistring._alignment.Alignment*] = None*)

Bases: `object`

A bidirectionally transformed string.

A *bistr* can be constructed from only a single string, which will give it identical original and modified strings and an identity alignment:

```
>>> s = bistr('test')
>>> s.original
'test'
>>> s.modified
'test'
>>> s.alignment
Alignment.identity(4)
```

You can also explicitly specify both the original and modified string. The inferred alignment will be as course as possible:

```
>>> s = bistr('TEST', 'test')
>>> s.original
'TEST'
>>> s.modified
'test'
>>> s.alignment
Alignment([(0, 0), (4, 4)])
```

Finally, you can specify the alignment explicitly too, if you know it:

```
>>> s = bistr('TEST', 'test', Alignment.identity(4))
>>> s[1:3]
bistr('ES', 'es', Alignment.identity(2))
```

original: str
The original string, before any modifications.

modified: str
The current value of the string, after all modifications.

alignment: *bistring._alignment.Alignment*
The sequence alignment between *original* and *modified*.

classmethod infer(*original*, *modified*, *cost_fn=None*)
Create a *bistr*, automatically inferring an alignment between the *original* and *modified* strings.

This method can be useful if the modified string was produced by some method out of your control. If at all possible, you should start with `bistr(original)` and perform non-destructive operations to get to the modified string instead.

```
>>> s = bistr.infer('color', 'colour')
>>> print(s[0:3])
'col'
>>> print(s[3:5])
('o'  'ou')
>>> print(s[5:6])
'r'
```

infer() tries to be intelligent about certain aspects of Unicode, which enables it to guess good alignments between strings that have been case-mapped, normalized, etc.:

```
>>> s = bistr.infer(
...     ' ,       ',
...     'the quick brown fox jumps over the lazy dog',
... )
>>> print(s[0:3])
('' 'the')
>>> print(s[4:9])
('' 'quick')
>>> print(s[10:15])
('' 'brown')
>>> print(s[16:19])
('' 'fox')
```

Warning: this operation has time complexity `O(N*M)`, where *N* and *M* are the lengths of the original and modified strings, and so should only be used for relatively short strings.

Parameters

- **original** (str) – The original string
- **modified** (str) – The modified string.
- **cost_fn** (Optional[Callable[[Optional[str], Optional[str]], Union[int, float]]]) – A function returning the cost of performing an edit (see *Alignment.infer()*).

Return type *bistr*

Returns A *bistr* with the inferred alignment.

__getitem__(*index: int*) → str
__getitem__(*index: slice*) → *bistring._bistr.bistr*
Indexing a *bistr* returns the nth character of the modified string:

```
>>> s = bistr('TEST').lower()
>>> s[1]
'e'
```

Slicing a *bistr* extracts a substring, complete with the matching part of the original string:

```
>>> s = bistr('TEST').lower()
>>> s[1:3]
bistr('ES', 'es', Alignment.identity(2))
```

Return type Union[str, *bistr*]

inverse()

Return type *bistr*

Returns

The inverse of this string, swapping the original and modified strings.

```
>>> s = bistr('HELLO WORLD').lower()
>>> s
bistr('HELLO WORLD', 'hello world', Alignment.identity(11))
>>> s.inverse()
bistr('hello world', 'HELLO WORLD', Alignment.identity(11))
```

chunks()

Return type Iterable[*bistr*]

Returns All the chunks of associated text in this string.

count(*sub*, *start=None*, *end=None*)
Like `str.count()`, counts the occurrences of *sub* in the string.

Return type int

find(*sub*, *start=None*, *end=None*)
Like `str.find()`, finds the position of *sub* in the string.

Return type int

find_bounds(*sub*, *start=None*, *end=None*)
Like *find()*, but returns both the start and end bounds for convenience.

Return type Tuple[int, int]

Returns The first *i, j* within *[start, end)* such that `self[i:j] == sub`, or `(-1, -1)` if not found.

rfind(*sub*, *start=None*, *end=None*)
Like `str.rfind()`, finds the position of *sub* in the string backwards.

Return type int

rfind_bounds(*sub*, *start=None*, *end=None*)
Like *rfind()*, but returns both the start and end bounds for convenience.

Return type Tuple[int, int]

Returns The last *i, j* within *[start, end)* such that `self[i:j] == sub`, or `(-1, -1)` if not found.

index(*sub*, *start=None*, *end=None*)
Like `str.index()`, finds the first position of *sub* in the string, otherwise raising a *ValueError*.

Return type int

index_bounds(*sub*, *start=None*, *end=None*)
Like *index()*, but returns both the start and end bounds for convenience. If the substring is not found, a ValueError is raised.

Return type Tuple[int, int]

Returns The first *i, j* within *[start, end)* such that `self[i:j] == sub`.

Raises ValueError if the substring is not found.

rindex(*sub*, *start=None*, *end=None*)
Like str.index(), finds the last position of *sub* in the string, otherwise raising a *ValueError*.

Return type int

rindex_bounds(*sub*, *start=None*, *end=None*)
Like *rindex()*, but returns both the start and end bounds for convenience. If the substring is not found, a ValueError is raised.

Return type Tuple[int, int]

Returns The last *i, j* within *[start, end)* such that `self[i:j] == sub`.

Raises ValueError if the substring is not found.

startswith(*prefix*, *start=None*, *end=None*)
Like str.startswith(), checks if the string starts with the given *prefix*.

Return type bool

endswith(*suffix*, *start=None*, *end=None*)
Like str.endswith(), checks if the string starts with the given *suffix*.

Return type bool

join(*iterable*)
Like str.join(), concatenates many (bi)strings together.

Return type *bistr*

split(*sep=None*, *maxsplit=- 1*)
Like str.split(), splits this string on a separator.

Return type List[*bistr*]

partition(*sep*)
Like str.partition(), splits this string into three chunks on a separator.

Return type Tuple[*bistr*, *bistr*, *bistr*]

rpartition(*sep*)
Like str.rpartition(), splits this string into three chunks on a separator, searching from the end.

Return type Tuple[*bistr*, *bistr*, *bistr*]

center(*width*, *fillchar=' '*)
Like str.center(), pads the start and end of the string to center it.

Return type *bistr*

ljust(*width*, *fillchar=' '*)
Like str.ljust(), pads the end of the string to a fixed width.

