15. Numeric Processing - Python in a Nutshell, 3rd Edition

Chapter 15. 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` in "The math and cmath Modules", `operator` in "The operator Module", `random` in "The random Module", `fractions` in "The fractions Module", and `decimal` in "The decimal Module". "The gmpy2 Module" also mentions the third-party module `gmpy2`, which further extends Python's numeric computation abilities. Numeric processing often requires, more specifically, the processing of *arrays* of numbers, covered in "Array Processing", focusing on the standard library module `array` and popular third-party extension NumPy.

The math and cmath Modules

The `math` module supplies mathematical functions on floating-point numbers; the `cmath` module supplies equivalent functions on 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 many calls to the modules' well-known mathematical functions, it's permissible, as an exception to the general guideline, to use at the top of your code `from math import *`, and afterward just call `sqrt`.

Each module exposes two `float` attributes bound to the values of fundamental mathematical constants, `e` and `pi`, and a variety of functions, including those shown in Table 15-1.

Table 15-1.

acos, asin, atan, cos, sin, tan	`acos(x)` Returns the arccosine, arcsine, arctangent, cosine, sine, or tangent of `x`, respectively, in radians.	math and cmath
acosh, asinh, atanh, cosh, sinh, tanh	`acosh(x)` Returns the arc hyperbolic cosine, arc hyperbolic sine, arc hyperbolic tangent, hyperbolic cosine, hyperbolic sine, or hyperbolic tangent of `x` , respectively, in radians.	math and cmath

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

ceil	`ceil(x)` Returns `float(i)`, where `i` is the lowest integer such that `i>=x`.	math only
e	The mathematical constant *e* (2.718281828459045).	math and cmath
exp	`exp(x)` Returns `e**x`.	math and cmath
erf	`erf(x)` Returns the error function of `x` as used in statistical calculations.	math only
fabs	`fabs(x)` Returns the absolute value of `x`.	math only
factorial	`factorial(x)` Returns the factorial of `x`. Raises `ValueError` when `x` is negative or not integral.	math only
floor	`floor(x)` Returns `float(i)`, where `i` is the lowest integer such that `i<=x`.	math only
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`.	math only
fsum	`fsum(iterable)` Returns the floating-point sum of the values in `iterable` to greater precision than `sum`.	math only

frexp	`frexp(x)` Returns a pair `(m,e)` with the “mantissa” (pedantically speaking, the *significand*) and exponent of `x`. `m` is a floating-point number, and `e` is an integer such that `x==m*(2**e)` and `0.5<=abs(m)<1`, except that `frexp(0)` returns `(0.0,0)`.	math only
gcd	`gcd(x,y)` Returns the greatest common divisor of `x` and `y`. When `x` and `y` are both zero, returns `0`.	math only v3 only
hypot	`hypot(x,y)` Returns `sqrt(x*x+y*y)`.	math only
inf	`inf` A floating-point positive infinity, like `float('inf')`.	math only

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 9 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)`
	
	Don’t use == between floating-point numbers
	
	Given the approximate nature of floating-point arithmetic, it rarely makes sense to check whether two `float`s `x` and `y` are equal: tiny variations in how each was computed can easily result in accidental, minuscule, irrelevant differences. Avoid `x==y`; use `math.isclose(x, y)`.
		math and cmath
		
		v3 only

isfinite	`isfinite(x)` Returns `True` when `x` (in `cmath`, both the real and imaginary part of `x`) is neither infinity nor `NaN`; otherwise, returns `False`.	math and cmath v3 only
isinf	`isinf(x)` Returns `True` when `x` (in `cmath`, either the real or imaginary part of `x`) is positive or negative infinity; otherwise, returns `False`.	math and cmath

