7. Core Built-ins and Standard Library Modules

Chapter 7. Core Built-ins and Standard Library Modules

The term *built-in* has more than one meaning in Python. In most contexts, a *built-in* means an object directly accessible to Python code without an `import` statement. "Python built-ins" shows the mechanism that Python uses to allow this direct access. Built-in types in Python include numbers, sequences, dictionaries, sets, functions (all covered in Chapter 3), classes (covered in "Python Classes"), standard exception classes (covered in "Exception Objects"), and modules (covered in "Module Objects"). "The io Module" covers the (built-in, in v2) `file` type, and "Internal Types" covers some other built-in types intrinsic to Python's internal operation. This chapter provides additional coverage of core built-in types (in "Built-in Types") and covers built-in functions available in the module `builtins` (named `__builtins__` in v2) in "Built-in Functions".

As mentioned in "Python built-ins", some modules are known as "built-in" because they are an integral part of the Python standard library (even though it takes an `import` statement to access them), as distinguished from separate, optional add-on modules, also called Python *extensions*. This chapter covers some *core* built-in modules: namely, the modules `sys` in "The sys Module", `copy` in "The copy Module", `collections` in "The collections Module", `functools` in "The functools Module", `heapq` in "The heapq Module", `argparse` in "The argparse Module", and `itertools` in "The itertools Module". Chapter 8 covers some string-related core built-in modules (`string` in "The string Module", `codecs` in "The codecs Module", and `unicodedata` in "The unicodedata Module"), and Chapter 9 covers `re` in "Regular Expressions and the re Module". Parts III and IV of this book cover other Python standard library modules.

Built-in Types

Table 7-1 covers Python's core built-in types, such as `int`, `float`, `dict`, and many others. More details about many of these types, and about operations on their instances, are found throughout Chapter 3. In this section, by "number" we mean, specifically, "noncomplex number."

Table 7-1. Core built-in types

bool	`bool(x=False)` Returns `False` if `x` evaluates as false; returns `True` if `x` evaluates as true. (See "Boolean Values".) `bool` extends `int`: built-in names `False` and `True` refer to the only two instances of `bool`. These instances are also `int`s, equal to `0` and `1`, respectively, but `str(True)` is `'True'`, `str(False)` is `'False'`.
bytearray	`bytearray(x=b''[,codec[,errors]])` A mutable sequence of *bytes* (`int`s with values from `0` to `255`), supporting the usual methods of mutable sequences, plus methods of `str` (covered in Table 7-1). When `x` is a `str` in v3, or a `unicode` instance in v2, you must also pass `codec` and may pass `errors`; the result is like calling `bytearray(x.encode(codec,errors))`. When `x` is an `int`, it must be `>=0`: the resulting instance has a length of `x` and each item is initialized to `0`. When `x` conforms to the `buffer` interface, the read-only buffer of bytes from `x` initializes the instance. Otherwise, `x` must be an iterable yielding `int`s `>=0` and `<256`, which initialize the instance. For example, `bytearray([1,2,3,4])==bytearray(b'\x01\x02\x03\x04')`.

bytes

`bytes(x=b''[,codec[,errors]])`

In v2, a synonym of `str`. In v3, an immutable sequence of *bytes*, with the same nonmutating methods, and the same initialization behavior, as `bytearray`. *Beware*: `bytes(2)` is `b'\x00\x00'` in v3, but `'2'` in v2!

complex

`complex(real=0,imag=0)`

Converts any number, or a suitable string, to a complex number. `imag` may be present only when `real` is a number, and in that case it is the imaginary part of the resulting complex number. See also “Complex numbers”.

dict

`dict(x={})`

Returns a new dictionary with the same items as `x`. (Dictionaries are covered in “Dictionaries”.) When `x` is a `dict`, `dict(x)` returns a shallow copy of `x`, like `x.copy()`. Alternatively, `x` can be an iterable whose items are pairs (iterables with two items each). In this case, `dict(x)` returns a dictionary whose keys are the first items of each pair in `x`, and whose values are the corresponding second items. In other words, when `x` is a sequence, `c = dict(x)` is equivalent to:

```
c = {}
for key, value in x: c[key] = value
```

You can call `dict` with named arguments, in addition to, or instead of, positional argument `x`. Each named argument becomes an item in the dictionary, with the name as the key: it might overwrite an item from `x`.

float

`float(x=0.0)`

Converts any number, or a suitable string, to a floating-point number. See “Floating-point numbers”.

frozenset

`frozenset(seq=())`

Returns a new frozen (i.e., immutable) set object with the same items as iterable `seq`. When `seq` is a frozen set, `frozenset(seq)` returns `seq` itself, like `seq.copy()`. See “Set Operations”.

int

`int(x=0,radix=10)`

Converts any number, or a suitable string, to an `int`. When `x` is a number, `int` truncates toward `0`, dropping any fractional part. `radix` may be present only when `x` is a string: then, `radix` is the conversion base, between `2` and `36`, with `10` as the default. `radix` can be explicitly passed as `0`: the base is then `2`, `8`, `10`, or `16`, depending on the form of string `x`, just like for integer literals, as covered in “Integer numbers”.

list

`list(seq=())`

Returns a new list object with the same items as iterable `seq`, in the same order. When `seq` is a list, `list(seq)` returns a shallow copy of `seq`, like `seq[:]`. See “Lists”.

memoryview	`memoryview(x)` `x` must be an object supporting the buffer interface (for example, `bytes` and `bytearray` do; in v3 only, so do `array` instances, covered in "The array Module"). `memoryview` returns an object `m` "viewing" exactly the same underlying memory as `x`, with items of `m.itemsize` bytes each (always `1`, in v2); `len(m)` is the number of items. `m` can be indexed (returning, in v2, a `str` instance of length `1`; in v3, an `int`) and sliced (returning another instance of `memoryview` "viewing" the appropriate subset of the same underlying memory). `m` is mutable if `x` is (but `m`'s size cannot be changed, so a slice assignment must be from a sequence of the same length as the slice getting assigned). `m` supplies several read-only attributes and v3-only methods; see the online docs for details. Two useful methods available in both v2 and 3 are `m.tobytes()` (returns `m`'s data as an instance of `bytes`) and `m.tolist()` (returns `m`'s data as a list of `int`s).
object	`object()` Returns a new instance of `object`, the most fundamental type in Python. Direct instances of type `object` have no functionality: only use of such instances is as "sentinels"—that is, objects comparing `!=` to any distinct object.
set	`set(seq=())` Returns a new mutable set object with the same items as the iterable object `seq`. When `seq` is a set, `set(seq)` returns a shallow copy of `seq`, like `seq.copy`(). See "Sets".
slice	`slice([start,]stop[,step])` Returns a slice object with the read-only attributes `start`, `stop`, and `step` bound to the respective argument values, each defaulting to `None` when missing. For positive indices, such a slice signifies the same indices as `range(start,stop,step)`. The slicing syntax `obj[start:stop:step]` passes a slice object as the argument to the `__getitem__`, `__setitem__`, or `__delitem__` method of object `obj`. It is up to `obj`'s class to interpret the slice objects that its methods receive. See also "Container slicing".
str	`str(obj='')` Returns a concise, readable string representation of `obj`. If `obj` is a string, `str` returns `obj`. See also `repr` in Table 7-2 and `__str__` in Table 4-1. In v2, synonym of `bytes`; in v3, equivalent to v2's `unicode`.
super	`super(cls,obj)` Returns a super-object of object `obj` (which must be an instance of class `cls` or of any subclass of `cls`), suitable for calling superclass methods. Instantiate this built-in type only within a method's code. See "Cooperative superclass method calling". In v3, you can just call `super()`, without arguments, within a method, and Python automatically determines the `cls` and `obj` by introspection.
tuple	`tuple(seq=())` Returns a tuple with the same items as iterable `seq`, in order. When `seq` is a tuple, `tuple` returns `seq` itself, like `seq[:]`. See "Tuples".

type	`type(obj)` Returns the type object that is the type of `obj` (i.e., the most-derived, AKA *leafmost*, type of which `obj` is an instance). `type(x)` is the same as `x.__class__` for any `x`. Avoid checking equality or identity of types: see Type checking: avoid it below.
unicode	`unicode(string[,codec[,errors]])` (v2 only.) Returns the Unicode string object built by decoding byte-string `string`, just like `string.decode(codec,errors)`. `codec` names the codec to use. If `codec` is missing, `unicode` uses the default codec (normally `'ascii'`). `errors`, if present, is a string that specifies how to handle decoding errors. See also "Unicode", particularly for information about codecs and `errors`, and `__unicode__` in Table 4-1. In v3, Unicode strings are of type `str`.