Return type *bistr*

rjust(*width*, *fillchar=' '*)
Like `str.rjust()`, pads the start of the string to a fixed width.

Return type *bistr*

casefold()
Computes the case folded form of this string. Case folding is used for case-insensitive operations, and the result may not be suitable for displaying to a user. For example:

```
>>> s = bistr('straße').casefold()
>>> s.modified
'strasse'
>>> s[4:6]
bistr('ß', 'ss')
```

Return type *bistr*

lower(*locale=None*)
Converts this string to lowercase. Unless you specify the *locale* parameter, the current system locale will be used.

```
>>> bistr('HELLO WORLD').lower()
bistr('HELLO WORLD', 'hello world', Alignment.identity(11))
>>> bistr('I').lower('en_US')
bistr('I', 'i')
>>> bistr('I').lower('tr_TR')
bistr('I', 'ı')
```

Return type *bistr*

upper(*locale=None*)
Converts this string to uppercase. Unless you specify the *locale* parameter, the current system locale will be used.

```
>>> bistr('hello world').upper()
bistr('hello world', 'HELLO WORLD', Alignment.identity(11))
>>> bistr('i').upper('en_US')
bistr('i', 'I')
>>> bistr('i').upper('tr_TR')
bistr('i', 'İ')
```

Return type *bistr*

title(*locale=None*)
Converts this string to title case. Unless you specify the *locale* parameter, the current system locale will be used.

```
>>> bistr('hello world').title()
bistr('hello world', 'Hello World', Alignment.identity(11))
>>> bistr('istanbul').title('en_US')
bistr('istanbul', 'Istanbul', Alignment.identity(8))
>>> bistr('istanbul').title('tr_TR')
bistr('istanbul', 'İstanbul', Alignment.identity(8))
```

Return type *bistr*

capitalize(*locale=None*)

Capitalize the first character of this string, and lowercase the rest. Unless you specify the *locale* parameter, the current system locale will be used.

```
>>> bistr('hello WORLD').capitalize()
bistr('hello WORLD', 'Hello world', Alignment.identity(11))
>>> bistr('').capitalize('el_GR')
bistr('', '', Alignment.identity(2))
```

Return type *bistr*

swapcase(*locale=None*)

Swap the case of every letter in this string. Unless you specify the *locale* parameter, the current system locale will be used.

```
>>> bistr('hello WORLD').swapcase()
bistr('hello WORLD', 'HELLO world', Alignment.identity(11))
```

Some Unicode characters, such as title-case ligatures and digraphs, don't have a case-swapped equivalent:

```
>>> bistr('epòta').swapcase('hr_HR')
bistr('epòta', 'EPÒTA', Alignment.identity(6))
```

In these cases, compatibilty normalization may help:

```
>>> s = bistr('epòta')
>>> s = s.normalize('NFKC')
>>> s = s.swapcase('hr_HR')
>>> print(s)
('epòta'  'lJEPÒTA')
```

Return type *bistr*

expandtabs(*tabsize=8*)

Like `str.expandtabs()`, replaces tab (`\t`) characters with spaces to align on multiples of *tabsize*.

Return type *bistr*

replace(*old*, *new*, *count=None*)

Like `str.replace()`, replaces occurrences of *old* with *new*.

Return type *bistr*

sub(*regex*, *repl*)

Like `re.sub()`, replaces all matches of *regex* with the replacement *repl*.

Parameters

- **regex** (`Union[str, Pattern[str]]`) – The regex to match. Can be a string pattern or a compiled regex.
- **repl** (`Union[str, Callable[[Match[str]], str]]`) – The replacement to use. Can be a string, which is interpreted as in `re.Match.expand()`, or a *callable*, which will receive each match and return the replacement string.

Return type *bistr*

strip(*chars=None*)
Like `str.strip()`, removes leading and trailing characters (whitespace by default).

Return type *bistr*

lstrip(*chars=None*)
Like `str.lstrip()`, removes leading characters (whitespace by default).

Return type *bistr*

rstrip(*chars=None*)
Like `str.rstrip()`, removes trailing characters (whitespace by default).

Return type *bistr*

normalize(*form*)
Like `unicodedata.normalize()`, applies a Unicode normalization form. The choices for *form* are:

- `'NFC'`: Canonical Composition
- `'NFKC'`: Compatibility Composition
- `'NFD'`: Canonical Decomposition
- `'NFKD'`: Compatibilty Decomposition

Return type *bistr*

3.2 BistrBuilder

class bistring.**BistrBuilder**(*original*)
Bases: `object`

Bidirectionally transformed string builer.

A *BistrBuilder* builds a transformed version of a source string iteratively. Each builder has an immutable original string, a current string, and the in-progress modified string, with alignments between each. For example:

```
original: |The| |quick,| |brown| || |jumps| |over| |the| |lazy| ||
          |   | |      | |     | |  \ \     \ \    \ \   \ \    \ \   \
current:  |The| |quick,| |brown| |fox| |jumps| |over| |the| |lazy| |dog|
          |   | |      / /     /
modified: |the| |quick| |brown| ...
```

The modified string is built in pieces by calling *replace()* to change *n* characters of the current string into new ones in the modified string. Convenience methods like *skip()*, *insert()*, and *discard()* are implemented on top of this basic primitive.

```
>>> b = BistrBuilder('The quick, brown  jumps over the lazy ')
>>> b.skip(17)
>>> b.peek(1)
''
>>> b.replace(1, 'fox')
>>> b.skip(21)
>>> b.peek(1)
''
>>> b.replace(1, 'dog')
>>> b.is_complete
```

(continues on next page)

(continued from previous page)

```
True
>>> b.rewind()
>>> b.peek(3)
'The'
>>> b.replace(3, 'the')
>>> b.skip(1)
>>> b.peek(6)
'quick,'
>>> b.replace(6, 'quick')
>>> b.skip_rest()
>>> s = b.build()
>>> s.modified
'the quick brown fox jumps over the lazy dog'
```

Parameters **original** (Union[str, *bistr*]) – The string to start from.

property original: str
The original string being modified.

Return type str

property current: str
The current string before modifications.

Return type str

property modified: str
The modified string as built so far.

Return type str

property alignment: *bistring._alignment.Alignment*
The alignment built so far from self.current to self.modified.

