Chapter 16. Numeric Processing

You can perform some numeric computations with operators (covered in **"Numeric Operations"**) and built-in functions (covered in **"Built-in Functions"**). Python also provides modules that support additional numeric computations, covered in this chapter: `math` and `cmath`, `statistics`, `operator`, `random` and `secrets`, `fractions`, and `decimal`. Numeric processing often requires, more specifically, the processing of *arrays* of numbers; this topic is covered in **"Array Processing"**, focusing on the standard library module `array` and popular third-party extension NumPy. Finally, **"Additional numeric packages"** lists several additional numeric processing packages produced by the Python community. Most examples in this chapter assume you've imported the appropriate module; **`import`** statements are only included where the situation might be unclear.

Floating-Point Values

Python represents real numeric values (that is, those that are not integers) using variables of type `float`. Unlike integers, computers can rarely represent `floats` exactly, due to their internal implementation as a fixed-size binary integer *significand* (often incorrectly called "mantissa") and a fixed-size binary integer exponent. `floats` have several limitations (some of which can lead to unexpected results).

For most everyday applications, `floats` are sufficient for arithmetic, but they are limited in the number of decimal places they can represent:

```
>>> f = 1.1 + 2.2 - 3.3  # f should be equal to 0
>>> f
```

```
4.440892098500626e-16
```

They are also limited in the range of integer values they can accurately store (in the sense of being able to distinguish one from the next largest or smallest integer value):

```
>>> f = 2 ** 53
>>> f
```

```
9007199254740992
```

```
>>> f + 1
```

```
9007199254740993    # integer arithmetic is not bounded
```

```
>>> f + 1.0
```

```
9007199254740992.0  # float conversion loses integer precision at 2**53
```

Always keep in mind that `floats` are not entirely precise, due to their internal representation in the computer. The same consideration applies to `complex` numbers.

DON'T USE == BETWEEN FLOATING-POINT OR COMPLEX NUMBERS

Given that floating-point numbers and operations only approximate the behavior of mathematical "real numbers," it seldom makes sense to check two `floats` *x* and *y* for exact equality. Tiny variations in how each was computed can easily result in unexpected differences.

For testing floating-point or complex numbers for equality, use the function `isclose` exported by the built-in module `math`. The following code illustrates why:

```
>>> import math
>>> f = 1.1 + 2.2 - 3.3  # f intuitively equal to 0
>>> f == 0
```

```
False
```

```
>>> f
```

```
4.440892098500626e-16
```

```
>>> # default tolerance fine for this comparison
>>> math.isclose(-1, f-1)
```

```
True
```

For some values, you may have to set the tolerance value explicitly (this is *always* necessary when you're comparing with `0`):

```
>>> # near-0 comparison with default tolerances
>>> math.isclose(0, f)
```

```
False
```

```
>>> # must use abs_tol for comparison with 0
>>> math.isclose(0, f, abs_tol=1e-15)
```

```
True
```

You can also use `isclose` for safe looping.

DON'T USE A FLOAT AS A LOOP CONTROL VARIABLE

A common error is to use a floating-point value as the control variable of a loop, assuming that it will eventually equal some ending value, such as `0`. Instead, it most likely will end up looping forever.

The following loop, expected to loop five times and then end, will in fact loop forever:

```
>>> f = 1
>>> while f != 0:
...     f -= 0.2
```

Even though `f` started as an `int`, it's now a `float`. This code shows why:

```
>>> 1-0.2-0.2-0.2-0.2-0.2  # should be 0, but...
```

```
5.551115123125783e-17
```

Even using the inequality operator `>` results in incorrect behavior, looping six times instead of five (since the residual `float` value is still greater than `0`):

```
>>> f = 1
>>> count = 0
>>> while f > 0:
...     count += 1
...     f -= 0.2
>>> print(count)
```

```
6   # one loop too many!
```

If instead you use `math.isclose` for comparing *f* with `0`, the **`for`** loop repeats the correct number of times:

```
>>> f = 1
>>> count = 0
>>> while not math.isclose(0,f,abs_tol=1e-15):
...     count += 1
...     f -= 0.2
>>> print(count)
```

```
5   # just right this time!
```

In general, try to use an `int` for a loop's control variable, rather than a `float`.

Finally, mathematical operations that result in very large `floats` will often cause an `OverflowError`, or Python may return them as `inf` (infinity). The maximum `float` value usable on your computer is `sys.float_info.max`: on 64-bit computers, it's `1.7976931348623157e+308`. This may cause unexpected results when doing math using very large numbers. When you need to work with very large numbers, we recommend using the `decimal` module or third-party **gmpy** instead.

The math and cmath Modules

The `math` module supplies mathematical functions for working with floating-point numbers; the `cmath` module supplies equivalent functions for complex numbers. For example, `math.sqrt(-1)` raises an exception, but `cmath.sqrt(-1)` returns `1j`.

Just like for any other module, the cleanest, most readable way to use these is to have, for example, **`import`** `math` at the top of your code, and explicitly call, say, `math.sqrt` afterward. However, if your code includes a large number of calls to the modules' well-known mathematical functions, you might (though it may lose some clarity and readability) either use **`from`** `math` **`import`** `*`, or use **`from`** `math` **`import`** `sqrt`, and afterward just call `sqrt`.

Each module exposes three `float` attributes bound to the values of fundamental mathematical constants, `e`, `pi`, and **tau**, and a variety of functions, including those shown in **Table 16-1**. The `math` and `cmath` modules are not fully symmetric, so for each method the table indicates whether it is in `math`, `cmath`, or both. All examples assume you have imported the appropriate module.

Table 16-1. Methods and attributes of the `math` and `cmath` modules

		math
`acos`, `asin`,	`acos(`*`x`*`)`, etc. Return the trigonometric functions arccosine,	✓

		math
`atan`, `cos`, `sin`, `tan`	arcsine, arctangent, cosine, sine, or tangent, respectively, of *x*, given in radians.	
`acosh`, `asinh`, `atanh`, `cosh`, `sinh`, `tanh`	`acosh(x)`, etc. Return the arc hyperbolic cosine, arc hyperbolic sine, arc hyperbolic tangent, hyperbolic cosine, hyperbolic sine, or hyperbolic tangent, respectively, of *x*, given in radians.	✓

`atan2`	`atan2(y, x)`
	Like `atan(y/x)`, except that `atan2` properly takes into account the signs of both arguments. For example:
	`>>> math.atan(-1./-1.)`
	`0.7853981633974483`
	`>>> math.atan2(-1., -1.)`
	`-2.356194490192345`
	When *x* equals 0, `atan2` returns π/2, while dividing by *x* would raise `ZeroDivisionError`.
		✓