Type checking: avoid it

Use `isinstance` (covered in Table 7-2), *not* equality comparison of types, to check whether an instance belongs to a particular class, in order to properly support inheritance. Checking `type(x)` for equality or identity to some other type object is known as *type checking*. Type checking is inappropriate in production Python code, as it interferes with polymorphism. Just try to use `x` as if it were of the type you expect, handling any problems with a `try`/`except` statement, as discussed in "Error-Checking Strategies"; this is known as *duck typing*.

When you just *have* to type-check, typically for debugging purposes, use `isinstance` instead. `isinstance(x, atype)`, although in a more general sense it, too, is type checking, nevertheless is a lesser evil than `type(x) is atype`, since it accepts an `x` that is an instance of any subclass of `atype`, not just a direct instance of `atype` itself. In particular, `isinstance` is perfectly fine when you're checking specifically for an ABC (abstract base class: see "Abstract Base Classes"); this newer idiom is known as *goose typing*.

Built-in Functions

Table 7-2 covers Python functions (and some types that in practice are only used as if they were functions) in the module `builtins` (in v2, `__builtins__`), in alphabetical order. Built-ins' names are *not* reserved words. You can bind, in local or global scope, an identifier that's a built-in name (although we recommend you avoid doing so; see the following warning). Names bound in local or global scope override names bound in built-in scope: local and global names *hide* built-in ones. You can also rebind names in built-in scope, as covered in "Python built-ins".

Don't hide built-ins

Avoid accidentally hiding built-ins: your code might need them later. It's tempting to use, for your own variables, natural names such as `input`, `list`, or `filter`, but *don't do it*: these are names of built-in Python types or functions. Unless you get into the habit of *never* hiding built-ins' names with your own, sooner or later you'll get mysterious bugs in your code caused by just such hiding occurring accidentally.

Several built-in functions work in slightly different ways in v3 than they do in v2. To remove some differences, start your v2 module with `from future_builtins import *`: this makes the built-ins `ascii`, `filter`, `hex`, `map`, `oct`, and `zip` work the v3 way. (To use the built-in `print` function in v2, however, use `from __future__ import print_function`.)

Most built-ins don't accept named arguments

Most built-ins don't accept named arguments

Most built-in functions and types cannot be called with named arguments, only with positional ones. In the following list, we specifically mention cases in which this limitation does not hold.

Table 7-2.

__import__

`__import__(module_name[,globals[,locals[,fromlist]]])`

Deprecated in modern Python; use, instead, `importlib.import_module`, covered in "Module Loading".

---

abs

`abs(x)`

Returns the absolute value of number `x`. When `x` is complex, `abs` returns the square root of `x.imag**2+x.real**2` (also known as the *magnitude* of the complex number). Otherwise, `abs` returns `-x` if `x` is `<0`, `x` if `x` is `>=0`. See also `__abs__`, `__invert__`, `__neg__`, `__pos__` in Table 4-4.

---

all

`all(seq)`

`seq` is any iterable (often a *generator expression*; see "Generator expressions"). `all` returns `False` when any item of `seq` is false; otherwise (including when `seq` is empty), `all` returns `True`. Like operators `and` and `or`, covered in "Short-Circuiting Operators", `all` stops evaluating, and returns a result, as soon as the answer is known; in the case of `all`, this means that evaluation stops as soon as a false item is reached, but proceeds throughout `seq` if all of `seq`'s items are true. Here is a typical toy example of the use of `all`:

```
if all(x>0 for x in the_numbers):
          'all of the numbers are
    print(positive'                         )
else:
          'some of the numbers are not
    print(positive'
)
```

---

any

`any(seq)`

`seq` is any iterable (often a *generator expression*; see "Generator expressions"). `any` returns `True` if any item of `seq` is true; otherwise (including when `seq` is empty), `any` returns `False`. Like operators `and` and `or`, covered in "Short-Circuiting Operators", `any` stops evaluating, and returns a result, as soon as the answer is known; in the case of `any`, this means that evaluation stops as soon as a true item is reached, but proceeds throughout `seq` if all of `seq`'s items are false. Here is a typical toy example of the use of `any`:

```
if any(x<0 for x in the_numbers):
          'some of the numbers are
    print(negative'
)
else:
          'none of the numbers are
    print(negative'
)
```

---

ascii	`ascii(x)` v3 only, unless you have `from future_builtins import *` at the start of your v2 module. Like `repr`, but escapes all non-ASCII characters in the string it returns, so the result is quite similar to that of `repr` in v2.
bin	`bin(x)` Returns a binary string representation of integer `x`.
callable	`callable(obj)` Returns `True` if `obj` can be called, and otherwise `False`. An object can be called if it is a function, method, class, type, or an instance of a class with a `__call__` method. See also `__call__` in Table 4-1.
chr	`chr(code)` Returns a string of length `1`, a single character corresponding to integer `code` in Unicode (in v2, `chr` works only for `code < 256`; use `unichr` to avoid this limitation, and get a `unicode` instance of length `1`). See also `ord` and `unichr` in this table.
compile	`compile(string,filename,kind)` Compiles a string and returns a code object usable by `exec` or `eval`. `compile` raises `SyntaxError` when `string` is not syntactically valid Python. When `string` is a multiline compound statement, the last character must be `'\n'`. `kind` must be `'eval'` when `string` is an expression and the result is meant for `eval`; otherwise, `kind` must be `'exec'`. `filename` must be a string, used only in error messages (if any error occurs). See also `eval` in this table and "Compile and Code Objects".
delattr	`delattr(obj,name)` Removes the attribute `name` from `obj`. `delattr(obj,'ident')` is like `del obj.ident`. If `obj` has an attribute named `name` just because its class has it (as is normally the case, for example, with methods of `obj`), you cannot delete that attribute from `obj` itself. You may be able to delete that attribute from the *class*, if the metaclass lets you. If you can delete the class attribute, `obj` ceases to have the attribute, and so does every other instance of that class.
dir	`dir([obj])` Called without arguments, `dir` returns a sorted list of all variable names that are bound in the current scope. `dir(obj)` returns a sorted list of names of attributes of `obj`, including ones coming from `obj`'s type or by inheritance. See also `vars` in this table.
divmod	`divmod(dividend,divisor)` Divides two numbers and returns a pair whose items are the quotient and remainder. See also `__divmod__` in Table 4-4.

enumerate `enumerate(iterable,start=0)`

Returns a new iterator object whose items are pairs. For each such pair, the second item is the corresponding item in `iterable`, while the first item is an integer: `start`, `start+1`, `start+2`.... For example, the following snippet loops on a list `L` of integers, changing `L` in-place by halving every even value:

```
for i, num in enumerate(L):
    if num % 2 == 0:
        L[i] = num // 2
```

eval `eval(expr,[globals[,locals]])`

Returns the result of an expression. `expr` may be a code object ready for evaluation, or a string; if a string, `eval` gets a code object by internally calling `compile(expr, '<string>', 'eval')`. `eval` evaluates the code object as an expression, using the `globals` and `locals` dictionaries as namespaces. When both arguments are missing, `eval` uses the current namespace. `eval` cannot execute statements; it only evaluates expressions. Nevertheless, `eval` is dangerous unless you know and trust that `expr` comes from a source that you are certain is safe. See also "Expressions" and `ast.literal_eval`, covered in "Standard Input".

exec `exec(statement,[globals[,locals]])`

In v3, like `eval`, but applies to any statement and returns `None`. In v2, `exec` works similarly, but it's a statement, not a function. In either version, `exec` is dangerous unless you know and trust that `statement` comes from a source that you are certain is safe. See also "Statements".

filter `filter(func,seq)`

In v2, returns a list of those items of `seq` for which `func` is true. `func` can be any callable object accepting a single argument, or `None`. `seq` can be any iterable. When `func` is callable, `filter` calls `func` on each item of `seq`, just like the following list comprehension:

```
[item for item in seq if func(item)]
```

In v2, when `seq` is a string or tuple, `filter`'s result is also a string or tuple rather than a list. When `func` is `None`, `filter` tests for true items, just like:

```
[item for item in seq if item]
```

In v3, whatever the type of `seq`, `filter` returns an iterator, rather than a list (or other sequence type) as in v2; therefore, in v3, `filter` is equivalent to a generator expression rather than to a list comprehension.

format `format(x,format_spec='')`

Returns `x.__format__(format_spec)`. See Table 4-1.