isnan	`isnan(x)` Returns `True` when `x` (in `cmath`, either the real or imaginary part of `x`) is `NaN`; otherwise, returns `False`.	math and cmath
ldexp	`ldexp(x,i)` Returns `x*(2**i)` (`i` must be an `int`; when `i` is a `float`, `ldexp` raises `TypeError`).	math only
log	`log(x)` Returns the natural logarithm of `x`.	math and cmath
log10	`log10(x)` Returns the base-10 logarithm of `x`. Also, `log2(x)` (`math` only, v3 only) returns the base-2 logarithm of `x`.	math and cmath
modf	`modf(x)` Returns a pair `(f,i)` with fractional and integer parts of `x`, meaning two `float`s with the same sign as `x` such that `i==int(i)` and `x==f+i`.	math only
nan	`nan` A floating-point "Not a Number" (`NaN`) value, like `float('nan')`.	math only
pi	The mathematical constant *π*, 3.141592653589793.	math and cmath
phase	`phase(x)` Returns the phase of `x`, as a float in the range (-*π*, *π*). Like `math.atan2(x.imag, x.real)`. See "Conversions to and from polar coordinates" in the Python online docs.	cmath only
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 Python online docs.	cmath only
pow	`pow(x,y)` Returns `x**y`.	math only
sqrt	`sqrt(x)` Returns the square root of `x`.	math and cmath
trunc	`trunc(x)` Returns `x` truncated to an `int`.	math only

Always keep in mind that `float`s are not entirely precise, due to their internal representation in the computer. The following example shows this, and also shows why the new function `isclose` may be useful:

```
                              # f is intuitively equal to
>>> f = 1.1 + 2.2 - 3.3  0
>>> f==0
False
>>> f
4.440892098500626e-16
>>> math.isclose(0,f,abs_tol=1e-15)
# abs_tol for near-0
comparison
True
>>> g = f-1
>>> g
-0.9999999999999996  # almost -1 but not
quite
                        # default is fine for this
>>> math.isclose(-1,g)  comparison
True
                                 # but you can set the
>>> isclose(-1,g,rel_tol=1e-15)  tolerances
True
                                 # including higher
>>> isclose(-1,g,rel_tol=1e-16)  precision
False
```

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 underscores): 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 `@`, in v3, but you must (as of this writing) implement it by defining your own `__matmul__()`, `__rmatmul__()`, and/or `__imatmul__()`; NumPy, however, does support `@` (but not yet `@=`) for matrix multiplication.

Table 15-2 lists some of the functions supplied by the `operator` module.

Table 15-2. Functions supplied by the operator module

Method	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]`

Method	Signature	Behaves like
delslice	delslice(a , b , c)	del a[b:c]
div	div(a , b)	a / b
eq	eq(a , b)	a == b
floordiv	floordiv(a , b)	a // b
ge	ge(a , b)	a >= b
getitem	getitem(a , b)	a [b]
getslice	getslice(a , b , c)	a [b : c]
gt	gt(a , b)	a > b
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)	is a 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
or_	or_(a , b)	a | b
pos	pos(a)	+ a
repeat	repeat(a , b)	a * b
rshift	rshift(a , b)	a >> b
setitem	setitem(a , b , c)	a [b]= c

Method	Signature	Behaves like
`setslice`	`setslice(a,b,c,d)`	`a[b:c]=d`
`sub`	`sub(a,b)`	`a - b`
`truediv`	`truediv(a,b)`	`a/b` `# "true" div -> no truncation`
`truth`	`truth(a)`	`not not a`, `bool(a)`
`xor`	`xor(a,b)`	`a ^ b`

The `operator` module also supplies two higher-order functions whose results are functions suitable for passing as named argument `key=` to the `sort` method of lists, the `sorted` built-in function, `itertools.groupby()`, and other built-in functions such as `min` and `max`.

attrgetter `attrgetter(attr)`

Returns a callable `f` such that `f(o)` is the same as `getattr(o,attr)`. The `attr` string 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')`.

`attrgetter(*attrs)`

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)`

Returns a callable `f` such that `f(o)` is the same as `getitem(o,key)`.

`itemgetter(*keys)`

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:

```
import operator
L.sort(key=operator.itemgetter(2, 0))
```

Random and Pseudorandom Numbers

The `random` module of the standard library generates pseudorandom numbers with various distributions. The underlying uniform pseudorandom generator uses the Mersenne Twister algorithm, with a period of length `2**19937-1`.