`cbrt`	`cbrt(x)` 3.11+ Returns the cube root of *x*.	✓

		math
`ceil`	`ceil(x)` Returns `float(i)`, where `i` is the smallest integer such that `i>=x`.	✓
`comb`	`comb(n, k)` 3.8+ Returns the number of *combinations* of `n` items taken `k` items at a time, regardless of order. When counting the number of combinations taken from three items *A*, *B*, and *C*, two at a time, `comb(3, 2)` returns `3` because, for example, *A*-*B* and *B*-*A* are considered the *same* combination (contrast this with `perm`, later in this table). Raises `ValueError` when `k` or `n` is negative; raises `TypeError` when `k` or `n` is not an `int`. When `k>n`, just returns `0`, raising no exceptions.	✓
`copysign`	`copysign(x, y)` Returns the absolute value of `x` with the sign of `y`.	✓
`degrees`	`degrees(x)` Returns the degree measure of the angle `x` given in radians.	✓
`dist`	`dist(pt0, pt1)` 3.8+ Returns the Euclidean distance between two *n*-dimensional points, where each point is represented as a sequence of values (coordinates). Raises `ValueError` if `pt0` and `pt1` are sequences of different lengths.	✓

		math
`e`	The mathematical constant *e* (2.718281828459045).	✓
`erf`	`erf(`*x*`)` Returns the error function of *x* as used in statistical calculations.	✓
`erfc`	`erfc(`*x*`)` Returns the complementary error function at *x*, defined as `1.0 - erf(`*x*`)`.	✓
`exp`	`exp(`*x*`)` Returns e^x.	✓
`exp2`	`exp2(`*x*`)` 3.11+ Returns 2^x.	✓
`expm1`	`expm1(`*x*`)` Returns $e^x - 1$. Inverse of `log1p`.	✓
`fabs`	`fabs(`*x*`)` Returns the absolute value of *x*. Always returns a `float`, even if *x* is an `int` (unlike the built-in `abs` function).	✓
`factorial`	`factorial(`*x*`)` Returns the factorial of *x*. Raises `ValueError` when *x* is negative, and `TypeError` when *x* is not integral.	✓
`floor`	`floor(`*x*`)` Returns `float(`*i*`)`, where *i* is the greatest integer such that *i*`<=`*x*.	✓

		math
`fmod`	`fmod(x, y)` Returns the `float` *r*, with the same sign as *x*, such that `r==x-n*y` for some integer *n*, and `abs(r)<abs(y)`. Like `x%y`, except that, when *x* and *y* differ in sign, `x%y` has the same sign as *y*, not the same sign as *x*.	✓
`frexp`	`frexp(x)` Returns a pair `(m, e)` where *m* is a floating-point number and *e* is an integer such that `x==m*(2**e)` and `0.5<=abs(m)<1`,[a] except that `frexp(0)` returns `(0.0, 0)`.	✓
`fsum`	`fsum(iterable)` Returns the floating-point sum of the values in *iterable* to greater precision than the `sum` built-in function.	✓
`gamma`	`gamma(x)` Returns the Gamma function evaluated at *x*.	✓
`gcd`	`gcd(x, y)` Returns the greatest common divisor of *x* and *y*. When *x* and *y* are both zero, returns `0`. (3.9+ `gcd` can accept any number of values; `gcd()` without arguments returns `0`.)	✓
`hypot`	`hypot(x, y)` Returns `sqrt(x*x+y*y)`. (3.8+ `hypot` can accept any number of values, to compute a hypotenuse length in *n* dimensions.)	✓

		math
`inf`	A floating-point positive infinity, like `float('inf')`.	✓
`infj`	A `complex` imaginary infinity, equal to `complex(0, float('inf'))`.	

`isclose`	`isclose(x, y, rel_tol=1e-09, abs_tol=0.0)`
	Returns **`True`** when *x* and *y* are approximately equal, within relative tolerance `rel_tol`, with minimum absolute tolerance of `abs_tol`; otherwise, returns **`False`**. Default is `rel_tol` within nine decimal digits. `rel_tol` must be greater than `0`. `abs_tol` is used for comparisons near zero: it must be at least `0.0`. `NaN` is not considered close to any value (including `NaN` itself); each of `-inf` and `inf` is only considered close to itself. Except for behavior at +/– `inf`, `isclose` is like:
	`abs(x-y) <= max(rel_tol*max(abs(x), abs(y)),abs_tol)`
		✓

`isfinite`	`isfinite(x)` Returns **`True`** when *x* (in `cmath`, both the real and imaginary parts of *x*) is neither infinity nor `NaN`; otherwise, returns **`False`**.	✓
`isinf`	`isinf(x)` Returns **`True`** when *x* (in `cmath`, either the real or imaginary part of *x*, or both) is positive or negative infinity; otherwise, returns **`False`**.	✓

		math
`isnan`	`isnan(x)` Returns **`True`** when *x* (in `cmath`, either the real or imaginary part of *x*, or both) is `NaN`; otherwise, returns **`False`**.	✓
`isqrt`	`isqrt(x)` **3.8+** Returns `int(sqrt(x))`.	✓
`lcm`	`lcm(x, ...)` **3.9+** Returns the least common multiple of the given `ints`. If not all values are `ints`, raises `TypeError`.	✓
`ldexp`	`ldexp(x, i)` Returns `x*(2**i)` (`i` must be an `int`; when `i` is a `float`, `ldexp` raises `TypeError`). Inverse of `frexp`.	✓
`lgamma`	`lgamma(x)` Returns the natural logarithm of the absolute value of the Gamma function evaluated at *x*.	✓
`log`	`log(x)` Returns the natural logarithm of *x*.	✓
`log10`	`log10(x)` Returns the base-10 logarithm of *x*.	✓
`log1p`	`log1p(x)` Returns the natural logarithm of `1+x`. Inverse of `expm1`.	✓

		math
log2	log2(*x*) Returns the base-2 logarithm of *x*.	✓
modf	modf(*x*) Returns a pair (*f*, *i*) with the fractional and integer parts of *x*, meaning two floats with the same sign as *x* such that *i*==int(*i*) and *x*==*f*+*i*.	✓
nan	nan A floating-point “Not a Number” (NaN) value, like float('nan') or complex('nan').	✓
nanj	A complex number with a 0.0 real part and floating-point “Not a Number” (NaN) imaginary part.	
nextafter	nextafter(*a*, *b*) 3.9+ Returns the next higher or lower float value from *a* in the direction of *b*.	✓
perm	perm(*n*, *k*) 3.8+ Returns the number of *permutations* of *n* items taken *k* items at a time, where selections of the same items but in differing order are counted separately. When counting the number of permutations of three items *A*, *B*, and *C*, taken two at a time, perm(3, 2) returns 6, because, for example, *A*-*B* and *B*-*A* are considered to be different permutations (contrast this with comb, earlier in this table). Raises ValueError when *k* or *n* is negative; raises TypeError when *k* or *n* is not an int.	✓

