

Navigation

- [index](#)
- [modules](#) |
- [next](#) |
- [previous](#) |
- [Python v2.6.4 documentation](#) »
- [The Python Standard Library](#) »
- [8. String Services](#) »

8.4. difflib — Helpers for computing deltas¶

New in version 2.1.

This module provides classes and functions for comparing sequences. It can be used for example, for comparing files, and can produce difference information in various formats, including HTML and context and unified diffs. For comparing directories and files, see also, the [filecmp](#) module.

`class difflib.SequenceMatcher¶`

This is a flexible class for comparing pairs of sequences of any type, so long as the sequence elements are [hashable](#). The basic algorithm predates, and is a little fancier than, an algorithm published in the late 1980's by Ratcliff and Obershelp under the hyperbolic name "gestalt pattern matching." The idea is to find the longest contiguous matching subsequence that contains no "junk" elements (the Ratcliff and Obershelp algorithm doesn't address junk). The same idea is then applied recursively to the pieces of the sequences to the left and to the right of the matching subsequence. This does not yield minimal edit sequences, but does tend to yield matches that "look right" to people.

Timing: The basic Ratcliff-Obershelp algorithm is cubic time in the worst case and quadratic time in the expected case. [SequenceMatcher](#) is quadratic time for the worst case and has expected-case behavior dependent in a complicated way on how many elements the sequences have in common; best case time is linear.

`class difflib.Differ¶`

This is a class for comparing sequences of lines of text, and producing human-readable differences or deltas. Differ uses [SequenceMatcher](#) both to compare sequences of lines, and to compare sequences of characters within similar (near-matching) lines.

Each line of a [Differ](#) delta begins with a two-letter code:

Code	Meaning
'-'	line unique to sequence 1
+'	line unique to sequence 2
' '	line common to both sequences
'?'	line not present in either input sequence

Lines beginning with '?' attempt to guide the eye to intraline differences, and were not present in either input sequence. These lines can be confusing if the sequences contain tab characters.

`class difflib.HtmlDiff¶`

This class can be used to create an HTML table (or a complete HTML file containing the table) showing a side by side, line by line comparison of text with inter-line and intra-line change highlights. The table can be generated in either full or contextual difference mode.

The constructor for this class is:

```
__init__([tabsize][, wrapcolumn][, linejunk][, charjunk])¶
```

Initializes instance of [HtmlDiff](#).

tabsize is an optional keyword argument to specify tab stop spacing and defaults to 8.

wrapcolumn is an optional keyword to specify column number where lines are broken and wrapped, defaults to `None` where lines are not wrapped.

linejunk and *charjunk* are optional keyword arguments passed into `ndiff()` (used by [HtmlDiff](#) to generate the side by side HTML differences). See `ndiff()` documentation for argument default values and descriptions.

The following methods are public:

```
make_file(fromlines, tolines[, fromdesc][, todesc][, context][, numlines])¶
```

Compares *fromlines* and *tolines* (lists of strings) and returns a string which is a complete HTML file containing a table showing line by line differences with inter-line and intra-line changes highlighted.

fromdesc and *todesc* are optional keyword arguments to specify from/to file column header strings (both default to an empty string).

context and *numlines* are both optional keyword arguments. Set *context* to `True` when contextual differences are to be shown, else the default is `False` to show the full files. *numlines* defaults to 5. When *context* is `True` *numlines* controls the number of context lines which surround the difference highlights. When *context* is `False` *numlines* controls the number of lines which are shown before a difference highlight when using the “next” hyperlinks (setting to zero would cause the “next” hyperlinks to place the next difference highlight at the top of the browser without any leading context).

```
make_table(fromlines, tolines[, fromdesc[, todesc[, context[, numlines]])
```

Compares *fromlines* and *tolines* (lists of strings) and returns a string which is a complete HTML table showing line by line differences with inter-line and intra-line changes highlighted.

The arguments for this method are the same as those for the [make_file\(\)](#) method.

`Tools/scripts/diff.py` is a command-line front-end to this class and contains a good example of its use.

New in version 2.4.

```
difflib.context_diff(a, b[, fromfile[, tofile[, fromfiledate[, tofiledate[, n[, lineterm]])
```

Compare *a* and *b* (lists of strings); return a delta (a [generator](#) generating the delta lines) in context diff format.

