

Importing data with `genfromtxt`

NumPy provides several functions to create arrays from tabular data. We focus here on the `genfromtxt` function.

In a nutshell, `genfromtxt` runs two main loops. The first loop converts each line of the file in a sequence of strings. The second loop converts each string to the appropriate data type. This mechanism is slower than a single loop, but gives more flexibility. In particular, `genfromtxt` is able to take missing data into account, when other faster and simpler functions like `loadtxt` cannot.

Note

When giving examples, we will use the following conventions:

```
>>> import numpy as np
>>> from io import StringIO
```

Defining the input

The only mandatory argument of `genfromtxt` is the source of the data. It can be a string, a list of strings, a generator or an open file-like object with a `read` method, for example, a file or `io.StringIO` object. If a single string is provided, it is assumed to be the name of a local or remote file. If a list of strings or a generator returning strings is provided, each string is treated as one line in a file. When the URL of a remote file is passed, the file is automatically downloaded to the current directory and opened.

Recognized file types are text files and archives. Currently, the function recognizes `gzip` and `bz2` (`bzip2`) archives. The type of the archive is determined from the extension of the file: if the filename ends with `'.gz'`, a `gzip` archive is expected; if it ends with `'bz2'`, a `bzip2` archive is assumed.

Splitting the lines into columns





The `delimiter` argument

Once the file is defined and open for reading, `genfromtxt` splits each non-empty line into a sequence of strings. Empty or commented lines are just skipped. The `delimiter` keyword is used to define how the splitting should take place.

Quite often, a single character marks the separation between columns. For example, comma-separated files (CSV) use a comma (,) or a semicolon (;) as delimiter:

```
>>> data = u"1, 2, 3\n4, 5, 6"
>>> np.genfromtxt(StringIO(data), delimiter=",")
array([[1.,  2.,  3.],
       [4.,  5.,  6.]])
```

Another common separator is `"\t"`, the tabulation character. However, we are not limited to a single character, any string will do. By default, `genfromtxt` assumes `delimiter=None`, meaning that the line is split along white spaces (including tabs) and that consecutive white spaces are considered as a single white space.

- [What is NumPy?](#)
- [Installation](#) 
- [NumPy quickstart](#)
- [NumPy: the absolute basics for beginners](#)
- [NumPy fundamentals](#) 
 - [Array creation](#)
 - [Indexing on ndarrays](#)
 - [I/O with NumPy](#) 
 - [Importing data with genfromtxt](#)
 - [Data types](#)
 - [Broadcasting](#)
 - [Byte-swapping](#)
 - [Structured arrays](#)
 - [Writing custom array containers](#)
 - [Subclassing ndarray](#)
 - [Universal functions \(ufunc\)](#)
 - [basics](#)
 - [Copies and views](#)
 - [Interoperability with NumPy](#)
- [Miscellaneous](#)
- [NumPy for MATLAB users](#)
- [Building from source](#)
- [Using NumPy C-API](#)
- [NumPy Tutorials](#) 
- [NumPy How Tos](#)
- [For downstream package authors](#)
- [F2PY user guide and reference manual](#)
- [Glossary](#)
- [Under-the-hood Documentation for developers](#)
- [Reporting bugs](#)
- [Release notes](#)
- [NumPy license](#)

Alternatively, we may be dealing with a fixed-width file, where columns are defined as a given number of characters. In that case, we need to set `delimiter` to a single integer (if all the columns have the same size) or to a sequence of integers (if columns can have different sizes):

```
>>> data = u" 1 2 3\n 4 5 67\n890123 4"
>>> np.genfromtxt(StringIO(data), delimiter=3)
array([[ 1.,  2.,  3.],
       [ 4.,  5., 67.],
       [890., 123.,  4.]])
>>> data = u"123456789\n 4 7 9\n 4567 9"
>>> np.genfromtxt(StringIO(data), delimiter=(4, 3, 2))
array([[1234., 567., 89.],
       [ 4.,  7.,  9.],
       [ 4., 567.,  9.]])
```

The `autostrip` argument


By default, when a line is decomposed into a series of strings, the individual entries are not stripped of leading nor trailing white spaces. This behavior can be overwritten by setting the optional argument `autostrip` to a value of `True`:

```
>>> data = u"1, abc , 2\n 3, xxx, 4"
>>> # Without autostrip
>>> np.genfromtxt(StringIO(data), delimiter=",", dtype="<U5")
array([[ '1', ' abc ', ' 2'],
       [ '3', ' xxx', ' 4']], dtype='<U5')
>>> # With autostrip
>>> np.genfromtxt(StringIO(data), delimiter=",", dtype="<U5", autostrip=True)
array([[ '1', 'abc', '2'],
       [ '3', 'xxx', '4']], dtype='<U5')
```