		math
`pi`	The mathematical constant π, 3.141592653589793.	✓
`phase`	`phase(`*`x`*`)` Returns the *phase* of *x*, a `float` in the range (–π, π). Like `math.atan2(`*`x`*`.imag, `*`x`*`.real)`. See "Conversions to and from polar coordinates" in the **online docs** for details.	
`polar`	`polar(`*`x`*`)` Returns the polar coordinate representation of *x*, as a pair `(`*`r`*`, `*`phi`*`)` where *r* is the modulus of *x* and *phi* is the phase of *x*. Like `(abs(`*`x`*`), cmath.phase(`*`x`*`))`. See "Conversions to and from polar coordinates" in the **online docs** for details.	
`pow`	`pow(`*`x`*`, `*`y`*`)` Returns `float(`*`x`*`)**float(`*`y`*`)`. For large `int` values of *x* and *y*, to avoid `OverflowError` exceptions, use *x*`**`*y* or the `pow` built-in function instead (which does not convert to `floats`).	✓
`prod`	`prod(`*`seq`*`, start=1)` **3.8+** Returns the product of all values in the sequence, beginning with the given `start` value, which defaults to `1`. If *seq* is empty, returns the start value.	✓
`radians`	`radians(`*`x`*`)` Returns the radian measure of the angle *x* given in degrees.	✓

		math
`rect`	`rect(r, phi)` Returns the `complex` value representing the polar coordinates `(r, phi)` converted to rectangular coordinates as `(x + yj)`.	
`remainder`	`remainder(x, y)` Returns the signed remainder from dividing `x/y` (the result may be negative if `x` or `y` is negative).	✓
`sqrt`	`sqrt(x)` Returns the square root of `x`.	✓
`tau`	The mathematical constant $\tau = 2\pi$, or 6.283185307179586.	✓
`trunc`	`trunc(x)` Returns `x` truncated to an `int`.	✓
`ulp`	`ulp(x)` **3.9+** Returns the least significant bit of floating-point value `x`. For positive values, equals `math.nextafter(x, x+1) - x`. For negative values, equals `ulp(-x)`. If `x` is `NaN` or `inf`, returns `x`. `ulp` stands for ***unit of least precision***.	✓

a Formally, `m` is the significand, and `e` is the exponent. Used to render a cross-pla table representation of a floating-point value.

The statistics Module

The `statistics` module supplies the class `NormalDist` to perform distribution analytics, and the functions listed in **Table 16-2** to compute common statistics.

Table 16-2. Functions of the `statistics` module (with functions added in versions 3.8 and 3.10)

	3.8+	3.10+
`harmonic_mean`	`fmean`	`correlation`
`mean`	`geometric_mean`	`covariance`
`median`	`multimode`	`linear_regression`
`median_grouped`	`quantiles`	
`median_high`	`NormalDist`	
`median_low`		
`mode`		
`pstdev`		
`pvariance`		
`stdev`		
`variance`		

The **online docs** contain detailed information on the signatures and use of these functions.

The operator Module

The `operator` module supplies functions that are equivalent to Python's operators. These functions are handy in cases where callables must be stored, passed as arguments, or returned as function results. The functions in `operator` have the same names as the corresponding special methods (covered in **"Special Methods"**). Each function is available with two names, with and without "dunder" (leading and trailing double un-

derscores): for example, both `operator.add(a, b)` and `operator.__add__(a, b)` return `a + b`.

Matrix multiplication support has been added for the infix operator `@`, but you must implement it by defining your own `__matmul__`, `__rmatmul__`, and/or `__imatmul__` methods; NumPy currently supports `@` (but, as of this writing, not yet `@=`) for matrix multiplication.

Table 16-3 lists some of the functions supplied by the `operator` module. For detailed information on these functions and their use, see the **online docs**.

Table 16-3. Functions supplied by the `operator` module

Function	Signature	Behaves like
abs	abs(*a*)	abs(*a*)
add	add(*a*, *b*)	*a* + *b*
and_	and_(*a*, *b*)	*a* & *b*
concat	concat(*a*, *b*)	*a* + *b*
contains	contains(*a*, *b*)	*b* **in** *a*
countOf	countOf(*a*, *b*)	*a*.count(*b*)
delitem	delitem(*a*, *b*)	**del** *a*[*b*]
delslice	delslice(*a*, *b*, *c*)	**del** *a*[*b*:*c*]
eq	eq(*a*, *b*)	*a* == *b*
floordiv	floordiv(*a*, *b*)	*a* // *b*
ge	ge(*a*, *b*)	*a* >= *b*

Function	Signature	Behaves like
getitem	getitem(*a*, *b*)	*a*[*b*]
getslice	getslice(*a*, *b*, *c*)	*a*[*b*:*c*]
gt	gt(*a*, *b*)	*a* > *b*
index	index(*a*)	*a*.__index__()
indexOf	indexOf(*a*, *b*)	*a*.index(*b*)
invert, inv	invert(*a*), inv(*a*)	~*a*
is_	is_(*a*, *b*)	*a* **is** *b*
is_not	is_not(*a*, *b*)	*a* **is not** *b*
le	le(*a*, *b*)	*a* <= *b*
lshift	lshift(*a*, *b*)	*a* << *b*
lt	lt(*a*, *b*)	*a* < *b*
matmul	matmul(*m1*, *m2*)	*m1* @ *m2*
mod	mod(*a*, *b*)	*a* % *b*
mul	mul(*a*, *b*)	*a* * *b*
ne	ne(*a*, *b*)	*a* != *b*
neg	neg(*a*)	-*a*
not_	not_(*a*)	**not** *a*

Function	Signature	Behaves like
or_	or_(*a*, *b*)	*a* | *b*
pos	pos(*a*)	+*a*
pow	pow(*a*, *b*)	a ** b
repeat	repeat(*a*, *b*)	*a* * *b*
rshift	rshift(*a*, *b*)	*a* >> *b*
setitem	setitem(*a*, *b*, *c*)	*a*[*b*] = *c*
setslice	setslice(*a*, *b*, *c*, *d*)	*a*[*b*:*c*] = *d*
sub	sub(*a*, *b*)	*a* - *b*
truediv	truediv(*a*, *b*)	*a*/*b* # *no truncation*
truth	truth(*a*)	bool(*a*), **not not** a
xor	xor(*a*, *b*)	*a* ^ *b*

The operator module also supplies additional higher-order functions, listed in **Table 16-4**. Three of these functions, attrgetter, itemgetter, and methodcaller, return functions suitable for passing as named argument key to the sort method of lists, the sorted, min, and max built-in functions, and several functions in standard library modules, such as heapq and itertools (discussed in **Chapter 8**).

