Creating_Functions_part_2

May 8, 2023

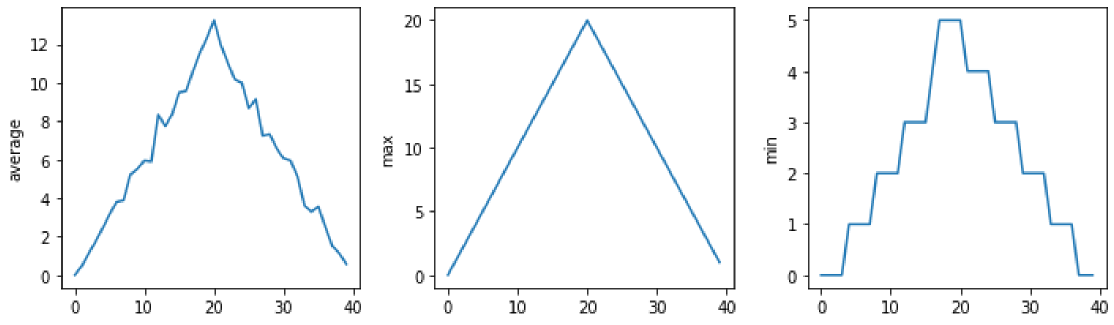
[17]:
```
import numpy
import glob
import matplotlib
import matplotlib.pyplot
```

[21]:
```
def visualize(filename):

    data = numpy.loadtxt(fname = filename, delimiter = ',')

    fig = matplotlib.pyplot.figure(figsize=(10.0, 3.0))

    axes1 = fig.add_subplot(1, 3, 1)
    axes2 = fig.add_subplot(1, 3, 2)
    axes3 = fig.add_subplot(1, 3, 3)

    axes1.set_ylabel('average')
    axes1.plot(numpy.mean(data, axis=0))

    axes2.set_ylabel('max')
    axes2.plot(numpy.amax(data, axis=0))

    axes3.set_ylabel('min')
    axes3.plot(numpy.amin(data, axis=0))

    fig.tight_layout()
    matplotlib.pyplot.show()
```

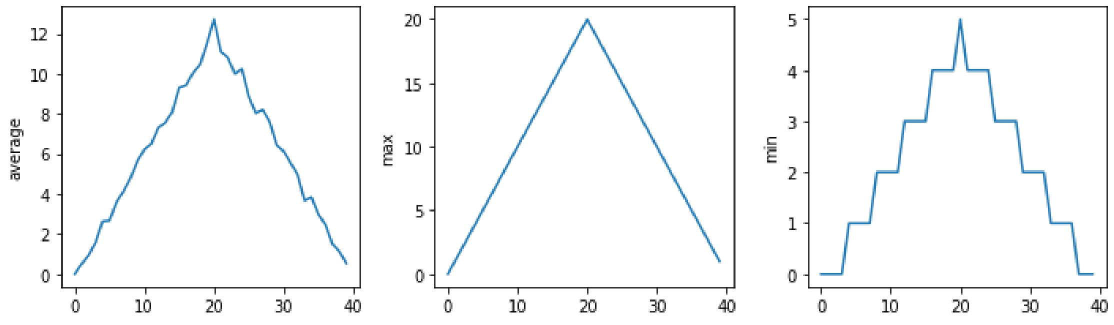
[22]:
```
def detect_problems(filename):

    data = numpy.loadtxt(fname = filename, delimiter = ',')

    if numpy.amax(data, axis = 0)[0] == 0 and numpy.amax(data, axis = 0)[20] ==␣
 ↪20:
        print('Suspicious looking maxima!')
    elif numpy.sum(numpy.amin(data, axis = 0)) == 0:
        print('Minima add up to zero!')
    else:
        print('seems ok!')
```

[25]:
```
filenames = sorted(glob.glob('inflammation*.csv'))

for filename in filenames:
    print(filename)
    visualize(filename)
    detect_problems(filename)
```

