

CHAPTER 2

Vectors, Matrices, and Multidimensional Arrays

Vectors, matrices, and arrays of higher dimensions are essential tools in numerical computing. When a computation must be repeated for a set of input values, it is natural and advantageous to represent the data as arrays and the computation in terms of array operations. Computations that are formulated this way are said to be vectorized.¹ Vectorized computing eliminates the need for many explicit loops over the array elements by applying batch operations on the array data. The result is concise and more maintainable code, and it enables delegating the implementation of (e.g., elementwise) array operations to more efficient low-level libraries. Vectorized computations can therefore be significantly faster than sequential element-by-element computations. This is particularly important in an interpreted language such as Python, where looping over arrays element by element entails a significant performance overhead.

In Python's scientific computing environment, efficient data structures for working with arrays are provided by the NumPy library. The core of NumPy is implemented in C and provides efficient functions for manipulating and processing arrays. At a first glance, NumPy arrays bear some resemblance to Python's list data structure. But an important difference is that while Python lists are generic containers of objects, NumPy arrays are homogenous and typed arrays of fixed size. Homogenous means that all elements in the array have the same data type. Fixed size means that an array cannot be resized (without creating a new array). For these and other reasons, operations and functions acting on NumPy arrays can be much more efficient than those using Python lists. In addition to

¹Many modern processors provide instructions that operate on arrays. These are also known as vectorized operations, but here vectorized refers to high-level array-based operations, regardless of how they are implemented at the processor level.

the data structures for arrays, NumPy also provides a large collection of basic operators and functions that act on these data structures, as well as submodules with higher-level algorithms such as linear algebra and fast Fourier transform.

In this chapter we first look at the basic NumPy data structure for arrays and various methods to create such NumPy arrays. Next we look at operations for manipulating arrays and for doing computations with arrays. The multidimensional data array provided by NumPy is a foundation for nearly all numerical libraries for Python. Spending time on getting familiar with NumPy and developing an understanding of how NumPy works is therefore important.

NumPy The NumPy library provides data structures for representing a rich variety of arrays and methods and functions for operating on such arrays. NumPy provides the numerical backend for nearly every scientific or technical library for Python. It is therefore a very important part of the scientific Python ecosystem. At the time of writing, the latest version of NumPy is 1.14.2. More information about NumPy is available at www.numpy.org.

Importing the Modules

In order to use the NumPy library, we need to import it in our program. By convention, the `numpy` module imported under the alias `np`, like so:

```
In [1]: import numpy as np
```

After this, we can access functions and classes in the `numpy` module using the `np` namespace. Throughout this book, we assume that the NumPy module is imported in this way.

The NumPy Array Object

The core of the NumPy library is the data structures for representing multidimensional arrays of homogeneous data. Homogeneous refers to all elements in an array having the same data type.² The main data structure for multidimensional arrays in NumPy

²This does not necessarily need to be the case for Python lists, which therefore can be heterogenous.

is the `ndarray` class. In addition to the data stored in the array, this data structure also contains important metadata about the array, such as its shape, size, data type, and other attributes. See Table 2-1 for a more detailed description of these attributes. A full list of attributes with descriptions is available in the `ndarray` docstring, which can be accessed by calling `help(np.ndarray)` in the Python interpreter or `np.ndarray?` in an IPython console.

Table 2-1. *Basic Attributes of the ndarray Class*

Attribute	Description
Shape	A tuple that contains the number of elements (i.e., the length) for each dimension (axis) of the array.
Size	The total number elements in the array.
Ndim	Number of dimensions (axes).
nbytes	Number of bytes used to store the data.
dtype	The data type of the elements in the array.

The following example demonstrates how these attributes are accessed for an instance data of the class `ndarray`:

```
In [2]: data = np.array([[1, 2], [3, 4], [5, 6]])
In [3]: type(data)
Out[3]: <class 'numpy.ndarray'>
In [4]: data
Out[4]: array([[1, 2],
               [3, 4],
               [5, 6]])
In [5]: data.ndim
Out[5]: 2
In [6]: data.shape
Out[6]: (3, 2)
In [7]: data.size
Out[7]: 6
```

```
In [8]: data.dtype
Out[8]: dtype('int64')
In [9]: data.nbytes
Out[9]: 48
```

Here the ndarray instance `data` is created from a nested Python list using the function `np.array`. More ways to create ndarray instances from data and from rules of various kinds are introduced later in this chapter. In the preceding example, the data is a two-dimensional array (`data.ndim`) of shape 3×2 , as indicated by `data.shape`, and in total it contains six elements (`data.size`) of type `int64` (`data.dtype`), which amounts to a total size of 48 bytes (`data.nbytes`).

Data Types

In the previous section, we encountered the `dtype` attribute of the ndarray object. This attribute describes the data type of each element in the array (remember, since NumPy arrays are homogeneous, all elements have the same data type). The basic numerical data types supported in NumPy are shown in Table 2-2. Nonnumerical data types, such as strings, objects, and user-defined compound types, are also supported.

Table 2-2. Basic Numerical Data Types Available in NumPy

dtype	Variants	Description
int	int8, int16, int32, int64	Integers
uint	uint8, uint16, uint32, uint64	Unsigned (nonnegative) integers
bool	Bool	Boolean (True or False)
float	float16, float32, float64, float128	Floating-point numbers
complex	complex64, complex128, complex256	Complex-valued floating-point numbers

For numerical work the most important data types are `int` (for integers), `float` (for floating-point numbers), and `complex` (for complex floating-point numbers). Each of these data types comes in different sizes, such as `int32` for 32-bit integers, `int64` for 64-bit integers, etc. This offers more fine-grained control over data types than the standard Python types, which only provides one type for integers and one type for floats.

It is usually not necessary to explicitly choose the bit size of the data type to work with, but it is often necessary to explicitly choose whether to use arrays of integers, floating-point numbers, or complex values.

The following example demonstrates how to use the `dtype` attribute to generate arrays of integer-, float-, and complex-valued elements:

```
In [10]: np.array([1, 2, 3], dtype=np.int)
Out[10]: array([1, 2, 3])
In [11]: np.array([1, 2, 3], dtype=np.float)
Out[11]: array([ 1.,  2.,  3.])
In [12]: np.array([1, 2, 3], dtype=np.complex)
Out[12]: array([ 1.+0.j,  2.+0.j,  3.+0.j])
```

Once a NumPy array is created, its `dtype` cannot be changed, other than by creating a new copy with type-casted array values. Typecasting an array is straightforward and can be done using either the `np.array` function:

```
In [13]: data = np.array([1, 2, 3], dtype=np.float)
In [14]: data
Out[14]: array([ 1.,  2.,  3.])
In [15]: data.dtype
Out[15]: dtype('float64')
In [16]: data = np.array(data, dtype=np.int)
In [17]: data.dtype
Out[17]: dtype('int64')
In [18]: data
Out[18]: array([1, 2, 3])
```

or by using the `astype` method of the `ndarray` class:

```
In [19]: data = np.array([1, 2, 3], dtype=np.float)
In [20]: data
Out[20]: array([ 1.,  2.,  3.])
In [21]: data.astype(np.int)
Out[21]: array([1, 2, 3])
```

When computing with NumPy arrays, the data type might get promoted from one type to another, if required by the operation. For example, adding float-valued and complex-valued arrays, the resulting array is a complex-valued array:

```
In [22]: d1 = np.array([1, 2, 3], dtype=float)
In [23]: d2 = np.array([1, 2, 3], dtype=complex)
In [24]: d1 + d2
Out[24]: array([ 2.+0.j,  4.+0.j,  6.+0.j])
In [25]: (d1 + d2).dtype
Out[25]: dtype('complex128')
```

In some cases, depending on the application and its requirements, it is essential to create arrays with data type appropriately set to, for example, `int` or `complex`. The default type is `float`. Consider the following example:

```
In [26]: np.sqrt(np.array([-1, 0, 1]))
Out[26]: RuntimeWarning: invalid value encountered in sqrt
         array([ nan,   0.,   1.])
In [27]: np.sqrt(np.array([-1, 0, 1], dtype=complex))
Out[27]: array([ 0.+1.j,  0.+0.j,  1.+0.j])
```

Here, using the `np.sqrt` function to compute the square root of each element in an array gives different results depending on the data type of the array. Only when the data type of the array is `complex` is the square root of `-1` resulting in the imaginary unit (denoted as `1j` in Python).

Real and Imaginary Parts

Regardless of the value of the `dtype` attribute, all NumPy array instances have the attributes `real` and `imag` for extracting the real and imaginary parts of the array, respectively:

```
In [28]: data = np.array([1, 2, 3], dtype=complex)
In [29]: data
Out[29]: array([ 1.+0.j,  2.+0.j,  3.+0.j])
In [30]: data.real
Out[30]: array([ 1.,  2.,  3.])
In [31]: data.imag
Out[31]: array([ 0.,  0.,  0.])
```

The same functionality is also provided by the functions `np.real` and `np.imag`, which also can be applied to other array-like objects, such as Python lists. Note that Python itself has support of complex numbers, and the `imag` and `real` attributes are also available for Python scalars.

Order of Array Data in Memory

Multidimensional arrays are stored as contiguous data in memory. There is a freedom of choice in how to arrange the array elements in this memory segment. Consider the case of a two-dimensional array, containing rows and columns: one possible way to store this array as a consecutive sequence of values is to store the rows after each other, and another equally valid approach is to store the columns one after another. The former is called row-major format and the latter is column-major format. Whether to use row-major or column-major is a matter of conventions, and row-major format is used, for example, in the C programming language, and Fortran uses the column-major format. A NumPy array can be specified to be stored in row-major format, using the keyword argument `order= 'C'`, and column-major format, using the keyword argument `order= 'F'`, when the array is created or reshaped. The default format is row-major. The 'C' or 'F' ordering of NumPy array is particularly relevant when NumPy arrays are used in interfaces with software written in C and Fortran, which is often required when working with numerical computing with Python.

Row-major and column-major ordering are special cases of strategies for mapping the index used to address an element, to the offset for the element in the array's memory segment. In general, the NumPy array attribute `ndarray.strides` defines exactly how this mapping is done. The `strides` attribute is a tuple of the same length as the number of axes (dimensions) of the array. Each value in `strides` is the factor by which the index for the corresponding axis is multiplied when calculating the memory offset (in bytes) for a given index expression.

For example, consider a C-order array `A` with shape `(2, 3)`, which corresponds to a two-dimensional array with two and three elements along the first and the second dimensions, respectively. If the data type is `int32`, then each element uses 4 bytes, and the total memory buffer for the array therefore uses $2 \times 3 \times 4 = 24$ bytes. The `strides` attribute of this array is therefore `(4 × 3, 4 × 1) = (12, 4)`, because each increment of `m` in `A[n, m]` increases the memory offset with one item, or 4 bytes. Likewise, each increment of `n` increases the memory offset with three items or 12 bytes (because the second

dimension of the array has length 3). If, on the other hand, the same array were stored in 'F' order, the `strides` would instead be (4, 8). Using strides to describe the mapping of array index to array memory offset is clever because it can be used to describe different mapping strategies, and many common operations on arrays, such as for example the transpose, can be implemented by simply changing the `strides` attribute, which can eliminate the need for moving data around in the memory. Operations that only require changing the `strides` attribute result in new `ndarray` objects that refer to the same data as the original array. Such arrays are called views. For efficiency, NumPy strives to create views rather than copies when applying operations on arrays. This is generally a good thing, but it is important to be aware of that some array operations result in views rather than new independent arrays, because modifying their data also modifies the data of the original array. Later in this chapter, we will see several examples of this behavior.

Creating Arrays

In the previous section, we looked at NumPy's basic data structure for representing arrays, the `ndarray` class, and we looked at the basic attributes of this class. In this section we focus on functions from the NumPy library that can be used to create `ndarray` instances.

Arrays can be generated in a number of ways, depending on their properties and the applications they are used for. For example, as we saw in the previous section, one way to initialize an `ndarray` instance is to use the `np.array` function on a Python list, which, for example, can be explicitly defined. However, this method is obviously limited to small arrays. In many situations it is necessary to generate arrays with elements that follow some given rule, such as filled with constant values, increasing integers, uniformly spaced numbers, random numbers, etc. In other cases we might need to create arrays from data stored in a file. The requirements are many and varied, and the NumPy library provides a comprehensive set of functions for generating arrays of various types. In this section we look in more detail at many of these functions. For a complete list, see the NumPy reference manual or the docstrings that are available by typing `help(np)` or using the autocompletion `np.<TAB>`. A summary of frequently used array-generating functions is given in Table 2-3.

Table 2-3. *Summary of NumPy Functions for Generating Arrays*

Function Name	Type of Array
<code>np.array</code>	Creates an array for which the elements are given by an array-like object, which, for example, can be a (nested) Python list, a tuple, an iterable sequence, or another ndarray instance.
<code>np.zeros</code>	Creates an array with the specified dimensions and data type that is filled with zeros.
<code>np.ones</code>	Creates an array with the specified dimensions and data type that is filled with ones.
<code>np.diag</code>	Creates a diagonal array with specified values along the diagonal and zeros elsewhere.
<code>np.arange</code>	Creates an array with evenly spaced values between the specified start, end, and increment values.
<code>np.linspace</code>	Creates an array with evenly spaced values between specified start and end values, using a specified number of elements.
<code>np.logspace</code>	Creates an array with values that are logarithmically spaced between the given start and end values.
<code>np.meshgrid</code>	Generates coordinate matrices (and higher-dimensional coordinate arrays) from one-dimensional coordinate vectors.
<code>np.fromfunction</code>	Creates an array and fills it with values specified by a given function, which is evaluated for each combination of indices for the given array size.
<code>np.fromfile</code>	Creates an array with the data from a binary (or text) file. NumPy also provides a corresponding function <code>np.tofile</code> with which NumPy arrays can be stored to disk and later read back using <code>np.fromfile</code> .
<code>np.genfromtxt</code> , <code>np.loadtxt</code>	Create an array from data read from a text file, for example, a comma-separated value (CSV) file. The function <code>np.genfromtxt</code> also supports data files with missing values.
<code>np.random.rand</code>	Generates an array with random numbers that are uniformly distributed between 0 and 1. Other types of distributions are also available in the <code>np.random</code> module.

Arrays Created from Lists and Other Array-Like Objects

Using the `np.array` function, NumPy arrays can be constructed from explicit Python lists, iterable expressions, and other array-like objects (such as other `ndarray` instances). For example, to create a one-dimensional array from a Python list, we simply pass the Python list as an argument to the `np.array` function:

```
In [32]: np.array([1, 2, 3, 4])
Out[32]: array([ 1,  2,  3,  4])
In [33]: data.ndim
Out[33]: 1
In [34]: data.shape
Out[34]: (4,)
```

To create a two-dimensional array with the same data as in the previous example, we can use a nested Python list:

```
In [35]: np.array([[1, 2], [3, 4]])
Out[35]: array([[1,  2],
                [3,  4]])
In [36]: data.ndim
Out[36]: 2
In [37]: data.shape
Out[37]: (2, 2)
```

Arrays Filled with Constant Values

The functions `np.zeros` and `np.ones` create and return arrays filled with zeros and ones, respectively. They take, as first argument, an integer or a tuple that describes the number of elements along each dimension of the array. For example, to create a 2×3 array filled with zeros, and an array of length 4 filled with ones, we can use

```
In [38]: np.zeros((2, 3))
Out[38]: array([[ 0.,  0.,  0.],
                [ 0.,  0.,  0.]])
In [39]: np.ones(4)
Out[39]: array([ 1.,  1.,  1.,  1.]])
```

Like other array-generating functions, the `np.zeros` and `np.ones` functions also accept an optional keyword argument that specifies the data type for the elements in the array. By default, the data type is `float64`, and it can be changed to the required type by explicitly specifying the `dtype` argument.

```
In [40]: data = np.ones(4)
In [41]: data.dtype
Out[41]: dtype('float64')
In [42]: data = np.ones(4, dtype=np.int64)
In [43]: data.dtype
Out[43]: dtype('int64')
```

An array filled with an arbitrary constant value can be generated by first creating an array filled with ones and then multiplying the array with the desired fill value. However, NumPy also provides the function `np.full` that does exactly this in one step. The following two ways of constructing arrays with ten elements, which are initialized to the numerical value 5.4 in this example, produces the same results, but using `np.full` is slightly more efficient since it avoids the multiplication.

```
In [44]: x1 = 5.4 * np.ones(10)
In [45]: x2 = np.full(10, 5.4)
```

An already created array can also be filled with constant values using the `np.fill` function, which takes an array and a value as arguments, and set all elements in the array to the given value. The following two methods to create an array therefore give the same results:

```
In [46]: x1 = np.empty(5)
In [47]: x1.fill(3.0)
In [48]: x1
Out[48]: array([ 3.,  3.,  3.,  3.,  3.])
In [49]: x2 = np.full(5, 3.0)
In [50]: x2
Out[50]: array([ 3.,  3.,  3.,  3.,  3.])
```

In this last example, we also used the `np.empty` function, which generates an array with uninitialized values, of the given size. This function should only be used when the initialization of all elements can be guaranteed by other means, such as an explicit loop over the array elements or another explicit assignment. This function is described in more detail later in this chapter.

Arrays Filled with Incremental Sequences

In numerical computing it is very common to require arrays with evenly spaced values between a starting value and ending value. NumPy provides two similar functions to create such arrays: `np.arange` and `np.linspace`. Both functions take three arguments, where the first two arguments are the start and end values. The third argument of `np.arange` is the increment, while for `np.linspace` it is the total number of points in the array.

For example, to generate arrays with values between 1 and 10, with increment 1, we could use either of the following:

```
In [51]: np.arange(0.0, 10, 1)
Out[51]: array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
In [52]: np.linspace(0, 10, 11)
Out[52]: array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.])
```

However, note that `np.arange` does not include the end value (10), while by default `np.linspace` does (although this behavior can be changed using the optional endpoint keyword argument). Whether to use `np.arange` or `np.linspace` is mostly a matter of personal preference, but it is generally recommended to use `np.linspace` whenever the increment is a noninteger.

Arrays Filled with Logarithmic Sequences

The function `np.logspace` is similar to `np.linspace`, but the increments between the elements in the array are logarithmically distributed, and the first two arguments, for the start and end values, are the powers of the optional base keyword argument (which defaults to 10). For example, to generate an array with logarithmically distributed values between 1 and 100, we can use

```
In [53]: np.logspace(0, 2, 5) # 5 data points between 10**0=1 to 10**2=100
Out[53]: array([ 1. ,  3.16227766, 10. , 31.6227766 , 100.])
```

Meshgrid Arrays

Multidimensional coordinate grids can be generated using the function `np.meshgrid`. Given two one-dimensional coordinate arrays (i.e., arrays containing a set of coordinates along a given dimension), we can generate two-dimensional coordinate arrays using the `np.meshgrid` function. An illustration of this is given in the following example:

```
In [54]: x = np.array([-1, 0, 1])
In [55]: y = np.array([-2, 0, 2])
In [56]: X, Y = np.meshgrid(x, y)
In [57]: X
Out[57]: array([[ -1,  0,  1],
                [ -1,  0,  1],
                [ -1,  0,  1]])

In [58]: Y
Out[58]: array([[ -2, -2, -2],
                [  0,  0,  0],
                [  2,  2,  2]])
```

A common use-case of the two-dimensional coordinate arrays, like `X` and `Y` in this example, is to evaluate functions over two variables x and y . This can be used when plotting functions over two variables, as colormap plots and contour plots. For example, to evaluate the expression $(x+y)^2$ at all combinations of values from the `x` and `y` arrays in the preceding section, we can use the two-dimensional coordinate arrays `X` and `Y`:

```
In [59]: Z = (X + Y) ** 2
In [60]: Z
Out[60]: array([[ 9,  4,  1],
                [ 1,  0,  1],
                [ 1,  4,  9]])
```

It is also possible to generate higher-dimensional coordinate arrays by passing more arrays as argument to the `np.meshgrid` function. Alternatively, the functions `np.mgrid` and `np.ogrid` can also be used to generate coordinate arrays, using a slightly different syntax based on indexing and slice objects. See their docstrings or the NumPy documentation for details.

Creating Uninitialized Arrays

To create an array of specific size and data type, but without initializing the elements in the array to any particular values, we can use the function `np.empty`. The advantage of using this function, for example, instead of `np.zeros`, which creates an array initialized with zero-valued elements, is that we can avoid the initiation step. If all elements are guaranteed to be initialized later in the code, this can save a little bit of time, especially when working with large arrays. To illustrate the use of the `np.empty` function, consider the following example:

```
In [61]: np.empty(3, dtype=np.float)
Out[61]: array([ 1.28822975e-231,  1.28822975e-231,  2.13677905e-314])
```

Here we generated a new array with three elements of type `float`. There is no guarantee that the elements have any particular values, and the actual values will vary from time to time. For this reason it is important that all values are explicitly assigned before the array is used; otherwise unpredictable errors are likely to arise. Often the `np.zeros` function is a safer alternative to `np.empty`, and if the performance gain is not essential, it is better to use `np.zeros`, to minimize the likelihood of subtle and hard-to-reproduce bugs due to uninitialized values in the array returned by `np.empty`.

Creating Arrays with Properties of Other Arrays

It is often necessary to create new arrays that share properties, such as shape and data type, with another array. NumPy provides a family of functions for this purpose: `np.ones_like`, `np.zeros_like`, `np.full_like`, and `np.empty_like`. A typical use-case is a function that takes arrays of unspecified type and size as arguments and requires working arrays of the same size and type. For example, a boilerplate example of this situation is given in the following function:

```
def f(x):
    y = np.ones_like(x)
    # compute with x and y
    return y
```

At the first line of the body of this function, a new array `y` is created using `np.ones_like`, which results in an array of the same size and data type as `x`, and filled with ones.