Return type *Alignment*

property position: int
The position of the builder in self.current.

Return type int

property remaining: int
The number of characters of the current string left to process.

Return type int

property is_complete: bool
Whether we've completely processed the string. In other words, whether the modified string aligns with the end of the current string.

Return type bool

peek(*n*)
Peek at the next *n* characters of the original string.

Return type str

skip(*n*)
Skip the next *n* characters, copying them unchanged.

Return type None

skip_rest()
Skip the rest of the string, copying it unchanged.

Return type None

insert(*string*)
Insert a substring into the string.

Return type None

discard(*n*)
Discard a portion of the original string.

Return type None

discard_rest()
Discard the rest of the original string.

Return type None

replace(*n*, *repl*)
Replace the next *n* characters with a new string.

Return type None

append(*bs*)
Append a bistr. The original value of the bistr must match the current string being processed.

Return type None

skip_match(*regex*)
Skip a substring matching a regex, copying it unchanged.

Parameters **regex** (Union[str, Pattern[str]]) – The (possibly compiled) regular expression to match.

Return type bool

Returns Whether a match was found.

discard_match(*regex*)
Discard a substring that matches a regex.

Parameters **regex** (Union[str, Pattern[str]]) – The (possibly compiled) regular expression to match.

Return type bool

Returns Whether a match was found.

replace_match(*regex*, *repl*)
Replace a substring that matches a regex.

Parameters

- **regex** (Union[str, Pattern[str]]) – The (possibly compiled) regular expression to match.
- **repl** (Union[str, Callable[[Match[str]], str]]) – The replacement to use. Can be a string, which is interpreted as in re.Match.expand(), or a *callable*, which will receive each match and return the replacement string.

Return type bool

Returns Whether a match was found.

replace_next(*regex*, *repl*)
Replace the next occurence of a regex.

Parameters

- **regex** (Union[str, Pattern[str]]) – The (possibly compiled) regular expression to match.
- **repl** (Union[str, Callable[[Match[str]], str]]) – The replacement to use.

Return type bool

Returns Whether a match was found.

replace_all(*regex*, *repl*)
Replace all occurences of a regex.

Parameters

- **regex** (Union[str, Pattern[str]]) – The (possibly compiled) regular expression to match.
- **repl** (Union[str, Callable[[Match[str]], str]]) – The replacement to use.

Return type None

build()
Build the *bistr*.

Return type *bistr*

Returns A *bistr* from the original string to the new modified string.

Raises ValueError if the modified string is not completely built yet.

rewind()
Reset this builder to apply another transformation.

Raises ValueError if the modified string is not completely built yet.

Return type None

3.3 Alignment

class bistring.**Alignment**(*values*)
Bases: object

An alignment between two related sequences.

Consider this alignment between two strings:

```
|it's| |aligned!|
|    \ \         |
|it is| |aligned|
```

An alignment stores all the indices that are known to correspond between the original and modified sequences. For the above example, it would be

```
>>> a = Alignment([
...     (0, 0),
...     (4, 5),
```

(continues on next page)

(continued from previous page)

```
...     (5, 6),
...     (13, 13),
... ])
```

Alignments can be used to answer questions like, "what's the smallest range of the original sequence that is guaranteed to contain this part of the modified sequence?" For example, the range `(0, 5)` ("it is") is known to match the range `(0, 4)` ("it's") of the original sequence:

```
>>> a.original_bounds(0, 5)
(0, 4)
```

Results may be imprecise if the alignment is too course to match the exact inputs:

```
>>> a.original_bounds(0, 2)
(0, 4)
```

A more granular alignment like this:

```
|i|t|'s| |a|l|i|g|n|e|d|!|
| | |  \ \ \ \ \ \ \ \ \ /
|i|t| is| |a|l|i|g|n|e|d|
```

```
>>> a = Alignment([
...     (0, 0), (1, 1), (2, 2), (4, 5), (5, 6), (6, 7), (7, 8),
...     (8, 9), (9, 10), (10, 11), (11, 12), (12, 13), (13, 13),
... ])
```

Can be more precise:

```
>>> a.original_bounds(0, 2)
(0, 2)
```

Parameters **values** (Iterable[Tuple[int, int]]) – The sequence of aligned indices. Each element should be a tuple `(x, y)`, where *x* is the original sequence position and *y* is the modified sequence position.

classmethod **identity**(*__length: int*) → *bistring._alignment.Alignment*
classmethod **identity**(*__start: int*, *__stop: int*) → *bistring._alignment.Alignment*
classmethod **identity**(*__bounds: Union[range, slice, Tuple[int, int]]*) → *bistring._alignment.Alignment*

Create an identity alignment, which maps all intervals to themselves. You can pass the size of the sequence:

```
>>> Alignment.identity(5)
Alignment.identity(5)
```

or the start and end positions:

```
>>> Alignment.identity(1, 5)
Alignment.identity(1, 5)
```

or a range-like object (range, slice, or Tuple[int, int]):

```
>>> Alignment.identity(range(1, 5))
Alignment.identity(1, 5)
```

Return type *Alignment*

classmethod infer(*original*, *modified*, *cost_fn=None*)
Infer the alignment between two sequences with the lowest edit distance.

```
>>> Alignment.infer('color', 'color')
Alignment.identity(5)
>>> a = Alignment.infer('color', 'colour')
>>> # 'ou' -> 'o'
>>> a.original_bounds(3, 5)
(3, 4)
```

Warning: this operation has time complexity `O(N*M)`, where *N* and *M* are the lengths of the original and modified sequences, and so should only be used for relatively short sequences.

Parameters

- **original** (Sequence[TypeVar(T)]) – The original sequence.
- **modified** (Sequence[TypeVar(U)]) – The modified sequence.
- **cost_fn** (Optional[Callable[[Optional[TypeVar(T)], Optional[TypeVar(U)]], Union[int, float]]]) – A function returning the cost of performing an edit. `cost_fn(a, b)` returns the cost of replacing *a* with *b*. `cost_fn(a, None)` returns the cost of deleting *a*, and `cost_fn(None, b)` returns the cost of inserting *b*. By default, all operations have cost 1 except replacing identical elements, which has cost 0.

Return type *Alignment*

Returns The inferred alignment.