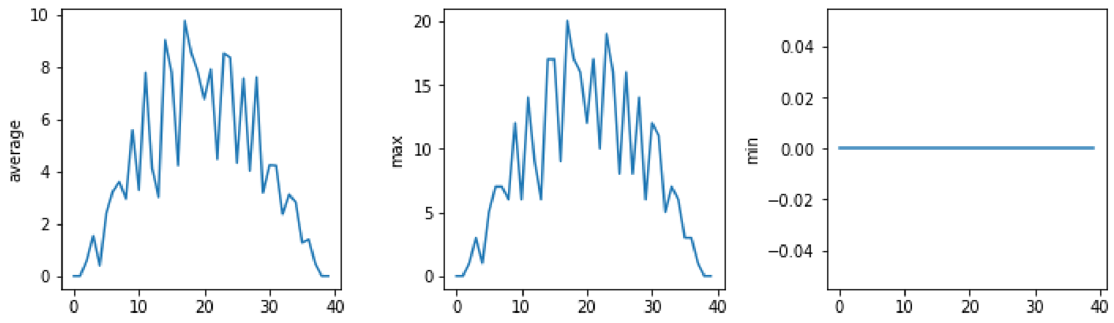
```
inflammation-01.csv
```



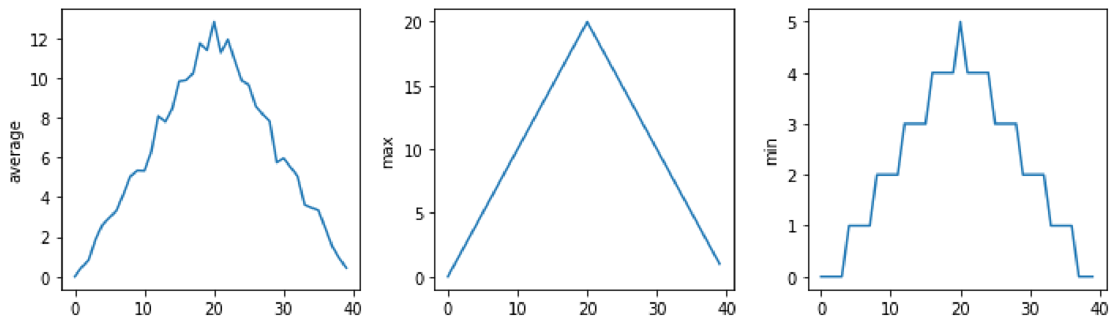
```
Suspicious looking maxima!
inflammation-02.csv
```



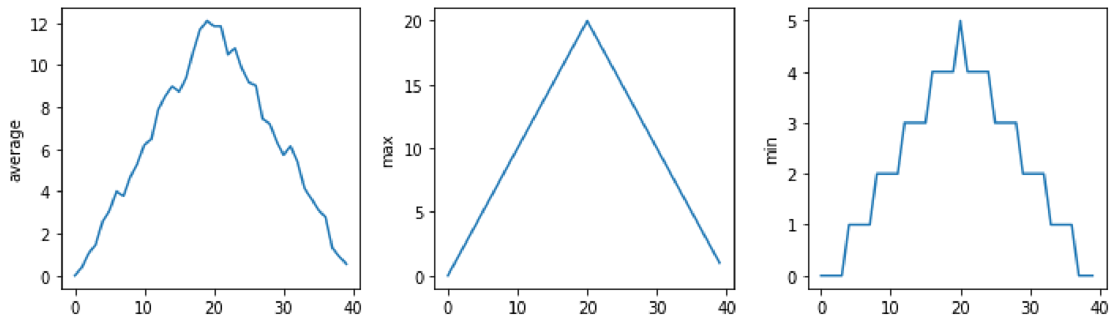
```
Suspicious looking maxima!
inflammation-03.csv
```



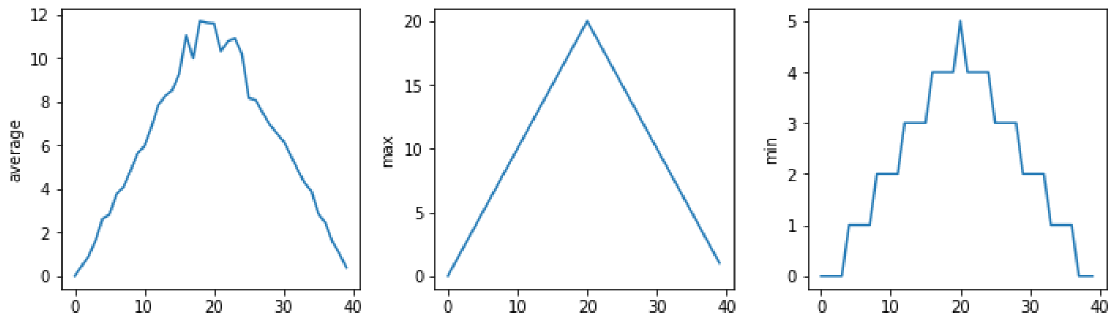
```
Minima add up to zero!
inflammation-04.csv
```