Table 16-4. Higher-order functions supplied by the operator module

attrgetter	attrgetter(*attr*), attrgetter(**attrs*) Returns a callable *f* such that *f*(*o*) is the same as

	`getattr(o, attr)`. The string *attr* can include dots (`.`), in which case the callable result of `attrgetter` calls `getattr` repeatedly. For example, `operator.attrgetter('a.b')` is equivalent to `lambda o: getattr(getattr(o, 'a'), 'b')`. When you call `attrgetter` with multiple arguments, the resulting callable extracts each attribute thus named and returns the resulting tuple of values.

`itemgetter`	`itemgetter(key)`, `itemgetter(*keys)`
	Returns a callable *f* such that *f*(*o*) is the same as `getitem(o, key)`.
	When you call `itemgetter` with multiple arguments, the resulting callable extracts each item thus keyed and returns the resulting tuple of values.
	For example, say that *L* is a list of lists, with each sublist at least three items long: you want to sort *L*, in place, based on the third item of each sublist, with sublists having equal third items sorted by their first items. The simplest way to do this is:
	`L.sort(key=operator.itemgetter(2, 0))`

`length_hint`	`length_hint(`*`iterable`*`, default=0)` Used to try to preallocate storage for items in *`iterable`*. Calls object *`iterable`*'s `__len__` method to try to get an exact length. If `__len__` is not implemented, then Python tries calling *`iterable`*'s `__length_hint__` method. If this is also not implemented, `length_hint` returns the given `default`. Any mistake in using this "hint" helper may result in a performance issue, but not in silent, incorrect behavior.
`method` `caller`	`methodcaller(`*`methodname`*`, args...)` Returns a callable *`f`* such that *`f`*`(`*`o`*`)` is the same as `o.`*`methodname`*`(args, ...)`. The optional `args` may be given as positional or named arguments.

Random and Pseudorandom Numbers

The `random` module of the standard library generates pseudorandom numbers with various distributions. The underlying uniform pseudorandom generator uses the powerful, popular **Mersenne Twister algorithm**, with a (huge!) period of length $2^{19937}-1$.

The random Module

All functions of the `random` module are methods of one hidden global instance of the class `random.Random`. You can instantiate `Random` explicitly to get multiple generators that do not share state. Explicit instantiation is advisable if you require random numbers in multiple threads (threads are covered in **Chapter 15**). Alternatively, instantiate `SystemRandom` if you require higher-quality random numbers (see the following section for details). **Table 16-5** documents the most frequently used functions exposed by the `random` module.

Table 16-5. Useful functions supplied by the `random` module

`choice`	`choice(`*`seq`*`)` Returns a random item from nonempty sequence *`seq`*.
`choices`	`choices(`*`seq`*`, `*`weights`*`=None, *, cum_weights=None, k=1)` Returns `k` elements from nonempty sequence *`seq`*, with replacement. By default, elements are chosen with equal probability. If the optional *`weights`*, or the named argument `cum_weights`, is passed (as a list of `floats` or `ints`), then the respective choices are weighted by that amount during choosing. The `cum_weights` argument accepts a list of `floats` or `ints` as would be returned by `itertools.accumulate(`*`weights`*`)`; e.g., if *`weights`* for a *`seq`* containing three items were `[1, 2, 1]`, then the corresponding `cum_weights` would be `[1, 3, 4]`. (Only one of *`weights`* or `cum_weights` may be specified, and must be the same length as *`seq`*. If used, `cum_weights` and `k` must be given as named arguments.)
`getrandbits`	`getrandbits(`*`k`*`)` Returns an `int >= 0` with *`k`* random bits, like `randrange(2 **` *`k`*`)` (but faster, and with no problems for large *`k`*).
`getstate`	`getstate()` Returns a hashable and pickleable object *`S`* representing the current state of the generator. You can later pass *`S`* to the function `setstate` to restore the generator's state.
`jumpahead`	`jumpahead(`*`n`*`)` Advances the generator state as if *`n`* random numbers

	had been generated. This is faster than generating and ignoring *n* random numbers.
`randbytes`	`randbytes(`*`k`*`)` 3.9+ Generates *k* random bytes. To generate bytes for secure or cryptographic applications, use `secrets.randbits(`*`k`*` * 8)`, then unpack the `int` it returns into *k* bytes, using `int.to_bytes(`*`k`*`, 'big')`.
`randint`	`randint(`*`start`*`, `*`stop`*`)` Returns a random `int` *i* from a uniform distribution such that *start* <= *i* <= *stop*. Both endpoints are included: this is quite unnatural in Python, so you would normally prefer `randrange`.
`random`	`random()` Returns a random `float` *r* from a uniform distribution, `0 <=` *r* `< 1`.
`randrange`	`randrange([`*`start`*`,]`*`stop`*`[,`*`step`*`])` Like `choice(range(`*`start`*`, `*`stop`*`, `*`step`*`))`, but much faster.
`sample`	`sample(`*`seq`*`, `*`k`*`)` Returns a new list whose *k* items are unique items randomly drawn from *seq*. The list is in random order, so that any slice of it is an equally valid random sample. *seq* may contain duplicate items. In this case, each occurrence of an item is a candidate for selection in the sample, and the sample may also contain such duplicates.
`seed`	`seed(`*`x`*`=`**`None`**`)` Initializes the generator state. *x* can be any `int`, `float`, `str`, `bytes`, or `bytearray`. When *x* is **`None`**, and when the module `random` is first loaded, `seed` uses the

	current system time (or some platform-specific source of randomness, if any) to get a seed. *x* is normally an `int` up to 2^{256}, a `float`, or a `str`, `bytes`, or `bytearray` up to 32 bytes in size.[a] Larger *x* values are accepted, but may produce the same generator state as smaller ones. `seed` is useful in simulation or modeling for repeatable runs, or to write tests that require a reproducible sequence of random values.
`setstate`	`setstate(`*S*`)` Restores the generator state. *S* must be the result of a previous call to `getstate` (such a call may have occurred in another program, or in a previous run of this program, as long as object *S* has correctly been transmitted, or saved and restored).
`shuffle`	`shuffle(`*seq*`)` Shuffles, in place, mutable sequence *seq*.
`uniform`	`uniform(`*a, b*`)` Returns a random floating-point number *r* from a uniform distribution such that *a* <= *r* < *b*.

[a] As defined in the Python language specification. Specific Python implementations may support larger seed values for generating unique random number sequences.

The `random` module also supplies several other functions that generate pseudorandom floating-point numbers from other probability distributions (Beta, Gamma, exponential, Gauss, Pareto, etc.) by internally calling `random.random` as their source of randomness. See the **online docs** for details.