__getitem__(*index: int*) → Tuple[int, int]
__getitem__(*index: slice*) → *bistring._alignment.Alignment*
Indexing an alignment returns the nth pair of aligned positions:

```
>>> a = Alignment.identity(5)
>>> a[3]
(3, 3)
```

Slicing an alignment returns a new alignment with a subrange of its values:

```
>>> a[1:5]
Alignment.identity(1, 4)
```

Return type Union[Tuple[int, int], *Alignment*]

shift(*delta_o*, *delta_m*)
Shift this alignment.

Parameters

- **delta_o** (int) – The distance to shift the original sequence.
- **delta_m** (int) – The distance to shift the modified sequence.

Return type *Alignment*

Returns An alignment with all the positions shifted by the given amounts.

original_bounds(**args*)

Maps a subrange of the modified sequence to the original sequence. Can be called with either two arguments:

```
>>> a = Alignment.identity(5).shift(1, 0)
>>> a.original_bounds(1, 3)
(2, 4)
```

or with a range-like object:

```
>>> a.original_bounds(range(1, 3))
(2, 4)
```

With no arguments, returns the bounds of the entire original sequence:

```
>>> a.original_bounds()
(1, 6)
```

Return type Tuple[int, int]

Returns The corresponding bounds in the original sequence.

original_range(**args*)

Like *original_bounds()*, but returns a range.

Return type range

original_slice(**args*)

Like *original_bounds()*, but returns a slice.

Return type slice

modified_bounds(**args*)

Maps a subrange of the original sequence to the modified sequence. Can be called with either two arguments:

```
>>> a = Alignment.identity(5).shift(1, 0)
>>> a.modified_bounds(2, 4)
(1, 3)
```

or with a range-like object:

```
>>> a.modified_bounds(range(2, 4))
(1, 3)
```

With no arguments, returns the bounds of the entire modified sequence:

```
>>> a.modified_bounds()
(0, 5)
```

Return type Tuple[int, int]

Returns The corresponding bounds in the modified sequence.

modified_range(**args*)

Like *modified_bounds()*, but returns a range.

Return type range

modified_slice(**args*)
Like *modified_bounds()*, but returns a range.

Return type slice

slice_by_original(**args*)
Slice this alignment by a span of the original sequence.

```
>>> a = Alignment.identity(5).shift(1, 0)
>>> a.slice_by_original(2, 4)
Alignment([(2, 1), (3, 2), (4, 3)])
```

Return type *Alignment*

Returns The slice of this alignment that corresponds with the given span of the original sequence.

slice_by_modified(**args*)
Slice this alignment by a span of the modified sequence.

```
>>> a = Alignment.identity(5).shift(1, 0)
>>> a.slice_by_modified(1, 3)
Alignment([(2, 1), (3, 2), (4, 3)])
```

Return type *Alignment*

Returns The slice of this alignment that corresponds with the given span of the modified sequence.

compose(*other*)

Return type *Alignment*

Returns A new alignment equivalent to applying this one first, then the *other*.

inverse()

Return type *Alignment*

Returns The inverse of this alignment, from the modified to the original sequence.

3.4 Tokenization

class bistring.**Token**(*text*, *start*, *end*)
Bases: object

A token extracted from a string.

Parameters

- **text** (Union[str, *bistr*]) – The text of this token.
- **start** (int) – The starting index of this token.
- **end** (int) – The ending index of this token.

text: *bistring._bistr.bistr*
The actual text of the token.

start: *int*
The start position of the token.

end: *int*
The end position of the token.

property original: *str*
The original value of this token.

Return type str

property modified: *str*
The modified value of this token.

Return type str

classmethod slice(*text*, *start*, *end*)
Create a Token from a slice of a bistr.

Parameters

- **text** (Union[str, *bistr*]) – The (bi)string to slice.
- **start** (int) – The starting index of the token.
- **end** (int) – The ending index of the token.

Return type *Token*

class bistring.**Tokenization**(*text*, *tokens*)
Bases: object

A string and its tokenization.

Parameters

- **text** (Union[str, *bistr*]) – The text from which the tokens have been extracted.
- **tokens** (Iterable[*Token*]) – The tokens extracted from the text.

text: *bistring._bistr.bistr*
The text that was tokenized.

alignment: *bistring._alignment.Alignment*
The alignment from text indices to token indices.

classmethod infer(*text*, *tokens*)
Infer a *Tokenization* from a sequence of tokens.

```
>>> tokens = Tokenization.infer('hello, world!', ['hello', 'world'])
>>> tokens[0]
Token(bistr('hello'), start=0, end=5)
>>> tokens[1]
Token(bistr('world'), start=7, end=12)
```

Due to the possibility of ambiguity, it is much better to use a *Tokenizer* or some other method of producing *Token*s with their positions explicitly set.

Return type *Tokenization*

Returns The inferred tokenization, with token positions found by simple forward search.

Raises ValueError if the tokens can't be found in the source string.

__getitem__(*index: int*) → *bistring._token.Token*

__getitem__(*index: slice*) → *bistring._token.Tokenization*

Indexing a *Tokenization* returns the nth token:

```
>>> tokens = Tokenization.infer(
...     "The quick, brown fox",
...     ["The", "quick", "brown", "fox"],
... )
>>> tokens[0]
Token(bistr('The'), start=0, end=3)
```

Slicing a *Tokenization* returns a new one with the requested slice of tokens:

```
>>> tokens = tokens[1:-1]
>>> tokens[0]
Token(bistr('quick'), start=4, end=9)
```

Return type Union[*Token*, *Tokenization*]

substring(**args*)

Map a span of tokens to the corresponding substring. With no arguments, returns the substring from the first to the last token.

Return type *bistr*

text_bounds(**args*)

Map a span of tokens to the bounds of the corresponding text. With no arguments, returns the bounds from the first to the last token.

Return type Tuple[int, int]

original_bounds(**args*)

Map a span of tokens to the bounds of the corresponding original text. With no arguments, returns the bounds from the first to the last token.

Return type Tuple[int, int]

bounds_for_text(**args*)

Map a span of text to the bounds of the corresponding span of tokens.

Return type Tuple[int, int]

bounds_for_original(**args*)

Map a span of original text to the bounds of the corresponding span of tokens.

Return type Tuple[int, int]

slice_by_text(**args*)

Map a span of text to the corresponding span of tokens.