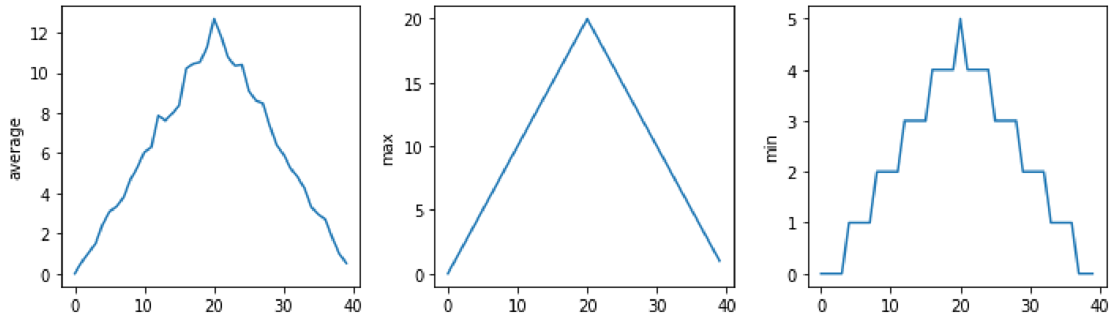
```
Suspicious looking maxima!
inflammation-05.csv
```



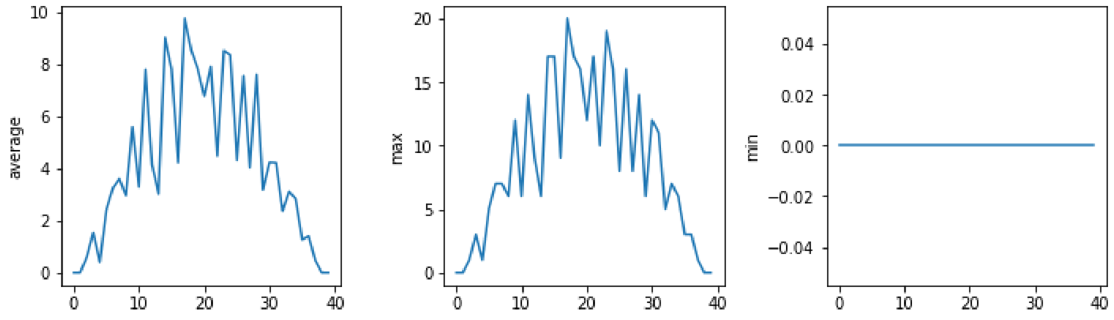
```
Suspicious looking maxima!
inflammation-06.csv
```



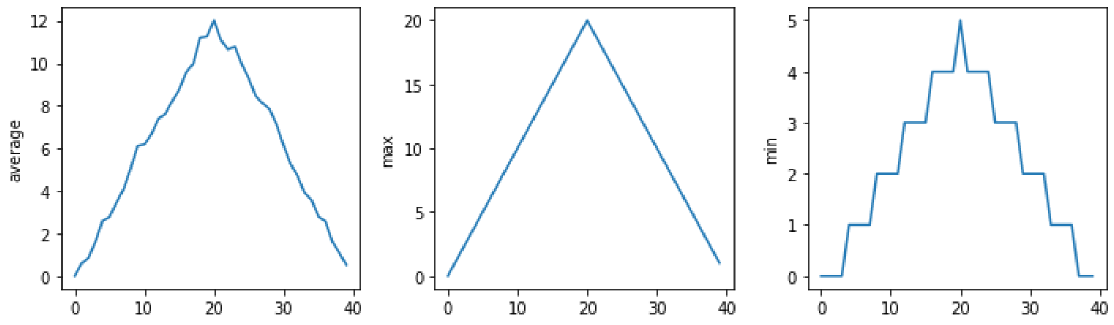
```
Suspicious looking maxima!
inflammation-07.csv
```