Physically Random and Cryptographically Strong Random Numbers

Pseudorandom numbers provided by the `random` module, while very good, are not of cryptographic quality. If you want higher-quality random numbers, you can call `os.urandom` (from the module `os`, *not* `random`), or instantiate the class `SystemRandom` from `random` (which calls `os.urandom` for you).

urandom	`urandom(n)` Returns `n` random bytes, read from physical sources of random bits such as */dev/urandom* on older Linux releases. In v3 only, uses the `getrandom()` syscall on Linux 3.17 and above. (On OpenBSD 5.6 and newer, the C `getrandom()` function is now used.) Uses cryptographical-strength sources such as the `CryptGenRandom` API on Windows. If no suitable source exists on the current system, `urandom` raises `NotImplementedError`.

An alternative source of physically random numbers: *http://www.fourmilab.ch/hotbits*.

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 14). Alternatively, instantiate `SystemRandom` if you require higher-quality random numbers. (See "Physically Random and Cryptographically Strong Random Numbers".) This section documents the most frequently used functions exposed by module `random`:

choice	`choice(seq)` Returns a random item from nonempty sequence `seq`.
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 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.
randint	`randint(`*`start`*`, `*`stop`*`)` Returns a random `int i` from a uniform distribution such that *`start`*`<=i<=`*`stop`*. Both end-points are included: this is quite unnatural in Python, so `randrange` is usually preferred.
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 hashable object. 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 integer up to `27814431486575`. Larger `x` values are accepted, but may produce the same generator state as smaller ones.
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(alist)` Shuffles, in place, mutable sequence `alist`.
uniform	`uniform(a,b)` Returns a random floating-point number `r` from a uniform distribution such that `a<=r<b`.

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.

The fractions Module

The `fractions` module supplies a rational number class called `Fraction` whose instances can be constructed from a pair of integers, another rational number, or a string. You can pass a pair of (optionally signed) integers: the *numerator* and *denominator*. When the denominator is 0, a `ZeroDivisionError` is raised. 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` also supports construction from `decimal.Decimal` instances, and from `float`s (although the latter may not provide the result you'd expect, given `float`s' bounded precision). `Fraction` class instances have the properties `numerator` and `denominator`.

Reduced 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 numerator and denominator originally passed to `Fraction` are not recoverable from the built instance.

```
from fractions import                          Fraction(1,
Fraction                     >>> Fraction(1,10)10)            >>> Fraction(Decimal('
      Fraction(1,                         Fraction(1,
0.1'))10)            >>> Fraction('0.1')10)            >>> Fraction('1/10')
Fraction(1,                     Fraction(3602879701896397,
10)             >>> Fraction(0.1)36028797018963968)                           >>>
                Fraction(-1,                        Fraction(1,
Fraction(-1, 10)10)             >>> Fraction(-1,-10)10)
```

`Fraction` also supplies several 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 dictionaries and members of sets, as well as being used in arithmetic operations with other numbers. See the `fractions` docs for more complete coverage.

The `fractions` module, in both v2 and v3, also supplies a function called `gcd` that works just like `math.gcd` (which exists in v3 only), covered in Table 15-1.

The decimal Module

A Python `float` is a binary floating-point number, normally in accordance with the standard known as IEEE 754 and implemented in hardware in modern computers. A concise, practical introduction to floating-point arithmetic and its issues can be found in David Goldberg's essay What Every Computer Scientist Should Know about Floating-Point Arithmetic. A Python-focused essay on the same issues is part of the online tutorial; another excellent summary is also 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, for base 10, in the standard library module `decimal`. The module has excellent documentation for both v2 and v3: there you can find complete reference documentation, pointers to the applicable standards, a tutorial, and advocacy for `decimal`. Here, we cover only a small subset of `decimal`'s functionality that corresponds to 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 integer, float, string, or tuple. If you start with a `float`, it is converted losslessly to the exact decimal equivalent (which may require 53 digits or more of precision):

```
from decimal import Decimaldf = Decimal(0.1)dfDecimal('
0.1000000000000000055511151231257827021181583404541015625')
```