getattr	`getattr(obj,name[,default])` Returns `obj`'s attribute named by string `name`. `getattr(obj,'ident')` is like `obj.ident`. When `default` is present and `name` is not found in `obj`, `getattr` returns `default` instead of raising `AttributeError`. See also "Object attributes and items" and "Attribute Reference Basics".
globals	`globals()` Returns the `__dict__` of the calling module (i.e., the dictionary used as the global namespace at the point of call). See also `locals` in this table.
hasattr	`hasattr(obj,name)` Returns `False` when `obj` has no attribute `name` (i.e., when `getattr(obj,name)` raises `AttributeError`). Otherwise, `hasattr` returns `True`. See also "Attribute Reference Basics".
hash	`hash(obj)` Returns the hash value for `obj`. `obj` can be a dictionary key, or an item in a set, only if `obj` can be hashed. All objects that compare equal must have the same hash value, even if they are of different types. If the type of `obj` does not define equality comparison, `hash(obj)` normally returns `id(obj)`. See also `__hash__` in Table 4-1.
hex	`hex(x)` Returns a hexadecimal string representation of integer `x`. See also `__hex__` in Table 4-4.
id	`id(obj)` Returns the integer value that denotes the identity of `obj`. The `id` of `obj` is unique and constant during `obj`'s lifetime (but may be reused at any later time after `obj` is garbage-collected, so, don't rely on storing or checking `id` values). When a type or class does not define equality comparison, Python uses `id` to compare and hash instances. For any objects `x` and `y`, identity check `x is y` is the same as `id(x)==id(y)`, but more readable and better-performing.
input	`input(prompt='')` In v2, `input(prompt)` is a shortcut for `eval(raw_input(prompt))`. In other words, in v2, `input` prompts the user for a line of input, evaluates the resulting string as an expression, and returns the expression's result. The implicit `eval` may raise `SyntaxError` or other exceptions. `input` is rather user-unfriendly and inappropriate for most programs, but it can sometimes be handy for small experiments and small exploratory scripts. See also `eval` and `raw_input` in this table. In v3, `input` is equivalent to v2's `raw_input`—that is, it always returns a `str` and does no `eval`.
intern	`intern(string)` Ensures that `string` is held in a table of interned strings and returns `string` itself or a copy. Interned strings may compare for equality slightly faster than other strings because you can use operator `is` instead of operator `==` for such comparisons. However, garbage collection can never recover the memory used for interned strings, so interning strings can slow down your program by making it take up too much memory. We do not cover interned strings in this book. In v3, function `intern` is not a built-in: rather, it lives, more appropriately, in the module `sys`.

isinstance	`isinstance(obj,cls)` Returns `True` when `obj` is an instance of class `cls` (or of any subclass of `cls`); otherwise, it returns `False`. `cls` can be a tuple whose items are classes: in this case, `isinstance` returns `True` if `obj` is an instance of any of the items of `cls`; otherwise, it returns `False`. See also "Abstract Base Classes".
issubclass	`issubclass(cls1,cls2)` Returns `True` when `cls1` is a direct or indirect subclass of `cls2`; otherwise, it returns `False`. `cls1` and `cls2` must be classes. `cls2` can also be a tuple whose items are classes. In this case, `issubclass` returns `True` when `cls1` is a direct or indirect subclass of any of the items of `cls2`; otherwise, it returns `False`. For any class `C`, `issubclass(C,C)` returns `True`.

iter	`)` `iter(objiter( func,sentinel)`	

Creates and returns an *iterator*, an object that you can repeatedly pass to the `next` built-in function to get one item at a time (see "Iterators"). When called with one argument, `iter(obj)` normally returns `obj.__iter__()`. When `obj` is a sequence without a special method `__iter__`, `iter(obj)` is equivalent to the generator:

```
def iter_sequence(obj):
    i = 0
    while True:
        try: yield obj[i]
        except IndexError: raise
StopIteration
        i += 1
```

See also "Sequences" and `__iter__` in Table 4-2.

When called with two arguments, the first argument must be callable without arguments, and `iter(func,sentinel)` is equivalent to the generator:

```
def iter_sentinel(func, sentinel):
    while True:
        item = func()
        if item == sentinel: raise
StopIteration
        yield item
```

Don't call iter in a for clause

As discussed in "The for Statement", the statement `for x in obj` is exactly equivalent to `for x in iter(obj)`; therefore, do *not* call `iter` in such a `for` statement: it would be redundant, and therefore bad Python style, slower, and less readable.

`iter` is *idempotent*. In other words, when `x` is an iterator, `iter(x)` is `x`, as long as `x`'s class supplies an `__iter__` method whose body is just `return self`, as an iterator's class should.

len `len(container)`

Returns the number of items in `container`, which may be a sequence, a mapping, or a set. See also `__len__` in "Container methods".

locals `locals()`

Returns a dictionary that represents the current local namespace. Treat the returned dictionary as read-only; trying to modify it may or may not affect the values of local variables, and might raise an exception. See also `globals` and `vars` in this table.

map	`map(func,seq,*seqs)` `map` calls `func` on every item of iterable `seq` and returns the sequence of results. When `map` is called with `n+1` arguments, the first one, `func`, can be any callable object that accepts `n` arguments; all remaining arguments to `map` must be iterable. `map` repeatedly calls `func` with `n` arguments (one corresponding item from each iterable). In v3, `map` returns an iterator yielding the results. For example, `map(func, seq)` is just like the generator expression `(func(item) for item in seq)`. When `map`'s iterable arguments have different lengths, in v3, `map` acts as if the longer ones were truncated. In v2, `map` returns a list of the results. For example, `map(func, seq)` is just like the list comprehension `[func(item) for item in seq]`. When `map`'s iterable arguments have different lengths, in v2, `map` acts as if the shorter ones were padded with `None`. Further, in v2 only, `func` can be `None`: in this case, each result is a tuple with `n` items (one item from each iterable).
max	`max(s,*args,key=None[,default=...])` Returns the largest item in the only positional argument `s` (`s` must then be iterable) or the largest one of multiple arguments. `max` is one of the built-in functions that you can call with named arguments: specifically, you can pass a `key=` argument, with the same semantics covered in "Sorting a list". In v3 only, you can also pass a `default=` argument, the value to return if the only positional argument `s` is empty; when you don't pass `default`, and `s` is empty, `max` raises `ValueError`.
min	`min(s,*args,key=None[,default=...])` Returns the smallest item in the only positional argument `s` (`s` must then be iterable) or the smallest one of multiple arguments. `min` is one of the built-in functions that you can call with named arguments: specifically, you can pass a `key=` argument, with the same semantics covered in "Sorting a list". In v3 only, you can also pass a `default=` argument, the value to return if the only positional argument `s` is empty; when you don't pass `default`, and `s` is empty, `min` raises `ValueError`.
next	`next(it[,default])` Returns the next item from iterator `it`, which advances to the next item. When `it` has no more items, `next` returns `default`, or, when you don't pass `default`, raises `StopIteration`.
oct	`oct(x)` Converts integer `x` to an octal string representation. See also `__oct__` in Table 4-4.
open	`open(filename,mode='r',bufsize=-1)` Opens or creates a file and returns a new file object. In v3, `open` accepts many optional parameters, and, in v2, you can `from io import open` to override the built-in function `open` with one very similar to v3's. See "The io Module".

ord

`ord(ch)`

In v2, returns the ASCII/ISO integer code between `0` and `255` (inclusive) for the single-character `str ch`. When, in v2, `ch` is of type `unicode` (and always, in v3), `ord` returns an integer code between `0` and `sys.maxunicode` (inclusive). See also `chr` and `unichr` in this table.

pow

`pow(x,y[,z])`

When `z` is present, `pow(x,y,z)` returns `x**y%z`. When `z` is missing, `pow(x,y)` returns `x**y`. See also `__pow__` in Table 4-4.

print

`, ..., sep=' ', end='\n', file=sys.stdout,`
`print(valueflush=False)`

In v3, formats with `str`, and emits to stream `file`, each `value`, separated by `sep`, with `end` after all of them (then flushes the stream if `flush` is true). In v2, `print` is a statement, unless you start your module with `from __future__ import print_function`, as we *highly* recommend (and which we assume in *every* example in this book), in which case `print` works just as it does in v3.

range

`range([start,]stop[,step=1])`

In v2, returns a list of integers in arithmetic progression:

```
[start, start+step, start+2*step, ...]
```

When `start` is missing, it defaults to `0`. When `step` is missing, it defaults to `1`. When `step` is `0`, `range` raises `ValueError`. When `step` is greater than `0`, the last item is the largest `start+i*step` strictly less than `stop`. When `step` is less than `0`, the last item is the smallest `start+i*step` strictly greater than `stop`. The result is an empty list when `start` is greater than or equal to `stop` and `step` is greater than `0`, or when `start` is less than or equal to `stop` and `step` is less than `0`. Otherwise, the first item of the result list is always `start`. See also `xrange` in this table.