Context diffs are a compact way of showing just the lines that have changed plus a few lines of context. The changes are shown in a before/after style. The number of context lines is set by *n* which defaults to three.

By default, the diff control lines (those with `***` or `---`) are created with a trailing newline. This is helpful so that inputs created from [file.readlines\(\)](#) result in diffs that are suitable for use with [file.writelines\(\)](#) since both the inputs and outputs have trailing newlines.

For inputs that do not have trailing newlines, set the *lineterm* argument to `" "` so that the output will be uniformly newline free.

The context diff format normally has a header for filenames and modification times. Any or all of these may be specified using strings for *fromfile*, *tofile*, *fromfiledate*, and *tofiledate*. The modification times are normally expressed in the format returned by [time.ctime\(\)](#). If not specified, the strings default to blanks.

```
>>> s1 = ['bacon\n', 'eggs\n', 'ham\n', 'guido\n']
>>> s2 = ['python\n', 'eggy\n', 'hamster\n', 'guido\n']
>>> for line in context_diff(s1, s2, fromfile='before.py', tofile='after.py'):
...     sys.stdout.write(line) # doctest: +NORMALIZE_WHITESPACE
*** before.py
--- after.py
*****
*** 1,4 ****
! bacon
! eggs
! ham
  guido
--- 1,4 ----
! python
! eggy
! hamster
  guido
```

See [A command-line interface to difflib](#) for a more detailed example.

New in version 2.3.

```
difflib.get_close_matches(word, possibilities[, n[, cutoff])
```

Return a list of the best “good enough” matches. *word* is a sequence for which close matches are desired (typically a string), and *possibilities* is a list of sequences against which to match *word* (typically a list of strings).

Optional argument *n* (default 3) is the maximum number of close matches to return; *n* must be greater than 0.

Optional argument *cutoff* (default 0.6) is a float in the range [0, 1]. Possibilities that don't score at least that similar to *word* are ignored.

The best (no more than *n*) matches among the possibilities are returned in a list, sorted by similarity score, most similar first.

```
>>> get_close_matches('appel', ['ape', 'apple', 'peach', 'puppy'])
['apple', 'ape']
>>> import keyword
>>> get_close_matches('wheel', keyword.kwlist)
['while']
>>> get_close_matches('apple', keyword.kwlist)
[]
```

```
>>> get_close_matches('accept', keyword.kwlist)
['except']
```

```
difflib.ndiff(a, b[, linejunk[, charjunk]])
```

Compare *a* and *b* (lists of strings); return a [Differ](#)-style delta (a [generator](#) generating the delta lines).

Optional keyword parameters *linejunk* and *charjunk* are for filter functions (or `None`):

linejunk: A function that accepts a single string argument, and returns true if the string is junk, or false if not. The default is (`None`), starting with Python 2.3. Before then, the default was the module-level function [IS_LINE_JUNK\(\)](#), which filters out lines without visible characters, except for at most one pound character ('#'). As of Python 2.3, the underlying [SequenceMatcher](#) class does a dynamic analysis of which lines are so frequent as to constitute noise, and this usually works better than the pre-2.3 default.

charjunk: A function that accepts a character (a string of length 1), and returns if the character is junk, or false if not. The default is module-level function [IS_CHARACTER_JUNK\(\)](#), which filters out whitespace characters (a blank or tab; note: bad idea to include newline in this!).

`Tools/scripts/ndiff.py` is a command-line front-end to this function.

```
>>> diff = ndiff('one\ntwo\nthree\n'.splitlines(1),
...             'ore\ntree\nemu\n'.splitlines(1))
>>> print ''.join(diff),
- one
? ^
+ ore
? ^
- two
- three
? -
+ tree
+ emu
```

```
difflib.restore(sequence, which)
```

Return one of the two sequences that generated a delta.

Given a *sequence* produced by [Differ.compare\(\)](#) or [ndiff\(\)](#), extract lines originating from file 1 or 2 (parameter *which*), stripping off line prefixes.

Example:

```
>>> diff = ndiff('one\ntwo\nthree\n'.splitlines(1),
...             'ore\ntree\nemu\n'.splitlines(1))
>>> diff = list(diff) # materialize the generated delta into a list
>>> print ''.join	restore(diff, 1)),
one
two
three
>>> print ''.join	restore(diff, 2)),
ore
tree
emu
```

```
difflib.unified_diff(a, b[, fromfile[, tofile[, fromfiledate[, tofiledate[, n[, lineterm]])])
```

Compare *a* and *b* (lists of strings); return a delta (a [generator](#) generating the delta lines) in unified diff format.