The `comments` argument

The optional argument `comments` is used to define a character string that marks the beginning of a comment. By default, `genfromtxt` assumes `comments='#'`. The comment marker may occur anywhere on the line. Any character present after the comment marker(s) is simply ignored:

```
>>> data = u"""#
... # Skip me !
... # Skip me too !
... 1, 2
... 3, 4
... 5, 6 #This is the third line of the data
... 7, 8
... # And here comes the last line
... 9, 0
... """
>>> np.genfromtxt(StringIO(data), comments="#", delimiter=",")
array([[1., 2.],
       [3., 4.],
       [5., 6.],
       [7., 8.],
       [9., 0.]])
```

 **New in version 1.7.0:** When `comments` is set to `None`, no lines are treated as comments.

Note

There is one notable exception to this behavior: if the optional argument `names=True`, the first commented line will be examined for names.

Skipping lines and choosing columns

The `skip_header` and `skip_footer` arguments

The presence of a header in the file can hinder data processing. In that case, we need to use the `skip_header` optional argument. The values of this argument must be an integer which corresponds to the number of lines to skip at the beginning of the file, before any other action is performed. Similarly, we can skip the last `n` lines of the file by using the `skip_footer` attribute and giving it a value of `n`:

```
>>> data = u"\n".join(str(i) for i in range(10))
>>> np.genfromtxt(StringIO(data),)
array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
>>> np.genfromtxt(StringIO(data),
...               skip_header=3, skip_footer=5)
array([3., 4.])
```

By default, `skip_header=0` and `skip_footer=0`, meaning that no lines are skipped.

The `usecols` argument

In some cases, we are not interested in all the columns of the data but only a few of them. We can select which columns to import with the `usecols` argument. This argument accepts a single integer or a sequence of integers corresponding to the indices of the columns to import. Remember that by convention, the first column has an index of 0. Negative integers behave the same as regular Python negative indexes.

For example, if we want to import only the first and the last columns, we can use `usecols=(0, -1)`:

```
>>> data = u"1 2 3\n4 5 6"
>>> np.genfromtxt(StringIO(data), usecols=(0, -1))
array([[1., 3.],
       [4., 6.]])
```

If the columns have names, we can also select which columns to import by giving their name to the `usecols` argument, either as a sequence of strings or a comma-separated string:

```
>>> data = u"1 2 3\n4 5 6"
>>> np.genfromtxt(StringIO(data),
...               names="a, b, c", usecols=("a", "c"))
array([(1., 3.), (4., 6.)], dtype=[('a', '<f8'), ('c', '<f8')])
>>> np.genfromtxt(StringIO(data),
...               names="a, b, c", usecols="a, c")
array([(1., 3.), (4., 6.)], dtype=[('a', '<f8'), ('c', '<f8')])
```

Choosing the data type

The main way to control how the sequences of strings we have read from the file are converted to other types is to set the `dtype` argument. Acceptable values for this argument are:

- a single type, such as `dtype=float`. The output will be 2D with the given dtype, unless a name has been associated with each column with the use of the `names` argument (see below). Note that `dtype=float` is the default for `genfromtxt`.
- a sequence of types, such as `dtype=(int, float, float)`.
- a comma-separated string, such as `dtype="i4,f8,U3"`.
- a dictionary with two keys `'names'` and `'formats'`.
- a sequence of tuples `(name, type)`, such as `dtype=[('A', int), ('B', float)]`.
- an existing `numpy.dtype` object.
- the special value `None`. In that case, the type of the columns will be determined from the data itself (see below).

In all the cases but the first one, the output will be a 1D array with a structured dtype. This dtype has as many fields as items in the sequence. The field names are defined with the `names` keyword.

When `dtype=None`, the type of each column is determined iteratively from its data. We start by checking whether a string can be converted to a boolean (that is, if the string matches `true` or `false` in lower cases); then whether it can be converted to an integer, then to a float, then to a complex and eventually to a string.

The option `dtype=None` is provided for convenience. However, it is significantly slower than setting the dtype explicitly.

Setting the names

The `names` argument

A natural approach when dealing with tabular data is to allocate a name to each column. A first possibility is to use an explicit structured dtype, as mentioned previously:

```
>>> data = StringIO("1 2 3\n 4 5 6")
>>> np.genfromtxt(data, dtype=[_, int] for _ in "abc"])
array([(1, 2, 3), (4, 5, 6)],
      dtype=[('a', '<i8'), ('b', '<i8'), ('c', '<i8')])
```

Another simpler possibility is to use the `names` keyword with a sequence of strings or a comma-separated string:

```
>>> data = StringIO("1 2 3\n 4 5 6")
>>> np.genfromtxt(data, names="A, B, C")
array([(1., 2., 3.), (4., 5., 6.)],
      dtype=[('A', '<f8'), ('B', '<f8'), ('C', '<f8')])
```

In the example above, we used the fact that by default, `dtype=float`. By giving a sequence of names, we are forcing the output to a structured dtype.