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

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

If you wish, you can easily write a factory function for ease of experimentation, particularly 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'))dqDecimal('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 `float`s (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 `float`s (to avoid unexpected loss of precision in the results), as demonstrated here:

```
>>> a = 1.1>>> d = Decimal('1.1')>>> a == dFalse>>> math.isclose(a, d)True>>> a + d
Traceback (most recent call
last):                                  File "<stdin>", line 1, in <module>TypeError:
unsupported operand type(s) for +:   'decimal.Decimal' and 'float'>>> d + Decimal(a)
                                    Decimal('2.200000000000000088817841970') #
# new decimal constructed from awhoops                                           >>>
                                    to decimal with
d + Decimal(str(a)) # convert astr(                 a)Decimal('2.20')
```

The online docs include useful recipes for monetary formatting, some trigonometric functions, and a list of Frequently Asked Questions (FAQ).

The gmpy2 Module

The `gmpy2` module is a C-coded extension that supports the GMP, MPFR, and MPC libraries, to extend and accelerate Python's abilities for multiple-precision arithmetic (arithmetic in which the precision of the numbers involved is bounded only by the amount of memory available). The main development branch of `gmpy2` supports thread-safe contexts. You can download and install `gmpy2` from PyPI.

Array Processing

You can represent arrays with lists (covered in "Lists"), as well as with the `array` standard library module (covered in "The array Module"). You can manipulate arrays with loops; indexing and slicing; list comprehensions; iterators; generators; 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, 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, 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`.

`array.array`'s advantage is that, compared to a list, it can save memory to hold 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: the type code of `a`'s items. Table 15-3 shows the possible type codes for `array`.

Table 15-3. Type codes for the array module

typecode	C type	Python type	Minimum size
`'c'`	`char`	`str (length 1)`	1 byte (v2 only)
`'b'`	`char`	`int`	1 byte
`'B'`	`unsigned char`	`int`	1 byte
`'u'`	`unicode char`	`unicode (length 1)`	2 bytes (4 if this Python is a "wide build")
`'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 (v3 only)

typecode	C type	Python type	Minimum size
`'Q'`	`unsigned long long`	`int`	8 bytes (v3 only)
`'f'`	`float`	`float`	4 bytes
`'d'`	`double`	`float`	8 bytes

Note

Note: `'c'` is v2 only. `'u'` is in both v2 and v3, with an item size of `2` if this Python is a "narrow build," and `4` if a "wide build." `q` and `Q` (v3 only) are available only if the platform supports C's `long long` (or, on Windows, `__int64`) type.

The size in bytes of each item may be larger than the minimum, depending on the machine's architecture, and is available as the read-only attribute `a.itemsize`. The module `array` supplies just the type object called `array`:

array `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 chars when `typecode` is `'c'` or `'u'`, otherwise of numbers): each item of the iterable initializes one item of `a`.

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, an array object `a` exposes the following methods.

byteswap `a.byteswap()`

Swaps the byte order of each item of `a`.

fromfile `a.fromfile(f,n)`

Reads `n` items, taken as machine values, from file object `f` and appends the items to `a`. Note that `f` should be open for reading in binary mode—for example, with mode `'rb'`. When fewer than `n` items are available in `f`, `fromfile` raises `EOFError` after appending the items that are available.

fromlist `a.fromlist(L)`

Appends to `a` all items of list `L`.

fromstring, frombytes `a.fromstring(s)` `a.frombytes(s)`

`fromstring` (v2 only) appends to `a` the bytes, interpreted as machine values, of string `s`. `len(s)` must be an exact multiple of `a.itemsize`. `frombytes` (v3 only) is identical (reading `s` as bytes).

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)`.
tostring, tobytes	`a.tostring() a.tobytes()` `tostring` (v2 only) returns the string with the bytes from all items of `a`, taken as machine values. For any `a`, `len(a.tostring())==len(a)*a.itemsize`. `f.write(a.tostring())` is the same as `a.tofile(f)`. `tobytes` (v3 only), similarly, returns the bytes representation of the array items.

Extensions for Numeric Array Computation

As you've seen, Python has great support for numeric processing. However, third-party library SciPy and packages such as NumPy, Matplotlib, Sympy, IPython/Jupyter, and pandas provide even more tools. We introduce NumPy here, then provide a brief description of SciPy and other packages (see "SciPy"), with pointers to their documentation.