Creating Matrix Arrays

Matrices, or two-dimensional arrays, are an important case for numerical computing. NumPy provides functions for generating commonly used matrices. In particular, the function `np.identity` generates a square matrix with ones on the diagonal and zeros elsewhere:

```
In [62]: np.identity(4)
Out[62]: array([[ 1.,  0.,  0.,  0.],
                [ 0.,  1.,  0.,  0.],
                [ 0.,  0.,  1.,  0.],
                [ 0.,  0.,  0.,  1.]])
```

The similar function `numpy.eye` generates matrices with ones on a diagonal (optionally offset). This is illustrated in the following example, which produces matrices with nonzero diagonals above and below the diagonal, respectively:

```
In [63]: np.eye(3, k=1)
Out[63]: array([[ 0.,  1.,  0.],
                [ 0.,  0.,  1.],
                [ 0.,  0.,  0.]])
```

```
In [64]: np.eye(3, k=-1)
Out[64]: array([[ 0.,  0.,  0.],
                [ 1.,  0.,  0.],
                [ 0.,  1.,  0.]])
```

To construct a matrix with an arbitrary one-dimensional array on the diagonal, we can use the `np.diag` function (which also takes the optional keyword argument `k` to specify an offset from the diagonal), as demonstrated here:

```
In [65]: np.diag(np.arange(0, 20, 5))
Out[65]: array([[0,  0,  0,  0],
                [0,  5,  0,  0],
                [0,  0, 10,  0],
                [0,  0,  0, 15]])
```

Here we gave a third argument to the `np.arange` function, which specifies the step size in the enumeration of elements in the array returned by the function. The resulting array therefore contains the values `[0, 5, 10, 15]`, which is inserted on the diagonal of a two-dimensional matrix by the `np.diag` function.

Indexing and Slicing

Elements and subarrays of NumPy arrays are accessed using the standard square bracket notation that is also used with Python lists. Within the square bracket, a variety of different index formats are used for different types of element selection. In general, the expression within the bracket is a tuple, where each item in the tuple is a specification of which elements to select from each axis (dimension) of the array.

One-Dimensional Arrays

Along a single axis, integers are used to select single elements, and so-called slices are used to select ranges and sequences of elements. Positive integers are used to index elements from the beginning of the array (index starts at 0), and negative integers are used to index elements from the end of the array, where the last element is indexed with `-1`, the second to last element with `-2`, and so on.

Slices are specified using the `:` notation that is also used for Python lists. In this notation, a range of elements can be selected using an expression like `m:n`, which selects elements starting with m and ending with $n - 1$ (note that the n th element is not included). The slice `m:n` can also be written more explicitly as `m : n : 1`, where the number 1 specifies that every element between m and n should be selected. To select every second element between m and n , use `m : n : 2`, and to select every p elements, use `m : n : p`, and so on. If p is negative, elements are returned in reversed order starting from m to $n+1$ (which implies that m has to be larger than n in this case). See Table 2-4 for a summary of indexing and slicing operations for NumPy arrays.

Table 2-4. *Examples of Array Indexing and Slicing Expressions*

Expression	Description
<code>a[m]</code>	Select element at index m , where m is an integer (start counting from 0).
<code>a[-m]</code>	Select the n th element from the end of the list, where n is an integer. The last element in the list is addressed as -1 , the second to last element as -2 , and so on.
<code>a[m:n]</code>	Select elements with index starting at m and ending at $n - 1$ (m and n are integers).
<code>a[:]</code> or <code>a[0:-1]</code>	Select all elements in the given axis.
<code>a[:n]</code>	Select elements starting with index 0 and going up to index $n - 1$ (integer).
<code>a[m:]</code> or <code>a[m:-1]</code>	Select elements starting with index m (integer) and going up to the last element in the array.
<code>a[m:n:p]</code>	Select elements with index m through n (exclusive), with increment p .
<code>a[::-1]</code>	Select all the elements, in reverse order.

The following examples demonstrate index and slicing operations for NumPy arrays. To begin with, consider an array with a single axis (dimension) that contains a sequence of integers between 0 and 10:

```
In [66]: a = np.arange(0, 11)
In [67]: a
Out[67]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

Note that the end value 11 is not included in the array. To select specific elements from this array, for example, the first, the last, and the 5th element, we can use integer indexing:

```
In [68]: a[0] # the first element
Out[68]: 0
In [69]: a[-1] # the last element
Out[69]: 10
In [70]: a[4] # the fifth element, at index 4
Out[70]: 4
```

To select a range of element, say from the second to the second-to-last element, selecting every element and every second element, respectively, we can use index slices:

```
In [71]: a[1:-1]
Out[71]: array([1, 2, 3, 4, 5, 6, 7, 8, 9])
In [72]: a[1:-1:2]
Out[72]: array([1, 3, 5, 7, 9])
```

To select the first five and the last five elements from an array, we can use the slices `:5` and `-5:`, since if `m` or `n` is omitted in `m:n`, the defaults are the beginning and the end of the array, respectively.

```
In [73]: a[:5]
Out[73]: array([0, 1, 2, 3, 4])
In [74]: a[-5:]
Out[74]: array([6, 7, 8, 9, 10])
```

To reverse the array and select only every second value, we can use the slice `::-2`, as shown in the following example:

```
In [75]: a[::-2]
Out[75]: array([10, 8, 6, 4, 2, 0])
```

Multidimensional Arrays

With multidimensional arrays, element selections like those introduced in the previous section can be applied on each axis (dimension). The result is a reduced array where each element matches the given selection rules. As a specific example, consider the following two-dimensional array:

```
In [76]: f = lambda m, n: n + 10 * m
In [77]: A = np.fromfunction(f, (6, 6), dtype=int)
In [78]: A
Out[78]: array([[ 0,  1,  2,  3,  4,  5],
                [10, 11, 12, 13, 14, 15],
                [20, 21, 22, 23, 24, 25],
                [30, 31, 32, 33, 34, 35],
                [40, 41, 42, 43, 44, 45],
                [50, 51, 52, 53, 54, 55]])
```

We can extract columns and rows from this two-dimensional array using a combination of slice and integer indexing:

```
In [79]: A[:, 1] # the second column
Out[79]: array([ 1, 11, 21, 31, 41, 51])
In [80]: A[1, :] # the second row
Out[80]: array([10, 11, 12, 13, 14, 15])
```

By applying a slice on each of the array axes, we can extract subarrays (submatrices in this two-dimensional example):

```
In [81]: A[:3, :3] # upper half diagonal block matrix
Out[81]: array([[ 0,  1,  2],
               [10, 11, 12],
               [20, 21, 22]])
In [82]: A[3:, :3] # lower left off-diagonal block matrix
Out[82]: array([[30, 31, 32],
               [40, 41, 42],
               [50, 51, 52]])
```

With element spacing other than 1, submatrices made up from nonconsecutive elements can be extracted:

```
In [83]: A[::2, ::2] # every second element starting from 0, 0
Out[83]: array([[ 0,  2,  4],
               [20, 22, 24],
               [40, 42, 44]])
In [84]: A[1::2, 1::3] # every second and third element starting from 1, 1
Out[84]: array([[11, 14],
               [31, 34],
               [51, 54]])
```

This ability to extract subsets of data from a multidimensional array is a simple but very powerful feature with many data processing applications.

Views

Subarrays that are extracted from arrays using slice operations are alternative *views* of the same underlying array data. That is, they are arrays that refer to the same data in the memory as the original array, but with a different `strides` configuration. When elements in a view are assigned new values, the values of the original array are therefore also updated. For example,

```
In [85]: B = A[1:5, 1:5]
In [86]: B
Out[86]: array([[11, 12, 13, 14],
                [21, 22, 23, 24],
                [31, 32, 33, 34],
                [41, 42, 43, 44]])

In [87]: B[:, :] = 0
In [88]: A
Out[88]: array([[ 0,  1,  2,  3,  4,  5],
                [10,  0,  0,  0,  0, 15],
                [20,  0,  0,  0,  0, 25],
                [30,  0,  0,  0,  0, 35],
                [40,  0,  0,  0,  0, 45],
                [50, 51, 52, 53, 54, 55]])
```

Here, assigning new values to the elements in an array `B`, which is created from the array `A`, also modifies the values in `A` (since both arrays refer to the same data in the memory). The fact that extracting subarrays results in views rather than new independent arrays eliminates the need for copying data and improves performance. When a copy rather than a view is needed, the view can be copied explicitly by using the `copy` method of the `ndarray` instance.

```
In [89]: C = B[1:3, 1:3].copy()
In [90]: C
Out[90]: array([[0, 0],
                [0, 0]])

In [91]: C[:, :] = 1 # this does not affect B since C is a copy of the
view B[1:3, 1:3]
In [92]: C
```

```

Out[92]: array([[1, 1],
               [1, 1]])
In [93]: B
Out[93]: array([[0, 0, 0, 0],
               [0, 0, 0, 0],
               [0, 0, 0, 0],
               [0, 0, 0, 0]])

```

In addition to the `copy` attribute of the `ndarray` class, an array can also be copied using the function `np.copy` or, equivalently, using the `np.array` function with the keyword argument `copy=True`.

Fancy Indexing and Boolean-Valued Indexing

In the previous section, we looked at indexing NumPy arrays with integers and slices, to extract individual elements or ranges of elements. NumPy provides another convenient method to index arrays, called fancy indexing. With fancy indexing, an array can be indexed with another NumPy array, a Python list, or a sequence of integers, whose values select elements in the indexed array. To clarify this concept, consider the following example: we first create a NumPy array with 11 floating-point numbers, and then index the array with another NumPy array (and Python list), to extract element numbers 0, 2, and 4 from the original array:

```

In [94]: A = np.linspace(0, 1, 11)
Out[94]: array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9,  1. ])
In [95]: A[np.array([0, 2, 4])]
Out[95]: array([ 0. ,  0.2,  0.4])
In [96]: A[[0, 2, 4]] # The same thing can be accomplished by indexing with a
Python list
Out[96]: array([ 0. ,  0.2,  0.4])

```

This method of indexing can be used along each axis (dimension) of a multidimensional NumPy array. It requires that the elements in the array or list used for indexing are integers.

Another variant of indexing NumPy arrays is to use Boolean-valued index arrays. In this case, each element (with values `True` or `False`) indicates whether or not to select the element from the list with the corresponding index. That is, if element n in the indexing array of Boolean values is `True`, then element n is selected from the indexed array. If the value is `False`, then element n is not selected. This index method is handy when filtering out elements from an array. For example, to select all the elements from the array `A` (as defined in the preceding section) that exceed the value 0.5, we can use the following combination of the comparison operator applied to a NumPy array and indexing using a Boolean-valued array:

```
In [97]: A > 0.5
Out[97]: array([False, False, False, False, False, False, True, True, True,
               True, True], dtype=bool)
In [98]: A[A > 0.5]
Out[98]: array([ 0.6,  0.7,  0.8,  0.9,  1.  ])
```

Unlike arrays created by using slices, the arrays returned using fancy indexing and Boolean-valued indexing are not views but rather new independent arrays. Nonetheless, it is possible to assign values to elements selected using fancy indexing:

```
In [99]: A = np.arange(10)
In [100]: indices = [2, 4, 6]
In [101]: B = A[indices]
In [102]: B[0] = -1 # this does not affect A
In [103]: A
Out[103]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
In [104]: A[indices] = -1 # this alters A
In [105]: A
Out[105]: array([ 0,  1, -1,  3, -1,  5, -1,  7,  8,  9])
```

and likewise for Boolean-valued indexing:

```
In [106]: A = np.arange(10)
In [107]: B = A[A > 5]
In [108]: B[0] = -1 # this does not affect A
In [109]: A
Out[109]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [110]: A[A > 5] = -1 # this alters A
```

```
In [111]: A
```

```
Out[111]: array([ 0,  1,  2,  3,  4,  5, -1, -1, -1, -1])
```

A visual summary of different methods to index NumPy arrays is given in Figure 2-1. Note that each type of indexing we have discussed here can be independently applied to each dimension of an array.

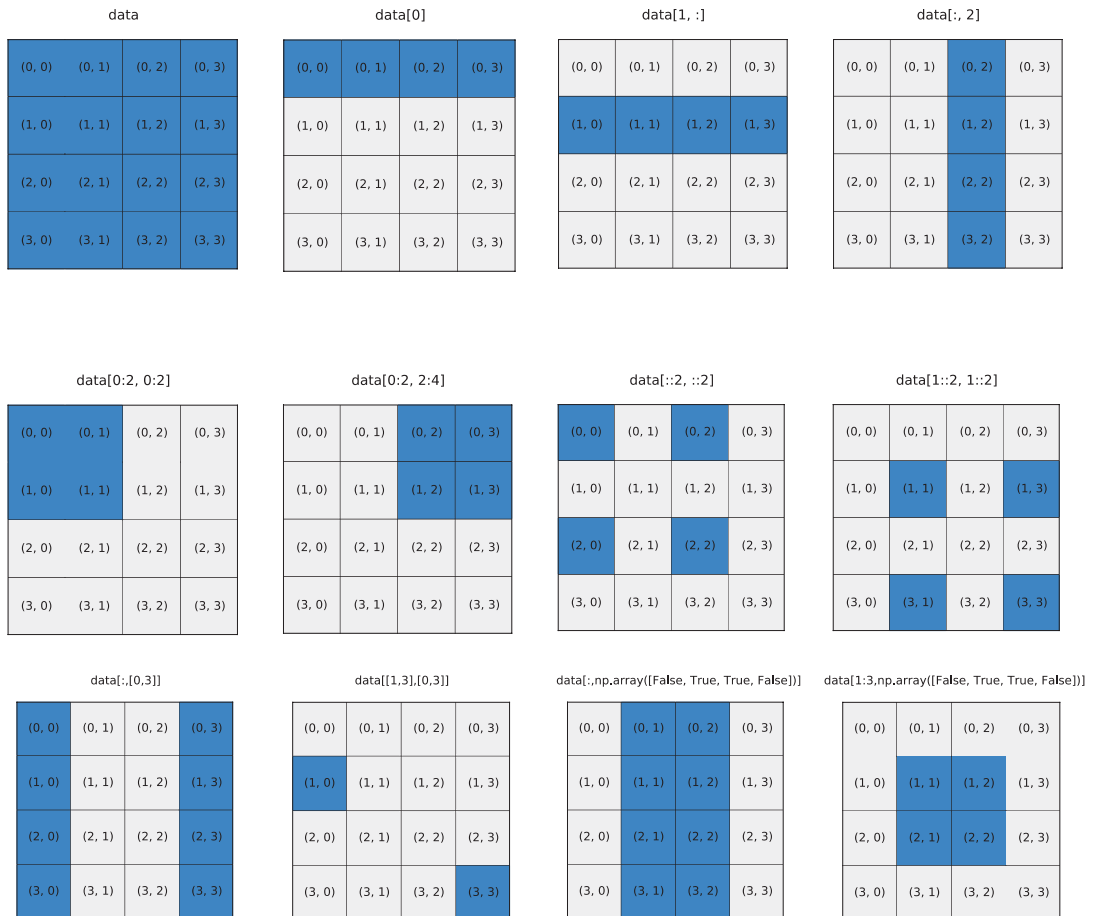


Figure 2-1. Visual summary of indexing methods for NumPy arrays. These diagrams represent NumPy arrays of shape (4, 4), and the highlighted elements are those that are selected using the indexing expression shown above the block representations of the arrays.

Reshaping and Resizing

When working with data in array form, it is often useful to rearrange arrays and alter the way they are interpreted. For example, an $N \times N$ matrix array could be rearranged into a vector of length N^2 , or a set of one-dimensional arrays could be concatenated together or stacked next to each other to form a matrix. NumPy provides a rich set of functions of this type of manipulation. See Table 2-5 for a summary of a selection of these functions.

Table 2-5. *Summary of NumPy Functions for Manipulating the Dimensions and the Shape of Arrays*

Function/Method	Description
<code>np.reshape,</code> <code>np.ndarray.reshape</code>	Reshape an N-dimensional array. The total number of elements must remain the same.
<code>np.ndarray.flatten</code>	Creates a copy of an N-dimensional array, and reinterpret it as a one-dimensional array (i.e., all dimensions are collapsed into one).
<code>np.ravel,</code> <code>np.ndarray.ravel</code>	Create a view (if possible, otherwise a copy) of an N-dimensional array in which it is interpreted as a one-dimensional array.
<code>np.squeeze</code>	Removes axes with length 1.
<code>np.expand_dims,</code> <code>np.newaxis</code>	Add a new axis (dimension) of length 1 to an array, where <code>np.newaxis</code> is used with array indexing.
<code>np.transpose,</code> <code>np.ndarray.transpose,</code> <code>np.ndarray.T</code>	Transpose the array. The transpose operation corresponds to reversing (or more generally, permuting) the axes of the array.
<code>np.hstack</code>	Stacks a list of arrays horizontally (along axis 1): for example, given a list of column vectors, appends the columns to form a matrix.
<code>np.vstack</code>	Stacks a list of arrays vertically (along axis 0): for example, given a list of row vectors, appends the rows to form a matrix.
<code>np.dstack</code>	Stacks arrays depth-wise (along axis 2).
<code>np.concatenate</code>	Creates a new array by appending arrays after each other, along a given axis.

(continued)

Table 2-5. *(continued)*

Function/Method	Description
<code>np.resize</code>	Resizes an array. Creates a new copy of the original array, with the requested size. If necessary, the original array will be repeated to fill up the new array.
<code>np.append</code>	Appends an element to an array. Creates a new copy of the array.
<code>np.insert</code>	Inserts a new element at a given position. Creates a new copy of the array.
<code>np.delete</code>	Deletes an element at a given position. Creates a new copy of the array.

Reshaping an array does not require modifying the underlying array data; it only changes in how the data is interpreted, by redefining the array's `strides` attribute. An example of this type of operation is a 2×2 array (matrix) that is reinterpreted as a 1×4 array (vector). In NumPy, the function `np.reshape`, or the `ndarray` class method `reshape`, can be used to reconfigure how the underlying data is interpreted. It takes an array and the new shape of the array as arguments:

```
In [112]: data = np.array([[1, 2], [3, 4]])
In [113]: np.reshape(data, (1, 4))
Out[113]: array([[1, 2, 3, 4]])
In [114]: data.reshape(4)
Out[114]: array([1, 2, 3, 4])
```

It is necessary that the requested new shape of the array match the number of elements in the original size. However, the number of axes (dimensions) does not need to be conserved, as illustrated in the previous example, where in the first case, the new array has dimension 2 and shape `(1, 4)`, while in the second case, the new array has dimension 1 and shape `(4,)`. This example also demonstrates two different ways of invoking the reshape operation: using the function `np.reshape` and the `ndarray` method `reshape`. Note that reshaping an array produces a view of the array, and if an independent copy of the array is needed, the view has to be copied explicitly (e.g., using `np.copy`).

The `np.ravel` (and its corresponding `ndarray` method) is a special case of `reshape`, which collapses all dimensions of an array and returns a flattened one-dimensional array with a length that corresponds to the total number of elements in the original array. The `ndarray` method `flatten` performs the same function but returns a copy instead of a view.

```
In [115]: data = np.array([[1, 2], [3, 4]])
In [116]: data
Out[116]: array([[1, 2],
                  [3, 4]])
In [117]: data.flatten()
Out[117]: array([ 1,  2,  3,  4])
In [118]: data.flatten().shape
Out[118]: (4,)
```

While `np.ravel` and `np.flatten` collapse the axes of an array into a one-dimensional array, it is also possible to introduce new axes into an array, either by using `np.reshape` or, when adding new empty axes, using indexing notation and the `np.newaxis` keyword at the place of a new axis. In the following example, the array `data` has one axis, so it should normally be indexed with a tuple with one element. However, if it is indexed with a tuple with more than one element, and if the extra indices in the tuple have the value `np.newaxis`, then the corresponding new axes are added:

```
In [119]: data = np.arange(0, 5)
In [120]: column = data[:, np.newaxis]
In [121]: column
Out[121]: array([[0],
                  [1],
                  [2],
                  [3],
                  [4]])
In [122]: row = data[np.newaxis, :]
In [123]: row
Out[123]: array([[0, 1, 2, 3, 4]])
```

The function `np.expand_dims` can also be used to add new dimensions to an array, and in the preceding example, the expression `data[:, np.newaxis]` is equivalent to `np.expand_dims(data, axis=1)`, and `data[np.newaxis, :]` is equivalent to `np.expand_dims(data, axis=0)`. Here the `axis` argument specifies the location relative to the existing axes where the new axis is to be inserted.

We have up to now looked at methods to rearrange arrays in ways that do not affect the underlying data. Earlier in this chapter, we also looked at how to extract subarrays using various indexing techniques. In addition to reshaping and selecting subarrays, it is often necessary to merge arrays into bigger arrays, for example, when joining separately computed or measured data series into a higher-dimensional array, such as a matrix. For this task, NumPy provides the functions `np.vstack`, for vertical stacking of, for example, rows into a matrix, and `np.hstack` for horizontal stacking of, for example, columns into a matrix. The function `np.concatenate` provides similar functionality, but it takes a keyword argument `axis` that specifies the axis along which the arrays are to be concatenated.

The shape of the arrays passed to `np.hstack`, `np.vstack`, and `np.concatenate` is important to achieve the desired type of array joining. For example, consider the following cases: say we have one-dimensional arrays of data, and we want to stack them vertically to obtain a matrix where the rows are made up of the one-dimensional arrays. We can use `np.vstack` to achieve this

```
In [124]: data = np.arange(5)
In [125]: data
Out[125]: array([0, 1, 2, 3, 4])
In [126]: np.vstack((data, data, data))
Out[126]: array([[0, 1, 2, 3, 4],
                  [0, 1, 2, 3, 4],
                  [0, 1, 2, 3, 4]])
```

If we instead want to stack the arrays horizontally, to obtain a matrix where the arrays are the column vectors, we might first attempt something similar using `np.hstack`:

```
In [127]: data = np.arange(5)
In [128]: data
Out[128]: array([0, 1, 2, 3, 4])
In [129]: np.hstack((data, data, data))
Out[129]: array([0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0, 1, 2, 3, 4])
```

This indeed stacks the arrays horizontally, but not in the way intended here. To make `np.hstack` treat the input arrays as columns and stack them accordingly, we need to make the input arrays two-dimensional arrays of shape `(1, 5)` rather than one-dimensional arrays of shape `(5,)`. As discussed earlier, we can insert a new axis by indexing with `np.newaxis`:

```
In [130]: data = data[:, np.newaxis]
In [131]: np.hstack((data, data, data))
Out[131]: array([[0, 0, 0],
                  [1, 1, 1],
                  [2, 2, 2],
                  [3, 3, 3],
                  [4, 4, 4]])
```

The behavior of the functions for horizontal and vertical stacking, as well as concatenating arrays using `np.concatenate`, is clearest when the stacked arrays have the same number of dimensions as the final array and when the input arrays are stacked along an axis for which they have length 1.

The number of elements in a NumPy array cannot be changed once the array has been created. To insert, append, and remove elements from a NumPy array, for example, using the function `np.append`, `np.insert`, and `np.delete`, a new array must be created and the data copied to it. It may sometimes be tempting to use these functions to grow or shrink the size of a NumPy array, but due to the overhead of creating new arrays and copying the data, it is usually a good idea to preallocate arrays with size such that they do not later need to be resized.

Vectorized Expressions

The purpose of storing numerical data in arrays is to be able to process the data with concise vectorized expressions that represent batch operations that are applied to all elements in the arrays. Efficient use of vectorized expressions eliminates the need of many explicit for loops. This results in less verbose code, better maintainability, and higher-performing code. NumPy implements functions and vectorized operations corresponding to most fundamental mathematical functions and operators. Many of these functions and operations act on arrays on an elementwise basis, and binary operations require all arrays in an expression to be of compatible size. The meaning of

compatible size is normally that the variables in an expression represent either scalars or arrays of the same size and shape. More generally, a binary operation involving two arrays is well defined if the arrays can be *broadcasted* into the same shape and size.

In the case of an operation between a scalar and an array, broadcasting refers to the scalar being distributed and the operation applied to each element in the array. When an expression contains arrays of unequal sizes, the operations may still be well defined if the smaller of the array can be broadcasted (“effectively expanded”) to match the larger array according to NumPy’s broadcasting rule: an array can be broadcasted over another array if their axes on a one-by-one basis either have the same length or if either of them have length 1. If the number of axes of the two arrays is not equal, the array with fewer axes is padded with new axes of length 1 from the left until the numbers of dimensions of the two arrays agree.

Two simple examples that illustrate array broadcasting are shown in Figure 2-2: a 3×3 matrix is added to a 1×3 row vector and a 3×1 column vector, respectively, and in both cases the result is a 3×3 matrix. However, the elements in the two resulting matrices are different, because the way the elements of the row and column vectors are broadcasted to the shape of the larger array is different depending on the shape of the arrays, according to NumPy’s broadcasting rule.

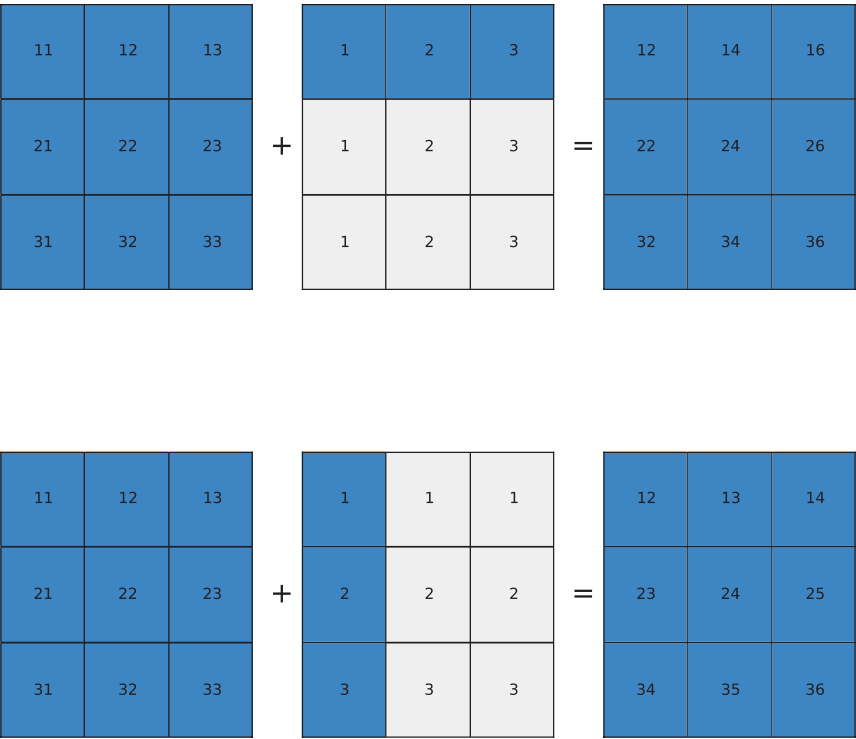


Figure 2-2. Visualization of broadcasting of row and column vectors into the shape of a matrix. The highlighted elements represent true elements of the arrays, while the light gray-shaded elements describe the broadcasting of the elements of the array of smaller size.

Arithmetic Operations

The standard arithmetic operations with NumPy arrays perform elementwise operations. Consider, for example, the addition, subtraction, multiplication, and division of equal-sized arrays:

```
In [132]: x = np.array([[1, 2], [3, 4]])
In [133]: y = np.array([[5, 6], [7, 8]])
In [134]: x + y
Out[134]: array([[ 6,  8],
                 [10, 12]])
```

```

In [135]: y - x
Out[135]: array([[4, 4],
                 [4, 4]])
In [136]: x * y
Out[136]: array([[ 5, 12],
                 [21, 32]])
In [137]: y / x
Out[137]: array([[ 5.          ,  3.          ],
                 [ 2.33333333,  2.          ]])

```

In operations between scalars and arrays, the scalar value is applied to each element in the array, as one could expect:

```

In [138]: x * 2
Out[138]: array([[2, 4],
                 [6, 8]])
In [139]: 2 ** x
Out[139]: array([[ 2,  4],
                 [ 8, 16]])
In [140]: y / 2
Out[140]: array([[ 2.5,  3. ],
                 [ 3.5,  4. ]])
In [141]: (y / 2).dtype
Out[141]: dtype('float64')

```

Note that the dtype of the resulting array for an expression can be promoted if the computation requires it, as shown in the preceding example with division between an integer array and an integer scalar, which in that case resulted in an array with a dtype that is `np.float64`.

If an arithmetic operation is performed on arrays with incompatible size or shape, a `ValueError` exception is raised:

```

In [142]: x = np.array([1, 2, 3, 4]).reshape(2, 2)
In [143]: z = np.array([1, 2, 3, 4])
In [144]: x / z

```

```

-----
ValueError                                Traceback (most recent call last)
<ipython-input-144-b88ced08eb6a> in <module>()
----> 1 x / z
ValueError: operands could not be broadcast together with shapes (2,2) (4,)

```

Here the array `x` has shape `(2, 2)` and the array `z` has shape `(4,)`, which cannot be broadcasted into a form that is compatible with `(2, 2)`. If, on the other hand, `z` has shape `(2,)`, `(2, 1)`, or `(1, 2)`, then it can be broadcasted to the shape `(2, 2)` by effectively repeating the array `z` along the axis with length 1. Let's first consider an example with an array `z` of shape `(1, 2)`, where the first axis (axis 0) has length 1:

```

In [145]: z = np.array([[2, 4]])
In [146]: z.shape
Out[146]: (1, 2)

```

Dividing the array `x` with array `z` is equivalent to dividing `x` with an array `zz` that is constructed by repeating (here using `np.concatenate`) the row vector `z` to obtain an array `zz` that has the same dimensions as `x`:

```

In [147]: x / z
Out[147]: array([[ 0.5,  0.5],
                  [ 1.5,  1. ]])
In [148]: zz = np.concatenate([z, z], axis=0)
In [149]: zz
Out[149]: array([[2, 4],
                  [2, 4]])
In [150]: x / zz
Out[150]: array([[ 0.5,  0.5],
                  [ 1.5,  1. ]])

```

Let's also consider the example in which the array `z` has shape `(2, 1)` and where the second axis (axis 1) has length 1:

```

In [151]: z = np.array([[2], [4]])
In [152]: z.shape
Out[152]: (2, 1)

```


In this case, dividing x with z is equivalent to dividing x with an array zz that is constructed by repeating the column vector z until a matrix with the same dimension as x is obtained.

```
In [153]: x / z
```

```
Out[153]: array([[ 0.5 ,  1.  ],
                  [ 0.75,  1.  ]])
```

```
In [154]: zz = np.concatenate([z, z], axis=1)
```

```
In [155]: zz
```

```
Out[155]: array([[2, 2],
                  [4, 4]])
```

```
In [156]: x / zz
```

```
Out[156]: array([[ 0.5 ,  1.  ],
                  [ 0.75,  1.  ]])
```

In summary, these examples show how arrays with shape $(1, 2)$ and $(2, 1)$ are broadcasted to the shape $(2, 2)$ of the array x when the operation x / z is performed. In both cases, the result of the operation x / z is the same as first repeating the smaller array z along its axis of length 1 to obtain a new array zz with the same shape as x and then performing the equal-sized array operation x / zz . However, the implementation of the broadcasting does not explicitly perform this expansion and the corresponding memory copies, but it can be helpful to think of the array broadcasting in these terms.

A summary of the operators for arithmetic operations with NumPy arrays is given in Table 2-6. These operators use the standard symbols used in Python. The result of an arithmetic operation with one or two arrays is a new independent array, with its own data in the memory. Evaluating complicated arithmetic expression might therefore trigger many memory allocation and copy operations, and when working with large arrays, this can lead to a large memory footprint and impact the performance negatively. In such cases, using inplace operation (see Table 2-6) can reduce the memory footprint and improve performance. As an example of inplace operators, consider the following two statements, which have the same effect:

```
In [157]: x = x + y
```

```
In [158]: x += y
```

Table 2-6. *Operators for Elementwise Arithmetic Operation on NumPy Arrays*

Operator	Operation
<code>+, +=</code>	Addition
<code>-, -=</code>	Subtraction
<code>*, *=</code>	Multiplication
<code>/, /=</code>	Division
<code>//, //=</code>	Integer division
<code>**, **=</code>	Exponentiation

The two expressions have the same effect, but in the first case, `x` is reassigned to a new array, while in the second case, the values of array `x` are updated inplace. Extensive use of inplace operators tends to impair code readability, and inplace operators should therefore be used only when necessary.

Elementwise Functions

In addition to arithmetic expressions using operators, NumPy provides vectorized functions for elementwise evaluation of many elementary mathematical functions and operations. Table 2-7 gives a summary of elementary mathematical functions in NumPy.³ Each of these functions takes a single array (of arbitrary dimension) as input and returns a new array of the same shape, where for each element the function has been applied to the corresponding element in the input array. The data type of the output array is not necessarily the same as that of the input array.

³Note that this is not a complete list of the available elementwise functions in NumPy. See the NumPy reference documentations for comprehensive lists.

Table 2-7. *Selection of NumPy Functions for Elementwise Elementary Mathematical Functions*

NumPy Function	Description
<code>np.cos</code> , <code>np.sin</code> , <code>np.tan</code>	Trigonometric functions.
<code>np.arccos</code> , <code>np.arcsin</code> , <code>np.arctan</code>	Inverse trigonometric functions.
<code>np.cosh</code> , <code>np.sinh</code> , <code>np.tanh</code>	Hyperbolic trigonometric functions.
<code>np.arccosh</code> , <code>np.arcsinh</code> , <code>np.arctanh</code>	Inverse hyperbolic trigonometric functions.
<code>np.sqrt</code>	Square root.
<code>np.exp</code>	Exponential.
<code>np.log</code> , <code>np.log2</code> , <code>np.log10</code>	Logarithms of base e, 2, and 10, respectively.

For example, the `np.sin` function (which takes only one argument) is used to compute the sine function for all values in the array:

```
In [159]: x = np.linspace(-1, 1, 11)
In [160]: x
Out[160]: array([-1. , -0.8, -0.6, -0.4, -0.2,  0. ,  0.2,  0.4,  0.6,  0.8,  1.])
In [161]: y = np.sin(np.pi * x)
In [162]: np.round(y, decimals=4)
Out[162]: array([-0., -0.5878, -0.9511, -0.9511, -0.5878,  0.,  0.5878,  0.9511,
                0.9511,  0.5878,  0.])
```

Here we also used the constant `np.pi` and the function `np.round` to round the values of `y` to four decimals. Like the `np.sin` function, many of the elementary math functions take one input array and produce one output array. In contrast, many of the mathematical operator functions (Table 2-8) operates on two input arrays returns one array:

```
In [163]: np.add(np.sin(x) ** 2, np.cos(x) ** 2)
Out[163]: array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.])
In [164]: np.sin(x) ** 2 + np.cos(x) ** 2
Out[164]: array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.])
```

Table 2-8. *Summary of NumPy Functions for Elementwise Mathematical Operations*

NumPy Function	Description
np.add, np.subtract, np.multiply, np.divide	Addition, subtraction, multiplication, and division of two NumPy arrays.
np.power	Raises first input argument to the power of the second input argument (applied elementwise).
np.remainder	The remainder of division.
np.reciprocal	The reciprocal (inverse) of each element.
np.real, np.imag, np.conj	The real part, imaginary part, and the complex conjugate of the elements in the input arrays.
np.sign, np.abs	The sign and the absolute value.
np.floor, np.ceil, np rint	Convert to integer values.
np.round	Rounds to a given number of decimals.

Note that in this example, `np.add` and the operator `+` are equivalent, and for normal use the operator should be used.

Occasionally it is necessary to define new functions that operate on NumPy arrays on an element-by-element basis. A good way to implement such functions is to express it in terms of already existing NumPy operators and expressions, but in cases when this is not possible, the `np.vectorize` function can be a convenient tool. This function takes a nonvectorized function and returns a vectorized function. For example, consider the following implementation of the Heaviside step function, which works for scalar input:

```
In [165]: def heaviside(x):
...:     return 1 if x > 0 else 0
In [166]: heaviside(-1)
Out[166]: 0
In [167]: heaviside(1.5)
Out[167]: 1
```

However, unfortunately this function does not work for NumPy array input:

```
In [168]: x = np.linspace(-5, 5, 11)
In [169]: heaviside(x)
...
ValueError: The truth value of an array with more than one element is
ambiguous. Use a.any() or a.all()
```

Using `np.vectorize` the scalar Heaviside function can be converted into a vectorized function that works with NumPy arrays as input:

```
In [170]: heaviside = np.vectorize(heaviside)
In [171]: heaviside(x)
Out[171]: array([0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1])
```

Although the function returned by `np.vectorize` works with arrays, it will be relatively slow since the original function must be called for each element in the array. There are much better ways to implementing this particular function using arithmetic with Boolean-valued arrays, as discussed later in this chapter:

```
In [172]: def heaviside(x):
...:     return 1.0 * (x > 0)
```

Nonetheless, `np.vectorize` can often be a quick and convenient way to vectorize a function written for scalar input.

In addition to NumPy's functions for elementary mathematical function, as summarized in Table 2-7, there are also numerous functions in NumPy for mathematical operations. A summary of a selection of these functions is given in Table 2-8.

Aggregate Functions

NumPy provides another set of functions for calculating aggregates for NumPy arrays, which take an array as input and by default return a scalar as output. For example, statistics such as averages, standard deviations, and variances of the values in the input array, and functions for calculating the sum and the product of elements in an array, are all aggregate functions.

A summary of aggregate functions is given in Table 2-9. All of these functions are also available as methods in the ndarray class. For example, `np.mean(data)` and `data.mean()` in the following example are equivalent:

```
In [173]: data = np.random.normal(size=(15,15))
In [174]: np.mean(data)
Out[174]: -0.032423651106794522
In [175]: data.mean()
Out[175]: -0.032423651106794522
```

Table 2-9. NumPy Functions for Calculating Aggregates of NumPy Arrays

NumPy Function	Description
<code>np.mean</code>	The average of all values in the array.
<code>np.std</code>	Standard deviation.
<code>np.var</code>	Variance.
<code>np.sum</code>	Sum of all elements.
<code>np.prod</code>	Product of all elements.
<code>np.cumsum</code>	Cumulative sum of all elements.
<code>np.cumprod</code>	Cumulative product of all elements.
<code>np.min</code> , <code>np.max</code>	The minimum/maximum value in an array.
<code>np.argmax</code> , <code>np.argmin</code>	The index of the minimum/maximum value in an array.
<code>np.all</code>	Returns True if all elements in the argument array are nonzero.
<code>np.any</code>	Returns True if any of the elements in the argument array is nonzero.

By default, the functions in Table 2-9 aggregate over the entire input array. Using the `axis` keyword argument with these functions, and their corresponding ndarray methods, it is possible to control over which axis in the array aggregation is carried out. The `axis` argument can be an integer, which specifies the axis to aggregate values over. In many cases the `axis` argument can also be a tuple of integers, which specifies multiple axes to aggregate over. The following example demonstrates how calling the aggregate

function `np.sum` on the array of shape (5, 10, 15) reduces the dimensionality of the array depending on the values of the axis argument:

```
In [176]: data = np.random.normal(size=(5, 10, 15))
In [177]: data.sum(axis=0).shape
Out[177]: (10, 15)
In [178]: data.sum(axis=(0, 2)).shape
Out[178]: (10,)
In [179]: data.sum()
Out[179]: -31.983793284860798
```

A visual illustration of how aggregation over all elements, over the first axis, and over the second axis of a 3×3 array is shown in Figure 2-3. In this example, the data array is filled with integers between 1 and 9:

```
In [180]: data = np.arange(1,10).reshape(3,3)
In [181]: data
Out[181]: array([[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9]])
```

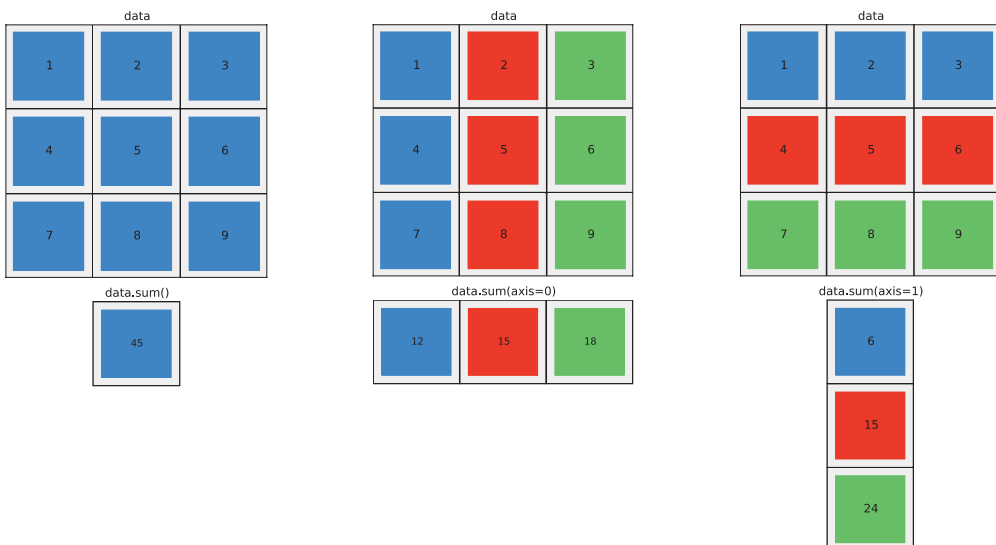


Figure 2-3. Illustration of array aggregation functions along all axes (left), the first axis (center), and the second axis (right) of a two-dimensional array of shape 3×3

and we compute the aggregate sum of the entire array, over the axis 0, and over axis 1, respectively:

```
In [182]: data.sum()
Out[182]: 45
In [183]: data.sum(axis=0)
Out[183]: array([12, 15, 18])
In [184]: data.sum(axis=1)
Out[184]: array([ 6, 15, 24])
```

Boolean Arrays and Conditional Expressions

When computing with NumPy arrays, there is often a need to compare elements in different arrays and perform conditional computations based on the results of such comparisons. Like with arithmetic operators, NumPy arrays can be used with the usual comparison operators, for example, `>`, `<`, `>=`, `<=`, `==`, and `!=`, and the comparisons are made on an element-by-element basis. The broadcasting rules also apply to comparison operators, and if two operators have compatible shapes and sizes, the result of the comparison is a new array with Boolean values (with dtype as `np.bool`) that gives the result of the comparison for each element:

```
In [185]: a = np.array([1, 2, 3, 4])
In [186]: b = np.array([4, 3, 2, 1])
In [187]: a < b
Out[187]: array([ True,  True, False, False], dtype=bool)
```

To use the result of a comparison between arrays in, for example, an `if` statement, we need to aggregate the Boolean values of the resulting arrays in some suitable fashion, to obtain a single `True` or `False` value. A common use-case is to apply the `np.all` or `np.any` aggregation functions, depending on the situation at hand:

```
In [188]: np.all(a < b)
Out[188]: False
In [189]: np.any(a < b)
Out[189]: True
In [190]: if np.all(a < b):
```



```

...: print("All elements in a are smaller than their corresponding
      element in b")
...: elif np.any(a < b):
...: print("Some elements in a are smaller than their corresponding
      element in b")
...: else:
...: print("All elements in b are smaller than their corresponding
      element in a")

```

Some elements in a are smaller than their corresponding element in b

The advantage of Boolean-valued arrays, however, is that they often make it possible to avoid conditional `if` statements altogether. By using Boolean-valued arrays in arithmetic expressions, it is possible to write conditional computations in vectorized form. When appearing in an arithmetic expression together with a scalar number, or another NumPy array with a numerical data type, a Boolean array is converted to a numerical-valued array with values 0 and 1 in place of `False` and `True`, respectively.

```

In [191]: x = np.array([-2, -1, 0, 1, 2])
In [192]: x > 0
Out[192]: array([False, False, False,  True,  True], dtype=bool)
In [193]: 1 * (x > 0)
Out[193]: array([0, 0, 0, 1, 1])
In [194]: x * (x > 0)
Out[194]: array([0, 0, 0, 1, 2])

```

This is a useful property for conditional computing, such as when defining piecewise functions. For example, if we need to define a function describing a pulse of a given height, width, and position, we can implement this function by multiplying the height (a scalar variable) with two Boolean-valued arrays for the spatial extension of the pulse:

```

In [195]: def pulse(x, position, height, width):
...:     return height * (x >= position) * (x <= (position + width))
In [196]: x = np.linspace(-5, 5, 11)
In [197]: pulse(x, position=-2, height=1, width=5)
Out[197]: array([0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0])
In [198]: pulse(x, position=1, height=1, width=5)
Out[198]: array([0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1])

```

In this example, the expression `(x >= position) * (x <= (position + width))` is a multiplication of two Boolean-valued arrays, and for this case the multiplication operator acts as an elementwise AND operator. The function `pulse` could also be implemented using NumPy's function for elementwise AND operations, `np.logical_and`:

```
In [199]: def pulse(x, position, height, width):
...:     return height * np.logical_and(x >= position, x <= (position +
...:                                     width))
```

There are also functions for other logical operations, such as NOT, OR, and XOR, and functions for selectively picking values from different arrays depending on a given condition `np.where`, a list of conditions `np.select`, and an array of indices `np.choose`. See Table 2-10 for a summary of such functions, and the following examples demonstrate the basic usage of some of these functions. The `np.where` function selects elements from two arrays (second and third arguments), given a Boolean-valued array condition (the first argument). For elements where the condition is True, the corresponding values from the array given as second argument are selected, and if the condition is False, elements from the third argument array are selected:

```
In [200]: x = np.linspace(-4, 4, 9)
In [201]: np.where(x < 0, x**2, x**3)
Out[201]: array([ 16.,   9.,   4.,   1.,   0.,   1.,   8.,  27.,  64.])
```

Table 2-10. *NumPy Functions for Conditional and Logical Expressions*

Function	Description
<code>np.where</code>	Chooses values from two arrays depending on the value of a condition array.
<code>np.choose</code>	Chooses values from a list of arrays depending on the values of a given index array.
<code>np.select</code>	Chooses values from a list of arrays depending on a list of conditions.
<code>np.nonzero</code>	Returns an array with indices of nonzero elements.
<code>np.logical_and</code>	Performs an elementwise AND operation.
<code>np.logical_or</code> , <code>np.logical_xor</code>	Elementwise OR/XOR operations.
<code>np.logical_not</code>	Elementwise NOT operation (inverting).

The `np.select` function works similarly, but instead of a Boolean-valued condition array, it expects a list of Boolean-valued condition arrays and a corresponding list of value arrays:

```
In [202]: np.select([x < -1, x < 2, x >= 2],
...:                [x**2, x**3, x**4])
Out[202]: array([ 16.,   9.,   4.,  -1.,   0.,   1.,  16.,
                  81., 256.])
```

The `np.choose` takes as a first argument a list or an array with indices that determine from which array in a given list of arrays an element is picked from:

```
In [203]: np.choose([0, 0, 0, 1, 1, 1, 2, 2, 2],
...:                [x**2, x**3, x**4])
Out[203]: array([ 16.,   9.,   4.,  -1.,   0.,   1.,  16.,
                  81., 256.])
```

The function `np.nonzero` returns a tuple of indices that can be used to index the array (e.g., the one that the condition was based on). This has the same results as indexing the array directly with `abs(x) > 2`, but it uses fancy indexing with the indices returned by `np.nonzero` rather than Boolean-valued array indexing.