Unified diffs are a compact way of showing just the lines that have changed plus a few lines of context. The changes are shown in a inline style (instead of separate before/after blocks). The number of context lines is set by *n* which defaults to three.

By default, the diff control lines (those with `---`, `+++`, or `@@`) are created with a trailing newline. This is helpful so that inputs created from [file.readlines\(\)](#) result in diffs that are suitable for use with [file.writelines\(\)](#) since both the inputs and outputs have trailing newlines.

For inputs that do not have trailing newlines, set the *lineterm* argument to `" "` so that the output will be uniformly newline free.

The context diff format normally has a header for filenames and modification times. Any or all of these may be specified using strings for *fromfile*, *tofile*, *fromfiledate*, and *tofiledate*. The modification times are normally expressed in the format returned by [time.ctime\(\)](#). If not specified, the strings default to blanks.

```
>>> s1 = ['bacon\n', 'eggs\n', 'ham\n', 'guido\n']
>>> s2 = ['python\n', 'eggy\n', 'hamster\n', 'guido\n']
>>> for line in unified_diff(s1, s2, fromfile='before.py', tofile='after.py'):
...     sys.stdout.write(line) # doctest: +NORMALIZE_WHITESPACE
```

```

--- before.py
+++ after.py
@@ -1,4 +1,4 @@
-bacon
-eggs
-ham
+python
+eggy
+hamster
guido

```

See [A command-line interface to difflib](#) for a more detailed example.

New in version 2.3.

`difflib.IS_LINE_JUNK(line)`

Return true for ignorable lines. The line *line* is ignorable if *line* is blank or contains a single '#', otherwise it is not ignorable. Used as a default for parameter *linejunk* in `ndiff()` before Python 2.3.

`difflib.IS_CHARACTER_JUNK(ch)`

Return true for ignorable characters. The character *ch* is ignorable if *ch* is a space or tab, otherwise it is not ignorable. Used as a default for parameter *charjunk* in `ndiff()`.

See also

[Pattern Matching: The Gestalt Approach](#)

Discussion of a similar algorithm by John W. Ratcliff and D. E. Metzener. This was published in [Dr. Dobbs's Journal](#) in July, 1988.

8.4.1. SequenceMatcher Objects

The [SequenceMatcher](#) class has this constructor:

```
class difflib.SequenceMatcher([isjunk[, a[, b]])
```

Optional argument *isjunk* must be `None` (the default) or a one-argument function that takes a sequence element and returns true if and only if the element is "junk" and should be ignored. Passing `None` for *isjunk* is equivalent to passing `lambda x: 0`; in other words, no elements are ignored. For example, pass:

```
lambda x: x in " \t"
```

if you're comparing lines as sequences of characters, and don't want to synch up on blanks or hard tabs.

The optional arguments *a* and *b* are sequences to be compared; both default to empty strings. The elements of both sequences must be [hashable](#).

[SequenceMatcher](#) objects have the following methods:

`set_seqs(a, b)`

Set the two sequences to be compared.

[SequenceMatcher](#) computes and caches detailed information about the second sequence, so if you want to compare one sequence against many sequences, use `set_seq2()` to set the commonly used sequence once and call `set_seq1()` repeatedly, once for each of the other sequences.

`set_seq1(a)`

Set the first sequence to be compared. The second sequence to be compared is not changed.

`set_seq2(b)`

Set the second sequence to be compared. The first sequence to be compared is not changed.

`find_longest_match(alo, ahi, blo, bhi)`

Find longest matching block in `a[alo:ahi]` and `b[blo:bhi]`.