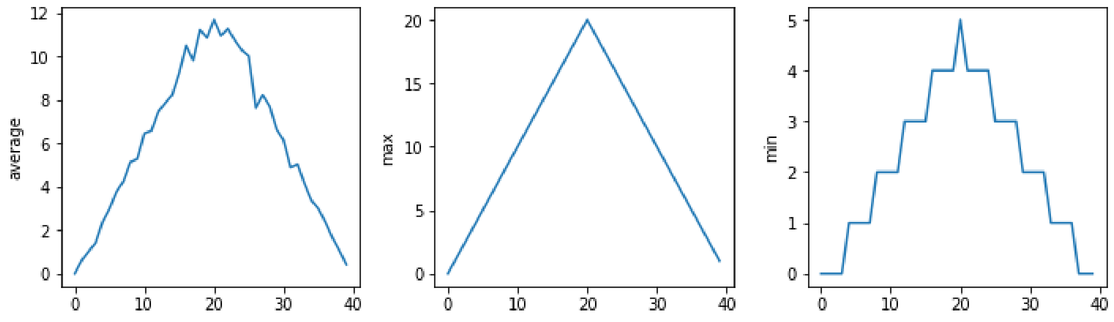
```
Suspicious looking maxima!
inflammation-08.csv
```



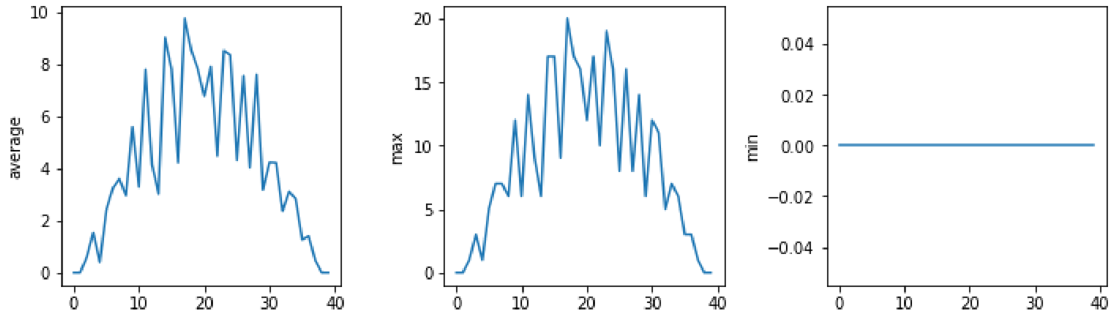
```
Minima add up to zero!
inflammation-09.csv
```



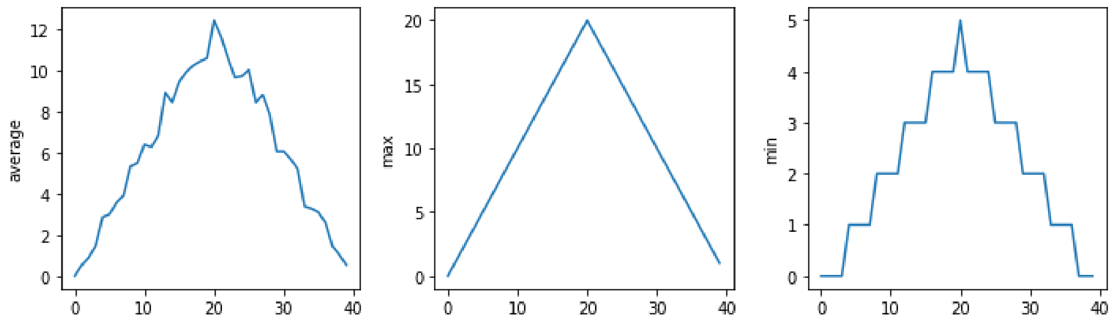
```
Suspicious looking maxima!
inflammation-10.csv
```



```
Suspicious looking maxima!
inflammation-11.csv
```



```
Minima add up to zero!
inflammation-12.csv
```



```
Suspicious looking maxima!
```

[29]:
```python
def offset_mean(data, target_mean_value):
    return(data - numpy.mean(data)) + target_mean_value
```

[30]:
```python
z = numpy.zeros((2,2))
print(offset_mean(z, 3))
```

```
[[3. 3.]
 [3. 3.]]
```

[32]:
```python
data = numpy.loadtxt(fname = 'inflammation-01.csv', delimiter = ',')

print(offset_mean(data, 0))
```

```
[[-6.14875 -6.14875 -5.14875 … -3.14875 -6.14875 -6.14875]
 [-6.14875 -5.14875 -4.14875 … -5.14875 -6.14875 -5.14875]
 [-6.14875 -5.14875 -5.14875 … -4.14875 -5.14875 -5.14875]
 …
 [-6.14875 -5.14875 -5.14875 … -5.14875 -5.14875 -5.14875]
 [-6.14875 -6.14875 -6.14875 … -6.14875 -4.14875 -6.14875]
 [-6.14875 -6.14875 -5.14875 … -5.14875 -5.14875 -6.14875]]
```