```
In [204]: np.nonzero(abs(x) > 2)
Out[204]: (array([0, 1, 7, 8]),)
In [205]: x[np.nonzero(abs(x) > 2)]
Out[205]: array([-4., -3.,  3.,  4.])
In [206]: x[abs(x) > 2]
Out[206]: array([-4., -3.,  3.,  4.])
```

Set Operations

The Python language provides a convenient *set* data structure for managing unordered collections of unique objects. The NumPy array class `ndarray` can also be used to describe such sets, and NumPy contains functions for operating on sets stored as NumPy arrays. These functions are summarized in Table 2-11. Using NumPy arrays to describe and operate on sets allows expressing certain operations in vectorized form. For example, testing if the values in a NumPy array are included in a set can be done using the `np.in1d` function, which tests for the existence of each element of its first

argument in the array passed as the second argument. To see how this works, consider the following example: first, to ensure that a NumPy array is a proper set, we can use the `np.unique` function, which returns a new array with unique values:

```
In [207]: a = np.unique([1, 2, 3, 3])
In [208]: b = np.unique([2, 3, 4, 4, 5, 6, 5])
In [209]: np.in1d(a, b)
Out[209]: array([False,  True,  True], dtype=bool)
```

Table 2-11. NumPy Functions for Operating on Sets

Function	Description
<code>np.unique</code>	Creates a new array with unique elements, where each value only appears once.
<code>np.in1d</code>	Tests for the existence of an array of elements in another array.
<code>np.intersect1d</code>	Returns an array with elements that are contained in two given arrays.
<code>np.setdiff1d</code>	Returns an array with elements that are contained in one, but not the other, of two given arrays.
<code>np.union1d</code>	Returns an array with elements that are contained in either, or both, of two given arrays.

Here, the existence of each element in `a` in the set `b` was tested, and the result is a Boolean-valued array. Note that we can use the `in` keyword to test for the existence of single elements in a set represented as NumPy array:

```
In [210]: 1 in a
Out[210]: True
In [211]: 1 in b
Out[211]: False
```

To test if `a` is a subset of `b`, we can use the `np.in1d`, as in the previous example, together with the aggregation function `np.all` (or the corresponding ndarray method):

```
In [212]: np.all(np.in1d(a, b))
Out[212]: False
```

The standard set operations union (the set of elements included in either or both sets), intersection (elements included in both sets), and difference (elements included in one of the sets but not the other) are provided by `np.union1d`, `np.intersect1d`, and `np.setdiff1d`, respectively:

```
In [213]: np.union1d(a, b)
Out[213]: array([1, 2, 3, 4, 5, 6])
In [214]: np.intersect1d(a, b)
Out[214]: array([2, 3])
In [215]: np.setdiff1d(a, b)
Out[215]: array([1])
In [216]: np.setdiff1d(b, a)
Out[216]: array([4, 5, 6])
```

Operations on Arrays

In addition to elementwise and aggregation functions, some operations act on arrays as a whole and produce a transformed array of the same size. An example of this type of operation is the transpose, which flips the order of the axes of an array. For the special case of a two-dimensional array, i.e., a matrix, the transpose simply exchanges rows and columns:

```
In [217]: data = np.arange(9).reshape(3, 3)
In [218]: data
Out[218]: array([[0, 1, 2],
                  [3, 4, 5],
                  [6, 7, 8]])
In [219]: np.transpose(data)
Out[219]: array([[0, 3, 6],
                  [1, 4, 7],
                  [2, 5, 8]])
```

The transpose function `np.transpose` also exists as a method in `ndarray` and as the special method name `ndarray.T`. For an arbitrary N-dimensional array, the transpose operation reverses all the axes, as can be seen from the following example (note that the `shape` attribute is used here to display the number of values along each axis of the array) :

```
In [220]: data = np.random.randn(1, 2, 3, 4, 5)
In [221]: data.shape
Out[221]: (1, 2, 3, 4, 5)
In [222]: data.T.shape
Out[222]: (5, 4, 3, 2, 1)
```

The `np.fliplr` (flip left-right) and `np.flipud` (flip up-down) functions perform operations that are similar to the transpose: they reshuffle the elements of an array so that the elements in rows (`np.fliplr`) or columns (`np.flipud`) are reversed, and the shape of the output array is the same as the input. The `np.rot90` function rotates the elements in the first two axes in an array by 90 degrees, and like the transpose function, it can change the shape of the array. Table 2-12 gives a summary of NumPy functions for common array operations.

Table 2-12. *Summary of NumPy Functions for Array Operations*

Function	Description
<code>np.transpose</code> , <code>np.ndarray.transpose</code> , <code>np.ndarray.T</code>	The transpose (reverse axes) of an array.
<code>np.fliplr</code> / <code>np.flipud</code>	Reverse the elements in each row/column.
<code>np.rot90</code>	Rotates the elements along the first two axes by 90 degrees.
<code>np.sort</code> , <code>np.ndarray.sort</code>	Sort the elements of an array along a given specified axis (which default to the last axis of the array). The <code>np.ndarray</code> method <code>sort</code> performs the sorting in place, modifying the input array.

Matrix and Vector Operations

We have so far discussed general N-dimensional arrays. One of the main applications of such arrays is to represent the mathematical concepts of vectors, matrices, and tensors, and in this use-case, we also frequently need to calculate vector and matrix operations

such as scalar (inner) products, dot (matrix) products, and tensor (outer) products. A summary of NumPy's functions for matrix operations is given in Table 2-13.

Table 2-13. *Summary of NumPy Functions for Matrix Operations*

NumPy Function	Description
<code>np.dot</code>	Matrix multiplication (dot product) between two given arrays representing vectors, arrays, or tensors.
<code>np.inner</code>	Scalar multiplication (inner product) between two arrays representing vectors.
<code>np.cross</code>	The cross product between two arrays that represent vectors.
<code>np.tensordot</code>	Dot product along specified axes of multidimensional arrays.
<code>np.outer</code>	Outer product (tensor product of vectors) between two arrays representing vectors.
<code>np.kron</code>	Kronecker product (tensor product of matrices) between arrays representing matrices and higher-dimensional arrays.
<code>np.einsum</code>	Evaluates Einstein's summation convention for multidimensional arrays.

In NumPy, the `*` operator is used for elementwise multiplication. For two two-dimensional arrays *A* and *B*, the expression *A* * *B* therefore does not compute a matrix product (in contrast to many other computing environments). Currently there is no operator for denoting matrix multiplication,⁴ and instead the NumPy function `np.dot` is used for this purpose. There is also a corresponding method in the `ndarray` class. To compute the product of two matrices *A* and *B*, of size $N \times M$ and $M \times P$, which results in a matrix of size $N \times P$, we can use:

```
In [223]: A = np.arange(1, 7).reshape(2, 3)
In [224]: A
Out[224]: array([[1, 2, 3],
                  [4, 5, 6]])
In [225]: B = np.arange(1, 7).reshape(3, 2)
In [226]: B
```

⁴Python recently adopted the `@` symbol for denoting matrix multiplication, and as of Python 3.5, this operator is now available. However, at the time of writing, this operator is still not widely used. See <http://legacy.python.org/dev/peps/pep-0465> for details.

```

Out[226]: array([[1, 2],
                [3, 4],
                [5, 6]])
In [227]: np.dot(A, B)
Out[227]: array([[22, 28],
                [49, 64]])
In [228]: np.dot(B, A)
Out[228]: array([[ 9, 12, 15],
                [19, 26, 33],
                [29, 40, 51]])

```

The `np.dot` function can also be used for matrix-vector multiplication (i.e., multiplication of a two-dimensional array, which represents a matrix, with a one-dimensional array representing a vector) . For example,

```

In [229]: A = np.arange(9).reshape(3, 3)
In [230]: A
Out[230]: array([[0, 1, 2],
                [3, 4, 5],
                [6, 7, 8]])
In [231]: x = np.arange(3)
In [232]: x
Out[232]: array([0, 1, 2])
In [233]: np.dot(A, x)
Out[233]: array([5, 14, 23])

```

In this example, `x` can be either a two-dimensional array of shape `(1, 3)` or a one-dimensional array with shape `(3,)`. In addition to the function `np.dot`, there is also a corresponding method `dot` in `ndarray`, which can be used as in the following example:

```

In [234]: A.dot(x)
Out[234]: array([5, 14, 23])

```


Unfortunately, nontrivial matrix multiplication expressions can often become complex and hard to read when using either `np.dot` or `np.ndarray.dot`. For example, even a relatively simple matrix expression like the one for a similarity transform, $A' = BAB^{-1}$, must be represented with relatively cryptic nested expressions,⁵ such as either

```
In [235]: A = np.random.rand(3,3)
In [236]: B = np.random.rand(3,3)
In [237]: Ap = np.dot(B, np.dot(A, np.linalg.inv(B)))
```

or

```
In [238]: Ap = B.dot(A.dot(np.linalg.inv(B)))
```

To improve this situation, NumPy provides an alternative data structure to `ndarray` named `matrix`, for which expressions like `A * B` are implemented as matrix multiplication. It also provides some convenient special attributes, like `matrix.I` for the inverse matrix and `matrix.H` for the complex conjugate transpose of a matrix. Using instances of this `matrix` class, one can therefore use the vastly more readable expression:

```
In [239]: A = np.matrix(A)
In [240]: B = np.matrix(B)
In [241]: Ap = B * A * B.I
```

This may seem like a practical compromise, but unfortunately using the `matrix` class does have a few disadvantages, and its use is therefore often discouraged. The main objection against using `matrix` is that expression like `A * B` is then context dependent: that is, it is not immediately clear if `A * B` denotes elementwise or matrix multiplication, because it depends on the type of `A` and `B`, and this creates another code-readability problem. This can be a particularly relevant issue if `A` and `B` are user-supplied arguments to a function, in which case it would be necessary to cast all input arrays explicitly to `matrix` instances, using, for example, `np.asmatrix` or the function `np.matrix` (since there would be no guarantee that the user calls the function with arguments of type `matrix` rather than `ndarray`). The `np.asmatrix` function creates a view of the original array in the form of an `np.matrix` instance. This does not add much in computational costs, but explicitly casting arrays back and forth between `ndarray` and `matrix` does

⁵With the new infix matrix multiplication operator, this same expression can be expressed as the considerably more readable: `Ap = B @ A @ np.linalg.inv(B)`.

offset much of the benefits of the improved readability of matrix expressions. A related issue is that some functions that operate on arrays and matrices might not respect the type of the input and may return an `ndarray` even though it was called with an input argument of type `matrix`. This way, a matrix of type `matrix` might be unintentionally converted to `ndarray`, which in turn would change the behavior of expressions like `A * B`. This type of behavior is not likely to occur when using NumPy's array and matrix functions, but it is not unlikely to happen when using functions from other packages. However, in spite of all the arguments for not using `matrix` matrices too extensively, personally I think that using `matrix` class instances for complicated matrix expressions is an important use-case, and in these cases, it might be a good idea to explicitly cast arrays to matrices before the computation and explicitly cast the result back to the `ndarray` type, following the pattern:

```
In [242]: A = np.asmatrix(A)
In [243]: B = np.asmatrix(B)
In [244]: Ap = B * A * B.I
In [245]: Ap = np.asarray(Ap)
```

The inner product (scalar product) between two arrays representing vectors can be computed using the `np.inner` function:

```
In [246]: np.inner(x, x)
Out[246]: 5
```

or, equivalently, using `np.dot`:

```
In [247]: np.dot(x, x)
Out[247]: 5
```

The main difference is that `np.inner` expects two input arguments with the same dimension, while `np.dot` can take input vectors of shape $1 \times N$ and $N \times 1$, respectively:

```
In [248]: y = x[:, np.newaxis]
In [249]: y
Out[249]: array([[0],
                 [1],
                 [2]])
In [250]: np.dot(y.T, y)
Out[250]: array([[5]])
```

While the inner product maps two vectors to a scalar, the outer product performs the complementary operation of mapping two vectors to a matrix.

```
In [251]: x = np.array([1, 2, 3])
```

```
In [252]: np.outer(x, x)
```

```
Out[252]: array([[1, 2, 3],
                 [2, 4, 6],
                 [3, 6, 9]])
```

The outer product can also be calculated using the Kronecker product using the function `np.kron`, which, however, in contrast to `np.outer`, produces an output array of shape $(M \times P, N \times Q)$ if the input arrays have shapes (M, N) and (P, Q) , respectively. Thus, for the case of two one-dimensional arrays of length M and P , the resulting array has shape $(M \times P,)$:

```
In [253]: np.kron(x, x)
```

```
Out[253]: array([1, 2, 3, 2, 4, 6, 3, 6, 9])
```

To obtain the result that corresponds to `np.outer(x, x)`, the input array `x` must be expanded to shape $(N, 1)$ and $(1, N)$, in the first and second argument to `np.kron`, respectively:

```
In [254]: np.kron(x[:, np.newaxis], x[np.newaxis, :])
```

```
Out[254]: array([[1, 2, 3],
                 [2, 4, 6],
                 [3, 6, 9]])
```

In general, while the `np.outer` function is primarily intended for vectors as input, the `np.kron` function can be used for computing tensor products of arrays of arbitrary dimension (but both inputs must have the same number of axes). For example, to compute the tensor product of two 2×2 matrices, we can use:

```
In [255]: np.kron(np.ones((2,2)), np.identity(2))
```

```
Out[255]: array([[ 1.,  0.,  1.,  0.],
                 [ 0.,  1.,  0.,  1.],
                 [ 1.,  0.,  1.,  0.],
                 [ 0.,  1.,  0.,  1.]])
```

```
In [256]: np.kron(np.identity(2), np.ones((2,2)))
Out[256]: array([[ 1.,  1.,  0.,  0.],
                  [ 1.,  1.,  0.,  0.],
                  [ 0.,  0.,  1.,  1.],
                  [ 0.,  0.,  1.,  1.]])
```

When working with multidimensional arrays, it is often possible to express common array operations concisely using Einstein's summation convention, in which an implicit summation is assumed over each index that occurs multiple times in an expression. For example, the scalar product between two vectors x and y is compactly expressed as $x_n y_n$, and the matrix multiplication of two matrices A and B is expressed as $A_{mk} B_{kn}$. NumPy provides the function `np.einsum` for carrying out Einstein summations. Its first argument is an index expression, followed by an arbitrary number of arrays that are included in the expression. The index expression is a string with comma-separated indices, where each comma separates the indices of each array. Each array can have any number of indices. For example, the scalar product expression $x_n y_n$ can be evaluated with `np.einsum` using the index expression "n,n", that is using `np.einsum("n,n", x, y)`:

```
In [257]: x = np.array([1, 2, 3, 4])
In [258]: y = np.array([5, 6, 7, 8])
In [259]: np.einsum("n,n", x, y)
Out[259]: 70
In [260]: np.inner(x, y)
Out[260]: 70
```

Similarly, the matrix multiplication $A_{mk} B_{kn}$ can be evaluated using `np.einsum` and the index expression "mk,kn":

```
In [261]: A = np.arange(9).reshape(3, 3)
In [262]: B = A.T
In [263]: np.einsum("mk,kn", A, B)
Out[263]: array([[ 5, 14, 23],
                  [14, 50, 86],
                  [23, 86, 149]])
In [264]: np.alltrue(np.einsum("mk,kn", A, B) == np.dot(A, B))
Out[264]: True
```

The Einstein summation convention can be particularly convenient when dealing with multidimensional arrays, since the index expression that defines the operation makes it explicit which operation is carried out and along which axes it is performed. An equivalent computation using, for example, `np.tensordot` might require giving the axes along which the dot product is to be evaluated.

Summary

In this chapter we have given a brief introduction to array-based programming with the NumPy library that can serve as a reference for the following chapters in this book. NumPy is a core library for computing with Python that provides a foundation for nearly all computational libraries for Python. Familiarity with the NumPy library and its usage patterns is a fundamental skill for using Python for scientific and technical computing. Here we started with introducing NumPy's data structure for N-dimensional arrays – the `ndarray` object – and we continued by discussing functions for creating and manipulating arrays, including indexing and slicing for extracting elements from arrays. We also discussed functions and operators for performing computations with `ndarray` objects, with an emphasis on vectorized expressions and operators for efficient computation with arrays. Throughout the rest of this book, we will see examples of higher-level libraries for specific fields in scientific computing that use the array framework provided by NumPy.

Further Reading

The NumPy library is the topic of several books, including the *Guide to NumPy*, by the creator of the NumPy T. Oliphant, available for free online at <http://web.mit.edu/dvp/Public/numpybook.pdf>, and a series of books by Ivan Idris: *Numpy Beginner's Guide* (2015), *NumPy Cookbook* (2012), and *Learning NumPy Array* (2014). NumPy is also covered in fair detail in McKinney (2013).

CHAPTER 4

Plotting and Visualization

Visualization is a universal tool for investigating and communicating results of computational studies, and it is hardly an exaggeration to say that the end product of nearly all computations – be it numeric or symbolic – is a plot or a graph of some sort. It is when visualized in graphical form that knowledge and insights can be most easily gained from computational results. Visualization is therefore a tremendously important part of the workflow in all fields of computational studies.

In the scientific computing environment for Python, there are a number of high-quality visualization libraries. The most popular general-purpose visualization library is Matplotlib, which mainly focuses on generating static publication-quality 2D and 3D graphs. Many other libraries focus on niche areas of visualization. A few prominent examples are Bokeh (<http://bokeh.pydata.org>) and Plotly (<http://plot.ly>), which both primarily focus on interactivity and web connectivity, Seaborn (<http://stanford.edu/~mwaskom/software/seaborn>) which is a high-level plotting library which targets statistical data analysis and which is based on the Matplotlib library, and the Mayavi library (<http://docs.enthought.com/mayavi/mayavi>) for high-quality 3D visualization, which uses the venerable VTK software (<http://www.vtk.org>) for heavy-duty scientific visualization. It is also worth noting that other VTK-based visualization software, such as ParaView (www.paraview.org), is scriptable with Python and can also be controlled from Python applications. In the 3D visualization space, there are also more recent players, such as VisPy (<http://vispy.org>), which is an OpenGL-based 2D and 3D visualization library with great interactivity and connectivity with browser-based environments, such as the Jupyter Notebook.

The visualization landscape in the scientific computing environment for Python is vibrant and diverse, and it provides ample options for various visualization needs. In this chapter we focus on exploring traditional scientific visualization in Python using the Matplotlib library. With traditional visualization, I mean plots and figures that are commonly used to visualize results and data in scientific and technical disciplines, such as line plots, bar plots, contour plots, colormap plots, and 3D surface plots.

Matplotlib Matplotlib is a Python library for publication-quality 2D and 3D graphics, with support for a variety of different output formats. At the time of writing, the latest version is 2.2.2. More information about Matplotlib is available at the project's web site www.matplotlib.org. This web site contains detailed documentation and an extensive gallery that showcases the various types of graphs that can be generated using the Matplotlib library, together with the code for each example. This gallery is a great source of inspiration for visualization ideas, and I highly recommend exploring Matplotlib by browsing this gallery.

There are two common approaches to creating scientific visualizations: using a graphical user interface to manually build up graphs and using a programmatic approach where the graphs are created with code. Both approaches have their advantages and disadvantages. In this chapter we will take the programmatic approach, and we will explore how to use the Matplotlib API to create graphs and control every aspect of their appearance. The programmatic approach is a particularly suitable method for creating graphics for scientific and technical applications and in particular for creating publication-quality figures. An important part of the motivation for this is that programmatically created graphics can guarantee consistency across multiple figures, can be made reproducible, and can easily be revised and adjusted without having to redo potentially lengthy and tedious procedures in a graphical user interface.

Importing Modules

Unlike most Python libraries, Matplotlib actually provides multiple entry points into the library, with different application programming interfaces (APIs). Specifically, it provides a stateful API and an object-oriented API, both provided by the module `matplotlib.pyplot`. I strongly recommend to only use the object-oriented approach, and the remainder of this chapter will solely focus on this part of Matplotlib.¹

¹Although the stateful API may be convenient and simple for small examples, the readability and maintainability of code written for stateful APIs scale poorly, and the context-dependent nature of such code makes it hard to rearrange or reuse. I therefore recommend to avoid it altogether and to only use the object-oriented API.

To use the object-oriented Matplotlib API, we first need to import its Python modules. In the following we will assume that Matplotlib is imported using the following standard convention:

```
In [1]: %matplotlib inline
In [2]: import matplotlib as mpl
In [3]: import matplotlib.pyplot as plt
In [4]: from mpl_toolkits.mplot3d.axes3d import Axes3D
```

The first line is assuming that we are working in an IPython environment and more specifically in the Jupyter Notebook or the IPython QtConsole. The IPython magic command `%matplotlib inline` configures the Matplotlib to use the “inline” backend, which results in the created figures being displayed directly in, for example, the Jupyter Notebook, rather than in a new window. The statement `import matplotlib as mpl` imports the main Matplotlib module, and the import statement `import matplotlib.pyplot as plt`, is for convenient access to the submodule `matplotlib.pyplot` that provides the functions that we will use to create new Figure instances.

Throughout this chapter we also make frequent use of the NumPy library, and as in Chapter 2, we assume that NumPy is imported using