In v3, `range` is a built-in type, a compact and efficient representation of the equivalent of a *read-only* list of integers in arithmetic progression; it is equivalent to v2's `range` or `xrange` in most practical uses, but more efficient. If you do need specifically a `list` in arithmetic progression, in v3, call `list(range(...))`.

raw_input

`raw_input(prompt='')`

v2 only: writes `prompt` to standard output, reads a line from standard input, and returns the line (without `\n`) as a string. When at end-of-file, `raw_input` raises `EOFError`. See also `input` in this table. In v3, this function is named `input`.

reduce

`reduce(func,seq[,init])`

(In v3, function `reduce` is not a built-in: rather, it lives in the module `functools`.) Applies `func` to the items of `seq`, from left to right, to reduce the iterable to a single value. `func` must be callable with two arguments. `reduce` calls `func` on the first two items of `seq`, then on the result of the first call and the third item, and so on. `reduce` returns the result of the last such call. When `init` is present, it is used before `seq`'s first item, if any. When `init` is missing, `seq` must be nonempty. When `init` is missing and `seq` has only one item, `reduce` returns `seq[0]`. Similarly, when `init` is present and `seq` is empty, `reduce` returns `init`. `reduce` is thus roughly equivalent to:

```
def reduce_equivalent(func,seq,init=None):    seq = iter(seq)    if init
is None: init = next(seq)    for item in seq: init = func(init,item)
return init
```

An example use of `reduce` is to compute the product of a sequence of numbers:

```
theprod = reduce(operator.mul, seq, 1)
```

reload

`reload(module)`

Reloads and reinitializes the module object `module`, and returns `module`. In v3, function `reload` is not a built-in: rather, it lives in module `imp` in early versions of Python 3, `importlib` in current ones, 3.4, and later.

repr

`repr(obj)`

Returns a complete and unambiguous string representation of `obj`. When feasible, `repr` returns a string that you can pass to `eval` in order to create a new object with the same value as `obj`. See also `str` in Table 7-1 and `__repr__` in Table 4-1.

reversed

`reversed(seq)`

Returns a new iterator object that yields the items of `seq` (which must be specifically a sequence, not just any iterable) in reverse order.

round

`round(x,n=0)`

Returns a `float` whose value is number `x` rounded to `n` digits after the decimal point (i.e., the multiple of `10**-n` that is closest to `x`). When two such multiples are equally close to `x`, `round`, in v2, returns the one that is farther from `0`; in v3, `round` returns the *even* multiple. Since today's computers represent floating-point numbers in binary, not in decimal, most of `round`'s results are not exact, as the online tutorial explains in detail. See also "The decimal Module".

setattr

`setattr(obj,name,value)`

Binds `obj`'s attribute `name` to `value`. `setattr(`*`obj`*`,'`*`ident`*`',`*`val`*`)` is like *`obj.ident=val`*. See also built-in `getattr` covered in this table, "Object attributes and items", and "Setting an attribute".

sorted	`sorted(seq,cmp=None,key=None,reverse=False)`
	
	Returns a list with the same items as iterable `seq`, in sorted order. Same as:
	
	`def sorted(seq,cmp=None,key=None,reverse=False):    result = list(seq) result.sort(cmp,key,reverse)    return result`
	
	Argument `cmp` exists only in v2, not in v3; see `cmp_to_key` in Table 7-4. See "Sorting a list" for the meaning of the arguments; `sorted` is one of the built-in functions that's callable with named arguments, specifically so you can optionally pass `key=` and/or `reverse=`.

sum	`sum(seq,start=0)` Returns the sum of the items of iterable `seq` (which should be numbers, and, in particular, cannot be strings) plus the value of `start`. When `seq` is empty, returns `start`. To "sum" (concatenate) an iterable of strings, in order, use `''.join(iterofstrs)`, as covered in Table 8-1 and "Building up a string from pieces".
unichr	`unichr(code)` v2 only: returns a Unicode string whose single character corresponds to `code`, where `code` is an integer between `0` and `sys.maxunicode` (inclusive). See also `str` and `ord` in Table 7-1. In v3, use `chr` for this purpose.
vars	`vars([obj])` When called with no argument, `vars` returns a dictionary with all variables that are bound in the current scope (like `locals`, covered in this table). Treat this dictionary as read-only. `vars(obj)` returns a dictionary with all attributes currently bound in `obj`, as covered in `dir` in this table. This dictionary may be modifiable, depending on the type of `obj`.
xrange	`xrange([start,]stop[,step=1])` v2 only: an iterable of integers in arithmetic progression, and otherwise similar to `range`. In v3, `range` plays this role. See `range` in this table.
zip	`zip(seq,*seqs)` In v2, returns a list of tuples, where the `n`th tuple contains the `n`th element from each of the argument sequences. `zip` must be called with at least one argument, and all arguments must be iterable. If the iterables have different lengths, `zip` returns a list as long as the shortest iterable, ignoring trailing items in the other iterable objects. See also `map` in this table and `izip_longest` in Table 7-5. In v3, `zip` returns an iterator, rather than a list, and therefore it's equivalent to a generator expression rather than to a list comprehension.

The sys Module

The attributes of the `sys` module are bound to data and functions that provide information on the state of the Python interpreter or affect the interpreter directly. Table 7-3 covers the most frequently used attributes of `sys`, in alphabetical order. Most `sys` attributes we don't cover are meant specifically for use in debuggers, profilers, and integrated development environments; see the online docs for more information. Platform-specific information is best accessed using the `platform` module, covered online, which we do not cover in this book.

Table 7-3.

argv	The list of command-line arguments passed to the main script. `argv[0]` is the name or full path of the main script, or `'-c'` if the command line used the `-c` option. See "The argparse Module" for one good way to use `sys.argv`.
byteorder	`'little'` on little-endian platforms, `'big'` on big-endian ones. See Wikipedia for more information on endianness.
builtin_module_names	A tuple of strings, the name of all the modules compiled into this Python interpreter.

displayhook	`displayhook(value)`
	
	In interactive sessions, the Python interpreter calls `displayhook`, passing it the result of each expression statement you enter. The default `displayhook` does nothing if `value` is `None`; otherwise, it preserves (in the built-in variable `_`), and displays via `repr`, `value`:
	
	`def _default_sys_displayhook(value):`
	`    if value is not None:`
	`        __builtins__._ = value`
	`        print(repr(value))`
	
	You can rebind `sys.displayhook` in order to change interactive behavior. The original value is available as `sys.__displayhook__`.

dont_write_bytecode	If true, Python does not write a bytecode file (with extension *.pyc* or, in v2, *.pyo*) to disk, when it imports a source file (with extension *.py*). Handy, for example, when importing from a read-only filesystem.
excepthook	`excepthook(type,value,traceback)` When an exception is not caught by any handler, propagating all the way up the call stack, Python calls `excepthook`, passing it the exception class, object, and traceback, as covered in "Exception Propagation". The default `excepthook` displays the error and traceback. You can rebind `sys.excepthook` to change how uncaught exceptions (just before Python returns to the interactive loop or terminates) are displayed and/or logged. The original value is available as `sys.__excepthook__`.
exc_info	`exc_info()` If the current thread is handling an exception, `exc_info` returns a tuple with three items: the class, object, and traceback for the exception. If the current thread is not handling an exception, `exc_info` returns `(None,None,None)`. To display information from a traceback, see "The traceback Module". **Holding on to a traceback object can make some garbage uncollectable** A traceback object indirectly holds references to all variables on the call stack; if you hold a reference to the traceback (e.g., indirectly, by binding a variable to the tuple that `exc_info` returns), Python must keep in memory data that might otherwise be garbage-collected. Make sure that any binding to the traceback object is of short duration, for example with a `try`/`finally` statement (discussed in "try/finally").