Crypto-Quality Random Numbers: The secrets

Module

Pseudorandom numbers provided by the `random` module, while sufficient for simulation and modeling, are not of cryptographic quality. To get random numbers and sequences for use in security and cryptography applications, use the functions defined in the `secrets` module. These functions use the `random.SystemRandom` class, which in turn calls `os.urandom`. `os.urandom` returns random bytes, read from physical sources of random bits such as */dev/urandom* on older Linux releases, or the **`getrandom()`** syscall on Linux 3.17 and above. On Windows, `os.urandom` uses cryptographical-strength sources such as the `CryptGenRandom` API. If no suitable source exists on the current system, `os.urandom` raises `NotImplementedError`.

SECRETS FUNCTIONS CANNOT BE RUN WITH A KNOWN SEED

Unlike the `random` module, which includes a `seed` function to support generation of repeatable sequences of random values, the `secrets` module has no such capability. To write tests dependent on specific sequences of random values generated by the `secrets` module functions, developers must emulate those functions with their own mock versions.

The `secrets` module supplies the functions listed in **Table 16-6**.

Table 16-6. Functions of the `secrets` module

`choice`	`choice(`*`seq`*`)` Returns a randomly selected item from nonempty sequence *`seq`*.
`randbelow`	`randbelow(`*`n`*`)` Returns a random int *x* in the range `0 <=` *x* `<` *n*.
`randbits`	`randbits(`*`k`*`)` Returns an `int` with *k* random bits.

`token_bytes`	`token_bytes(`*n*`)` Returns a `bytes` object of *n* random bytes. When you omit *n*, uses a default value, usually `32`.
`token_hex`	`token_hex(`*n*`)` Returns a string of hexadecimal characters from *n* random bytes, with two characters per byte. When you omit *n*, uses a default value, usually `32`.
`token_urlsafe`	`token_urlsafe(`*n*`)` Returns a string of Base64-encoded characters from *n* random bytes; the resulting string's length is approximately 1.3 times *n*. When you omit *n*, uses a default value, usually `32`.

Additional recipes and best cryptographic practices are provided in Python's **online documentation**.

Alternative sources of physically random numbers are available online, e.g. from **Fourmilab**.

The fractions Module

The `fractions` module supplies a rational number class, `Fraction`, whose instances you can construct from a pair of integers, another rational number, or a `str`. `Fraction` class instances have read-only attributes `numerator` and `denominator`. You can pass a pair of (optionally signed) `int`s as the *numerator* and *denominator*. A denominator of `0` raises `ZeroDivisionError`. A string can be of the form `'3.14'`, or can include an optionally signed numerator, a slash (`/`) , and a denominator, such as `'-22/7'`.

FRACTION REDUCES TO LOWEST TERMS

`Fraction` reduces the fraction to the lowest terms—for example, `f = Fraction(226, 452)` builds an instance `f` equal to one built by `Fraction(1, 2)`. The specific numerator and denominator originally passed to `Fraction` are not recoverable from the resulting instance.

`Fraction` also supports construction from `decimal.Decimal` instances, and from `floats` (the latter may not provide the results you expect, given `floats`' bounded precision). Here are some examples of using `Fraction` with various inputs.

```
>>> from fractions import Fraction
>>> from decimal import Decimal
>>> Fraction(1,10)
```

```
Fraction(1, 10)
```

```
>>> Fraction(Decimal('0.1'))
```

```
Fraction(1, 10)
```

```
>>> Fraction('0.1')
```

```
Fraction(1, 10)
```

```
>>> Fraction('1/10')
```

```
Fraction(1, 10)
```

```
>>> Fraction(0.1)
```

```
Fraction(3602879701896397, 36028797018963968)
```

```
>>> Fraction(-1, 10)
```

```
Fraction(-1, 10)
```

```
>>> Fraction(-1,-10)
```

```
Fraction(1, 10)
```

The `Fraction` class supplies methods including `limit_denominator`, which allows you to create a rational approximation of a `float`—for example, `Fraction(0.0999).limit_denominator(10)` returns `Fraction(1, 10)`. `Fraction` instances are immutable and can be keys in `dicts` or members of `sets`, as well as being usable in arithmetic operations with other numbers. See the **online docs** for complete coverage.

The `fractions` module also supplies a function `gcd` that's just like `math.gcd`, covered in **Table 16-1**.

The decimal Module

A Python `float` is a binary floating-point number, normally according to the standard known as IEEE 754 implemented in hardware in modern computers. An excellent, concise, practical introduction to floating-point arithmetic and its issues can be found in David Goldberg's paper **"What Every Computer Scientist Should Know About Floating-Point Arithmetic"**. A Python-focused essay on the same issues is part of the **tutorial** in the Python docs; another excellent summary (not focused on Python), Bruce Bush's "The Perils of Floating Point," is also available **online**.

Often, particularly for money-related computations, you may prefer to use *decimal* floating-point numbers. Python supplies an implementation of the standard known as IEEE 854,[1] for base 10, in the standard library module `decimal`. The module has excellent **documentation**: there, you can find complete reference material, pointers to the applicable standards, a tutorial, and advocacy for `decimal`. Here, we cover only a small subset of `decimal`'s functionality, the most frequently used parts of the module.

The `decimal` module supplies a `Decimal` class (whose immutable instances are decimal numbers), exception classes, and classes and functions to deal with the *arithmetic context*, which specifies such things as precision, rounding, and which computational anomalies (such as division by zero, overflow, underflow, and so on) raise exceptions when they occur. In the default context, precision is 28 decimal digits, rounding is "half-even" (round results to the closest representable decimal number; when a result is exactly halfway between two such numbers, round to the one whose last digit is even), and the anomalies that raise exceptions are invalid operation, division by zero, and overflow.

To build a decimal number, call `Decimal` with one argument: an `int`, `float`, `str`, or `tuple`. If you start with a `float`, Python converts it losslessly to the exact decimal equivalent (which may require 53 digits or more of precision):

```
>>> from decimal import Decimal
>>> df = Decimal(0.1)
>>> df
```

```
Decimal('0.1000000000000000055511151231257827021181583404541015625')
```

If this is not the behavior you want, you can pass the `float` as a `str`; for example:

```
>>> ds = Decimal(str(0.1))  # or, more directly, Decimal('0.1')
>>> ds
```

```
Decimal('0.1')
```

You can easily write a factory function for ease of interactive experimentation with `decimal`:

```
def dfs(x):
    return Decimal(str(x))
```

Now `dfs(0.1)` is just the same thing as `Decimal(str(0.1))`, or `Decimal('0.1')`, but more concise and handier to write.