```
In [5]: import numpy as np
```

and we also use the SymPy library, imported as:

```
In [6]: import sympy
```

Getting Started

Before we delve deeper into the details of how to create graphics with Matplotlib, we begin here with a quick example of how to create a simple but typical graph. We also cover some of the fundamental principles of the Matplotlib library, to build up an understanding for how graphics can be produced with the library.

A graph in Matplotlib is structured in terms of a Figure instance and one or more Axes instances within the figure. The Figure instance provides a canvas area for drawing, and the Axes instances provide coordinate systems that are assigned to fixed regions of the total figure canvas; see Figure 4-1.

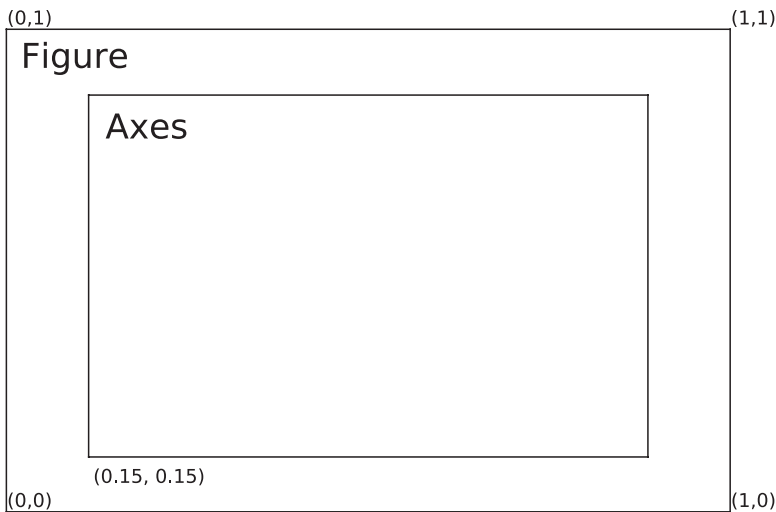


Figure 4-1. Illustration of the arrangement of a Matplotlib Figure instance and an Axes instance. The Axes instance provides a coordinate system for plotting, and the Axes instance itself is assigned to a region within the figure canvas. The figure canvas has a simple coordinate system where (0, 0) is the lower-left corner and (1,1) is the upper-right corner. This coordinate system is only used when placing elements, such as an Axes, directly on the figure canvas.

A Figure can contain multiple Axes instances, for example, to show multiple panels in a figure or to show insets within another Axes instance. An Axes instance can manually be assigned to an arbitrary region of a figure canvas, or, alternatively, Axes instances can be automatically added to a figure canvas using one of several layout managers provided by Matplotlib. The Axes instance provides a coordinate system that can be used to plot data in a variety of plot styles, including line graphs, scatter plots, bar plots, and many other styles. In addition, the Axes instance also determines how the coordinate axes are displayed, for example, with respect to the axis labels, ticks and tick labels, and so on. In fact, when working with Matplotlib’s object-oriented API, most functions that are needed to tune the appearance of a graph are methods of the Axes class.

As a simple example for getting started with Matplotlib, say that we would like to graph the function $y(x) = x^3 + 5x^2 + 10$, together with its first and second derivatives, over the range $x \in [-5, 2]$. To do this we first create NumPy arrays for the x range and then compute the three functions we want to graph. When the data for the graph is prepared, we need to create Matplotlib Figure and Axes instances, then use the plot method of the Axes instance to plot the data, and set basic graph properties such as x and y axis labels,

using the `set_xlabel` and `set_ylabel` methods and generating a legend using the `legend` method. These steps are carried out in the following code, and the resulting graph is shown in Figure 4-2.

```
In [7]: x = np.linspace(-5, 2, 100)
...: y1 = x**3 + 5*x**2 + 10
...: y2 = 3*x**2 + 10*x
...: y3 = 6*x + 10
...:
...: fig, ax = plt.subplots()
...: ax.plot(x, y1, color="blue", label="y(x)")
...: ax.plot(x, y2, color="red", label="y'(x)")
...: ax.plot(x, y3, color="green", label="y''(x)")
...: ax.set_xlabel("x")
...: ax.set_ylabel("y")
...: ax.legend()
```

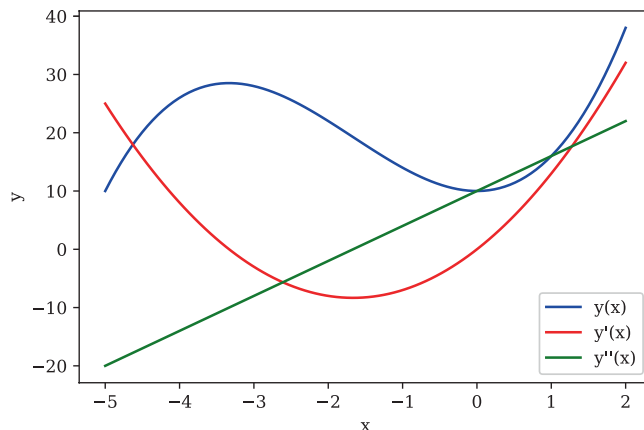


Figure 4-2. Example of a simple graph created with Matplotlib

Here we used the `plt.subplots` function to generate Figure and Axes instances. This function can be used to create grids of Axes instances within a newly created Figure instance, but here it was merely used as a convenient way of creating a Figure and an Axes instance in one function call. Once the Axes instance is available, note that all the remaining steps involve calling methods of this Axes instance. To create the actual graphs, we use `ax.plot`, which takes as first and second arguments NumPy arrays with

numerical data for the x and y values of the graph, and it draws a line connecting these data points. We also used the optional `color` and `label` keyword arguments to specify the color of each line and assign a text label to each line that is used in the legend. These few lines of code are enough to generate the graph we set out to produce, but as a bare minimum, we should also set labels on the x and y axes and if suitable add a legend for the curves we have plotted. The axis labels are set with `ax.set_xlabel` and `ax.set_ylabel` methods, which takes as argument a text string with the corresponding label. The legend is added using the `ax.legend` method, which does not require any arguments in this case since we used the `label` keyword argument when plotting the curves.

These are the typical steps required to create a graph using Matplotlib. While this graph, Figure 4-2, is complete and fully functional, there is certainly room for improvements in many aspects of its appearance. For example, to meet publication or production standards, we may need to change the font and the fontsize of the axis labels, the tick labels, and the legend, and we should probably move the legend to a part of the graph where it does not interfere with the curves we are plotting. We might even want to change the number of axis ticks and label and add annotations and additional help lines to emphasize certain aspects of the graph and so on. With a few changes along these lines, the figure may, for example, appear like in Figure 4-3, which is considerably more presentable. In the remainder of this chapter, we look at how to fully control the appearance of the graphics produced using Matplotlib.

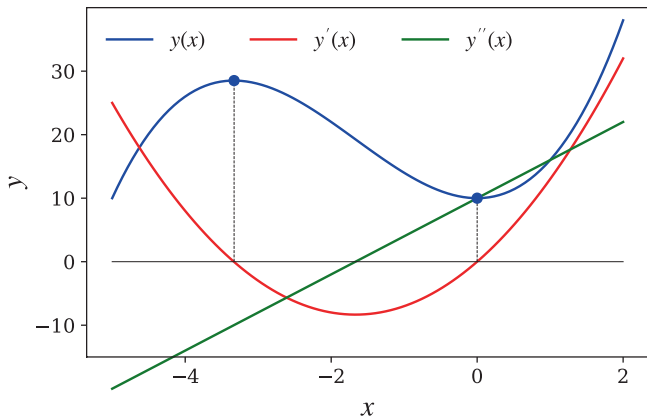


Figure 4-3. Revised version of Figure 4-2

Interactive and Noninteractive Modes

The Matplotlib library is designed to work well with many different environments and platforms. As such, the library does not only contain routines for generating graphs, but it also contains support for displaying graphs in different graphical environments. To this end, Matplotlib provides *backends* for generating graphics in different formats (e.g., PNG, PDF, Postscript, and SVG) and for displaying graphics in a graphical user interface using a variety of different widget toolkits (e.g., Qt, GTK, wxWidgets, and Cocoa for Mac OS X) that are suitable for different platforms.

Which backend to use can be selected in that Matplotlib resource file,² or using the function `mpl.use`, which must be called right after importing `matplotlib`, before importing the `matplotlib.pyplot` module. For example, to select the Qt4Agg backend, we can use

```
import matplotlib as mpl
mpl.use('qt4agg')
import matplotlib.pyplot as plt
```

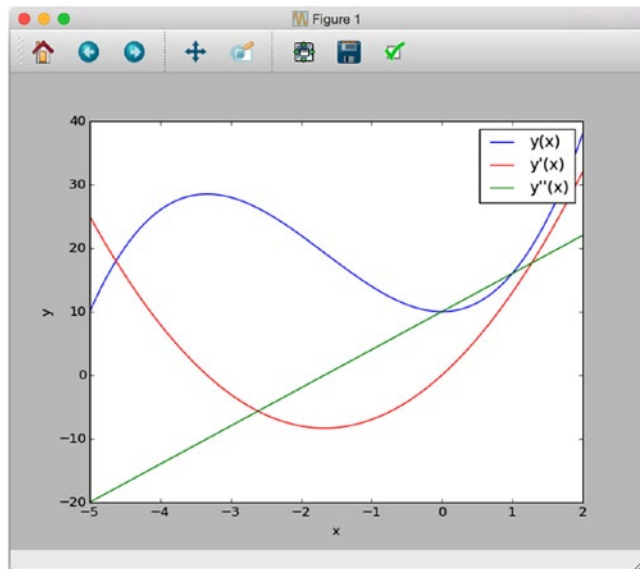


Figure 4-4. A screenshot of the Matplotlib graphical user interface for displaying figures, using the Qt4 backend on Mac OS X. The detailed appearance varies across platforms and backends, but the basic functionality is the same.

²The Matplotlib resource file, `matplotlibrc`, can be used to set default values of many Matplotlib parameters, including which backend to use. The location of the file is platform dependent. For details, see <http://matplotlib.org/users/customizing.html>.

The graphical user interface for displaying Matplotlib figures, as shown in Figure 4-4, is useful for interactive use with Python script files or the IPython console, and it allows to interactively explore figures, for example, by zooming and panning. When using an interactive backend, which displays the figure in a graphical user interface, it is necessary to call the function `plt.show` to get the window to appear on the screen. By default, the `plt.show` call will hang until the window is closed. For a more interactive experience, we can activate *interactive mode* by calling the function `plt.ion`. This instructs Matplotlib to take over the GUI event loop and show a window for a figure as soon as it is created, returning the control flow to the Python or IPython interpreter. To have the changes to a figure take effect, we need to issue a redraw command using the function `plt.draw`. We can deactivate the interactive mode using the function `plt.ioff`, and we can use the function `mpl.is_interactive` to check if Matplotlib is in interactive or noninteractive mode.

While the interactive graphical user interfaces have unique advantages, when working the Jupyter Notebook or Qtconsole, it is often more convenient to display Matplotlib-produced graphics embedded directly in the notebook. This behavior is activated using the IPython command `%matplotlib inline`, which activates the “inline backend” provided by IPython. This configures Matplotlib to use a noninteractive backend to generate graphics images, which are then displayed as static images in, for example, the Jupyter Notebook. The IPython “inline backend” for Matplotlib can be fine-tuned using the IPython `%config` command. For example, we can select the output format for the generated graphics using the `InlineBackend.figure_format` option,³ which, for example, we can set to ‘svg’ to generate SVG graphics rather than PNG files:

```
In [8]: %matplotlib inline
```

```
In [9]: %config InlineBackend.figure_format='svg'
```

With this approach the interactive aspect of the graphical user interface is lost (e.g., zooming and panning), but embedding the graphics directly in the notebook has many other advantages. For example, keeping the code that was used to generate a figure together with the resulting figure in the same document eliminates the need for rerunning the code to display a figure, and interactive nature of the Jupyter Notebook itself replaces some of the interactivity of Matplotlib’s graphical user interface.

³For Max OS X users, `%config InlineBackend.figure_format='retina'` is another useful option, which improves the quality of the Matplotlib graphics when viewed on retina displays.

When using the IPython inline backend, it is not necessary to use `plt.show` and `plt.draw`, since the IPython rich display system is responsible for triggering the rendering and the displaying of the figures. In this book, I will assume that code examples are executed in the Jupyter Notebooks, and the calls to the function `plt.show` are therefore not in the code examples. When using an interactive backend, it is necessary to add this function call at the end of each example.

Figure

As introduced in the previous section, the `Figure` object is used in Matplotlib to represent a graph. In addition to providing a canvas on which, for example, `Axes` instances can be placed, the `Figure` object also provides methods for performing actions on figures, and it has several attributes that can be used to configure the properties of a figure.

A `Figure` object can be created using the function `plt.figure`, which takes several optional keyword arguments for setting figure properties. In particular, it accepts the `figsize` keyword argument, which should be assigned to a tuple on the form `(width, height)`, specifying the width and height of the figure canvas in inches. It can also be useful to specify the color of the figure canvas by setting the `facecolor` keyword argument.

Once a `Figure` is created, we can use the `add_axes` method to create a new `Axes` instance and assign it to a region on the figure canvas. The `add_axes` takes one mandatory argument: a list containing the coordinates of the lower-left corner and the width and height of the `Axes` in the figure canvas coordinate system, on the format `(left, bottom, width, height)`.⁴ The coordinates and the width and height of the `Axes` object are expressed as fractions of total canvas width and height; see Figure 4-1. For example, an `Axes` object that completely fills the canvas corresponds to `(0, 0, 1, 1)`, but this leaves no space for axis labels and ticks. A more practical size could be `(0.1, 0.1, 0.8, 0.8)`, which corresponds to a centered `Axes` instance that covers 80% of the width and height of the canvas. The `add_axes` method takes a large number of keyword arguments for setting properties of the new `Axes` instance. These will be described in more detail later in this chapter, when we discuss the `Axes` object in depth. However,

⁴An alternative to passing a coordinate and size tuple to `add_axes` is to pass an already existing `Axes` instance.

one keyword argument that is worth to emphasize here is `facecolor`, with which we can assign a background color for the Axes object. Together with the `facecolor` argument of `plt.figure`, this allows selecting colors of both the canvas and the regions covered by Axes instances.

With the Figure and Axes objects obtained from `plt.figure` and `fig.add_axes`, we have the necessary preparations to start plotting data using the methods of the Axes objects. For more details on this, see the next section of this chapter. However, once the required plots have been created, there are more methods in the Figure objects that are important in the graph creation workflow. For example, to set an overall figure title, we can use `suptitle`, which takes a string with the title as argument. To save a figure to a file, we can use the `savefig` method. This method takes a string with the output filename as first argument, as well as several optional keyword arguments. By default, the output file format will be determined from the file extension of the filename argument, but we can also specify the format explicitly using the `format` argument. The available output formats depend on which Matplotlib backend is used, but commonly available options are PNG, PDF, EPS, and SVG formats. The resolution of the generated image can be set with the `dpi` argument. DPI stands for “dots per inch,” and since the figure size is specified in inches using the `figsize` argument, multiplying these numbers gives the output image size in pixels. For example, with `figsize=(8, 6)` and `dpi=100`, the size of the generated image is 800x600 pixels. The `savefig` method also takes some arguments that are similar to those of the `plt.figure` function, such as the `facecolor` argument. Note that even though the `facecolor` argument is used with `plt.figure`, it also needs to be specified with `savefig` for it to apply to the generated image file. Finally, the figure canvas can also be made transparent using the `transparent=True` argument to `savefig`. The following code listing illustrates these techniques, and the result is shown in Figure 4-5.

```
In [10]: fig = plt.figure(figsize=(8, 2.5), facecolor="#f1f1f1")
...:
...: # axes coordinates as fractions of the canvas width and height
...: left, bottom, width, height = 0.1, 0.1, 0.8, 0.8
...: ax = fig.add_axes((left, bottom, width, height),
...:                   facecolor="#e1e1e1")
...:
...: x = np.linspace(-2, 2, 1000)
...: y1 = np.cos(40 * x)
```

```

...: y2 = np.exp(-x**2)
...:
...: ax.plot(x, y1 * y2)
...: ax.plot(x, y2, 'g')
...: ax.plot(x, -y2, 'g')
...: ax.set_xlabel("x")
...: ax.set_ylabel("y")
...:
...: fig.savefig("graph.png", dpi=100, facecolor="#f1f1f1")

```

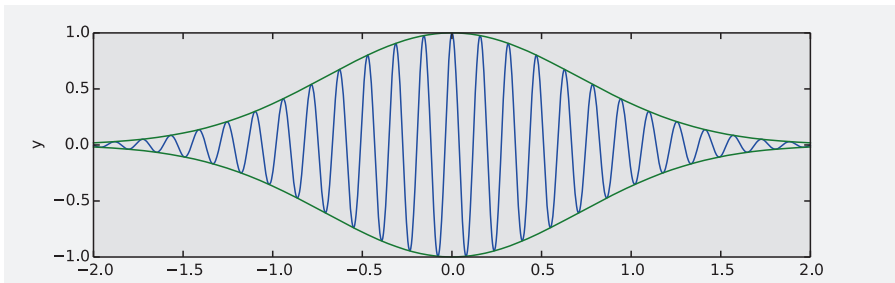


Figure 4-5. Graph showing the result of setting the size of a figure with `figsize`, adding a new `Axes` instance with `add_axes`, setting the background colors of the `Figure` and `Axes` objects using `facecolor`, and finally saving the figure to file using `savefig`

Axes

The `Figure` object introduced in the previous section provides the backbone of a Matplotlib graph, but all the interesting content is organized within or around `Axes` instances. We have already encountered `Axes` objects on a few occasions earlier in this chapter. The `Axes` object is central to most plotting activities with the Matplotlib library. It provides the coordinate system in which we can plot data and mathematical functions, and in addition it contains the axis objects that determine where the axis labels and the axis ticks are placed. The functions for drawing different types of plots are also methods of this `Axes` class. In this section we first explore different types of plots that can be drawn using `Axes` methods and how to customize the appearance of the x and y axes and the coordinate systems used with an `Axes` object.

We have seen how new `Axes` instances can be added to a figure explicitly using the `add_axes` method. This is a flexible and powerful method for placing `Axes` objects at arbitrary positions, which has several important applications, as we will see later in the chapter. However, for most common use-cases, it is tedious to specify explicitly the coordinates of the `Axes` instances within the figure canvas. This is especially true when using multiple panels of `Axes` instances within a figure, for example, in a grid layout. Matplotlib provides several different `Axes` layout managers, which create and place `Axes` instances within a figure canvas following different strategies. Later in this chapter, we look into more detail of how to use such layout managers. However, to facilitate the forthcoming examples, we here briefly look at one of these layout managers: the `plt.subplots` function. Earlier in this chapter, we already used this function to conveniently generate new `Figure` and `Axes` objects in one function call. However, the `plt.subplots` function is also capable of filling a figure with a grid of `Axes` instances, which is specified using the first and the second arguments, or alternatively with the `nrows` and `ncols` arguments, which, as the names imply, create a grid of `Axes` objects, with the given number of rows and columns. For example, to generate a grid of `Axes` instances in a newly created `Figure` object, with three rows and two columns, we can use

```
fig, axes = plt.subplots(nrows=3, ncols=2)
```

Here, the function `plt.subplots` returns a tuple (`fig`, `axes`), where `fig` is a `Figure` instance and `axes` is a NumPy array of size (`nrows`, `ncols`), in which each element is an `Axes` instance that has been appropriately placed in the corresponding figure canvas. At this point we can also specify that columns and/or rows should share x and y axes, using the `sharex` and `sharey` arguments, which can be set to `True` or `False`.

The `plt.subplots` function also takes two special keyword arguments `fig_kw` and `subplot_kw`, which are dictionaries with keyword arguments that are used when creating the `Figure` and `Axes` instances, respectively. This allows us to set and retain full control of the properties of the `Figure` and `Axes` objects with `plt.subplots` in a similar way as when directly using `plt.figure` and the `make_axes` method.

Plot Types

Effective scientific and technical visualization of data requires a wide variety of graphing techniques. Matplotlib implements many types of plotting techniques as methods of the `Axes` object. For example, in the previous examples, we have already used the `plot` method, which draws curves in the coordinate system provided by the `Axes` object.

In the following sections, we explore some of Matplotlib's plotting functions in more depth by using these functions in example graphs. A summary of commonly used 2D plot functions is shown in Figure 4-6. Other types of graphs, such as color maps and 3D graphs, are discussed later in this chapter. All plotting functions in Matplotlib expect data as NumPy arrays as input, typically as arrays with x and y coordinates as the first and second arguments. For details, see the docstrings for each method shown in Figure 4-6, using, for example, `help(plt.Axes.bar)`.

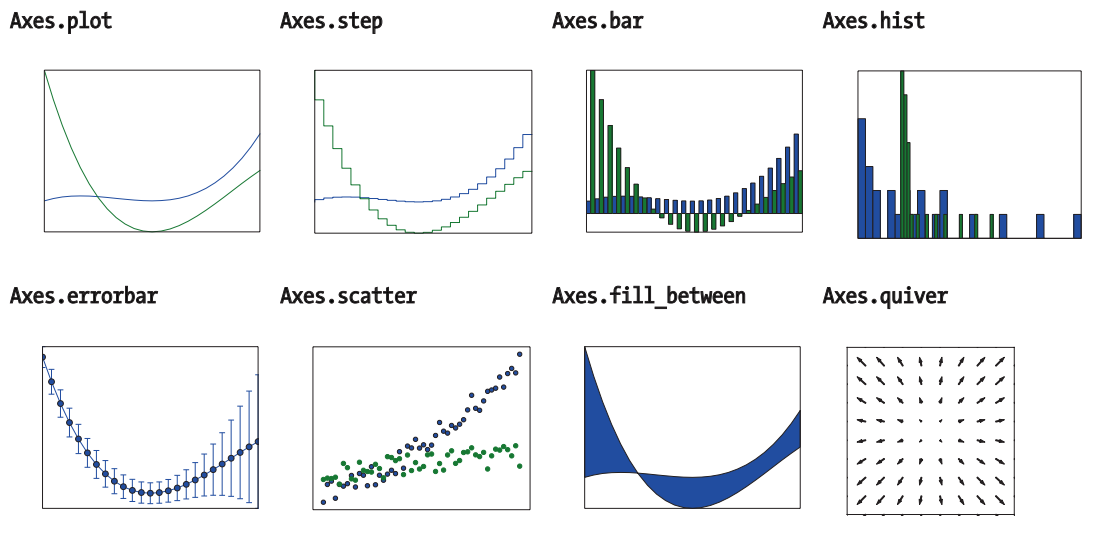


Figure 4-6. Overview of selected 2D graph types. The name of the Axes method for generating each type of graph is shown together with the corresponding graph.

Line Properties

The most basic type of plot is the simple line plot. It may, for example, be used to depict the graph of a univariate function or to plot data as a function of a control variable.

In line plots, we frequently need to configure properties of the lines in the graph, for example, the line width, line color, and line style (solid, dashed, dotted, etc.). In Matplotlib we set these properties with keyword arguments to the plot methods, such as `plot`, `step`, and `bar`. A few of these graph types are shown in Figure 4-6. Many of the plot methods have their own specific arguments, but basic properties such as colors and line width are shared among most plotting methods. These basic properties and the corresponding keyword arguments are summarized in Table 4-1.

Table 4-1. *Basic Line Properties and Their Corresponding Argument Names for Use with the Matplotlib Plotting Methods*

Argument	Example Values	Description
color	A color specification can be a string with a color name, such as “red,” “blue,” etc., or a RGB color code on the form “#aabbcc.”	A color specification.
alpha	Float number between 0.0 (completely transparent) and 1.0 (completely opaque).	The amount of transparency.
linewidth, lw	Float number.	The width of a line.
linestyle, ls	“-” – solid “--” – dashed “.” – dotted “-.” – dash-dotted	The style of the line, i.e., whether the line is to be drawn as a solid line or if it should be, for example, dotted or dashed.
marker	+ , o , * = cross, circle, star s = square . = small dot 1, 2, 3, 4, ... = triangle-shaped symbols with different angles.	Each data point, whether or not it is connected with adjacent data points, can be represented with a marker symbol as specified with this argument.
markersize	Float number.	The marker size.
markerfacecolor	Color specification (see in the preceding text).	The fill color for the marker.
markeredgewidth	Float number.	The line width of the marker edge.
markeredgecolor	Color specification (see above).	The marker edge color.

To illustrate the use of these properties and arguments, consider the following code, which draws horizontal lines with various values of the line width, line style, marker symbol, color, and size. The resulting graph is shown in Figure 4-7.