exit	`exit(arg=0)` Raises a `SystemExit` exception, which normally terminates execution after executing cleanup handlers installed by `try`/`finally` statements, `with` statements, and the `atexit` module. When `arg` is an `int`, Python uses `arg` as the program's exit code: `0` indicates successful termination, while any other value indicates unsuccessful termination of the program. Most platforms require exit codes to be between `0` and `127` . When `arg` is not an `int`, Python prints `arg` to `sys.stderr`, and the exit code of the program is `1` (a generic "unsuccessful termination" code).
float_info	A read-only object whose attributes hold low-level details about the implementation of the `float` type in this Python interpreter. See the online docs for details.
getrefcount	`getrefcount(object)` Returns the reference count of `object`. Reference counts are covered in "Garbage Collection".
getrecursionlimit	`getrecursionlimit()` Returns the current limit on the depth of Python's call stack. See also "Recursion" and `setrecursionlimit` in this table.
getsizeof	`getsizeof(obj,[default])` Returns the size in bytes of `obj` (not counting any items or attributes `obj` may refer to), or `default` when `obj` does not provide a way to retrieve its size (in the latter case, when `default` is absent, `getsizeof` raises `TypeError`).
maxint	(v2 only.) The largest `int` in this version of Python (at least `2**31-1`; that is, `2147483647`). Negative `int`s can go down to `-maxint-1`, due to two's complement representation. In v3, an `int` is of unbounded size (like a `long` in v2), so there *is* no "largest" `int`.
maxsize	Maximum number of bytes in an object in this version of Python (at least `2**31-1`; that is, `2147483647`).
maxunicode	The largest codepoint for a Unicode character in this version of Python (at least `2**16-1`, that is, `65535`). In v3, always `1114111` (`0x10FFFF`).
modules	A dictionary whose items are the names and module objects for all loaded modules. See "Module Loading" for more information on `sys.modules`.
path	A list of strings that specifies the directories and ZIP files that Python searches when looking for a module to load. See "Searching the Filesystem for a Module" for more information on `sys.path`.
platform	A string that names the platform on which this program is running. Typical values are brief operating system names, such as `'darwin'`, `'linux2'`, and `'win32'`. To check for Linux specifically, use `sys.platform.startswith('linux')`, for portability among Linux versions and between v2 and v3.

ps1, ps2

`ps1` and `ps2` specify the primary and secondary interpreter prompt strings, initially `'>>> '` and `'... '`, respectively. These attributes exist only in interactive interpreter sessions. If you bind either attribute to a nonstring object `x`, Python prompts by calling `str(x)` on the object each time a prompt is output. This feature allows dynamic prompting: code a class that defines `__str__`, then assign an instance of that class to `sys.ps1` and/or `sys.ps2`. For example, to get numbered prompts:

```
>>> import sys
>>> class Ps1(object):
...   def __init__(self):
...     self.p = 0
...   def __str__(self):
...     self.p += 1
...     return '[{}]>>> '.format(self.p)
...
>>> class Ps2(object):
...   def __str__(self):
...     return '[{}]... '.format(sys.ps1.p)
...
>>> sys.ps1 = Ps1(); sys.ps2 = Ps2()
[1]>>> (2 +
[1]... 2)
4
[2]>>>
```

setrecursionlimit

`setrecursionlimit(limit)`

Sets the limit on the depth of Python's call stack (the default is `1000`). The limit prevents runaway recursion from crashing Python. Raising the limit may be necessary for programs that rely on deep recursion, but most platforms cannot support very large limits on call-stack depth. More usefully, *lowering* the limit may help you check, during testing and debugging, that your program is gracefully degrading, rather than abruptly crashing with a `RuntimeError`, under situations of almost-runaway recursion. See also "Recursion" and `getrecursionlimit` in this table.

stdin, stdout, stderr

`stdin`, `stdout`, and `stderr` are predefined file-like objects that correspond to Python's standard input, output, and error streams. You can rebind `stdout` and `stderr` to file-like objects open for writing (objects that supply a `write` method accepting a string argument) to redirect the destination of output and error messages. You can rebind `stdin` to a file-like object open for reading (one that supplies a `readline` method returning a string) to redirect the source from which built-in functions `raw_input` (v2 only) and `input` read. The original values are available as `__stdin__`, `__stdout__`, and `__stderr__`. File objects are covered in "The io Module".

tracebacklimit	The maximum number of levels of traceback displayed for unhandled exceptions. By default, this attribute is not set (i.e., there is no limit). When `sys.tracebacklimit` is `<=0`, Python prints only the exception type and value, without traceback.
version	A string that describes the Python version, build number and date, and C compiler used. `sys.version[:3]` is `'2.7'` for Python 2.7, `'3.5'` for Python 3.5, and so on.

The copy Module

As discussed in "Assignment Statements", assignment in Python does not copy the righthand-side object being assigned. Rather, assignment adds a reference to the righthand-side object. When you want a copy of object `x`, you can ask `x` for a copy of itself, or you can ask `x`'s type to make a new instance copied from `x`. If `x` is a list, `list(x)` returns a copy of `x`, as does `x[:]`. If `x` is a dictionary, `dict(x)` and `x.copy()` return a copy of `x`. If `x` is a set, `set(x)` and `x.copy()` return a copy of `x`. In each case, we think it's best to use the uniform and readable idiom of calling the type, but there is no consensus on this style issue in the Python community.

The `copy` module supplies a `copy` function to create and return a copy of many types of objects. Normal copies, such as `list(x)` for a list `x` and `copy.copy(x)`, are known as *shallow* copies: when `x` has references to other objects (either as items or as attributes), a normal (shallow) copy of `x` has distinct references to the same objects. Sometimes, however, you need a *deep* copy, where referenced objects are deep-copied recursively; fortunately, this need is rare, since a deep copy can take a lot of memory and time. The `copy` module supplies a `deepcopy` function to create and return a deep copy.

copy	`copy(x)` Creates and returns a shallow copy of `x`, for `x` of many types (copies of several types, such as modules, classes, files, frames, and other internal types, are, however, not supported). If `x` is immutable, `copy.copy(x)` may return `x` itself as an optimization. A class can customize the way `copy.copy` copies its instances by having a special method `__copy__(self)` that returns a new object, a shallow copy of `self`.

deepcopy `deepcopy(x,[memo])`

Makes a deep copy of `x` and returns it. Deep copying implies a recursive walk over a directed (not necessarily acyclic) graph of references. A special precaution is needed to reproduce the graph's exact shape: when references to the same object are met more than once during the walk, distinct copies must not be made. Rather, references to the same copied object must be used. Consider the following simple example:

```
sublist = [1,2]
original = [sublist, sublist]
thecopy = copy.deepcopy(original)
```

`original[0] is original[1]` is `True` (i.e., the two items of `original` refer to the same object). This is an important property of `original` and anything claiming to "be a copy" must preserve it. The semantics of `copy.deepcopy` ensure that `thecopy[0] is thecopy[1]` is also `True`: the graphs of references of `original` and `thecopy` have the same shape. Avoiding repeated copying has an important beneficial side effect: it prevents infinite loops that would otherwise occur when the graph of references has cycles.

`copy.deepcopy` accepts a second, optional argument `memo`, a `dict` that maps the `id` of objects already copied to the new objects that are their copies. `memo` is passed by all recursive calls of `deepcopy` to itself; you may also explicitly pass it (normally as an originally empty `dict`) if you need to maintain a correspondence map between the identities of originals and copies.

A class can customize the way `copy.deepcopy` copies its instances by having a special method `__deepcopy__(self, memo)` that returns a new object, a deep copy of `self`. When `__deepcopy__` needs to deep copy some referenced object `subobject`, it must do so by calling `copy.deepcopy(subobject, memo)`. When a class has no special method `__deepcopy__`, `copy.deepcopy` on an instance of that class also tries calling the special methods `__getinitargs__`, `__getnewargs__`, `__getstate__`, and `__setstate__`, covered in "Pickling instances".

The collections Module

The `collections` module supplies useful types that are collections (i.e., containers), as well as the *abstract base classes* (ABCs) covered in "Abstract Base Classes". Since Python 3.4, the ABCs are in `collections.abc` (but, for backward compatibility, can still be accessed directly in `collections` itself: the latter access will cease working in some future release of v3).

ChainMap (v3 Only)

`ChainMap` "chains" multiple mappings together; given a `ChainMap` instance `c`, accessing `c[key]` returns the value in the first of the mappings that has that key, while *all* changes to `c` only affect the very first mapping in `c`. In v2, you could approximate this as follows:

```
class ChainMap(collections.MutableMapping):
    def __init__(self, *maps):
        self.maps = list(maps)
        self._keys = set()
        for m in self.maps: self._keys.update(m)
    def __len__(self): return len(self._keys)
    def __iter__(self): return iter(self._keys)
    def __getitem__(self, key):
        if key not in self._keys: raise KeyError(key
)
        for m in self.maps:
            try: return m[key]
            except KeyError: pass
    def __setitem__(self, key, value)
        self.maps[0][key] = value
        self._keys.add(key)
    def __delitem__(self, key)
        del self.maps[0][key]
        self._keys = set()
        for m in self.maps: self._keys.update(m)
```

Other methods could be defined for efficiency, but this is the minimum set that `MutableMapping` requires. A stable, production-level backport of `ChainMap` to v2 (and early versions of Python 3) is available on PyPI and can therefore be installed like all PyPI modules—for example, by running `pip install chainmap`.