If *isjunk* was omitted or `None`, `find_longest_match()` returns (*i*, *j*, *k*) such that `a[i:i+k]` is equal to `b[j:j+k]`, where `alo <= i <= i+k <= ahi` and `blo <= j <= j+k <= bhi`. For all (*i*', *j*', *k*') meeting those conditions, the additional conditions `k >= k'`, `i <= i'`, and if `i == i'`, `j <= j'` are also met. In other words, of all maximal matching blocks, return one that starts earliest in *a*, and of all those maximal matching blocks that start earliest in *a*, return the one that starts earliest in *b*.

```
>>> s = SequenceMatcher(None, "abcd", "abcdabcd")
```

```
>>> s.find_longest_match(0, 5, 0, 9)
```

```
Match(a=0, b=4, size=5)
```

If *isjunk* was provided, first the longest matching block is determined as above, but with the additional restriction that no junk element appears in the block. Then that block is extended as far as possible by matching (only) junk elements on both sides. So the resulting block never matches on junk except as identical junk happens to be adjacent to an interesting match.

Here's the same example as before, but considering blanks to be junk. That prevents 'abcd' from matching the 'abcd' at the tail end of the second sequence directly. Instead only the 'abcd' can match, and matches the leftmost 'abcd' in the second sequence:

```
>>> s = SequenceMatcher(lambda x: x==" ", "abcd", "abcd abcd")
>>> s.find_longest_match(0, 5, 0, 9)
Match(a=1, b=0, size=4)
```

If no blocks match, this returns (a0, b0, 0).

Changed in version 2.6: This method returns a [named tuple](#) Match(a, b, size).

`get_matching_blocks()`

Return list of triples describing matching subsequences. Each triple is of the form (i, j, n), and means that a[i:i+n] == b[j:j+n]. The triples are monotonically increasing in i and j.

The last triple is a dummy, and has the value (len(a), len(b), 0). It is the only triple with n == 0. If (i, j, n) and (i', j', n') are adjacent triples in the list, and the second is not the last triple in the list, then i+n != i' or j+n != j'; in other words, adjacent triples always describe non-adjacent equal blocks.

Changed in version 2.5: The guarantee that adjacent triples always describe non-adjacent blocks was implemented.

```
>>> s = SequenceMatcher(None, "abxcd", "abcd")
>>> s.get_matching_blocks()
[Match(a=0, b=0, size=2), Match(a=3, b=2, size=2), Match(a=5, b=4, size=0)]
```

`get_opcodes()`

Return list of 5-tuples describing how to turn a into b. Each tuple is of the form (tag, i1, i2, j1, j2). The first tuple has i1 == j1 == 0, and remaining tuples have i1 equal to the i2 from the preceding tuple, and, likewise, j1 equal to the previous j2.

The tag values are strings, with these meanings:

Value	Meaning
'replace'	a[i1:i2] should be replaced by b[j1:j2].
'delete'	a[i1:i2] should be deleted. Note that j1 == j2 in this case.
'insert'	b[j1:j2] should be inserted at a[i1:i1]. Note that i1 == i2 in this case.
'equal'	a[i1:i2] == b[j1:j2] (the sub-sequences are equal).

For example:

```
>>> a = "qabxcd"
>>> b = "abycdf"
>>> s = SequenceMatcher(None, a, b)
>>> for tag, i1, i2, j1, j2 in s.get_opcodes():
...     print ("%7s a[%d:%d] (%s) b[%d:%d] (%s)" %
...           (tag, i1, i2, a[i1:i2], j1, j2, b[j1:j2]))
delete a[0:1] (q) b[0:0] ()
equal a[1:3] (ab) b[0:2] (ab)
replace a[3:4] (x) b[2:3] (y)
equal a[4:6] (cd) b[3:5] (cd)
insert a[6:6] () b[5:6] (f)
```

`get_grouped_opcodes([n])`

Return a [generator](#) of groups with up to n lines of context.

Starting with the groups returned by `get_opcodes()`, this method splits out smaller change clusters and eliminates intervening ranges which have no changes.

The groups are returned in the same format as `get_opcodes()`.

New in version 2.3.

`ratio()`

Return a measure of the sequences' similarity as a float in the range [0, 1].

Where T is the total number of elements in both sequences, and M is the number of matches, this is 2.0*M / T. Note that this is 1.0 if the sequences are identical, and 0.0 if they have nothing in common.

This is expensive to compute if `get_matching_blocks()` or `get_opcodes()` hasn't already been called, in which case you may want to try `quick_ratio()` or `real_quick_ratio()` first to get an upper bound.

```
quick_ratio()
```

Return an upper bound on [ratio\(\)](#) relatively quickly.

This isn't defined beyond that it is an upper bound on [ratio\(\)](#), and is faster to compute.

```
real_quick_ratio()
```

Return an upper bound on [ratio\(\)](#) very quickly.