```

In [11]: x = np.linspace(-5, 5, 5)
...: y = np.ones_like(x)
...:
...: def axes_settings(fig, ax, title, ymax):
...:     ax.set_xticks([])
...:     ax.set_yticks([])
...:     ax.set_ylim(0, ymax+1)
...:     ax.set_title(title)
...:
...: fig, axes = plt.subplots(1, 4, figsize=(16,3))
...:
...: # Line width
...: linewidths = [0.5, 1.0, 2.0, 4.0]
...: for n, linewidth in enumerate(linewidths):
...:     axes[0].plot(x, y + n, color="blue", linewidth=linewidth)
...: axes_settings(fig, axes[0], "linewidth", len(linewidths))
...:
...: # Line style
...: linestyle = ['-', '-.', ':']
...: for n, linestyle in enumerate(linestyle):
...:     axes[1].plot(x, y + n, color="blue", lw=2, linestyle=linestyle)
...: # custom dash style
...: line, = axes[1].plot(x, y + 3, color="blue", lw=2)
...: length1, gap1, length2, gap2 = 10, 7, 20, 7
...: line.set_dashes([length1, gap1, length2, gap2])
...: axes_settings(fig, axes[1], "linetypes", len(linestyle) + 1)
...:
...: # marker types
...: markers = ['+', 'o', '*', 's', '.', '1', '2', '3', '4']
...: for n, marker in enumerate(markers):
...:     # lw = shorthand for linewidth, ls = shorthand for linestyle
...:     axes[2].plot(x, y + n, color="blue", lw=2, ls='*',
...:                 marker=marker)
...: axes_settings(fig, axes[2], "markers", len(markers))
...:

```

```

...: # marker size and color
...: markersizecolors = [(4, "white"), (8, "red"), (12, "yellow"),
...:                     (16, "lightgreen")]
...: for n, (markersize, markerfacecolor) in enumerate
...:     (markersizecolors):
...:     axes[3].plot(x, y + n, color="blue", lw=1, ls='-',
...:                 marker='o', markersize=markersize,
...:                 markerfacecolor=markerfacecolor,
...:                 markeredgewidth=2)
...: axes_settings(fig, axes[3], "marker size/color", len
...:                 (markersizecolors))

```

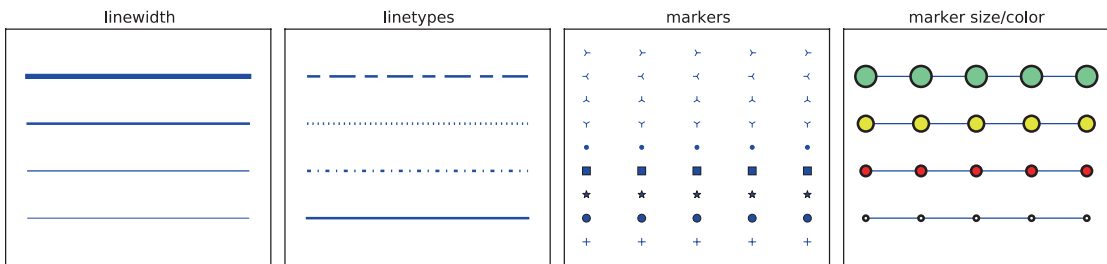


Figure 4-7. *Graphs showing the result of setting the line properties line width, line style, marker type and marker size, and color*

In practice, using different colors, line widths and line styles are important tools for making a graph easily readable. In a graph with a large number of lines, we can use a combination of colors and line style to make each line uniquely identifiable, for example, via a legend. The line width property is best used to give emphasis to important lines. Consider the following example, where the function $\sin(x)$ is plotted together with its first few series expansions around $x = 0$, as shown in Figure 4-8.

```

In [12]: # a symbolic variable for x, and a numerical array with specific
...:      values of x
...: sym_x = sympy.Symbol("x")
...: x = np.linspace(-2 * np.pi, 2 * np.pi, 100)
...:
...: def sin_expansion(x, n):
...:     """
...:     Evaluate the nth order Taylor. series expansion

```

```

...:     of sin(x) for the numerical values in the array x.
...:     """
...:     return sympy.lambdify(sym_x, sympy.sin(sym_x).series(n=n+1).
...:         remove0(), 'numpy')(x)
...:
...: fig, ax = plt.subplots()
...:
...: ax.plot(x, np.sin(x), linewidth=4, color="red", label='exact')
...:
...: colors = ["blue", "black"]
...: linestyles = [':', '-.', '--']
...: for idx, n in enumerate(range(1, 12, 2)):
...:     ax.plot(x, sin_expansion(x, n), color=colors[idx // 3],
...:         linestyle=linestyles[idx % 3], linewidth=3,
...:         label="order %d approx." % (n+1))
...:
...: ax.set_ylim(-1.1, 1.1)
...: ax.set_xlim(-1.5*np.pi, 1.5*np.pi)
...:
...: # place a legend outside of the Axes
...: ax.legend(bbox_to_anchor=(1.02, 1), loc=2, borderaxespad=0.0)
...: # make room for the legend to the right of the Axes
...: fig.subplots_adjust(right=.75)

```

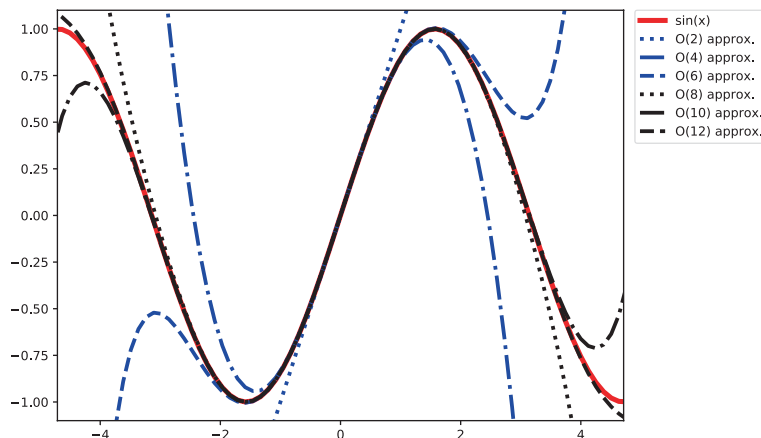


Figure 4-8. Graph for $\sin(x)$ together with its Taylor series approximation of the few lowest orders

Legends

A graph with multiple lines may often benefit from a legend, which displays a label along each line type somewhere within the figure. As we have seen in the previous example, a legend may be added to an Axes instance in a Matplotlib figure using the `legend` method. Only lines with assigned labels are included in the legend (to assign a label to a line, use the `label` argument of, for example, `Axes.plot`). The `legend` method accepts a large number of optional arguments. See `help(plt.legend)` for details. Here we emphasize a few of the more useful arguments. In the example in the previous section, we used the `loc` argument, which allows to specify where in the Axes area the legend is to be added: `loc=1` for upper-right corner, `loc=2` for upper-left corner, `loc=3` for the lower-left corner, and `loc=4` for lower-right corner, as shown in Figure 4-9.

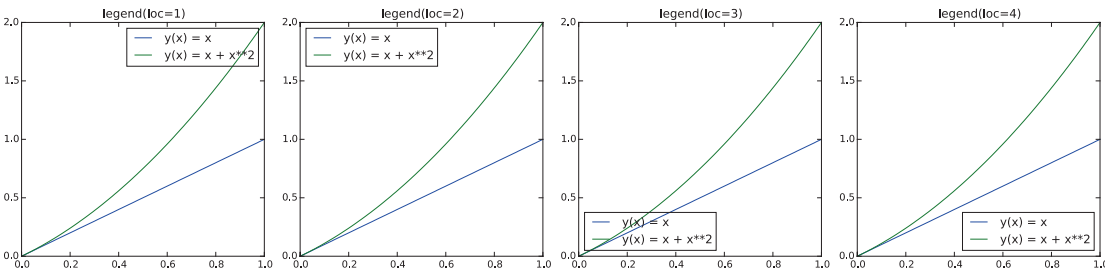


Figure 4-9. Legend at different positions within an Axes instance, specified using the `loc` argument of the method `legend`

In the example of the previous section, we also used the `bbox_to_anchor`, with which help the legend can be placed at an arbitrary location within the figure canvas. The `bbox_to_anchor` argument takes the value of a tuple on the form (x, y) , where x and y are the *canvas coordinates* within the Axes object. That is, the point $(0, 0)$ corresponds to the lower-left corner, and $(1, 1)$ corresponds to the upper-right corner. Note that x and y can be smaller than 0 and larger than 1 in this case, which indicates that the legend is to be placed outside the Axes area, as was used in the previous section.

By default all lines in the legend are shown in a vertical arrangement. Using the `ncols` argument, it is possible to split the legend labels into multiple columns, as illustrated in Figure 4-10.

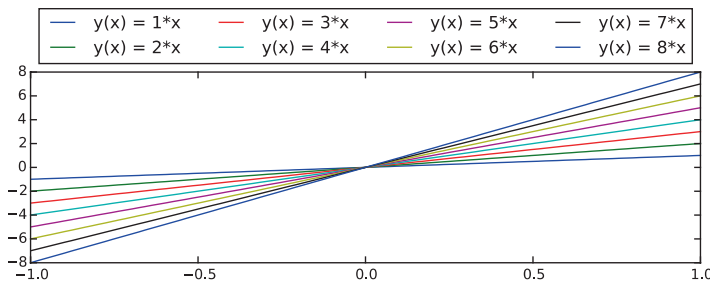


Figure 4-10. Legend displayed outside the Axes object and shown with four columns instead of the single one, here using `ax.legend(ncol=4, loc=3, bbox_to_anchor=(0, 1))`

Text Formatting and Annotations

Text labels, titles, and annotations are important components in most graphs, and having full control of, for example, the font types and font sizes that are used to render such texts is a basic requirement for producing publication-quality graphs. Matplotlib provides several ways of configuring font properties. The default values can be set in the Matplotlib resource file, and session-wide configuration can be set in the `mpl.rcParams` dictionary. This dictionary is a cache of the Matplotlib resource file, and changes to parameters within this dictionary are valid until the Python interpreter is restarted and Matplotlib is imported again. Parameters that are relevant to how text is displayed include, for example, `'font.family'` and `'font.size'`.

Tip Try `print(mpl.rcParams)` to get a list of possible configuration parameters and their current values. Updating a parameter is as simple as assigning a new value to the corresponding item in the dictionary `mpl.rcParams`, for example, `mpl.rcParams['savefig.dpi'] = 100`. See also the `mpl.rc` function, which can be used to update the `mpl.rcParams` dictionary, and `mpl.rcParams.defaults` for restoring the default values.

It is also possible to set text properties on a case-to-case basis, by passing a set of standard keyword arguments to functions that create text labels in a graph. Most Matplotlib functions that deal with text labels, in one way or another, accept the keyword arguments summarized in Table 4-2 (this list is an incomplete selection of common arguments; see `help(mpl.text.Text)` for a complete reference). For example, these

arguments can be used with the method `Axes.text`, which create a new text label at a given coordinate. They may also be used with `set_title`, `set_xlabel`, `set_ylabel`, etc. For more information on these methods, see the next section.

In scientific and technical visualization, it is clearly important to be able to render mathematical symbols and expressions in text labels. Matplotlib provides excellent support for this through LaTeX markup within its text labels: any text label in Matplotlib can include LaTeX math by enclosing it within `$` signs, for example, "Regular text: $f(x)=1-x^2$ ". By default, Matplotlib uses an internal LaTeX rendering, which supports a subset of LaTeX language. However, by setting the configuration parameter `mpl.rcParams["text.usetex"]=True`, it is also possible to use an external full-featured LaTeX engine (if it is available on your system).

When embedding LaTeX code in strings in Python, there is a common stumbling block: Python uses `\` as escape character, while in LaTeX it is used to denote the start of commands. To prevent the Python interpreter from escaping characters in strings containing LaTeX expressions, it is convenient to use raw strings, which are literal string expressions that are prepended with and an `r`, for example, `r"$\int f(x) dx$"` and `r'$x_{\rm A}$'`.

The following example demonstrates how to add text labels and annotations to a Matplotlib figure using `ax.text` and `ax.annotate`, as well as how to render a text label that includes an equation that is typeset in LaTeX. The resulting graph is shown in Figure 4-11.

```
In [13]: fig, ax = plt.subplots(figsize=(12, 3))
...:
...: ax.set_yticks([])
...: ax.set_xticks([])
...: ax.set_xlim(-0.5, 3.5)
...: ax.set_ylim(-0.05, 0.25)
...: ax.axhline(0)
...:
...: # text label
...: ax.text(0, 0.1, "Text label", fontsize=14, family="serif")
...:
...: # annotation
...: ax.plot(1, 0, "o")
...: ax.annotate("Annotation",
```

```

...:         fontsize=14, family="serif",
...:         xy=(1, 0), xycoords="data",
...:         xytext=(+20, +50), textcoords="offset points",
...:         arrowprops=dict(arrowstyle="->", connectionstyle="arc3,
...:                         rad=.5"))
...:
...: # equation
...: ax.text(2, 0.1, r"Equation:  $\hbar \partial_t \Psi = \hat{H} \Psi$ ",
...:         fontsize=14, family="serif")
...:

```

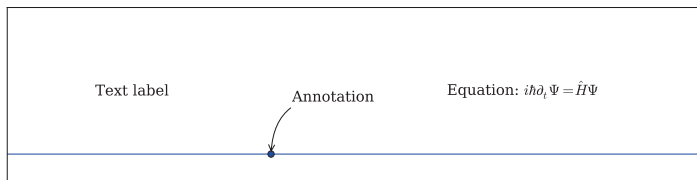


Figure 4-11. Example demonstrating the result of adding text labels and annotations using `ax.text` and `ax.annotate` and including LaTeX formatted equations in a Matplotlib text label

Table 4-2. Summary of Selected Font Properties and the Corresponding Keyword Arguments

Argument	Description
<code>fontsize</code>	The size of the font, in points.
<code>family</code> or <code>fontname</code>	The font type.
<code>backgroundcolor</code>	Color specification for the background color of the text label.
<code>color</code>	Color specification for the font color.
<code>alpha</code>	Transparency of the font color.
<code>rotation</code>	Rotation angle of the text label.

Axis Properties

After having created Figure and Axes objects, the data or functions are plotted using some of the many plot functions provided by Matplotlib, and the appearance of lines and markers are customized – the last major aspect of a graph that remains to be configured and fine-tuned is the Axis instances. A two-dimensional graph has two axis objects: for the horizontal x axis and the vertical y axis. Each axis can be individually configured with respect to attributes such as the axis labels, the placement of ticks and the tick labels, and the location and appearance of the axis itself. In this section we look into the details of how to control these aspects of a graph.

Axis Labels and Titles

Arguably the most important property of an axis, which needs to be set in nearly all cases, is the axis label. We can set the axis labels using the `set_xlabel` and `set_ylabel` methods: they both take a string with the label as first arguments. In addition, the optional `labelpad` argument specifies the spacing, in units of points, from the axis to the label. This padding is occasionally necessary to avoid overlap between the axis label and the axis tick labels. The `set_xlabel` and `set_ylabel` methods also take additional arguments for setting text properties, such as color, fontsize, and fontname, as discussed in detail in the previous section. The following code, which produces Figure 4-12, demonstrates how to use the `set_xlabel` and `set_ylabel` methods and the keyword arguments discussed here.

```
In [14]: x = np.linspace(0, 50, 500)
...: y = np.sin(x) * np.exp(-x/10)
...:
...: fig, ax = plt.subplots(figsize=(8, 2), subplot_kw={'facecolor':
...:         "#ebf5ff"})
...:
...: ax.plot(x, y, lw=2)
...:
...: ax.set_xlabel("x", labelpad=5, fontsize=18, fontname='serif',
...:         color="blue")
...: ax.set_ylabel("f(x)", labelpad=15, fontsize=18, fontname='serif',
...:         color="blue")
...: ax.set_title("axis labels and title example", fontsize=16,
...:         fontname='serif', color="blue")
```

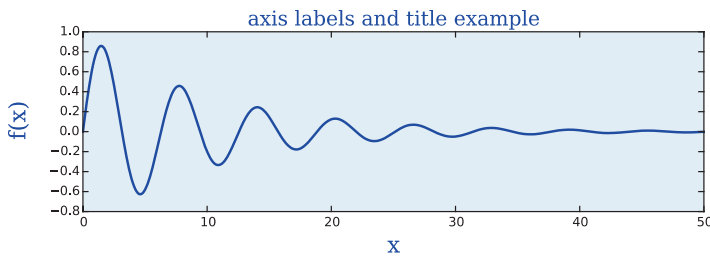


Figure 4-12. Graph demonstrating the result of using `set_xlabel` and `set_ylabel` for setting the x and y axis labels

In addition to labels on the x and y axes, we can also set a title of an Axes object, using the `set_title` method. This method takes mostly the same arguments as `set_xlabel` and `set_ylabel`, with the exception of the `loc` argument, which can be assigned to 'left', 'centered', to 'right', and which dictates that the title is to be left aligned, centered, or right aligned.

Axis Range

By default, the range of the x and y axes of a Matplotlib is automatically adjusted to the data that is plotted in the Axes object. In many cases these default ranges are sufficient, but in some situations, it may be necessary to explicitly set the axis ranges. In such cases, we can use the `set_xlim` and `set_ylim` methods of the Axes object. Both these methods take two arguments that specify the lower and upper limit that is to be displayed on the axis, respectively. An alternative to `set_xlim` and `set_ylim` is the `axis` method, which, for example, accepts the string argument 'tight', for a coordinate range that tightly fit the lines it contains, and 'equal', for a coordinate range where one unit length along each axis corresponds to the same number of pixels (i.e., a ratio preserving coordinate system).

It is also possible to use the `autoscale` method to selectively turn on and off autoscaling, by passing True and False as first argument, for the x and/or y axis by setting its axis argument to 'x', 'y', or 'both'. The example below shows how to use these methods to control axis ranges. The resulting graphs are shown in Figure 4-13.