See the ChainMap documentation in the online Python docs for more details and a collection of “recipes” on how to use `ChainMap`.

Counter

`Counter` is a subclass of `dict` with `int` values that are meant to *count* how many times the key has been seen (although values are allowed to be `<=0`); it roughly corresponds to types that other languages call “bag” or “multi-set.” A `Counter` instance is normally built from an iterable whose items are hashable: `c = collections.Counter(iterable)`. Then, you can index `c` with any of `iterable`’s items, to get the number of times that item appeared. When you index `c` with any missing key, the result is `0` (to remove an entry in `c`, use `del c[entry]`; setting `c[entry] = 0` leaves `entry` in `c`, just with a corresponding value of `0`).

`c` supports all methods of `dict`; in particular, `c.update(`*`otheriterable`*`)` updates all the counts, incrementing them according to occurrences in *`otheriterable`*. So, for example:

```
c = collections.Counter('moo')
c.update('foo')
```

leaves *`c['o']`* giving `4`, and *`c['f']`* and *`c['m']`* each giving `1`.

In addition to `dict` methods, `c` supports three extra methods:

elements	`c.elements()` Yields, in arbitrary order, keys in `c` with `c[key]>0`, yielding each key as many times as its count.
most_common	`c.most_common([n])` Returns a list of pairs for the `n` keys in `c` with the highest counts (all of them, if you omit `n`) in order of decreasing count (“ties” between keys with the same count are resolved arbitrarily); each pair is of the form `(k, c[k])` where `k` is one of the `n` most common keys in `c`.
subtract	`c.subtract(iterable)` Like `c.update(iterable)` “in reverse”—that is, *subtracting* counts rather than *adding* them. Resulting counts in `c` can be `<=0`.

See the Counter documentation in the online Python docs for more details and a collection of useful “recipes” on how to use `Counter`.

OrderedDict

`OrderedDict` is a subclass of `dict` that remembers the order in which keys were originally inserted (assigning a new value for an existing key does not change the order, but removing a key and inserting it again does). Given an `OrderedDict` instance `o`, iterating on `o` yields keys in order of insertion (oldest to newest key), and `o.popitem()` removes and returns the item at the key most recently inserted. Equality tests between two instances of `OrderedDict` are order-sensitive; equality tests between an instance of `OrderedDict` and a `dict` or other mapping are not. See the OrderedDict documentation in the online Python docs for more details and a collection of “recipes” on how to use `OrderedDict`.

defaultdict

`defaultdict` extends `dict` and adds one per-instance attribute, named `default_factory`. When an instance `d` of `defaultdict` has `None` as the value of *`d`*`.default_factory`, `d` behaves exactly like a `dict`. Otherwise, *`d`*`.default_factory` must be callable without arguments, and `d` behaves just like a `dict` except when you access `d` with a key `k` that is not `in d`. In this specific case, the indexing `d[k]` calls `d.default_factory()`, assigns the result as the value of `d[k]`, and returns the result. In other words, the type `defaultdict` behaves much like the following Python-coded class:

```
class defaultdict(dict):
    def __init__(self, default_factory, *a, **k):
        dict.__init__(self, *a, **k)
        self.default_factory = default_factory
    def __getitem__(self, key):
        if key not in self and self.default_factory is not None
:
            self[key] = self.default_factory()
        return dict.__getitem__(self, key)
```

As this Python equivalent implies, to instantiate `defaultdict` you pass it an extra first argument (before any other arguments, positional and/or named, if any, to be passed on to plain `dict`). That extra first argument becomes the initial value of `default_factory`; you can access and rebind `default_factory` .

All behavior of `defaultdict` is essentially as implied by this Python equivalent (except `str` and `repr`, which return strings different from those they'd return for a `dict`). Named methods, such as `get` and `pop`, are not affected. All behavior related to keys (method `keys`, iteration, membership test via operator `in`, etc.) reflects exactly the keys that are currently in the container (whether you put them there explicitly, or implicitly via an indexing that called `default_factory`).

A typical use of `defaultdict` is to set `default_factory` to `list`, to make a mapping from keys to lists of values:

```
def make_multi_dict(items):
    d = collections.defaultdict(list)
    for key, value in items: d[key].append(value
)
    return d
```

Called with any iterable whose items are pairs of the form `(key,value)`, with all keys being hashable, this `make_multi_dict` function returns a mapping that associates each key to the lists of one or more values that accompanied it in the iterable (if you want a pure `dict` result, change the last statement into `return dict(d)` — this is rarely necessary).

If you don't want duplicates in the result, and every `value` is hashable, use a `collections.defaultdict(set)`, and `add` rather than `append` in the loop.

deque

deque `deque(seq=(), maxlen=None)`

`deque` is a sequence type whose instances are "double-ended queues" (additions and removals at either end are fast). A `deque` instance `d` is a mutable sequence and can be indexed and iterated on (however, `d` cannot be sliced, only indexed one item at a time, whether for access, rebinding, or deletion). The initial items of `d` are those of `seq`, in the same order. `d.maxlen` is a read-only attribute: when `None`, `d` has no maximum length; when an `int`, it must be `>=0`, and `d`'s length is limited to `d.maxlen` (when too many items are added, items are silently dropped from the other side)—a useful data type to maintain "the latest *N* things seen," also known in other languages as a *ring buffer*.

`d` supplies the following methods:

append	`d.append(item)` Appends `item` at the right (end) of `d`.
appendleft	`d.appendleft(item)` Appends `item` at the left (start) of `d`.
clear	`d.clear()` Removes all items from `d`, leaving it empty.
extend	`d.extend(iterable)` Appends all items of `iterable` at the right (end) of `d`.
extendleft	`d.extendleft(item)` Appends all items of `iterable` at the left (start) of `d` in reverse order.
pop	`d.pop()` Removes and returns the last (rightmost) item from `d`. If `d` is empty, raises `IndexError`.
popleft	`d.popleft()` Removes and returns the first (leftmost) item from `d`. If `d` is empty, raises `IndexError`.
rotate	`d.rotate(n=1)` Rotates `d` `n` steps to the right (if `n<0`, rotates left).

namedtuple

`namedtuple` is a factory function, building and returning a subclass of `tuple` whose instances' items you can access by attribute reference, as well as by index.

namedtuple `namedtuple(typename,fieldnames)`

`typename` is a string that's a valid identifier (starts with a letter, may continue with letters, digits, and underscores; can't be a reserved word such as `'class'`) and names the new type that `namedtuple` builds and returns. `fieldnames` is a sequence of strings that are valid identifiers and name the new type's attributes, in order (for convenience, `fieldnames` can also be a single string with identifiers separated by spaces or commas). `namedtuple` returns a type: you can bind that type to a name, then use it to make immutable instances initialized with either positional or named arguments. Calling `repr` or `str` on those instances formats them in named-argument style. For example:

```
point = collections.namedtuple('point', 'x,y,z')p = point(x=1,y=2,z=3)
# can build with named
arguments                         x, y, z = p
# can unpack like a normal
tuple                             if p.x < p.y:
# can access items as
attributes                            print(p)
# formats with named
argument                       # prints point(x=1, y=2, z=3)
```

A `namedtuple` such as `point` and its instances such as `p` also supply attributes and methods (whose names start with an underscore only to avoid conflicts with field names, *not* to indicate they are in any way "private"):

_asdict	`p._asdict()` Returns a `dict` whose keys are `p`'s field names, with values from `p`'s corresponding items.
_fields	`point._fields` Returns a tuple of field name strings, here `('x', 'y', 'z')`.
_make	`point._make(seq)` Returns a `point` instance with items initialized from iterable `seq`, in order (`len(seq)` must equal `len(point._fields)`).
_replace	`p._replace(**kwargs)` Returns a copy of `p`, with `0` or more items replaced as per the named arguments.

For more details and advice about using `namedtuple`, see the online docs.

The functools Module

The `functools` module supplies functions and types supporting functional programming in Python, listed in Table 7-4.