Return type *Tokenization*

slice_by_original(**args*)

Map a span of the original text to the corresponding span of tokens.

Return type *Tokenization*

snap_text_bounds(**args*)

Expand a span of text to align it with token boundaries.

Return type Tuple[int, int]

snap_original_bounds(**args*)
Expand a span of original text to align it with token boundaries.

Return type Tuple[int, int]

3.5 Tokenizer

class bistring.**Tokenizer**
Bases: abc.ABC

Abstract base class for tokenizers.

abstract tokenize(*text*)
Tokenize some text.

Parameters **text** (Union[str, *bistr*]) – The text to tokenize, as either an *str* or *bistr*. A plain *str* should be converted to a *bistr* before processing.

Return type *Tokenization*

Returns A *Tokenization* holding the text and its tokens.

class bistring.**RegexTokenizer**(*regex*)
Bases: *bistring._token.Tokenizer*

Breaks text into tokens based on a regex.

```
>>> tokenizer = RegexTokenizer(r'\w+')
>>> tokens = tokenizer.tokenize('the quick brown fox jumps over the lazy dog')
>>> tokens[0]
Token(bistr('the'), start=0, end=3)
>>> tokens[1]
Token(bistr('quick'), start=4, end=9)
```

Parameters **regex** (Union[str, Pattern[str]]) – A (possibly compiled) regular expression that matches tokens to extract.

class bistring.**SplittingTokenizer**(*regex*)
Bases: *bistring._token.Tokenizer*

Splits text into tokens based on a regex.

```
>>> tokenizer = SplittingTokenizer(r'\s+')
>>> tokens = tokenizer.tokenize('the quick brown fox jumps over the lazy dog')
>>> tokens[0]
Token(bistr('the'), start=0, end=3)
>>> tokens[1]
Token(bistr('quick'), start=4, end=9)
```

Parameters **regex** (Union[str, Pattern[str]]) – A (possibly compiled) regular expression that matches the regions between tokens.

class bistring.**CharacterTokenizer**(*locale*)
Bases: bistring._token._IcuTokenizer

Splits text into user-perceived characters/grapheme clusters.

```
>>> tokenizer = CharacterTokenizer('th_TH')
>>> tokens = tokenizer.tokenize('')
>>> tokens[0]
Token(bistr(''), start=0, end=2)
>>> tokens[1]
Token(bistr(''), start=2, end=4)
>>> tokens[2]
Token(bistr(''), start=4, end=5)
```

Parameters **locale** (str) – The name of the locale to use for computing user-perceived character boundaries.

class bistring.**WordTokenizer**(*locale*)

Bases: bistring._token._IcuTokenizer

Splits text into words based on Unicode rules.

```
>>> tokenizer = WordTokenizer('en_US')
>>> tokens = tokenizer.tokenize('the quick brown fox jumps over the lazy dog')
>>> tokens[0]
Token(bistr('the'), start=0, end=3)
>>> tokens[1]
Token(bistr('quick'), start=4, end=9)
```

Parameters **locale** (str) – The name of the locale to use for computing word boundaries.

class bistring.**SentenceTokenizer**(*locale*)

Bases: bistring._token._IcuTokenizer

Splits text into sentences based on Unicode rules.

```
>>> tokenizer = SentenceTokenizer('en_US')
>>> tokens = tokenizer.tokenize(
...     'Word, sentence, etc. boundaries are hard. Luckily, Unicode can help.'
... )
>>> tokens[0]
Token(bistr('Word, sentence, etc. boundaries are hard. '), start=0, end=42)
>>> tokens[1]
Token(bistr('Luckily, Unicode can help.'), start=42, end=68)
```

Parameters **locale** (str) – The name of the locale to use for computing sentence boundaries.

CHAPTER

FOUR

JAVASCRIPT

4.1 BiString

class BiString(*original*, *modified*, *alignment*)

A bidirectionally transformed string.

A BiString can be constructed from only a single string, which will give it identical original and modified strings and an identity alignment:

```
new BiString("test");
```

You can also explicitly specify both the original and modified strings. The inferred alignment will be as course as possible:

```
new BiString("TEST", "test");
```

Finally, you can specify the alignment explicitly, if you know it:

```
new BiString("TEST", "test", Alignment.identity(4));
```

Arguments

- **original** (`string()`) – The original string, before any modifications.
- **modified** (`string()`) – The modified string, after any modifications.
- **alignment** (*Alignment()*) – The alignment between the original and modified strings.

BiString.**alignment**

type: Alignment

The sequence alignment between *original* and *modified*.

BiString.**length**

type: number

The length of the modified string.

BiString.**modified**

type: string

The current value of the string, after all modifications.

BiString.**original**

type: string

The original string, before any modifications.

BiString.**[iterator]**()
Iterates over the code points in the modified string.

Returns **IterableIterator<string>** –

BiString.**boundsOf**(*searchValue*, *fromIndex*)
Like *indexOf()*, but returns both the start and end positions for convenience.

Arguments

- **searchValue** (string()) –
- **fromIndex** (number()) –

Returns **Bounds** –

BiString.**charAt**(*pos*)
Like String.prototype.charAt(), returns a code unit as a string from the modified string.

Arguments

- **pos** (number()) –

Returns **string** –

BiString.**charCodeAt**(*pos*)
Like String.prototype.charCodeAt(), returns a code unit as a number from the modified string.

Arguments

- **pos** (number()) –

Returns **number** –

BiString.**codePointAt**(*pos*)
Like String.prototype.codePointAt(), returns a code point from the modified string.

Arguments

- **pos** (number()) –

Returns **undefined|number** –

BiString.**concat**(*...others*)
Concatenate this string together with one or more others. The additional strings can be either BiStrings or normal strings.

Arguments

- **others** (AnyString[]()) –

Returns **BiString** –

BiString.**endsWith**(*searchString*, *position*)
Like String.prototype.endsWith(), returns whether this string ends with the given prefix.

Arguments

- **searchString** (string()) –
- **position** (number()) –

Returns **boolean** –

BiString.**equals**(*other*)

Arguments

- **other** (*BiString()*) –

Returns **boolean** – Whether this BiString is equal to another.

BiString.**indexOf**(*searchValue*, *fromIndex*)
Like `String.prototype.indexOf()`, finds the first occurrence of a substring.