This isn't defined beyond that it is an upper bound on [ratio\(\)](#), and is faster to compute than either [ratio\(\)](#) or [quick_ratio\(\)](#).

The three methods that return the ratio of matching to total characters can give different results due to differing levels of approximation, although [quick_ratio\(\)](#) and [real_quick_ratio\(\)](#) are always at least as large as [ratio\(\)](#):

```
>>> s = SequenceMatcher(None, "abcd", "bcde")
>>> s.ratio()
0.75
>>> s.quick_ratio()
0.75
>>> s.real_quick_ratio()
1.0
```

8.4.2. SequenceMatcher Examples

This example compares two strings, considering blanks to be "junk:"

```
>>> s = SequenceMatcher(lambda x: x == " ",
...                     "private Thread currentThread;",
...                     "private volatile Thread currentThread;")
```

[ratio\(\)](#) returns a float in [0, 1], measuring the similarity of the sequences. As a rule of thumb, a [ratio\(\)](#) value over 0.6 means the sequences are close matches:

```
>>> print round(s.ratio(), 3)
0.866
```

If you're only interested in where the sequences match, [get_matching_blocks\(\)](#) is handy:

```
>>> for block in s.get_matching_blocks():
...     print "a[%d] and b[%d] match for %d elements" % block
a[0] and b[0] match for 8 elements
a[8] and b[17] match for 21 elements
a[29] and b[38] match for 0 elements
```

Note that the last tuple returned by [get_matching_blocks\(\)](#) is always a dummy, `(len(a), len(b), 0)`, and this is the only case in which the last tuple element (number of elements matched) is 0.

If you want to know how to change the first sequence into the second, use [get_opcodes\(\)](#):

```
>>> for opcode in s.get_opcodes():
...     print "%6s a[%d:%d] b[%d:%d]" % opcode
equal a[0:8] b[0:8]
insert a[8:8] b[8:17]
equal a[8:29] b[17:38]
```

See also

- The [get_close_matches\(\)](#) function in this module which shows how simple code building on [SequenceMatcher](#) can be used to do useful work.
- [Simple version control recipe](#) for a small application built with [SequenceMatcher](#).

8.4.3. Differ Objects

Note that [Differ](#)-generated deltas make no claim to be **minimal** diffs. To the contrary, minimal diffs are often counter-intuitive, because they synch up anywhere possible, sometimes accidental matches 100 pages apart. Restricting synch points to contiguous matches preserves some notion of locality, at the occasional cost of producing a longer diff.

The [Differ](#) class has this constructor:

```
class difflib.Differ([linejunk[, charjunk]])
```

Optional keyword parameters *linejunk* and *charjunk* are for filter functions (or `None`):

linejunk: A function that accepts a single string argument, and returns true if the string is junk. The default is `None`, meaning that no line is considered junk.

charjunk: A function that accepts a single character argument (a string of length 1), and returns true if the character is junk. The default is `None`, meaning that no character is considered junk.

[Differ](#) objects are used (deltas generated) via a single method:

```
compare(a, b)
```

Compare two sequences of lines, and generate the delta (a sequence of lines).

Each sequence must contain individual single-line strings ending with newlines. Such sequences can be obtained from the `readlines()` method of file-like objects. The delta generated also consists of newline-terminated strings, ready to be printed as-is via the `writelines()` method of a file-like object.

8.4.4. Differ Example

This example compares two texts. First we set up the texts, sequences of individual single-line strings ending with newlines (such sequences can also be obtained from the `readlines()` method of file-like objects):

```
>>> text1 = ''' 1. Beautiful is better than ugly.
... 2. Explicit is better than implicit.
... 3. Simple is better than complex.
... 4. Complex is better than complicated.
... '''.splitlines(1)
>>> len(text1)
4
>>> text1[0][-1]
'\n'
>>> text2 = ''' 1. Beautiful is better than ugly.
... 3. Simple is better than complex.
... 4. Complicated is better than complex.
... 5. Flat is better than nested.
... '''.splitlines(1)
```

Next we instantiate a `Differ` object:

```
>>> d = Differ()
```

Note that when instantiating a [Differ](#) object we may pass functions to filter out line and character “junk.” See the [Differ\(\)](#) constructor for details.