```
In [15]: x = np.linspace(0, 30, 500)
...: y = np.sin(x) * np.exp(-x/10)
...:
...:
...: fig, axes = plt.subplots(1, 3, figsize=(9, 3), subplot_
...:     kw={'facecolor': "#ebf5ff"})
```

```

...:
...: axes[0].plot(x, y, lw=2)
...: axes[0].set_xlim(-5, 35)
...: axes[0].set_ylim(-1, 1)
...: axes[0].set_title("set_xlim / set_y_lim")
...:
...: axes[1].plot(x, y, lw=2)
...: axes[1].axis('tight')
...: axes[1].set_title("axis('tight')")
...:
...: axes[2].plot(x, y, lw=2)
...: axes[2].axis('equal')
...: axes[2].set_title("axis('equal')")

```

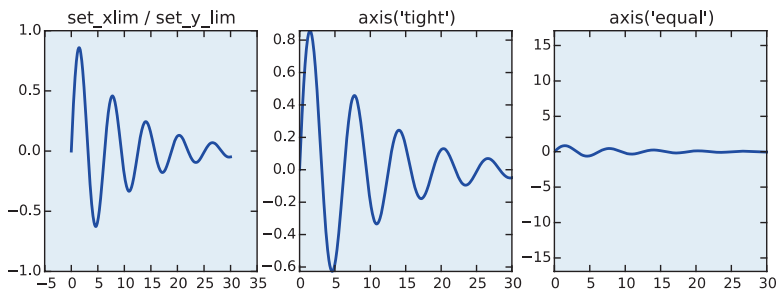


Figure 4-13. *Graphs that show the result of using the `set_xlim`, `set_ylim`, and `axis` methods for setting the axis ranges that are shown in a graph*

Axis Ticks, Tick Labels, and Grids

The final basic properties of the axis that remain to be configured are the placement of axis ticks and the placement and the formatting of the corresponding tick labels. The axis ticks are an important part of the overall appearance of a graph, and when preparing publication and production-quality graphs, it is often necessary to have detailed control over the axis ticks. Matplotlib module `mpl.ticker` provides a general and extensible tick management system that gives full control of the tick placement. Matplotlib distinguishes between major ticks and minor ticks. By default, every major tick has a corresponding label, and the distances between major ticks may be further marked with minor ticks that do not have labels, although this feature must be explicitly turned on. See Figure 4-14 for an illustration of major and minor ticks.

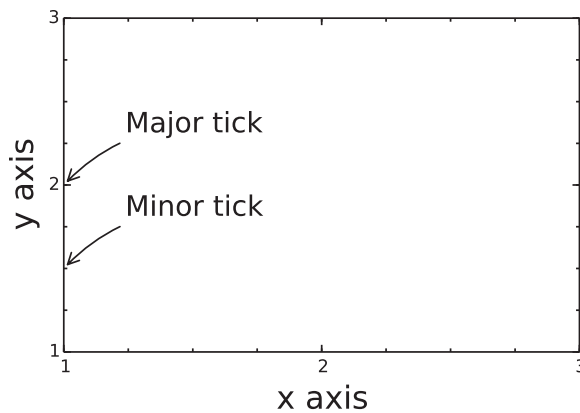


Figure 4-14. *The difference between major and minor ticks*

When approaching the configuration of ticks, the most common design target is to determine where the major tick with labels should be placed along the coordinate axis. The `mpl.ticker` module provides classes for different tick placement strategies. For example, the `mpl.ticker.MaxNLocator` can be used to set the maximum number ticks (at unspecified locations), the `mpl.ticker.MultipleLocator` can be used for setting ticks at multiples of a given base, and the `mpl.ticker.FixedLocator` can be used to place ticks at explicitly specified coordinates. To change ticker strategy, we can use the `set_major_locator` and the `set_minor_locator` methods in `Axes.xaxis` and `Axes.yaxis`. These methods accept an instance of a ticker class defined in `mpl.ticker` or a custom class that is derived from one of those classes.

When explicitly specifying tick locations, we can also use the methods `set_xticks` and `set_yticks`, which accept a list of coordinates for where to place major ticks. In this case, it is also possible to set custom labels for each tick using the `set_xticklabels` and `set_yticklabels`, which expects lists of strings to use as labels for the corresponding ticks. If possible, it is a good idea to use generic tick placement strategies, for example, `mpl.ticker.MaxNLocator`, because they dynamically adjust if the coordinate range is changed, whereas explicit tick placement using `set_xticks` and `set_yticks` then would require manual code changes. However, when the exact placement of ticks must be controlled, then `set_xticks` and `set_yticks` are convenient methods.

The following code demonstrates how to change the default tick placement using combinations of the methods discussed in the previous paragraphs, and the resulting graphs are shown in Figure 4-15.

```

In [16]: x = np.linspace(-2 * np.pi, 2 * np.pi, 500)
....: y = np.sin(x) * np.exp(-x**2/20)
....:
....: fig, axes = plt.subplots(1, 4, figsize=(12, 3))
....:
....: axes[0].plot(x, y, lw=2)
....: axes[0].set_title("default ticks")

....: axes[1].plot(x, y, lw=2)
....: axes[1].set_title("set_xticks")
....: axes[1].set_yticks([-1, 0, 1])
....: axes[1].set_xticks([-5, 0, 5])
....:
....: axes[2].plot(x, y, lw=2)
....: axes[2].set_title("set_major_locator")
....: axes[2].xaxis.set_major_locator(mpl.ticker.MaxNLocator(4))
....: axes[2].yaxis.set_major_locator(mpl.ticker.FixedLocator([-1, 0, 1]))
....: axes[2].xaxis.set_minor_locator(mpl.ticker.MaxNLocator(8))
....: axes[2].yaxis.set_minor_locator(mpl.ticker.MaxNLocator(8))
....:
....: axes[3].plot(x, y, lw=2)
....: axes[3].set_title("set_xticklabels")
....: axes[3].set_yticks([-1, 0, 1])
....: axes[3].set_xticks([-2 * np.pi, -np.pi, 0, np.pi, 2 * np.pi])
....: axes[3].set_xticklabels([r'$-2\pi$', r'$-\pi$', 0, r'$\pi$',
    r'$2\pi$'])
....: x_minor_ticker = mpl.ticker.FixedLocator([-3 * np.pi / 2,
    -np.pi / 2, 0,
    np.pi / 2, 3 * np.pi / 2])
....: axes[3].xaxis.set_minor_locator(x_minor_ticker)
....: axes[3].yaxis.set_minor_locator(mpl.ticker.MaxNLocator(4))

```

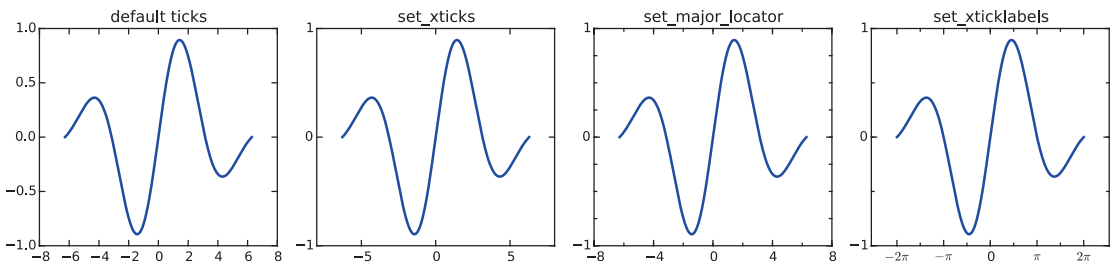


Figure 4-15. *Graphs that demonstrate different ways of controlling the placement and appearance of major and minor ticks along the x axis and the y axis*

A commonly used design element in graphs is grid lines, which are intended as a visual guide when reading values from the graph. Grids and grid lines are closely related to axis ticks, since they are drawn at the same coordinate values, and are therefore essentially extensions of the ticks that span across the graph. In Matplotlib, we can turn on axis grids using the `grid` method of an axes object. The `grid` method takes optional keyword arguments that are used to control the appearance of the grid. For example, like many of the plot functions in Matplotlib, the `grid` method accepts the arguments `color`, `linestyle`, and `linewidth`, for specifying the properties of the grid lines. In addition, it takes argument `which` and axis that can be assigned values `'major'`, `'minor'`, or `'both'`, and `'x'`, `'y'`, or `'both'`, respectively. These arguments are used to indicate which ticks along which axis the given style is to be applied to. If several different styles for the grid lines are required, multiple calls to `grid` can be used, with different values of `which` and axis. For an example of how to add grid lines and how to style them in different ways, see the following example, which produces the graphs shown in Figure 4-16.

```
In [17]: fig, axes = plt.subplots(1, 3, figsize=(12, 4))

...: x_major_ticker = mpl.ticker.MultipleLocator(4)
...: x_minor_ticker = mpl.ticker.MultipleLocator(1)
...: y_major_ticker = mpl.ticker.MultipleLocator(0.5)
...: y_minor_ticker = mpl.ticker.MultipleLocator(0.25)
...:
...: for ax in axes:
...:     ax.plot(x, y, lw=2)
...:     ax.xaxis.set_major_locator(x_major_ticker)
...:     ax.yaxis.set_major_locator(y_major_ticker)
...:     ax.xaxis.set_minor_locator(x_minor_ticker)
...:     ax.yaxis.set_minor_locator(y_minor_ticker)
```



```

...:
...: axes[0].set_title("default grid")
...: axes[0].grid()
...:
...: axes[1].set_title("major/minor grid")
...: axes[1].grid(color="blue", which="both", linestyle=':',
...:             linewidth=0.5)
...:
...: axes[2].set_title("individual x/y major/minor grid")
...: axes[2].grid(color="grey", which="major", axis='x', linestyle='-',
...:             linewidth=0.5)
...: axes[2].grid(color="grey", which="minor", axis='x', linestyle=':',
...:             linewidth=0.25)
...: axes[2].grid(color="grey", which="major", axis='y', linestyle='-',
...:             linewidth=0.5)

```

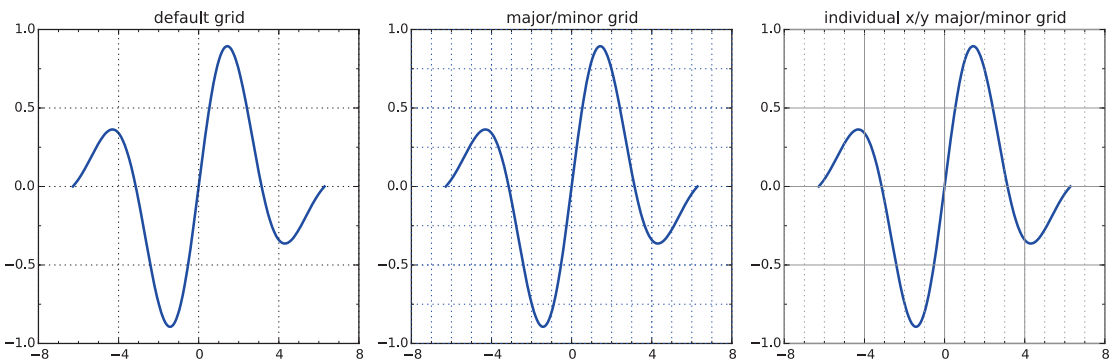


Figure 4-16. *Graphs demonstrating the result of using grid lines*

In addition to controlling the tick placements, the Matplotlib `mpl.ticker` module also provides classes for customizing the tick labels. For example, the `ScalarFormatter` from the `mpl.ticker` module can be used to set several useful properties related to displaying tick labels with scientific notation, for displaying axis labels for large numerical values. If scientific notation is activated using the `set_scientific` method, we can control the threshold for when scientific notation is used with the `set_powerlimits` method (by default, tick labels for small numbers are not displayed using the scientific notation), and we can use the `useMathText=True` argument when creating

the `ScalarFormatter` instance in order to have the exponents shown in math style rather than using code style exponents (e.g., `1e10`). See the following code for an example of using scientific notation in tick labels. The resulting graphs are shown in Figure 4-17.

```
In [19]: fig, axes = plt.subplots(1, 2, figsize=(8, 3))
...:
...: x = np.linspace(0, 1e5, 100)
...: y = x ** 2
...:
...: axes[0].plot(x, y, 'b.')
...: axes[0].set_title("default labels", loc='right')
...:
...: axes[1].plot(x, y, 'b')
...: axes[1].set_title("scientific notation labels", loc='right')
...:
...: formatter = mpl.ticker.ScalarFormatter(useMathText=True)
...: formatter.set_scientific(True)
...: formatter.set_powerlimits((-1,1))
...: axes[1].xaxis.set_major_formatter(formatter)
...: axes[1].yaxis.set_major_formatter(formatter)
```

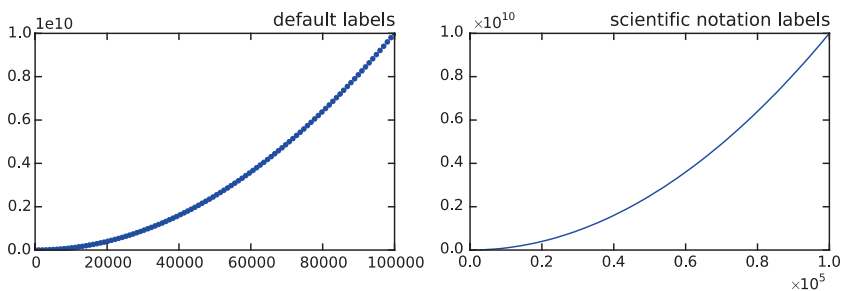


Figure 4-17. Graphs with tick labels in scientific notation. The left panel uses the default label formatting, while the right panel uses tick labels in scientific notation, rendered as math text.

Log Plots

In visualization of data that spans several orders of magnitude, it is useful to work with logarithmic coordinate systems. In Matplotlib, there are several plot functions for graphing functions in such coordinate systems, for example, `loglog`, `semilogx`, and `semilogy`, which use logarithmic scales for both the x and y axes, for only the x axis, and for only the y axis, respectively. Apart from the logarithmic axis scales, these functions behave similarly to the standard plot method. An alternative approach is to use the standard plot method and to separately configure the axis scales to be logarithmic using the `set_xscale` and/or `set_yscale` method with 'log' as first argument. These methods of producing log-scale plots are exemplified in the following section, and the resulting graphs are shown in Figure 4-18.

```
In [20]: fig, axes = plt.subplots(1, 3, figsize=(12, 3))
...:
...: x = np.linspace(0, 1e3, 100)
...: y1, y2 = x**3, x**4
...:
...: axes[0].set_title('loglog')
...: axes[0].loglog(x, y1, 'b', x, y2, 'r')
...:
...: axes[1].set_title('semilogy')
...: axes[1].semilogy(x, y1, 'b', x, y2, 'r')
...:
...: axes[2].set_title('plot / set_xscale / set_yscale')
...: axes[2].plot(x, y1, 'b', x, y2, 'r')
...: axes[2].set_xscale('log')
...: axes[2].set_yscale('log')
```

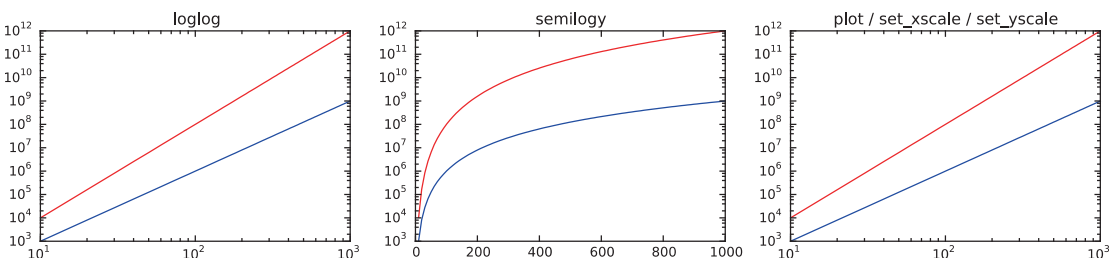


Figure 4-18. Examples of log-scale plots

Twin Axes

An interesting trick with axes that Matplotlib provides is the twin axis feature, which allows displaying two independent axes overlaid on each other. This is useful when plotting two different quantities, for example, with different units, within the same graph. A simple example that demonstrates this feature is shown as follows, and the resulting graph is shown in Figure 4-19. Here we use the `twinx` method (there is also a `twiny` method) to produce second Axes instance with shared x axis and a new independent y axis, which is displayed on the right side of the graph.

```
In [21]: fig, ax1 = plt.subplots(figsize=(8, 4))
...:
...: r = np.linspace(0, 5, 100)
...: a = 4 * np.pi * r ** 2 # area
...: v = (4 * np.pi / 3) * r ** 3 # volume
...:
...: ax1.set_title("surface area and volume of a sphere", fontsize=16)
...: ax1.set_xlabel("radius [m]", fontsize=16)
...:
...: ax1.plot(r, a, lw=2, color="blue")
...: ax1.set_ylabel(r"surface area ($m^2$)", fontsize=16, color="blue")
...: for label in ax1.get_yticklabels():
...:     label.set_color("blue")
...:
...: ax2 = ax1.twinx()
...: ax2.plot(r, v, lw=2, color="red")
...: ax2.set_ylabel(r"volume ($m^3$)", fontsize=16, color="red")
...: for label in ax2.get_yticklabels():
...:     label.set_color("red")
```

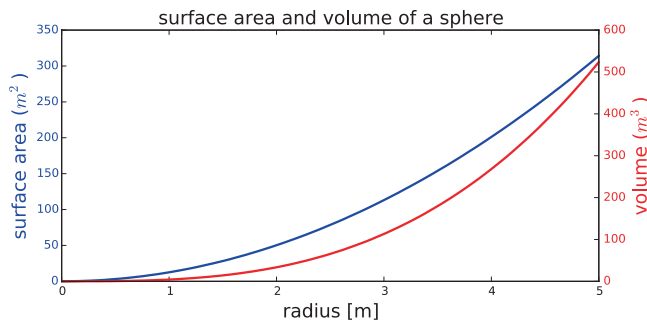


Figure 4-19. Example of graphs with twin axes

Spines

In all graphs generated so far, we have always had a box surrounding the Axes region. This is indeed a common style for scientific and technical graphs, but in some cases, for example, when representing schematic graphs, moving these coordinate lines may be desired. The lines that make up the surrounding box are called axis spines in Matplotlib, and we can use the `Axes.spines` attribute to change their properties. For example, we might want to remove the top and the right spines and move the spines to coincide with the origin of the coordinate systems.

The `spines` attribute of the `Axes` object is a dictionary with the keys `right`, `left`, `top`, and `bottom` that can be used to access each spine individually. We can use the `set_color` method to set the color to `'None'` to indicate that a particular spine should not be displayed, and in this case, we also need to remove the ticks associated with that spine, using the `set_ticks_position` method of `Axes.xaxis` and `Axes.yaxis` (which accepts the arguments `'both'`, `'top'`, or `'bottom'` and `'both'`, `'left'`, or `'right'`, respectively). With these methods we can transform the surrounding box to x and y coordinate axes, as demonstrated in the following example. The resulting graph is shown in Figure 4-20.

```
In [22]: x = np.linspace(-10, 10, 500)
...: y = np.sin(x) / x
...:
...: fig, ax = plt.subplots(figsize=(8, 4))
...:
...: ax.plot(x, y, linewidth=2)
...:
...: # remove top and right spines
```

```

...: ax.spines['right'].set_color('none')
...: ax.spines['top'].set_color('none')
...:
...: # remove top and right spine ticks
...: ax.xaxis.set_ticks_position('bottom')
...: ax.yaxis.set_ticks_position('left')
...:
...: # move bottom and left spine to x = 0 and y = 0
...: ax.spines['bottom'].set_position(('data', 0))
...: ax.spines['left'].set_position(('data', 0))
...:
...: ax.set_xticks([-10, -5, 5, 10])
...: ax.set_yticks([0.5, 1])
...:
...: # give each label a solid background of white, to not overlap with
...:   the plot line
...: for label in ax.get_xticklabels() + ax.get_yticklabels():
...:     label.set_bbox({'facecolor': 'white',
...:                     'edgecolor': 'white'})

```

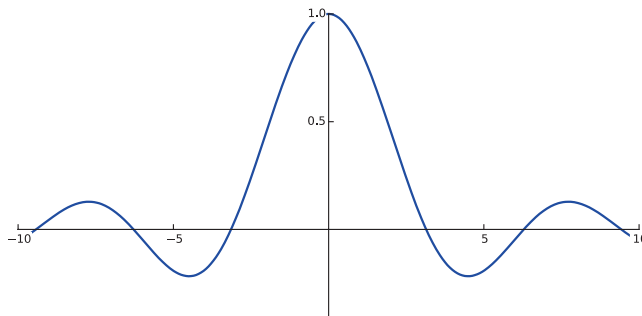


Figure 4-20. Example of a graph with axis spines

Advanced Axes Layouts

So far, we have repeatedly used `plt.figure`, `Figure.make_axes`, and `plt.subplots` to create new `Figure` and `Axes` instances, which we then used for producing graphs. In scientific and technical visualization, it is common to pack together multiple figures in different panels, for example, in a grid layout. In Matplotlib there are functions for automatically creating `Axes` objects and placing them on a figure canvas, using a variety of different layout strategies. We have already used the `plt.subplots` function, which is capable of generating a uniform grid of `Axes` objects. In this section we explore additional features of the `plt.subplots` function and introduce the `subplot2grid` and `GridSpec` layout managers, which are more flexible in how the `Axes` objects are distributed within a figure canvas.

Insets

Before diving into the details of how to use more advanced `Axes` layout managers, it is worth taking a step back and considering an important use-case of the very first approach we used to add `Axes` instances to a figure canvas: the `Figure.add_axes` method. This approach is well suited for creating so-called inset, which is a smaller graph that is displayed within the region of another graph. Insets are, for example, frequently used for displaying a magnified region of special interest in the larger graph or for displaying some related graphs of secondary importance.

In Matplotlib we can place additional `Axes` objects at arbitrary locations within a figure canvas, even if they overlap with existing `Axes` objects. To create an inset, we therefore simply add a new `Axes` object with `Figure.make_axes` and with the (figure canvas) coordinates for where the inset should be placed. A typical example of a graph with an inset is produced by the following code, and the graph that this code generates is shown in Figure 4-21. When creating the `Axes` object for the inset, it may be useful to use the argument `facecolor='none'`, which indicates that there should be no background color, that is, that the `Axes` background of the inset should be transparent.

```
In [23]: fig = plt.figure(figsize=(8, 4))
...:
...: def f(x):
...:     return 1/(1 + x**2) + 0.1/(1 + ((3 - x)/0.1)**2)
...:
```

```

...: def plot_and_format_axes(ax, x, f, fontsize):
...:     ax.plot(x, f(x), linewidth=2)
...:     ax.xaxis.set_major_locator(mpl.ticker.MaxNLocator(5))
...:     ax.yaxis.set_major_locator(mpl.ticker.MaxNLocator(4))
...:     ax.set_xlabel(r"$x$", fontsize=fontsize)
...:     ax.set_ylabel(r"$f(x)$", fontsize=fontsize)
...:
...: # main graph
...: ax = fig.add_axes([0.1, 0.15, 0.8, 0.8], facecolor="#f5f5f5")
...: x = np.linspace(-4, 14, 1000)
...: plot_and_format_axes(ax, x, f, 18)
...:
...: # inset
...: x0, x1 = 2.5, 3.5
...: ax.axvline(x0, ymax=0.3, color="grey", linestyle=":")
...: ax.axvline(x1, ymax=0.3, color="grey", linestyle=":")
...:
...: ax_insert = fig.add_axes([0.5, 0.5, 0.38, 0.42], facecolor='none')
...: x = np.linspace(x0, x1, 1000)
...: plot_and_format_axes(ax_insert, x, f, 14)

```

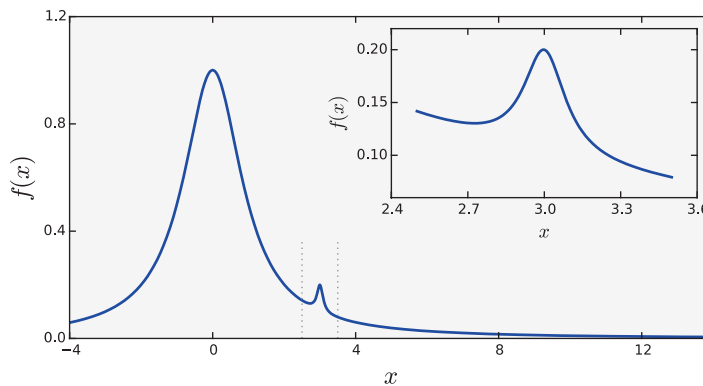


Figure 4-21. Example of a graph with an inset

Subplots

We have already used `plt.subplots` extensively, and we have noted that it returns a tuple with a `Figure` instance and a NumPy array with the `Axes` objects for each row and column that was requested in the function call. It is often the case when plotting grids of subplots that either the x or the y axis, or both, is shared among the subplots. Using the `sharex` and `sharey` arguments to `plt.subplots` can be useful in such situations, since it prevents the same axis labels to be repeated across multiple `Axes`.

It is also worth noting that the dimension of the NumPy array with `Axes` instances that is returned by `plt.subplots` is “squeezed” by default: that is, the dimensions with length 1 are removed from the array. If both the requested numbers of column and row are greater than one, then a two-dimensional array is returned, but if either (or both) the number of columns or rows is one, then a one-dimensional (or scalar, i.e., the only `Axes` object itself) is returned. We can turn off the squeezing of the dimensions of the NumPy arrays by passing the argument `squeeze=False` to the `plt.subplots` function. In this case the axes variable in `fig, axes = plt.subplots(nrows, ncols)` is always a two-dimensional array.

A final touch of configurability can be achieved using the `plt.subplots_adjust` function, which allows to explicitly set the left, right, bottom, and top coordinates of the overall `Axes` grid, as well as the width (`wspace`) and height spacing (`hspace`) between `Axes` instances in the grid. See the following code, and the corresponding Figure 4-22, for a step-by-step example of how to set up an `Axes` grid with shared x and y axes and with adjusted `Axes` spacing.