[33]:
```python
print('original mn, mean and max are:', numpy.amin(data), numpy.mean(data),␣
 ↪numpy.amax(data))
offset_data = offset_mean(data, 0)
print('min, mean, and max of offset data are:',
    numpy.amin(offset_data),
    numpy.mean(offset_data),
    numpy.amax(offset_data))
```

```
original mn, mean and max are: 0.0 6.14875 20.0
min, mean, and max of offset data are: -6.14875 2.842170943040401e-16 13.85125
```

[34]:
```python
print('std dev before and after:', numpy.std(data), numpy.std(offset_data))
```

```
std dev before and after: 4.613833197118566 4.613833197118566
```

[35]:
```python
print('difference in standard deviation before and after:',
      numpy.std(data)- numpy.std(offset_data))
```

```
difference in standard deviation before and after: 0.0
```

[36]:
```python
# offset_mean(data, target_mean_value):
# return a new array containing the original data with its mean offset to match␣
↪the desired value
def offset_mean(data, target_mean_value):
    return(data - numpy.mean(data)) + target_mean_value
```

[37]:
```python
def offset_mean(data, target_mean_value):
    """Return a new array containing the original data with its mean offset to␣
↪match the desired value"""
    return(data - numpy.mean(data)) + target_mean_value
```

[38]:
```python
help(offset_mean)
```

```
Help on function offset_mean in module __main__:

offset_mean(data, target_mean_value)
    Return a new array containing the original data with its mean offset to
match the desired value

```

[39]:
```python
def offset_mean(data, target_mean_value):
    """Return a new array containing the original data with its mean offset to␣
↪match desired value.

    Examples
    ----------

    >>>Offset_mean([1,2,3],0)
    array([-1., 0., 1.])
    """
    return (data - numpy.mean(data)) + target_mean_value
```

[40]:
```python
help(offset_mean)
```

```
Help on function offset_mean in module __main__:

offset_mean(data, target_mean_value)
    Return a new array containing the original data with its mean offset to
match desired value.

    Examples
    ----------

    >>>Offset_mean([1,2,3],0)
    array([-1., 0., 1.])
```

```
[41]: numpy.loadtxt('inflammation-01.csv', delimiter = ',')
```

```
[41]: array([[0., 0., 1., …, 3., 0., 0.],
             [0., 1., 2., …, 1., 0., 1.],
             [0., 1., 1., …, 2., 1., 1.],
             …,
             [0., 1., 1., …, 1., 1., 1.],
             [0., 0., 0., …, 0., 2., 0.],
             [0., 0., 1., …, 1., 1., 0.]])
```

```
[42]: def offset_mean(data, target_mean_value = 0.0):
          """Return a new array containing the original data with its mean offset to␣
       ↪match desired value, (0 by default).

          Examples
          ----------

          >>>offset_mean([1,2,3])
          array([-1., 0., 1.])
          """
          return (data - numpy.mean(data)) + target_mean_value
```

```
[43]: test_data = numpy.zeros((2,2))
      print(offset_mean(test_data, 3))
```

```
[[3. 3.]
 [3. 3.]]
```

```
[44]: print(offset_mean(test_data))
```

```
[[0. 0.]
 [0. 0.]]
```

```
[45]: def display(a=1, b=2, c=3):
          print('a:', a, 'b:', b, 'c:', c)

      print('no parameters:')
      display()
      print('one parameter:')
      display(55)
      print('two parameters')
      display(55,66)
```

```
no parameters:
a: 1 b: 2 c: 3
one parameter:
a: 55 b: 2 c: 3
two parameters
a: 55 b: 66 c: 3
```

[46]:
```python
print('only setting the value of c')
display(c = 77)
```

```
only setting the value of c
a: 1 b: 2 c: 77
```