Finally, we compare the two:

```
>>> result = list(d.compare(text1, text2))
```

`result` is a list of strings, so let's pretty-print it:

```
>>> from pprint import pprint
>>> pprint(result)
[' 1. Beautiful is better than ugly.\n',
'- 2. Explicit is better than implicit.\n',
'- 3. Simple is better than complex.\n',
'+ 3. Simple is better than complex.\n',
'? ++\n',
'- 4. Complex is better than complicated.\n',
'? ^ ---- ^\n',
'+ 4. Complicated is better than complex.\n',
'? ++++ ^ ^\n',
'+ 5. Flat is better than nested.\n']
```

As a single multi-line string it looks like this:

```
>>> import sys
>>> sys.stdout.writelines(result)
 1. Beautiful is better than ugly.
- 2. Explicit is better than implicit.
- 3. Simple is better than complex.
+ 3. Simple is better than complex.
? ++
- 4. Complex is better than complicated.
? ^ ---- ^
+ 4. Complicated is better than complex.
```

```
?      +++++ ^
+ 5. Flat is better than nested.
```

8.4.5. A command-line interface to `difflib`

This example shows how to use `difflib` to create a `diff`-like utility. It is also contained in the Python source distribution, as `Tools/scripts/diff.py`.

```
""" Command line interface to difflib.py providing diffs in four formats:

* ndiff:    lists every line and highlights interline changes.
* context:  highlights clusters of changes in a before/after format.
* unified:  highlights clusters of changes in an inline format.
* html:     generates side by side comparison with change highlights.

"""

import sys, os, time, difflib, optparse

def main():
    # Configure the option parser
    usage = "usage: %prog [options] fromfile tofile"
    parser = optparse.OptionParser(usage)
    parser.add_option("-c", action="store_true", default=False,
                    help='Produce a context format diff (default)')
    parser.add_option("-u", action="store_true", default=False,
                    help='Produce a unified format diff')
    hlp = 'Produce HTML side by side diff (can use -c and -l in conjunction)'
    parser.add_option("-m", action="store_true", default=False, help=hlp)
    parser.add_option("-n", action="store_true", default=False,
                    help='Produce a ndiff format diff')
    parser.add_option("-l", "--lines", type="int", default=3,
                    help='Set number of context lines (default 3)')
    (options, args) = parser.parse_args()

    if len(args) == 0:
        parser.print_help()
        sys.exit(1)
    if len(args) != 2:
        parser.error("need to specify both a fromfile and tofile")

    n = options.lines
    fromfile, tofile = args # as specified in the usage string

    # we're passing these as arguments to the diff function
    fromdate = time.ctime(os.stat(fromfile).st_mtime)
    todate = time.ctime(os.stat(tofile).st_mtime)
    fromlines = open(fromfile, 'U').readlines()
    tolines = open(tofile, 'U').readlines()

    if options.u:
        diff = difflib.unified_diff(fromlines, tolines, fromfile, tofile,
                                   fromdate, todate, n=n)
    elif options.n:
        diff = difflib.ndiff(fromlines, tolines)
    elif options.m:
        diff = difflib.HtmlDiff().make_file(fromlines, tolines, fromfile,
                                             tofile, context=options.c,
                                             numlines=n)
    else:
        diff = difflib.context_diff(fromlines, tolines, fromfile, tofile,
                                    fromdate, todate, n=n)

    # we're using writelines because diff is a generator
    sys.stdout.writelines(diff)

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

[8.4. difflib — Helpers for computing deltas](#)

- [8.4.1. SequenceMatcher Objects](#)
- [8.4.2. SequenceMatcher Examples](#)
- [8.4.3. Differ Objects](#)
- [8.4.4. Differ Example](#)
- [8.4.5. A command-line interface to difflib](#)

Previous topic

[8.3. struct — Interpret strings as packed binary data](#)

Next topic

[8.5. StringIO — Read and write strings as files](#)

This Page

- [Show Source](#)

Navigation

- [index](#)
- [modules](#) |
- [next](#) |
- [previous](#) |
- [Python v2.6.4 documentation](#) »
- [The Python Standard Library](#) »
- [8. String Services](#) »

© [Copyright](#) 1990-2010, Python Software Foundation.

The Python Software Foundation is a non-profit corporation. [Please donate.](#)

Last updated on Feb 26, 2010. Created using [Sphinx](#) 0.6.3.