Arguments

- **searchValue** (`string()`) –
- **fromIndex** (`number()`) –

Returns **number** –

BiString.**inverse**()

Returns **BiString** – The inverse of this string, swapping the original and modified strings.

BiString.**join**(*items*)
Like `Array.prototype.join()`, joins a sequence together with this *BiString* as the separator.

Arguments

- **items** (`Iterable`) –

Returns **BiString** –

BiString.**lastBoundsOf**(*searchValue*, *fromIndex*)
Like *lastIndexOf()*, but returns both the start and end positions for convenience.

Arguments

- **searchValue** (`string()`) –
- **fromIndex** (`number()`) –

Returns **Bounds** –

BiString.**lastIndexOf**(*searchValue*, *fromIndex*)
Like `String.prototype.lastIndexOf()`, finds the last occurrence of a substring.

Arguments

- **searchValue** (`string()`) –
- **fromIndex** (`number()`) –

Returns **number** –

BiString.**match**(*regexp*)
Like `String.prototype.match()`, returns the result of a regular expression match.

Arguments

- **regexp** (`RegExp()`) –

Returns **null|RegExpMatchArray** –

BiString.**matchAll**(*regexp*)
Like `String.prototype.matchAll()`, returns an iterator over all regular expression matches.

Arguments

- **regexp** (`RegExp()`) –

Returns **IterableIterator<RegExpMatchArray>** –

BiString.**normalize**(*form*)
Like `String.prototype.normalize()`, applies a Unicode normalization form.

Arguments

- **form** (`'NFC'|'NFD'|'NFKC'|'NFKD'()`) – The normalization form to apply, one of "NFC", "NFD", "NFKC", or "NFKD".

Returns **BiString** –

BiString.**padEnd**(*targetLength*, *padString=" "*)
Like `String.prototype.padEnd()`, pads a string at the end to a target length.

Arguments

- **targetLength** (`number()`) –
- **padString** (`string()`) –

Returns **BiString** –

BiString.**padStart**(*targetLength*, *padString=" "*)
Like `String.prototype.padStart()`, pads a string at the beginning to a target length.

Arguments

- **targetLength** (`number()`) –
- **padString** (`string()`) –

Returns **BiString** –

BiString.**replace**(*pattern*, *replacement*)
Like `String.prototype.replace()`, returns a new string with regex or fixed-string matches replaced.

Arguments

- **pattern** (`string|RegExp()`) –
- **replacement** (`string|Replacer()`) –

Returns **BiString** –

BiString.**search**(*regexp*)
Like `String.prototype.search()`, finds the position of the first match of a regular expression.

Arguments

- **regexp** (`RegExp()`) –

Returns **number** –

BiString.**searchBounds**(*regexp*)
Like *search()*, but returns both the start and end positions for convenience.

Arguments

- **regexp** (`RegExp()`) –

Returns **Bounds** –

BiString.**slice**(*start*, *end*)
Extract a slice of this BiString, with similar semantics to `String.prototype.slice()`.

Arguments

- **start** (`number()`) –
- **end** (`number()`) –

Returns **BiString** –

BiString.**split**(*separator*, *limit*)
Like `String.prototype.split()`, splits this string into chunks using a separator.

Arguments

- **separator** (`string|RegExp()`) –
- **limit** (`number()`) –

Returns **BiString[]** –

BiString.**startsWith**(*searchString*, *position*)
Like `String.prototype.startsWith()`, returns whether this string starts with the given prefix.

Arguments

- **searchString** (`string()`) –
- **position** (`number()`) –

Returns **boolean** –

BiString.**substring**(*start*, *end*)
Extract a substring of this BiString, with similar semantics to `String.prototype.substring()`.

Arguments

- **start** (`number()`) –
- **end** (`number()`) –

Returns **BiString** –

BiString.**toLowerCase**()
Like `String.prototype.toLowerCase()`, converts a string to lowercase.

Returns **BiString** –

BiString.**toUpperCase**()
Like `String.prototype.toUpperCase()`, converts a string to uppercase.

Returns **BiString** –

BiString.**trim**()
Like `String.prototype.trim()`, returns a new string with leading and trailing whitespace removed.

Returns **BiString** –

BiString.**trimEnd**()
Like `String.prototype.trim()`, returns a new string with trailing whitespace removed.

Returns **BiString** –

BiString.**trimStart**()
Like `String.prototype.trim()`, returns a new string with leading whitespace removed.

Returns **BiString** –

BiString.**from**(*str*)
Create a *BiString* from a string-like object.

Arguments

- **str** (`AnyString()`) – Either a *string* or a *BiString*.

Returns **BiString** – The input coerced to a *BiString*.

BiString.**infer**(*original*, *modified*)
Create a *BiString*, automatically inferring an alignment between the *original* and *modified* strings.

Arguments

- **original** (string()) – The original string.
- **modified** (string()) – The modified string.

Returns **BiString** –

4.2 BiStringBuilder

class BiStringBuilder(*original*)
Bidirectionally transformed string builder.

A *BistrBuilder* builds a transformed version of a source string iteratively. Each builder has an immutable original string, a current string, and the in-progress modified string, with alignments between each. For example:

```
original: |The| |quick,| |brown| || |jumps| |over| |the| |lazy| ||
          |   | |      | |     | |  \ \     \ \    \ \   \ \    \ \   \
current:  |The| |quick,| |brown| |fox| |jumps| |over| |the| |lazy| |dog|
          |   | |      / /     /
modified: |the| |quick| |brown| ...
```

The modified string is built in pieces by calling *replace()* to change *n* characters of the current string into new ones in the modified string. Convenience methods like *skip()*, *insert()*, and *discard()* are implemented on top of this basic primitive.

Construct a BiStringBuilder.

Arguments

- **original** (AnyString()) – Either an original string or a BiString to start from.

BiStringBuilder.**alignment**
type: Alignment

BiStringBuilder.**current**
type: string

BiStringBuilder.**isComplete**
type: boolean

BiStringBuilder.**modified**
type: string

BiStringBuilder.**original**
type: string

BiStringBuilder.**position**
type: number

BiStringBuilder.**remaining**
type: number

BiStringBuilder.**append**(*bs*)
Append a BiString. The original value of the BiString must match the current string being processed.