```
In [24]: fig, axes = plt.subplots(2, 2, figsize=(6, 6), sharex=True,
...: sharey=True, squeeze=False)
...:
...: x1 = np.random.randn(100)
...: x2 = np.random.randn(100)
...:
...: axes[0, 0].set_title("Uncorrelated")
...: axes[0, 0].scatter(x1, x2)
...:
...: axes[0, 1].set_title("Weakly positively correlated")
...: axes[0, 1].scatter(x1, x1 + x2)
...:
```

```

...: axes[1, 0].set_title("Weakly negatively correlated")
...: axes[1, 0].scatter(x1, -x1 + x2)
...:
...: axes[1, 1].set_title("Strongly correlated")
...: axes[1, 1].scatter(x1, x1 + 0.15 * x2)
...:
...: axes[1, 1].set_xlabel("x")
...: axes[1, 0].set_xlabel("x")
...: axes[0, 0].set_ylabel("y")
...: axes[1, 0].set_ylabel("y")
...:
...: plt.subplots_adjust(left=0.1, right=0.95, bottom=0.1, top=0.95,
...:                     wspace=0.1, hspace=0.2)

```

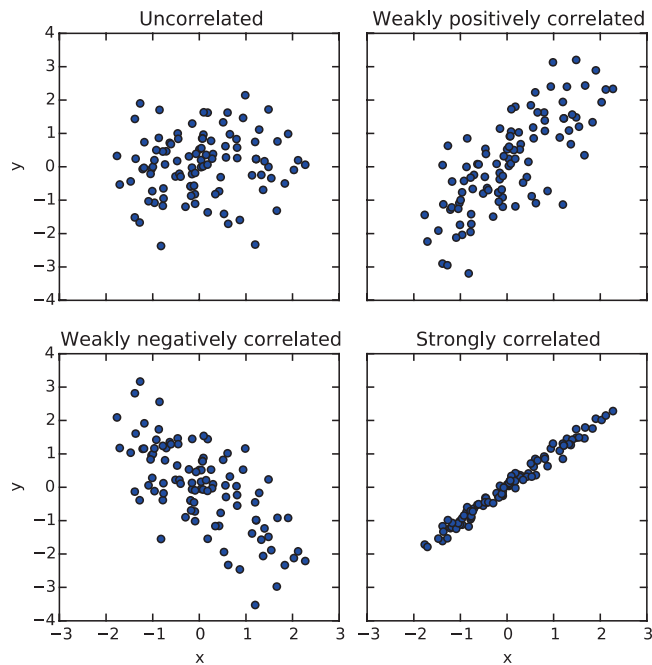
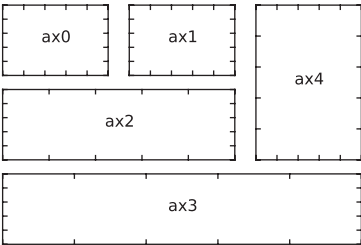


Figure 4-22. Example graph using `plt.subplot` and `plt.subplot_adjust`

Subplot2grid

The `plt.subplot2grid` function is an intermediary between `plt.subplots` and `gridspec` (see the next section) that provides a more flexible Axes layout management than `plt.subplots` while at the same time being simpler to use than `gridspec`. In particular, `plt.subplot2grid` is able to create grids with Axes instances that span multiple rows and/or columns. The `plt.subplot2grid` takes two mandatory arguments: the first argument is the shape of the Axes grid, in the form of a tuple (`nrows`, `ncols`), and the second argument is a tuple (`row`, `col`) that specifies the starting position within the grid. The two optional keyword arguments `colspan` and `rowspan` can be used to indicate how many rows and columns the new Axes instance should span. An example of how to use the `plt.subplot2grid` function is given in Table 4-3. Note that each call to the `plt.subplot2grid` function results in one new Axes instance, in contrast to `plt.subplots` which creates all Axes instances in one function call and returns them in a NumPy array.

Table 4-3. *Example of a Grid Layout Created with `plt.subplot2grid` and the Corresponding Code*

Axes Grid Layout	Code
	<pre>ax0 = plt.subplot2grid((3, 3), (0, 0)) ax1 = plt.subplot2grid((3, 3), (0, 1)) ax2 = plt.subplot2grid((3, 3), (1, 0), colspan=2) ax3 = plt.subplot2grid((3, 3), (2, 0), colspan=3) ax4 = plt.subplot2grid((3, 3), (0, 2), rowspan=2)</pre>

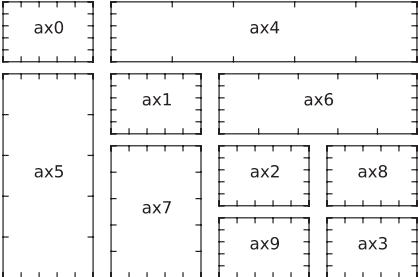
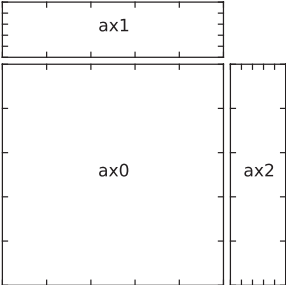
GridSpec

The final grid layout manager that we cover here is `GridSpec` from the `mpl.gridspec` module. This is the most general grid layout manager in Matplotlib, and in particular it allows creating grids where not all rows and columns have equal width and height, which is not easily achieved with the grid layout managers we have used earlier in this chapter.

A `GridSpec` object is only used to specify the grid layout, and by itself it does not create any `Axes` objects. When creating a new instance of the `GridSpec` class, we must specify the number of rows and columns in the grid. Like for other grid layout managers, we can also set the position of the grid using the keyword arguments `left`, `bottom`, `right`, and `top`, and we can set the width and height spacing between subplots using `wspace` and `hspace`. Additionally, `GridSpec` allows specifying the relative width and heights of columns and rows using the `width_ratios` and `height_ratios` arguments. These should both be lists with relative weights for the size of each column and row in the grid. For example, to generate a grid with two rows and two columns, where the first row and column is twice as big as the second row and column, we could use `mpl.gridspec.GridSpec(2, 2, width_ratios=[2, 1], height_ratios=[2, 1])`.

Once a `GridSpec` instance has been created, we can use the `Figure.add_subplot` method to create `Axes` objects and place them on a figure canvas. As argument to `add_subplot`, we need to pass an `mpl.gridspec.SubplotSpec` instance, which we can generate from the `GridSpec` object using an array-like indexing: for example, given a `GridSpec` instance `gs`, we obtain a `SubplotSpec` instance for the upper-left grid element using `gs[0, 0]` and for a `SubplotSpec` instance that covers the first row we use `gs[:, 0]` and so on. See Table 4-4 for concrete examples of how to use `GridSpec` and `add_subplot` to create `Axes` instance.

Table 4-4. *Examples of How to Use the Subplot Grid Manager `mpl.gridspec`. `GridSpec`*

Axes Grid Layout	Code
	<pre>fig = plt.figure(figsize=(6, 4)) gs = mpl.gridspec.GridSpec(4, 4) ax0 = fig.add_subplot(gs[0, 0]) ax1 = fig.add_subplot(gs[1, 1]) ax2 = fig.add_subplot(gs[2, 2]) ax3 = fig.add_subplot(gs[3, 3]) ax4 = fig.add_subplot(gs[0, 1:]) ax5 = fig.add_subplot(gs[1:, 0]) ax6 = fig.add_subplot(gs[1, 2:]) ax7 = fig.add_subplot(gs[2:, 1]) ax8 = fig.add_subplot(gs[2, 3]) ax9 = fig.add_subplot(gs[3, 2])</pre>
	<pre>fig = plt.figure(figsize=(4, 4)) gs = mpl.gridspec.GridSpec(2, 2, width_ratios=[4, 1], height_ratios=[1, 4], wspace=0.05, hspace=0.05) ax0 = fig.add_subplot(gs[1, 0]) ax1 = fig.add_subplot(gs[0, 0]) ax2 = fig.add_subplot(gs[1, 1])</pre>

Colormap Plots

We have so far only considered graphs of univariate functions or, equivalently, two-dimensional data in x - y format. The two-dimensional Axes objects that we have used for this purpose can also be used to visualize bivariate functions, or three-dimensional data on x - y - z format, using so-called color maps (or heat maps), where each pixel in the Axes

area is colored according to the z value corresponding to that point in the coordinate system. Matplotlib provides the functions `pcolor` and `imshow` for these types of plots, and the `contour` and `contourf` functions graph data on the same format by drawing contour lines rather than color maps. Examples of graphs generated with these functions are shown in Figure 4-23.

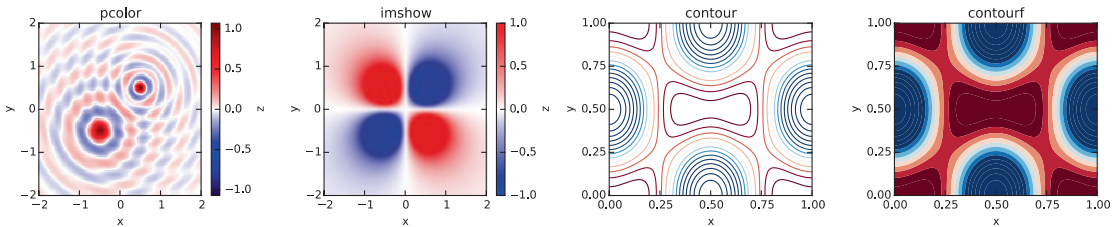


Figure 4-23. Example graphs generated with *pcolor*, *imshow*, *contour*, and *contourf*

To produce a colormap graph, for example, using `pcolor`, we first need to prepare the data in the appropriate format. While standard two-dimensional graphs expect one-dimensional coordinate arrays with x and y values, in the present case, we need to use two-dimensional coordinate arrays, as, for example, generated using the NumPy `meshgrid` function. To plot a bivariate function or data with two dependent variables, we start by defining one-dimensional coordinate arrays, x and y , that span the desired coordinate range or correspond to the values for which data is available. The x and y arrays can then be passed to the `np.meshgrid` function, which produces the required two-dimensional coordinate arrays X and Y . If necessary, we can use NumPy array computations with X and Y to evaluate bivariate functions to obtain a data array Z , as done in lines 1 to 3 in In [25] (see in the following section).

Once the two-dimensional coordinate and data arrays are prepared, they are easily visualized using, for example, `pcolor`, `contour`, or `contourf`, by passing the X , Y , and Z arrays as the first three arguments. The `imshow` method works similarly but only expects the data array Z as argument, and the relevant coordinate ranges must instead be set using the `extent` argument, which should be set to a list on the format `[xmin, xmax, ymin, ymax]`. Additional keyword arguments that are important for controlling the appearance of colormap graphs are `vmin`, `vmax`, `norm`, and `cmap`: the `vmin` and `vmax` can be used to set the range of values that are mapped to the color axis. This can equivalently be achieved by setting `norm=plt.colors.Normalize(vmin, vmax)`. The `cmap` argument

specifies a color map for mapping the data values to colors in the graph. This argument can either be a string with a predefined colormap name or a colormap instance. The predefined color maps in Matplotlib are available in `mpl.cm`. Try `help(mpl.cm)` or try to autocomplete in IPython on the `mpl.cm` module for a full list of available color maps.⁵

The last piece required for a complete colormap plot is the colorbar element, which gives the viewer of the graph a way to read off the numerical values that different colors correspond to. In Matplotlib we can use the `plt.colorbar` function to attach a colorbar to an already plotted colormap graph. It takes a handle to the plot as first argument, and it takes two optional arguments `ax` and `cax`, which can be used to control where in the graph the colorbar is to appear. If `ax` is given, the space will be taken from this Axes object for the new colorbar. If, on the other hand, `cax` is given, then the colorbar will draw on this Axes object. A colorbar instance `cb` has its own axis object, and the standard methods for setting axis attributes can be used on the `cb.ax` object, and we can use, for example, the `set_label`, `set_ticks`, and `set_ticklabels` method in the same manner as for `x` and `y` axes.

The steps outlined in the previous paragraphs are shown in the following code, and the resulting graph is shown in Figure 4-24. The functions `imshow`, `contour`, and `contourf` can be used in a nearly similar manner, although these functions take additional arguments for controlling their characteristic properties. For example, the `contour` and `contourf` functions additionally take an argument `N` that specifies the number of contour lines to draw.

```
In [25]: x = y = np.linspace(-10, 10, 150)
...: X, Y = np.meshgrid(x, y)
...: Z = np.cos(X) * np.cos(Y) * np.exp(-(X/5)**2-(Y/5)**2)
...:
...: fig, ax = plt.subplots(figsize=(6, 5))
...:
...: norm = mpl.colors.Normalize(-abs(Z).max(), abs(Z).max())
...: p = ax.pcolor(X, Y, Z, norm=norm, cmap=mpl.cm.bwr)
...:
...: ax.axis('tight')
...: ax.set_xlabel(r"$x$", fontsize=18)
```

⁵A nice visualization of all the available color maps is available at http://wiki.scipy.org/Cookbook/Matplotlib/Show_colormaps. This page also describes how to create new color maps.

```

...: ax.set_ylabel(r"$y$", fontsize=18)
...: ax.xaxis.set_major_locator(mpl.ticker.MaxNLocator(4))
...: ax.yaxis.set_major_locator(mpl.ticker.MaxNLocator(4))
...:
...: cb = fig.colorbar(p, ax=ax)
...: cb.set_label(r"$z$", fontsize=18)
...: cb.set_ticks([-1, -.5, 0, .5, 1])

```

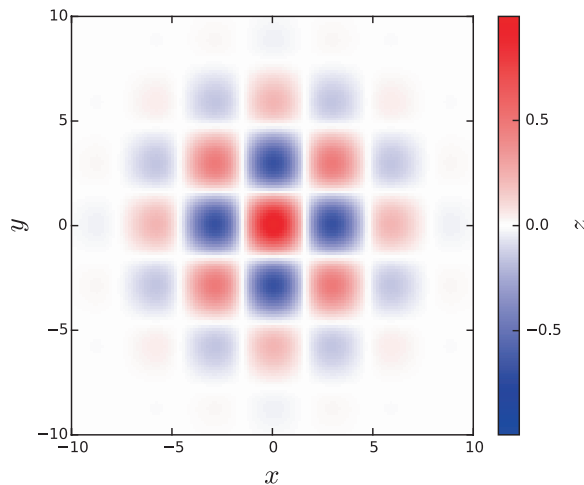


Figure 4-24. Example using *pcolor* to produce a colormap graph

3D Plots

The colormap graphs discussed in the previous section were used to visualize data with two dependent variables by color-coding data in 2D graphs. Another way of visualizing the same type of data is to use 3D graphs, where a third axis z is introduced and the graph is displayed in a perspective on the screen. In Matplotlib, drawing 3D graphs requires using a different axes object, namely, the `Axes3D` object that is available from the `mpl_toolkits.mplot3d` module. We can create a 3D-aware Axes instance explicitly using the constructor of the `Axes3D` class, by passing a Figure instance as argument: `ax = Axes3D(fig)`. Alternatively, we can use the `add_subplot` function with the `projection='3d'` argument:

```
ax = ax = fig.add_subplot(1, 1, 1, projection='3d')
```


or use `plt.subplots` with the `subplot_kw={'projection': '3d'}` argument:

```
fig, ax = plt.subplots(1, 1, figsize=(8, 6), subplot_kw={'projection': '3d'})
```

In this way, we can use all of the axes layout approaches we have previously used for 2D graphs, if only we specify the `projection` argument in the appropriate manner. Note that using `add_subplot`, it is possible to mix axes objects with 2D and 3D projections within the same figure, but when using `plt.subplots`, the `subplot_kw` argument applies to all the subplots added to a figure.

Having created and added 3D-aware Axes instances to a figure, for example, using one of the methods described in the previous paragraph, the Axes3D class methods – such as `plot_surface`, `plot_wireframe`, and `contour` – can be used to plot data as surfaces in a 3D perspective. These functions are used in a manner that is nearly the same as how the color map was used in the previous section: these 3D plotting functions all take two-dimensional coordinate and data arrays *X*, *Y*, and *Z* as first arguments. Each function also takes additional parameters for tuning specific properties. For example, the `plot_surface` function takes the arguments `rstride` and `cstride` (row and column stride) for selecting data from the input arrays (to avoid data points that are too dense). The `contour` and `contourf` functions take optional arguments `zdir` and `offset`, which is used to select a projection direction (the allowed values are “x,” “y,” and “z”) and the plane to display the projection on.

In addition to the methods for 3D surface plotting, there are also straightforward generalizations of the line and scatter plot functions that are available for 2D axes, for example, `plot`, `scatter`, `bar`, and `bar3d`, which in the version that is available in the Axes3D class takes an additional argument for the *z* coordinates. Like their 2D relatives, these functions expect one-dimensional data arrays rather than the two-dimensional coordinate arrays that are used for surface plots.

When it comes to axes titles, labels, ticks, and tick labels, all the methods used for 2D graphs, as described in detail earlier in this chapter, are straightforwardly generalized to 3D graphs. For example, there are new methods `set_zlabel`, `set_zticks`, and `set_zticklabels` for manipulating the attributes of the new *z* axis. The Axes3D object also provides new class methods for 3D specific actions and attributes. In particular, the `view_init` method can be used to change the angle from which the graph is viewed, and it takes the elevation and the azimuth, in degrees, as first and second arguments.

Examples of how to use these 3D plotting functions are given in the following section, and the produced graphs are shown in Figure 4-25.

```
In [26]: fig, axes = plt.subplots(1, 3, figsize=(14, 4), subplot_
        kw={'projection': '3d'})
...:
...: def title_and_labels(ax, title):
...:     ax.set_title(title)
...:     ax.set_xlabel("$x$", fontsize=16)
...:     ax.set_ylabel("$y$", fontsize=16)
...:     ax.set_zlabel("$z$", fontsize=16)
...:
...: x = y = np.linspace(-3, 3, 74)
...: X, Y = np.meshgrid(x, y)
...:
...: R = np.sqrt(X**2 + Y**2)
...: Z = np.sin(4 * R) / R
...:
...: norm = mpl.colors.Normalize(-abs(Z).max(), abs(Z).max())
...:
...: p = axes[0].plot_surface(X, Y, Z, rstride=1, cstride=1,
        linewidth=0, antialiased=False, norm=norm, cmap=mpl.cm.Blues)
...:
...: cb = fig.colorbar(p, ax=axes[0], shrink=0.6)
...: title_and_labels(axes[0], "plot_surface")
...:
...: p = axes[1].plot_wireframe(X, Y, Z, rstride=2, cstride=2,
        color="darkgrey")
...: title_and_labels(axes[1], "plot_wireframe")
...:
...: cset = axes[2].contour(X, Y, Z, zdir='z', offset=0, norm=norm,
        cmap=mpl.cm.Blues)
...: cset = axes[2].contour(X, Y, Z, zdir='y', offset=3, norm=norm,
        cmap=mpl.cm.Blues)
...: title_and_labels(axes[2], "contour")
```

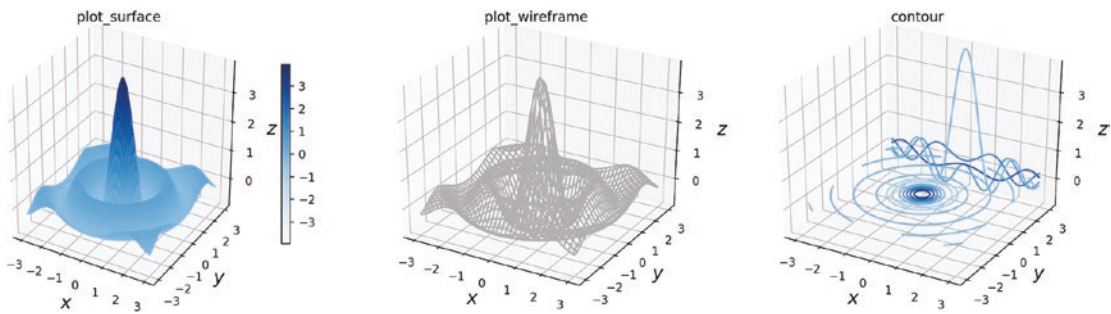


Figure 4-25. 3D surface and contour graphs generated by using `plot_surface`, `plot_wireframe`, and `contour`

Summary

In this chapter, we have covered the basics of how to produce 2D and 3D graphics using Matplotlib. Visualization is one of the most important tools for computational scientists and engineers, both as an analysis tool while working on computational problems and for presenting and communicating computational results. Visualization is therefore an integral part of the computational workflow, and it is equally important to be able to quickly visualize and explore data and to be able to produce picture-perfect publication-quality graphs, with detailed control over every graphical element. Matplotlib is a great general-purpose tool for both exploratory visualization and for producing publication-quality graphics. However, there are limitations to what can be achieved with Matplotlib, especially with respect to interactivity and high-quality 3D graphics. For more specialized use-cases, I therefore recommend to also explore some of the other graphic libraries that are available in the scientific Python ecosystem, some of which was briefly mentioned at the beginning of this chapter.

Further Reading

The Matplotlib is treated in books dedicated to the library, such as Tosi (2009) and Devert (2014), and in several books with a wider scope, for example, Milovanovi (2013) and McKinney (2013). For interesting discussions on data visualization and style guides and good practices in visualization, see, for example, Yau (2011) and J. Steele (2010).