We may sometimes need to define the column names from the data itself. In that case, we must use the `names` keyword with a value of `True`. The names will then be read from the first line (after the `skip_header` ones), even if the line is commented out:

```
>>> data = StringIO("So it goes\n# a b c\n1 2 3\n 4 5 6")
>>> np.genfromtxt(data, skip_header=1, names=True)
array([(1., 2., 3.), (4., 5., 6.)],
      dtype=[('a', '<f8'), ('b', '<f8'), ('c', '<f8')])
```

The default value of `names` is `None`. If we give any other value to the keyword, the new names will overwrite the field names we may have defined with the dtype:

```
>>> data = StringIO("1 2 3\n 4 5 6")
>>> ndtype=[('a',int), ('b', float), ('c', int)]
>>> names = ["A", "B", "C"]
>>> np.genfromtxt(data, names=names, dtype=ndtype)
array([(1, 2., 3), (4, 5., 6)],
      dtype=[('A', '<i8'), ('B', '<f8'), ('C', '<i8')])
```

The defaultfmt argument

If `names=None` but a structured dtype is expected, names are defined with the standard NumPy default of `"%fi"`, yielding names like `f0`, `f1` and so forth:


```
>>> data = StringIO("1 2 3\n 4 5 6")
>>> np.genfromtxt(data, dtype=(int, float, int))
array([(1, 2., 3), (4, 5., 6)],
      dtype=[('f0', '<i8'), ('f1', '<f8'), ('f2', '<i8')])
```

In the same way, if we don't give enough names to match the length of the dtype, the missing names will be defined with this default template:

```
>>> data = StringIO("1 2 3\n 4 5 6")
>>> np.genfromtxt(data, dtype=(int, float, int), names="a")
array([(1, 2., 3), (4, 5., 6)],
      dtype=[('a', '<i8'), ('f0', '<f8'), ('f1', '<i8')])
```

We can overwrite this default with the `defaultfmt` argument, that takes any format string:

```
>>> data = StringIO("1 2 3\n 4 5 6")
>>> np.genfromtxt(data, dtype=(int, float, int), defaultfmt="var_%02i")
array([(1, 2., 3), (4, 5., 6)],
      dtype=[('var_00', '<i8'), ('var_01', '<f8'), ('var_02', '<i8')])
```

 **Note**

We need to keep in mind that `defaultfmt` is used only if some names are expected but not defined.

Validating names

NumPy arrays with a structured dtype can also be viewed as [recarray](#), where a field can be accessed as if it were an attribute. For that reason, we may need to make sure that the field name doesn't contain any space or invalid character, or that it does not correspond to the name of a standard attribute (like `size` or `shape`), which would confuse the interpreter. [genfromtxt](#) accepts three optional arguments that provide a finer control on the names:

deletechars

Gives a string combining all the characters that must be deleted from the name. By default, invalid characters are `~!@#$$%^&*()-+=~\|]}[{';: /?.>,<.`

excludelist

Gives a list of the names to exclude, such as `return`, `file`, `print`... If one of the input name is part of this list, an underscore character (`'_'`) will be appended to it.

case_sensitive

Whether the names should be case-sensitive (`case_sensitive=True`), converted to upper case (`case_sensitive=False` or `case_sensitive='upper'`) or to lower case (`case_sensitive='lower'`).

Tweaking the conversion

The `converters` argument

Usually, defining a dtype is sufficient to define how the sequence of strings must be converted. However, some additional control may sometimes be required. For example, we may want to make sure that a date in a format `YYYY/MM/DD` is converted to a `datetime` object, or that a string like `xx%` is properly converted to a float between 0 and 1. In such cases, we should define conversion functions with the `converters` arguments.

The value of this argument is typically a dictionary with column indices or column names as keys and a conversion functions as values. These conversion functions can either be actual functions or lambda functions. In any case, they should accept only a string as input and output only a single element of the wanted type.