Alternatively, you may use the `quantize` method of `Decimal` to construct a new decimal by rounding a `float` to the number of significant digits you specify:

```
>>> dq = Decimal(0.1).quantize(Decimal('.00'))
>>> dq
```

```
Decimal('0.10')
```

If you start with a tuple, you need to provide three arguments: the sign (`0` for positive, `1` for negative), a tuple of digits, and the integer exponent:

```
>>> pidigits = (3, 1, 4, 1, 5)
>>> Decimal((1, pidigits, -4))
```

```
Decimal('-3.1415')
```

Once you have instances of `Decimal`, you can compare them, including comparison with `floats` (use `math.isclose` for this); `pickle` and `unpickle` them; and use them as keys in dictionaries and as members of sets. You may also perform arithmetic among them, and with integers, but not with `floats` (to avoid unexpected loss of precision in the results), as shown here:

```
>>> import math
>>> from decimal import Decimal
>>> a = 1.1
>>> d = Decimal('1.1')
>>> a == d
```

```
False
```

```
>>> math.isclose(a, d)
```

```
True
```

```
>>> a + d
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'float' and
'decimal.Decimal'
```

```
>>> d + Decimal(a) # new decimal constructed from 'a'
```

```
Decimal('2.200000000000000088817841970')
```

```
>>> d + Decimal(str(a)) # convert 'a' to decimal with str(a)
```

```
Decimal('2.20')
```

The **online docs** include useful **recipes** for monetary formatting, some trigonometric functions, and a list of FAQs.

Array Processing

You can represent what most languages call arrays, or vectors, with lists (covered in **"Lists"**), as well as with the `array` standard library module (covered in the following subsection). You can manipulate arrays with loops, indexing and slicing, list comprehensions, iterators, generators, and genexps (all covered in **Chapter 3**); built-ins such as `map`, `reduce`, and `filter` (all covered in **"Built-in Functions"**); and standard library modules such as `itertools` (covered in **"The itertools Module"**). If you only need a lightweight, one-dimensional array of instances of a simple type,

stick with `array`. However, to process large arrays of numbers, such functions may be slower and less convenient than third-party extensions such as NumPy and SciPy (covered in **"Extensions for Numeric Array Computation"**). When you're doing data analysis and modeling, Pandas, which is built on top of NumPy (but not discussed in this book), might be most suitable.

The array Module

The `array` module supplies a type, also called `array`, whose instances are mutable sequences, like lists. An `array` *a* is a one-dimensional sequence whose items can be only characters, or only numbers of one specific numeric type, fixed when you create *a*. The constructor for `array` is:

`array` **`class`** `array(`*`typecode, init=''`*`, /)`
Creates and returns an `array` object *a* with the given *`typecode`*. *`init`* can be a string (a bytestring, except for *`typecode`* `'u'`) whose length is a multiple of `itemsize`: the string's bytes, interpreted as machine values, directly initialize *a*'s items. Alternatively, *`init`* can be an iterable (of characters when *`typecode`* is `'u'`, otherwise of numbers): each item of the iterable initializes one item of *a*.

`array.array`'s advantage is that, compared to a list, it can save memory when you need to hold a sequence of objects all of the same (numeric or character) type. An `array` object *a* has a one-character, read-only attribute *a*`.typecode`, set on creation, which specifies the type of *a*'s items. **Table 16-7** shows the possible `typecode` values for `array`.

Table 16-7. Type codes for the `array` module

typecode	C type	Python type	Minimum size
`'b'`	`char`	`int`	1 byte

typecode	C type	Python type	Minimum size
'B'	unsigned char	int	1 byte
'u'	unicode char	str (length 1)	See note
'h'	short	int	2 bytes
'H'	unsigned short	int	2 bytes
'i'	int	int	2 bytes
'I'	unsigned int	int	2 bytes
'l'	long	int	4 bytes
'L'	unsigned long	int	4 bytes
'q'	long long	int	8 bytes
'Q'	unsigned long long	int	8 bytes
'f'	float	float	4 bytes
'd'	double	float	8 bytes

MINIMUM SIZE OF TYPECODE 'U'

'u' has an item size of 2 on a few platforms (notably, Windows) and 4 on just about every other platform. You can check the build type of a Python interpreter by using array.array('u').itemsize.

The size, in bytes, of each item of an `array` *a* may be larger than the minimum, depending on the machine's architecture, and is available as the read-only attribute *a*`.itemsize`.

Array objects expose all methods and operations of mutable sequences (as covered in **"Sequence Operations"**), except `sort`. Concatenation with `+` or `+=`, and slice assignment, require both operands to be arrays with the same `typecode`; in contrast, the argument to *a*`.extend` can be any iterable with items acceptable to *a*. In addition to the methods of mutable sequences (`append`, `extend`, `insert`, `pop`, etc.), an `array` object *a* exposes the methods and properties listed in **Table 16-8**.

Table 16-8. Methods and properties of an `array` object `a`

`buffer_info`	*a*`.buffer_info()` Returns a two-item tuple (*address*, *array_length*), where *array_length* is the number of items that you can store in *a*. The size of *a* in bytes is *a*`.buffer_info()[1] *` *a*`.itemsize`.
`byteswap`	*a*`.byteswap()` Swaps the byte order of each item of *a*.
`frombytes`	*a*`.frombytes(`*s*`)` Appends to *a* the bytes, interpreted as machine values, of `bytes` *s*. `len(`*s*`)` must be an exact multiple of *a*`.itemsize`.

`fromfile`	`a.fromfile(f, n)` Reads *n* items, taken as machine values, from file object *f* and appends the items to *a*. *f* should be open for reading in binary mode—typically, mode `'rb'` (see **"Creating a File Object with open"**). When fewer than *n* items are available in *f*, `fromfile` raises `EOFError` after appending the items that are available (so, be sure to catch this in a **`try/except`**, if that's OK in your app!).
`fromlist`	`a.fromlist(L)` Appends to *a* all items of list `L`.
`fromunicode`	`a.fromunicode(s)` Appends to *a* all characters from string *s*. *a* must have typecode `'u'`; otherwise, Python raises `ValueError`.
`itemsize`	`a.itemsize` Property that returns the size, in bytes, of each item in *a*.
`tobytes`	`a.tobytes()` `tobytes` returns the bytes representation of the items in a. For any *a*, `len(a.tobytes()) == len(a)*a.itemsize`. `f.write(a.tobytes())` is the same as `a.tofile(f)`.
`tofile`	`a.tofile(f)` Writes all items of *a*, taken as machine values, to file object *f*. Note that *f* should be open for writing in binary mode—for example, with mode `'wb'`.
`tolist`	`a.tolist()` Creates and returns a list object with the same items as *a*, like `list(a)`.

`tounicode`	*a*`.tounicode()` Creates and returns a string with the same items as *a*, like `''.join(`*a*`)`. *a* must have `typecode 'u'`; otherwise, Python raises `ValueError`.
`typecode`	*a*`.typecode` Property that returns the `typecode` used to create `a`.