Table 7-4.

cmp_to_key	`cmp_to_key(func)` `func` must be callable with two arguments and return a number: `<0` if the first argument is to be considered "less than" the second one, `>0` if vice versa, `0` if the two arguments are to be considered equal (like the old `cmp` built-in function, deprecated in v2 and removed in v3, and the old `cmp=` named argument to `sort` and `sorted`). `cmp_to_key` returns a callable `k` suitable as the `key=` named argument to functions and methods such as `sort`, `sorted`, `min`, `max`, and so on. This is useful to convert programs using old-style `cmp=` arguments to new-style `key=` ones, which is required to use v3 and highly recommended in v2.
lru_cache	`lru_cache(`*`max_size`*`=128, `*`typed`*`=False)` (v3 only; to use in v2, `pip install` Jason Coombs' backport.) A *memoizing* decorator suitable for decorating a function whose arguments are all hashable, adding to the function a cache storing the last `max_size` results (`max_size` should be a power of 2, or `None` to have the cache keep all previous results); when the decorated function is called again with arguments that are in the cache, it immediately returns the previously cached result, bypassing the underlying function's body code. When `typed` is true, arguments that compare equal but have different types, such as 23 and 23.0, are cached separately. For more details and examples, see the online docs.

partial	`partial(func,*a,**k)`
	
	`func` is any callable. `partial` returns another callable `p` that is just like `func`, but with some positional and/or named parameters already bound to the values given in `a` and `k`. In other words, `p` is a *partial application* of `func`, often also known (with debatable correctness, but colorfully) as a *currying* of `func` to the given arguments (named in honor of mathematician Haskell Curry). For example, say that we have a list of numbers `L` and want to clip the negative ones to `0`; one way to do it is:
	
	`L = map(functools.partial(max, 0), L)`
	
	as an alternative to the lambda-using snippet:
	
	`L = map(lambda x: max(0, x), L)`
	
	and to the most concise approach, a list comprehension:
	
	`L = [max(0, x) for x in L]`
	
	`functools.partial` comes into its own in situations that demand callbacks, such as event-driven programming for GUIs and networking applications (covered in Chapter 18).
	
	`partial` returns a callable with the attributes `func` (the wrapped function), `args` (the `tuple` of prebound positional arguments), and `keywords` (the `dict` of prebound named arguments, or `None`).

reduce `reduce(func,seq[,init])`

Like the built-in function `reduce`, covered in Table 7-2. In v3, `reduce` is not a built-in, but it's still available in `functools`.

total_ordering A class decorator suitable for decorating classes that supply at least one inequality comparison method, such as `__lt__`, and also supply `__eq__`. Based on the class's existing methods, the class decorator `total_ordering` adds to the class all other inequality comparison methods, removing the need for you to add boilerplate code for them.

wraps `wraps(wrapped)`

A decorator suitable for decorating functions that wrap another function `wrapped` (often nested functions within another decorator). `wraps` copies the `__name__`, `__doc__`, and `__module__` attributes of `wrapped` on the decorated function, thus improving the behavior of the built-in function `help`, and of doctests, covered in "The doctest Module".

The heapq Module

The `heapq` module uses *min heap* algorithms to keep a list in "nearly sorted" order as items are inserted and extracted. `heapq`'s operation is faster than calling a list's `sort` method after each insertion, and much faster than `bisect` (covered in the online docs). For many purposes, such as implementing "priority queues," the nearly sorted order supported by `heapq` is just as good as a fully sorted order, and faster to establish and maintain. The `heapq` module supplies the following functions:

heapify `heapify(alist)`

Permutes `alist` as needed to make it satisfy the (min) *heap condition*:

- for any `i>=0`
  - `alist[i]<=alist[2*i+1]` and
  - `alist[i]<=alist[2*i+2]`
- as long as all the indices in question are `<len(alist)`

If a list satisfies the (min) heap condition, the list's first item is the smallest (or equal-smallest) one. A sorted list satisfies the heap condition, but many other permutations of a list also satisfy the heap condition, without requiring the list to be fully sorted. `heapify` runs in `O(len(alist))` time.

heappop `heappop(alist)`

Removes and returns the smallest (first) item of `alist`, a list that satisfies the heap condition, and permutes some of the remaining items of `alist` to ensure the heap condition is still satisfied after the removal. `heappop` runs in `O(log(len(alist)))` time.

heappush `heappush(alist,item)`

Inserts the `item` in `alist`, a list that satisfies the heap condition, and permutes some items of `alist` to ensure the heap condition is still satisfied after the insertion. `heappush` runs in `O(log(len(alist)))` time.

heappushpop `heappushpop(alist,item)`

Logically equivalent to `heappush` followed by `heappop`, similar to:

```
def heappushpop(alist, item):
    heappush(alist, item)
    return heappop(alist)
```

`heappushpop` runs in `O(log(len(alist)))` time and is generally faster than the logically equivalent function just shown. `heappushpop` can be called on an empty `alist`: in that case, it returns the `item` argument, as it does when `item` is smaller than any existing item of `alist`.

heapreplace `heapreplace(alist,item)`

Logically equivalent to `heappop` followed by `heappush`, similar to:

```
def heapreplace(alist, item):
    try: return heappop(alist)
    finally: heappush(alist, item
)
```

`heapreplace` runs in `O(log(len(alist)))` time and is generally faster than the logically equivalent function just shown. `heapreplace` cannot be called on an empty `alist`: `heapreplace` always returns an item that was already in `alist`, not the `item` being pushed onto it.

merge `merge(*iterables)`

Returns an iterator yielding, in sorted order (smallest to largest), the items of the `iterables`, each of which must be smallest-to-largest sorted.

nlargest `nlargest(n,seq,key=None)`

Returns a reverse-sorted list with the `n` largest items of iterable `seq` (less than `n` if `seq` has fewer than `n` items); like `sorted(seq, reverse=True)[:n]` but faster for small values of `n`. You may also specify a `key=` argument, like you can for `sorted`.

nsmallest `nsmallest(n,seq,key=None)`

Returns a sorted list with the `n` smallest items of iterable `seq` (less than `n` if `seq` has fewer than `n` items); like `sorted(seq)[:n]` but faster for small values of `n`. You may also specify a `key=` argument, like you can for `sorted`.

The Decorate-Sort-Undecorate Idiom

Several functions in the `heapq` module, although they perform comparisons, do not accept a `key=` argument to customize the comparisons. This is inevitable, since the functions operate in-place on a plain list of the items: they have nowhere to “stash away” custom comparison keys computed once and for all.

When you need both heap functionality and custom comparisons, you can apply the good old *decorate-sort-undecorate (DSU)* idiom (which used to be crucial to optimize sorting in old versions of Python, before the `key=`

functionality was introduced).

The DSU idiom, as applied to `heapq`, has the following components:

1. *Decorate*: Build an auxiliary list `A` where each item is a tuple starting with the sort key and ending with the item of the original list `L`.
2. Call `heapq` functions on `A`, typically starting with `heapq.heapify(A)`.
3. *Undecorate*: When you extract an item from `A`, typically by calling `heapq.heappop(A)`, return just the last item of the resulting tuple (which corresponds to an item of the original list `L`).

When you add an item to `A` by calling `heapq.heappush(A, item)`, decorate the actual item you're inserting into a tuple starting with the sort key.

This sequencing is best wrapped up in a class, as in this example:

```
import heapq

class KeyHeap(object):
    def __init__(self, alist, key):
        self.heap = [
            (key(o), i, o)
            for i, o in enumerate(alist)]
        heapq.heapify(self.heap)
        self.key = key
        if alist:
            self.nexti = self.heap[-1][1] +
1
        else:
            self.nexti = 0

    def __len__(self):
        return len(self.heap)

    def push(self, o):
        heapq.heappush(
            self.heap,
            (self.key(o), self.nexti, o))
        self.nexti += 1

    def pop(self):
        return heapq.heappop(self.heap)[-1]
```

In this example, we use an increasing number in the middle of the decorated tuple (after the sort key, before the actual item) to ensure that actual items are never compared directly, even if their sort keys are equal (this semantic guarantee is an important aspect of the `key=` argument's functionality to `sort` and the like).

The argparse Module

When you write a Python program meant to be run from the command line (or from a "shell script" in Unix-like systems or a "batch file" in Windows), you often want to let the user pass to the program, on the command line, *command-line arguments* (including *command-line options*, which by convention are usually arguments starting with one or two dash characters). In Python, you can access the arguments as `sys.argv`, an attribute of module `sys` holding those arguments as a list of strings (`sys.argv[0]` is the name by which the user started your program; the arguments are in sublist `sys.argv[1:]`). The Python standard library offers three modules to process those arguments; we only cover the newest and most powerful one, `argparse`, and we only cover a small, *core* subset of `argparse`'s rich functionality. See the online reference and tutorial for much, much more.