NumPy

If you need a lightweight one-dimensional array of numbers, the standard library's `array` module may often suffice. If you are doing scientific computing, advanced image handling, 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 book is also available.

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`. In this section, we use all of these monikers.

`NumPy` provides class `ndarray`, which you can subclass to add functionality for your particular needs. An `ndarray` object has `n` dimensions of homogenous items (items can include containers of heterogenous types). An `ndarray` object `a` has a 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; however, the default numerical types are: `bool_`, one 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 `zeros`, `ones`, `empty`, which default to `dtype float64`, and `indices`, which defaults to `int64`
- with factory function `arange` (with the usual *start*, *stop*, *stride*), or with factory function `linspace` (*start*, *stop*, *quantity*) for better floating-point behavior
- reading data from files with other `np` functions (e.g., CSV with `genfromtxt`)

Here are examples of creating an array, as just listed:

```
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>
ValueError: only 2 non-keyword arguments accepted

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()
1.2002

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([  4.94065646e-324,   9.88131292e-324,   1.48219694e-323,
         1.97626258e-323,   2.47032823e-323,   2.96439388e-323,
         3.45845952e-323,   3.95252517e-323,   4.44659081e-323])

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])

import io
np.genfromtxt(io.BytesIO(b'1 2 3\n4 5 6'))  # using a pseudo-file
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])

with io.open('x.csv', 'wb') as f:
    f.write(b'2,4,6\n1,3,5')
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 a 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 for 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 `i:j` (`j-i` when `a.shape[0]>j>=i>=0`, and so on).

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—that is, `a = a.reshape(i,j)` or `b = a.reshape(i,j)`.

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

```
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 the same thing as `a[(i,j)]`, but the form without parentheses is more readable.

An indexing is a slicing when one or more of the tuple's items are slices, or (at most once per slicing) the special form `...` (also available, in v3 only, as 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)aarray([0, 1, 2, 3, 4, 5, 6, 7])a = a.reshape(2,4)aarray([[0, 1, 2, 3],
       [4, 5, 6, 7]])          print(a[1,2])6a[:,: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)0145
```

Matrix Operations in NumPy

As mentioned in "The operator Module", `NumPy` implements the new 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 promote it to a two-dimensional array, by temporarily appending or prepending a 1, as needed, to its shape. Do not use `@` with a scalar; use `*` instead (see the following example). Matrices also allow addition (using `+`) with a scalar (see example), as well as with vectors and other matrices (shapes must be compatible). Dot product is also available for matrices, using `np.dot(a1, a2)`. A few simple examples of these operators follow:

```
                              # a 2-d                                        # a
a = np.arange(6).reshape(2,3)  matrix         b = np.arange(3)                vector    a
                                          # adding a
array([[0, 1, 2],       [3, 4, 5]])a + 1    scalar          array([[1, 2, 3],       [4,
              # adding a
 5, 6]])a + b    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
vector                   array([ 5, 14])c = (a*2).reshape(3,2)
# using scalar multiplication to                         # another
create                               c                    matrix         array([[
                                              # matrix multiplying two 2-d
 0,  2],       [ 4,  6],       [ 8, 10]])a @ c    matrices
array([[20, 26],       [56, 80]])
```

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

SciPy

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

SciPy.org also hosts the documentation for a number of other packages, which are integrated with SciPy and NumPy: Matplotlib, which provides 2D plotting support; Sympy, which supports symbolic mathematics; IPython, a powerful interactive console shell and web-application kernel (the latter now blossoming as the Jupyter project); and pandas, which supports data analysis and modeling (you can find the pandas tutorials here, and many books and other materials here). Finally, if you’re interested in Deep Learning, consider using the open source TensorFlow, which has a Python API.

Specifically in Python 3.5

Note that `fromstring` and `tostring`, in v2, are renamed to `frombytes` and `tobytes` in v3 for clarity—`str` in v2 was bytes; in v3, `str` is Unicode.

Since Python 3.5