Arguments

- **bs** (*BiString()*) –

BiStringBuilder.**build**()
Build the *BiString()*.

Returns **BiString** –

BiStringBuilder.**discard**(*n*)
Discard a portion of the original string.

Arguments

- **n** (number()) –

BiStringBuilder.**discardMatch**(*pattern*)
Discard a substring that matches a regex.

Arguments

- **pattern** (RegExp()) – The pattern to match. Must have either the sticky flag, forcing it to match at the current position, or the global flag, finding the next match.

Returns **boolean** – Whether a match was found.

BiStringBuilder.**discardRest**()
Discard the rest of the original string.

BiStringBuilder.**insert**(*str*)
Insert a substring into the string.

Arguments

- **str** (string()) –

BiStringBuilder.**peek**(*n*)
Peek at the next few characters.

Arguments

- **n** (number()) – The number of characters to peek at.

Returns **string** –

BiStringBuilder.**replace**(*n*, *str*)
Replace the next *n* characters with a new string.

Arguments

- **n** (number()) –
- **str** (AnyString()) –

BiStringBuilder.**replaceAll**(*pattern*, *replacement*)
Replace all occurences of a regex, like String.prototype.replace().

Arguments

- **pattern** (RegExp()) – The pattern to match. The global flag (/g) must be set to get multiple matches.
- **replacement** (string|Replacer()) – The replacement string or function, as in String.prototype.replace().

BiStringBuilder.**replaceMatch**(*pattern*, *replacement*)
Replace a substring that matches a regex.

Arguments

- **pattern** (RegExp()) – The pattern to match. Must have either the sticky flag, forcing it to match at the current position, or the global flag, finding the next match.
- **replacement** (string|Replacer()) – The replacement string or function, as in String.prototype.replace().

Returns **boolean** – Whether a match was found.

BiStringBuilder.**rewind**()
Reset this builder to apply another transformation.

BiStringBuilder.**skip**(*n*)
Skip the next *n* characters, copying them unchanged.

Arguments

- **n** (number()) –

BiStringBuilder.**skipMatch**(*pattern*)
Skip a substring matching a regex, copying it unchanged.

Arguments

- **pattern** (RegExp()) – The pattern to match. Must have either the sticky flag, forcing it to match at the current position, or the global flag, finding the next match.

Returns **boolean** – Whether a match was found.

BiStringBuilder.**skipRest**()
Skip the rest of the string, copying it unchanged.

4.3 Alignment

class Alignment(*values*)
An alignment between two related sequences.

Consider this alignment between two strings:

```
|it's| |aligned!|
|    \ \        |
|it is| |aligned|
```

An alignment stores all the indices that are known to correspond between the original and modified sequences. For the above example, it would be

```
let a = new Alignment([
    [0, 0],
    [4, 5],
    [5, 6],
    [13, 13],
]);
```

Alignments can be used to answer questions like, "what's the smallest range of the original sequence that is guaranteed to contain this part of the modified sequence?" For example, the range *(0, 5)* ("it is") is known to match the range *(0, 4)* ("it's") of the original sequence.

```
console.log(a.original_bounds(0, 5));
// [0, 4]
```

Results may be imprecise if the alignment is too course to match the exact inputs:

```
console.log(a.original_bounds(0, 2));
// [0, 4]
```

A more granular alignment like this:

```
|i|t|'s| |a|l|i|g|n|e|d|!|
| | |  \ \ \ \ \ \ \ \ \ /
|i|t| is| |a|l|i|g|n|e|d|
```

```
a = new Alignment([
    [0, 0], [1, 1], [2, 2], [4, 5], [5, 6], [6, 7], [7, 8],
    [8, 9], [9, 10], [10, 11], [11, 12], [12, 13], [13, 13],
]);
```

Can be more precise:

```
console.log(a.original_bounds(0, 2));
// [0, 2]
```

Create a new Alignment.

Arguments

- **values** (Iterable) – The pairs of indices to align. Each element should be a pair destructurable as *[x, y]*, where *x* is the original sequence position and *y* is the modified sequence position.

Alignment.**length**
type: number

The number of entries in this alignment.

Alignment.**values**
type: readonly BiIndex[]

The pairs of aligned indices in this alignment.

Alignment.**compose**(*other*)

Arguments

- **other** (*Alignment()*) –

Returns **Alignment** – An alignment equivalent to applying *this* first, then *other*.

Alignment.**concat**(*...others*)
Concatenate this alignment together with one or more others.

Arguments

- **others** (Alignment[]()) –

Returns **Alignment** –

Alignment.**equals**(*other*)

Arguments

- **other** (*Alignment()*) –

Returns **boolean** – Whether this alignment is the same as *other*.

`Alignment.`**`inverse`**`()`

Returns **Alignment** – The inverse of this alignment, from the modified to the original sequence.

`Alignment.`**`modifiedBounds`**`()`

Returns **Bounds** – The bounds of the modified sequence.

`Alignment.`**`originalBounds`**`()`

Returns **Bounds** – The bounds of the original sequence.

`Alignment.`**`shift`**`(`*deltaO*, *deltaM*`)`
Shift this alignment.

Arguments

- **deltaO** (`number()`) – The distance to shift the original sequence.
- **deltaM** (`number()`) – The distance to shift the modified sequence.

Returns **Alignment** – An alignment with all the positions shifted by the given amounts.

`Alignment.`**`slice`**`(`*start*, *end*`)`
Extract a slice of this alignment.

Arguments

- **start** (`number()`) – The position to start from.
- **end** (`number()`) – The position to end at.

Returns **Alignment** – The requested slice as a new *Alignment*.

`Alignment.`**`sliceByModified`**`(`*start*, *end*`)`
Slice this alignment by a span of the modified sequence.

Arguments

- **start** (`number()`) – The start of the span in the modified sequence.
- **end** (`number()`) – The end of the span in the modified sequence.

Returns **Alignment** – The requested slice of this alignment.

`Alignment.`**`sliceByOriginal`**`(`*start*, *end*`)`
Slice this alignment by a span of the original sequence.

Arguments

- **start** (`number()`) – The start of the span in the original sequence.
- **end** (`number()`) – The end of the span in the original sequence.

Returns **Alignment** – The requested slice of this alignment.