Extensions for Numeric Array Computation

As you've seen, Python has great built-in support for numeric processing. The third-party library **SciPy**, and many, *many* other packages, such as **NumPy**, **Matplotlib**, **SymPy**, **Numba**, **Pandas**, **PyTorch**, **CuPy**, and **TensorFlow**, provide even more tools. We introduce NumPy here, then provide a brief description of SciPy and some other packages, with pointers to their documentation.

NumPy

If you need a lightweight, one-dimensional array of numbers, the standard library's `array` module may suffice. If your work involves scientific computing, image processing, multidimensional arrays, linear algebra, or other applications involving large amounts of data, the popular third-party NumPy package meets your needs. Extensive documentation is available **online**; a free PDF of Travis Oliphant's ***Guide to NumPy*** is also available.[2]

NUMPY OR NUMPY?

The docs variously refer to the package as NumPy or Numpy; however, in coding, the package is called `numpy`, and you usually import it with **`import`** `numpy` **`as`** `np`. This section follows those conventions.

NumPy provides the class `ndarray`, which you can **subclass** to add functionality for your particular needs. An `ndarray` object has *n* dimensions of

homogeneous items (items can include containers of heterogeneous types). Each `ndarray` object *a* has a certain number of dimensions (aka *axes*), known as its *rank*. A *scalar* (i.e., a single number) has rank `0`, a *vector* has rank `1`, a *matrix* has rank `2`, and so forth. An `ndarray` object also has a *shape*, which can be accessed as property `shape`. For example, for a matrix *m* with 2 columns and 3 rows, *m*`.shape` is `(3,2)`.

NumPy supports a wider range of **numeric types** (instances of `dtype`) than Python; the default numerical types are `bool_` (1 byte), `int_` (either `int64` or `int32`, depending on your platform), `float_` (short for `float64`), and `complex_` (short for `complex128`).

Creating a NumPy array

There are several ways to create an array in NumPy. Among the most common are:

- With the factory function `np.array`, from a sequence (often a nested one), with *type inference* or by explicitly specifying *dtype*
- With factory functions `np.zeros`, `np.ones`, or `np.empty`, which default to *dtype* `float64`
- With factory function `np.indices`, which defaults to *dtype* `int64`
- With factory functions `np.random.uniform`, `np.random.normal`, `np.random.binomial`, etc., which default to *dtype* `float64`
- With factory function `np.arange` (with the usual *start*, *stop*, *stride*), or with factory function `np.linspace` (with *start*, *stop*, *quantity*) for better floating-point behavior
- By reading data from files with other `np` functions (e.g., CSV with `np.genfromtxt`)

Here are some examples of creating an array using the various techniques just described:

```
>>> import numpy as np
>>> np.array([1, 2, 3, 4])  # from a Python list
```

```
array([1, 2, 3, 4])
```

```
>>> np.array(5, 6, 7)  # a common error: passing items separately (they
                       # must be passed as a sequence, e.g. a list)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: array() takes from 1 to 2 positional arguments, 3 were given
```

```
>>> s = 'alph', 'abet'  # a tuple of two strings
>>> np.array(s)
```

```
array(['alph', 'abet'], dtype='<U4')
```

```
>>> t = [(1,2), (3,4), (0,1)]  # a list of tuples
>>> np.array(t, dtype='float64')  # explicit type designation
```

```
array([[1., 2.],
       [3., 4.],
       [0., 1.]])
```

```
>>> x = np.array(1.2, dtype=np.float16)  # a scalar
>>> x.shape
```

```
()
```

```
>>> x.max()
```

```
1.2
```

```
>>> np.zeros(3)  # shape defaults to a vector
```

```
array([0., 0., 0.])
```

```
>>> np.ones((2,2))  # with shape specified
```

```
array([[1., 1.],
[1., 1.]])
```

```
>>> np.empty(9)  # arbitrary float64s
```

```
array([ 6.17779239e-31, -1.23555848e-30,  3.08889620e-31,
       -1.23555848e-30,  2.68733969e-30, -8.34001973e-31,
            3.08889620e-31, -8.34001973e-31,  4.78778910e-31])
```

```
>>> np.indices((3,3))
```

```
array([[[0, 0, 0],
        [1, 1, 1],
        [2, 2, 2]],
```

```
       [[0, 1, 2],
        [0, 1, 2],
        [0, 1, 2]]])
```

```
>>> np.arange(0, 10, 2)  # upper bound excluded
```

```
array([0, 2, 4, 6, 8])
```

```
>>> np.linspace(0, 1, 5)  # default: endpoint included
```

```
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

```
>>> np.linspace(0, 1, 5, endpoint=False)  # endpoint not included
```

```
array([0. , 0.2, 0.4, 0.6, 0.8])
```

```
>>> np.genfromtxt(io.BytesIO(b'1 2 3\n4 5 6'))  # using a pseudo-file
```

```
array([[1., 2., 3.],
       [4., 5., 6.]])
```

```
>>> with open('x.csv', 'wb') as f:
...     f.write(b'2,4,6\n1,3,5')
...
```

```
11
```

```
>>> np.genfromtxt('x.csv', delimiter=',')  # using an actual CSV file
```

```
array([[2., 4., 6.],
       [1., 3., 5.]])
```

Shape, indexing, and slicing

Each `ndarray` object *a* has an attribute `a.shape`, which is a tuple of `int`s. `len(a.shape)` is *a*'s *rank*; for example, a one-dimensional array of numbers (also known as a *vector*) has rank `1`, and `a.shape` has just one item. More generally, each item of `a.shape` is the length of the corresponding dimension of *a*. *a*'s number of elements, known as its *size*, is the product of all items of `a.shape` (also available as property `a.size`). Each dimension of *a* is also known as an *axis*. Axis indices are from `0` and up, as usual in Python. Negative axis indices are allowed and count from the right, so `-1` is the last (rightmost) axis.

Each array *a* (except a scalar, meaning an array of rank `0`) is a Python sequence. Each item `a[i]` of *a* is a subarray of *a*, meaning it is an array with a rank one less than *a*'s: `a[i].shape == a.shape[1:]`. For example, if *a* is a two-dimensional matrix (*a* is of rank `2`), `a[i]`, for any valid index `i`, is a one-dimensional subarray of *a* that corresponds to one row of the matrix. When *a*'s rank is `1` or `0`, *a*'s items are *a*'s elements (just one element, for rank `0` arrays). Since *a* is a sequence, you can index *a* with normal indexing syntax to access or change *a*'s items. Note that *a*'s items are *a*'s subarrays; only for an array of rank `1` or `0` are the array's *items* the same thing as the array's *elements*.

As with any other sequence, you can also *slice* *a*. After `b = a[i:j]`, *b* has the same rank as *a*, and `b.shape` equals `a.shape` except that `b.shape[0]` is

the length of the slice `a[i:j]`, (i.e., when `a.shape[0] > j >= i >= 0`, the length of the slice is `j - i`, as described in **"Slicing a sequence"**).

Once you have an array *a*, you can call `a.reshape` (or, equivalently, `np.reshape` with *a* as the first argument). The resulting shape must match `a.size`: when `a.size` is `12`, you can call `a.reshape(3, 4)` or `a.reshape(2, 6)`, but `a.reshape(2, 5)` raises `ValueError`. Note that `reshape` does not work in place: you must explicitly bind or rebind the array, for example, `a = a.reshape(i, j)` or `b = a.reshape(i, j)`.

You can also loop on (nonscalar) *a* with **`for`**, just as you can with any other sequence. For example, this:

```
for x in a:
    process(x)
```

means the same thing as:

```
for _ in range(len(a)):
    x = a[_]
    process(x)
```

In these examples, each item *x* of *a* in the **`for`** loop is a subarray of *a*. For example, if *a* is a two-dimensional matrix, each *x* in either of these loops is a one-dimensional subarray of *a* that corresponds to a row of the matrix.

You can also index or slice *a* by a tuple. For example, when *a*'s rank is `>= 2`, you can write `a[i][j]` as `a[i, j]`, for any valid `i` and `j`, for rebinding as well as for access; tuple indexing is faster and more convenient. *Do not put parentheses* inside the brackets to indicate that you are indexing *a* by a tuple: just write the indices one after the other, separated by commas. `a[i, j]` means exactly the same thing as `a[(i, j)]`, but the form without parentheses is more readable.

An indexing is a slicing in which one or more of the tuple's items are slices, or (at most once per slicing) the special form `...` (the Python built-in `Ellipsis`). `...` expands into as many all-axis slices (`:`) as needed to "fill" the rank of the array you're slicing. For example, *a*`[1,...,2]` is like *a*`[1,:,:,2]` when *a*'s rank is 4, but like *a*`[1,:,:,:,2]` when *a*'s rank is 6.

The following snippets show looping, indexing, and slicing:

```
>>> a = np.arange(8)
>>> a
```

```
array([0, 1, 2, 3, 4, 5, 6, 7])
```

```
>>> a = a.reshape(2,4)
>>> a
```

```
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
```

```
>>> print(a[1,2])
```

```
6
```

```
>>> a[:,:2]
```

```
array([[0, 1],
       [4, 5]])
```

```
>>> for row in a:
...     print(row)
...
```

```
[0 1 2 3]
[4 5 6 7]
```

```
>>> for row in a:
...     for col in row[:2]:  # first two items in each row
...         print(col)
...
```

```
0
1
4
5
```

Matrix operations in NumPy

As mentioned in **"The operator Module"**, NumPy implements the operator `@` for matrix multiplication of arrays. *a1* `@` *a2* is like `np.matmul(`*a1*`,` *a2*`)`. When both matrices are two-dimensional, they're treated as conventional matrices. When one argument is a vector, you conceptually promote it to a two-dimensional array, as if by temporarily appending or prepending a `1`, as needed, to its shape. Do not use `@` with a scalar; use `*` instead. Matrices also allow addition (using `+`) with a scalar, as well as with vectors and other matrices of compatible shapes. Dot product is also

available for matrices, using `np.dot(a1, a2)`. A few simple examples of these operators follow:

```
>>> a = np.arange(6).reshape(2,3)  # a 2D matrix
>>> b = np.arange(3)               # a vector
>>>
>>> a
```

```
array([[0, 1, 2],
       [3, 4, 5]])
```

```
>>> a + 1    # adding a scalar
```

```
array([[1, 2, 3],
       [4, 5, 6]])
```

```
>>> a + b    # adding a vector
```

```
array([[0, 2, 4],
       [3, 5, 7]])
```

```
>>> a * 2    # multiplying by a scalar
```

```
array([[ 0,  2,  4],
       [ 6,  8, 10]])
```

```
>>> a * b    # multiplying by a vector
```

```
array([[ 0,  1,  4],
       [ 0,  4, 10]])
```

```
>>> a @ b    # matrix-multiplying by a vector
```

```
array([ 5, 14])
```

```
>>> c = (a*2).reshape(3,2)  # using scalar multiplication to create
>>> c
```

```
array([[ 0,  2],
       [ 4,  6],
       [ 8, 10]])
```

```
>>> a @ c    # matrix-multiplying two 2D matrices
```

```
array([[20, 26],
       [56, 80]])
```

NumPy is rich and powerful enough to warrant whole books of its own; we have only touched on a few details. See the NumPy **documentation** for extensive coverage of its many, many features.

SciPy

Whereas NumPy contains classes and functions for handling arrays, the SciPy library supports more advanced numeric computation. For example, while NumPy provides a few linear algebra methods, SciPy provides advanced decomposition methods and supports more advanced functions, such as allowing a second matrix argument for solving generalized eigenvalue problems. In general, when you are doing advanced numeric computation, it's a good idea to install both SciPy and NumPy.

SciPy.org also hosts **docs** for a number of other packages, which are integrated with SciPy and NumPy, including **Matplotlib**, which provides 2D plotting support; **SymPy**, which supports symbolic mathematics; **Jupyter Notebook**, a powerful interactive console shell and web application kernel; and **Pandas**, which supports data analysis and modeling. You may also want to take a look at `mpmath`, for arbitrary precision, and `sagemath`, for even richer functionality.

Additional numeric packages

The Python community has produced many more packages in the field of numeric processing. A few of them are:

Anaconda
: A consolidated environment that simplifies the installation of Pandas, NumPy, and many related numerical processing, analytical, and visualization packages, and provides package management via its own `conda` package installer.

`gmpy2`
: A module[3] that supports the GMP/MPIR, MPFR, and MPC libraries, to extend and accelerate Python's abilities for multiple-precision arithmetic.

Numba
: A just-in-time compiler to convert Numba-decorated Python functions and NumPy code to LLVM. Numba-compiled numerical algorithms in Python can approach the speeds of C or FORTRAN.

PyTorch
: An open source machine learning framework.

TensorFlow
: A comprehensive machine learning platform that operates at large scale and in mixed environments, using dataflow graphs to represent computation, shared

state, and state manipulation operations. TensorFlow supports processing across multiple machines in a cluster, and within-machine across multicore CPUs, GPUs, and custom-designed ASICs. TensorFlow's main API uses Python.

1 Superseded, technically, by the more recent, very similar standard **754-2008**, but practically still useful!

2 Python and the NumPy project have worked closely together for many years, with Python introducing language features specifically for NumPy (such as the `@` operator and extended slicing) even though such novel language features are not (yet?) used anywhere in the Python standard library.

3 Originally derived from the work of one of this book's authors.