In the following example, the second column is converted from a string representing a percentage to a float between 0 and 1:

```
>>> convertfunc = lambda x: float(x.strip(b"%"))/100.
>>> data = u"1, 2.3%, 45.\n6, 78.9%, 0"
>>> names = ("i", "p", "n")
>>> # General case .....
>>> np.genfromtxt(StringIO(data), delimiter=",", names=names)
array([(1., nan, 45.), (6., nan, 0.)],
      dtype=[('i', '<f8'), ('p', '<f8'), ('n', '<f8')])
```

We need to keep in mind that by default, `dtype=float`. A float is therefore expected for the second column. However, the strings `' 2.3%'` and `' 78.9%'` cannot be converted to float and we end up having `np.nan` instead. Let's now use a converter:

```
>>> # Converted case ...
>>> np.genfromtxt(StringIO(data), delimiter=",", names=names,
...               converters={1: convertfunc})
array([(1., 0.023, 45.), (6., 0.789, 0.)],
      dtype=[('i', '<f8'), ('p', '<f8'), ('n', '<f8')])
```

The same results can be obtained by using the name of the second column (`"p"`) as key instead of its index (1):

```
>>> # Using a name for the converter ...
>>> np.genfromtxt(StringIO(data), delimiter=",", names=names,
...               converters={"p": convertfunc})
array([(1., 0.023, 45.), (6., 0.789, 0.)],
      dtype=[('i', '<f8'), ('p', '<f8'), ('n', '<f8')])
```

Converters can also be used to provide a default for missing entries. In the following example, the converter `convert` transforms a stripped string into the corresponding float or into -999 if the string is empty. We need to explicitly strip the string from white spaces as it is not done by default:

```
>>> data = u"1, , 3\n 4, 5, 6"
>>> convert = lambda x: float(x.strip() or -999)
>>> np.genfromtxt(StringIO(data), delimiter=",",
...               converters={1: convert})
array([[ 1., -999.,  3.],
       [ 4.,   5.,  6.]])
```

Using missing and filling values

Some entries may be missing in the dataset we are trying to import. In a previous example, we used a converter to transform an empty string into a float. However, user-defined converters may rapidly become cumbersome to manage.

The `genfromtxt` function provides two other complementary mechanisms: the `missing_values` argument is used to recognize missing data and a second argument, `filling_values`, is used to process these missing data.

missing_values

By default, any empty string is marked as missing. We can also consider more complex strings, such as `"N/A"` or `"???"` to represent missing or invalid data. The `missing_values` argument accepts three kinds of values:

a string or a comma-separated string

This string will be used as the marker for missing data for all the columns

a sequence of strings

In that case, each item is associated to a column, in order.

a dictionary

Values of the dictionary are strings or sequence of strings. The corresponding keys can be column indices (integers) or column names (strings). In addition, the special key `None` can be used to define a default applicable to all columns.

filling_values

We know how to recognize missing data, but we still need to provide a value for these missing entries. By default, this value is determined from the expected dtype according to this table:

Expected type	Default
<code>bool</code>	<code>False</code>
<code>int</code>	<code>-1</code>
<code>float</code>	<code>np.nan</code>
<code>complex</code>	<code>np.nan+0j</code>
<code>string</code>	<code>'???'</code>

We can get a finer control on the conversion of missing values with the `filling_values` optional argument. Like `missing_values`, this argument accepts different kind of values:

a single value

This will be the default for all columns

a sequence of values

Each entry will be the default for the corresponding column

a dictionary

Each key can be a column index or a column name, and the corresponding value should be a single object. We can use the special key `None` to define a default for all columns.

In the following example, we suppose that the missing values are flagged with `"N/A"` in the first column and by `"???"` in the third column. We wish to transform these missing values to 0 if they occur in the first and second column, and to -999 if they occur in the last column:

```
>>> data = u"N/A, 2, 3\n4, ,???"
>>> kwargs = dict(delimiter=",",
...               dtype=int,
...               names="a,b,c",
...               missing_values={0:"N/A", 'b':" ", 2:"???"},
...               filling_values={0:0, 'b':0, 2:-999})
>>> np.genfromtxt(StringIO(data), **kwargs)
array([(0, 2, 3), (4, 0, -999)],
      dtype=[('a', '<i8'), ('b', '<i8'), ('c', '<i8')])
```

usemask

We may also want to keep track of the occurrence of missing data by constructing a boolean mask, with **True** entries where data was missing and **False** otherwise. To do that, we just have to set the optional argument **usemask** to **True** (the default is **False**). The output array will then be a [MaskedArray](#).

Shortcut functions

In addition to [genfromtxt](#), the `numpy.lib.npyio` module provides several convenience functions derived from [genfromtxt](#). These functions work the same way as the original, but they have different default values.

`numpy.lib.npyio.recfromtxt`

Returns a standard [numpy.recarray](#) (if `usemask=False`) or a `numpy.ma.mrecords.MaskedRecords` array (if `usemaske=True`). The default dtype is `dtype=None`, meaning that the types of each column will be automatically determined.

`numpy.lib.npyio.recfromcsv`

Like `numpy.lib.npyio.recfromtxt`, but with a default `delimiter=","`.