`Alignment.`**`identity`**`(`*size*`)`
Create an identity alignment of the given size, which maps all intervals to themselves.

Arguments

- **size** (`number()`) – The size of the sequences to align.

Returns **Alignment** – The identity alignment for the bounds [0, *size*].

`Alignment.`**`infer`**`(`*original*, *modified*, *costFn*`)`
Infer the alignment between two sequences with the lowest edit distance.

Warning: this operation has time complexity `O(N*M)`, where *N* and *M* are the lengths of the original and modified sequences, and so should only be used for relatively short sequences.

Arguments

- **`original`** (`Iterable`) – The original sequence.
- **`modified`** (`Iterable`) – The modified sequence.
- **`costFn`** (`CostFn`) – A function returning the cost of performing an edit. *costFn(a, b)* returns the cost of replacing *a* with *b*. *costFn(a, undefined)* returns the cost of deleting *a*, and *costFn(undefined, b)* returns the cost of inserting *b*. By default, all operations have cost 1 except replacing identical elements, which has cost 0.

Returns **Alignment** – The inferred alignment.

4.4 Tokenization

`class Token``(`*text*, *start*, *end*`)`
A token extracted from a string.

Create a token.

Arguments

- **`text`** (`AnyString()`) – The text of this token.
- **`start`** (`number()`) – The start position of the token.
- **`end`** (`number()`) – The end position of the token.

`Token.`**`end`**
type: number

The end position of the token.

`Token.`**`modified`**
type: string

`Token.`**`original`**
type: string

`Token.`**`start`**
type: number

The start position of the token.

`Token.`**`text`**
type: BiString

The actual text of the token.

`Token.`**`slice`**`(`*text*, *start*, *end*`)`
Create a token from a slice of a string.

Arguments

- **`text`** (`AnyString()`) – The text to slice.

- **start** (number()) – The start index of the token.
- **end** (number()) – The end index of the token.

Returns **Token** –

class **Tokenization**(*text*, *tokens*)

A string and its tokenization.

Create a *Tokenization*.

Arguments

- **text** (AnyString()) – The text from which the tokens have been extracted.
- **tokens** (*Iterable*) – The tokens extracted from the text.

Tokenization.**alignment**

type: Alignment

The alignment between the text and the tokens.

Tokenization.**length**

type: number

The number of tokens.

Tokenization.**text**

type: BiString

The text that was tokenized.

Tokenization.**tokens**

type: readonly Token[]

The tokens extracted from the text.

Tokenization.**boundsForOriginal**(*start*, *end*)

Map a span of original text to the bounds of the corresponding span of tokens.

Arguments

- **start** (number()) –
- **end** (number()) –

Returns **Bounds** –

Tokenization.**boundsForText**(*start*, *end*)

Map a span of text to the bounds of the corresponding span of tokens.

Arguments

- **start** (number()) –
- **end** (number()) –

Returns **Bounds** –

Tokenization.**originalBounds**(*start*, *end*)

Map a span of tokens to the bounds of the corresponding original text.

Arguments

- **start** (number()) –
- **end** (number()) –

Returns **Bounds** –

Tokenization.**slice**(*start*, *end*)
Compute a slice of this tokenization.

Arguments

- **start** (number()) – The position to start from.
- **end** (number()) – The position to end at.

Returns **Tokenization** – The requested slice as a new *Tokenization*.

Tokenization.**sliceByOriginal**(*start*, *end*)
Map a span of original text to the corresponding span of tokens.

Arguments

- **start** (number()) –
- **end** (number()) –

Returns **Tokenization** –

Tokenization.**sliceByText**(*start*, *end*)
Map a span of text to the corresponding span of tokens.

Arguments

- **start** (number()) –
- **end** (number()) –

Returns **Tokenization** –

Tokenization.**snapOriginalBounds**(*start*, *end*)
Expand a span of original text to align it with token boundaries.

Arguments

- **start** (number()) –
- **end** (number()) –

Returns **Bounds** –

Tokenization.**snapTextBounds**(*start*, *end*)
Expand a span of text to align it with token boundaries.

Arguments

- **start** (number()) –
- **end** (number()) –

Returns **Bounds** –

Tokenization.**substring**(*start*, *end*)
Map a span of tokens to the corresponding substring.

Arguments

- **start** (number()) –
- **end** (number()) –

Returns **BiString** –

Tokenization.**textBounds**(*start*, *end*)
Map a span of tokens to the bounds of the corresponding text.

Arguments

- **start** (number()) –
- **end** (number()) –

Returns **Bounds** –

Tokenization.**infer**(*text*, *tokens*)

Infer a *Tokenization* from a sequence of tokens.

Due to the possibility of ambiguity, it is much better to use a *Tokenizer()* or some other method of producing *Token()*s with their positions explicitly set.

Arguments

- **text** (AnyString()) – The text that was tokenized.
- **tokens** (Iterable) – The extracted tokens.

Returns **Tokenization** – The inferred tokenization, with token positions found by simple forward search.

4.5 Tokenizer

class Tokenizer()

A tokenizer that produces *Tokenization()*s.

interface

Tokenizer.**tokenize**(*text*)

Tokenize a string.

Arguments

- **text** (AnyString()) – The text to tokenize, either a string or a *BiString()*.

Returns **Tokenization** – A *Tokenization()* holding the text and its tokens.

class RegExpTokenizer(*pattern*)

Breaks text into tokens based on a RegExp().

Implements:

- *Tokenizer()*

Create a *RegExpTokenizer*.

Arguments

- **pattern** (RegExp()) – The regex that will match tokens.

RegExpTokenizer.**tokenize**(*text*)

Tokenize a string.

Arguments

- **text** (AnyString()) –

Returns **Tokenization** – A *Tokenization()* holding the text and its tokens.

class SplittingTokenizer(*pattern*)

Splits text into tokens based on a RegExp().

Implements:

- *Tokenizer()*

Create a *SplittingTokenizer*.

Arguments

- **pattern** (RegExp()) – A regex that matches the regions between tokens.

SplittingTokenizer.**tokenize**(*text*)

Tokenize a string.

Arguments

- **text** (AnyString()) –

Returns **Tokenization** – A *Tokenization()* holding the text and its tokens.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

Symbols

A

B

C

D

E

F

I

J

L

M

N

O

P

R

S

T

U

W