[47]:
```python
help(numpy.loadtxt)
```

```
Help on function loadtxt in module numpy:

loadtxt(fname, dtype=<class 'float'>, comments='#', delimiter=None,
converters=None, skiprows=0, usecols=None, unpack=False, ndmin=0,
encoding='bytes', max_rows=None)
    Load data from a text file.

    Each row in the text file must have the same number of values.

    Parameters
    ----------
    fname : file, str, or pathlib.Path
        File, filename, or generator to read.  If the filename extension is
        ``.gz`` or ``.bz2``, the file is first decompressed. Note that
        generators should return byte strings for Python 3k.
    dtype : data-type, optional
        Data-type of the resulting array; default: float.  If this is a
        structured data-type, the resulting array will be 1-dimensional, and
        each row will be interpreted as an element of the array.  In this
        case, the number of columns used must match the number of fields in
        the data-type.
    comments : str or sequence of str, optional
        The characters or list of characters used to indicate the start of a
        comment. None implies no comments. For backwards compatibility, byte
        strings will be decoded as 'latin1'. The default is '#'.
    delimiter : str, optional
        The string used to separate values. For backwards compatibility, byte
        strings will be decoded as 'latin1'. The default is whitespace.
    converters : dict, optional
        A dictionary mapping column number to a function that will parse the
        column string into the desired value.  E.g., if column 0 is a date
        string: ``converters = {0: datestr2num}``.  Converters can also be
        used to provide a default value for missing data (but see also
        `genfromtxt`): ``converters = {3: lambda s: float(s.strip() or 0)}``.
        Default: None.
    skiprows : int, optional
        Skip the first `skiprows` lines, including comments; default: 0.
    usecols : int or sequence, optional
        Which columns to read, with 0 being the first. For example,
        ``usecols = (1,4,5)`` will extract the 2nd, 5th and 6th columns.
        The default, None, results in all columns being read.

        .. versionchanged:: 1.11.0
            When a single column has to be read it is possible to use
            an integer instead of a tuple. E.g ``usecols = 3`` reads the
            fourth column the same way as ``usecols = (3,)`` would.
    unpack : bool, optional
        If True, the returned array is transposed, so that arguments may be
        unpacked using ``x, y, z = loadtxt(...)``.  When used with a structured
        data-type, arrays are returned for each field.  Default is False.
    ndmin : int, optional
        The returned array will have at least `ndmin` dimensions.
        Otherwise mono-dimensional axes will be squeezed.
        Legal values: 0 (default), 1 or 2.

        .. versionadded:: 1.6.0
    encoding : str, optional
        Encoding used to decode the inputfile. Does not apply to input streams.
        The special value 'bytes' enables backward compatibility workarounds
        that ensures you receive byte arrays as results if possible and passes
        'latin1' encoded strings to converters. Override this value to receive
        unicode arrays and pass strings as input to converters.  If set to None
        the system default is used. The default value is 'bytes'.

        .. versionadded:: 1.14.0
    max_rows : int, optional
        Read `max_rows` lines of content after `skiprows` lines. The default
        is to read all the lines.

        .. versionadded:: 1.16.0

    Returns
    -------
    out : ndarray
        Data read from the text file.

    See Also
    --------
    load, fromstring, fromregex
    genfromtxt : Load data with missing values handled as specified.
    scipy.io.loadmat : reads MATLAB data files
```

```
    Notes
    -----
    This function aims to be a fast reader for simply formatted files.  The
    `genfromtxt` function provides more sophisticated handling of, e.g.,
    lines with missing values.

    .. versionadded:: 1.10.0

    The strings produced by the Python float.hex method can be used as
    input for floats.

    Examples
    --------
    >>> from io import StringIO   # StringIO behaves like a file object
    >>> c = StringIO(u"0 1\n2 3")
    >>> np.loadtxt(c)
    array([[0., 1.],
           [2., 3.]])

    >>> d = StringIO(u"M 21 72\nF 35 58")
    >>> np.loadtxt(d, dtype={'names': ('gender', 'age', 'weight'),
    …                        'formats': ('S1', 'i4', 'f4')})
    array([(b'M', 21, 72.), (b'F', 35, 58.)],
          dtype=[('gender', 'S1'), ('age', '<i4'), ('weight', '<f4')])

    >>> c = StringIO(u"1,0,2\n3,0,4")
    >>> x, y = np.loadtxt(c, delimiter=',', usecols=(0, 2), unpack=True)
    >>> x
    array([1., 3.])
    >>> y
    array([2., 4.])
```

```
[48]: numpy.loadtxt('inflammation-01.csv', delimiter = ',')
```

```
[48]: array([[0., 0., 1., …, 3., 0., 0.],
             [0., 1., 2., …, 1., 0., 1.],
             [0., 1., 1., …, 2., 1., 1.],
             …,
             [0., 1., 1., …, 1., 1., 1.],
             [0., 0., 0., …, 0., 2., 0.],
             [0., 0., 1., …, 1., 1., 0.]])
```

```
[ ]:
```