ArgumentParser	`ArgumentParser(**kwargs)` `ArgumentParser` is the class whose instances perform argument parsing. It accepts many named arguments, mostly meant to improve the help message that your program displays if command-line arguments include `-h` or `--help`. One named argument you should pass is `description=`, a string summarizing the purpose of your program.

Given an instance `ap` of `ArgumentParser`, prepare it by one or more calls to *`ap`*`.add_argument`; then, use it by calling `ap.parse_args()` without arguments (so it parses `sys.argv`): the call returns an instance of `argparse.Namespace`, with your program's args and options as attributes.

`add_argument` has a mandatory first argument: either an identifier string, for positional command-line arguments, or a flag name, for command-line options. In the latter case, pass one or more flag names; an option often has both a short name (one dash, then a single character) and a long name (two dashes, then an identifier).

After the positional arguments, pass to `add_argument` zero or more named arguments to control its behavior. Here are the common ones:

action	What the parser does with this argument. Default: `'store'`, store the argument's value in the namespace (at the name given by `dest`). Also useful: `'store_true'` and `'store_false'`, making an option into a `bool` one (defaulting to the opposite `bool` if the option is not present); and `'append'`, appending argument values to a list (and thus allowing an option to be repeated).
choices	A set of values allowed for the argument (parsing the argument raises an exception if the value is not among these); default, no constraints.
default	Value to use if the argument is not present; default, `None`.
dest	Name of the attribute to use for this argument; default, same as the first positional argument stripped of dashes.
help	A short string mentioning the argument's role, for help messages.
nargs	Number of command-line arguments used by this logical argument (by default, `1`, stored as-is in the namespace). Can be an integer `>0` (uses that many arguments and stores them as a list), `'?'` (one, or none in which case `default` is used), `'*'` (`0` or more, stored as a list), `'+'` (`1` or more, stored as a list), or `argparse.REMAINDER` (all remaining arguments, stored as a list).
type	A callable accepting a string, often a type such as `int`; used to transform values from strings to something else. Can be an instance of `argparse.FileType` to open the string as a filename (for reading if `FileType('r')`, for writing if `FileType('w')`, and so on).

Here's a simple example of `argparse`—save this code in a file called *greet.py*:

```
import argparse
ap = argparse.ArgumentParser(description='Just an example')
ap.add_argument('who', nargs='?', default='World')
ap.add_argument('--formal', action='store_true')
ns = ap.parse_args()
if ns.formal:
    greet = 'Most felicitous salutations, o {}.'
else:
    greet = 'Hello, {}!'
print(greet.format(ns.who))
```

Now, `python greet.py` prints `Hello, World!`, while `python greet.py --formal Cornelia` prints `Most felicitous salutations, o Cornelia.`

The itertools Module

The `itertools` module offers high-performance building blocks to build and manipulate iterators. To handle long processions of items, iterators are often better than lists, thanks to iterators' intrinsic "lazy evaluation" approach: an iterator produces items one at a time, as needed, while all items of a list (or other sequence) must be in memory at the same time. This approach even makes it feasible to build and use unbounded iterators, while lists must always have finite numbers of items (since any machine has a finite amount of memory).

Table 7-5 covers the most frequently used attributes of `itertools`; each of them is an iterator type, which you call to get an instance of the type in question, or a factory function behaving similarly. See the itertools documentation in the online Python docs for more `itertools` attributes, including *combinatoric* generators for permutations, combinations, and Cartesian products, as well as a useful taxonomy of `itertools` attributes.

The online docs also offer *recipes*, ways to combine and use `itertools` attributes. The recipes assume you have `from itertools import *` at the top of your module; this is *not* recommended use, just an assumption to make the recipes' code more compact. It's best to `import itertools as it`, then use references such as `it.something` rather than the more verbose `itertools.something`.

Table 7-5.

chain	`chain(*iterables)`
	
	Yields items from the first argument, then items from the second argument, and so on until the end of the last argument, just like the generator expression:
	
	`(item for iterable in iterables for item in iterable)`

chain.from_iterable	`chain.from_iterable(iterables)`
	
	Yields items from the iterables in the argument, in order, just like the genexp:
	
	`(item for iterable in iterables for item in iterable)`

compress

`compress(data,conditions)`

Yields each item from `data` corresponding to a true item in `conditions`, just like the generator expression:

```
(item for item, cond in zip(data, conditions) if cond)
```

count

`count(start=0, step=1)`

Yields consecutive integers starting from `start`, just like the generator:

```
def count(start=0, step=1):
    while True:
        yield start
        start += step
```

cycle

`cycle(iterable)`

Yields each item of `iterable`, endlessly repeating items from the beginning each time it reaches the end, just like the generator:

```
def cycle(iterable):
    saved = []
    for item in iterable:
        yield item
        saved.append(item)
    while saved:
        for item in saved: yield
item
```

dropwhile

`dropwhile(func,iterable)`

Drops the `0+` leading items of `iterable` for which `func` is true, then yields each other item, just like the generator:

```
def dropwhile(func,iterable):
    iterator = iter(iterable)
    for item in iterator:
        if not func(item):
            yield item
            break
    for item in iterator: yield
item
```

groupby

`groupby(iterable,key=None)`

`iterable` normally needs to be already sorted according to `key` (`None`, as usual, standing for the identity function). `groupby` yields pairs `(k,g)`, each pair representing a *group* of adjacent items from `iterable` having the same value `k` for `key(item)`; each `g` is an iterator yielding the items in the group. When the `groupby` object advances, previous iterators `g` become invalid (so, if a group of items needs to be processed later, store somewhere a `list` "snapshot" of it, `list(g)`).

Another way of looking at the groups `groupby` yields is that each terminates as soon as `key(item)` changes (which is why you normally call `groupby` only on an `iterable` that's already sorted by `key`).

For example, suppose that, given a `set` of lowercase words, we want a `dict` that maps each initial to the longest word having that initial (with "ties" broken arbitrarily):

```
import itertools as it, operator
def set2dict(aset):
    first = operator.itemgetter(0)
    words = sorted(aset, key=first)
    adict = {}
    for initial,
        group in it.groupby(words, key=first
):
        adict[initial] = max(group, key=len)
    return adict
```

ifilter

`ifilter(func,iterable)`

Yields those items of `iterable` for which `func` is true, just like the genexp:

```
(item for item in iterable if func(item))
```

`func` can be any callable object that accepts a single argument, or `None`. When `func` is `None`, `ifilter` yields true items, just like the genexp:

```
(item for item in iterable if item)
```

ifilterfalse

`ifilterfalse(func,iterable)`

Yields those items of `iterable` for which `func` is false, just like the genexp:

```
(item for item in iterable if not func(item))
```

`func` can be any callable accepting a single argument, or `None`. When `func` is `None`, `ifilterfalse` yields false items, just like the genexp:

```
(item for item in iterable if not item)
```

imap

`imap(func,*iterables)`

Yields the results of `func`, called with one argument from each of the `iterables`; stops when the shortest of the `iterables` is exhausted, just like the generator:

```
def imap(func,*iterables):
    iters = [iter(x) for x in iterables]
    while True: yield func(*(next(x) for x in iters
))
```

islice

`islice(iterable[,start],stop[,step])`

Yields items of `iterable`, skipping the first `start` ones (default `0`), until the `stop`th one excluded, advancing by steps of `step` (default `1`) at a time. All arguments must be nonnegative integers (or `None`), and `step` must be `>0`. Apart from checks and optional arguments, it's like the generator:

```
def islice(iterable,start,stop,step=1):
    en = enumerate(iterable)
    n = stop
    for n, item in en:
        if n>=start: break
    while n<stop:
        yield item
        for x in range(step): n, item = next(en
)
```

izip

`izip(*iterables)`

Yields tuples with one corresponding item from each of the `iterables`; stops when the shortest of the `iterables` is exhausted. Just like `imap(tuple,*iterables)`. v2 only; in v3, the built-in function `zip`, covered in Table 7-2, has this functionality.

izip_longest

`izip_longest(*iterables,fillvalue=None)`

Yields tuples with one corresponding item from each of the `iterables`; stops when the longest of the `iterables` is exhausted, behaving as if each of the others was "padded" to that same length with references to `fillvalue`.

repeat

`repeat(item[,times])`

Repeatedly yields `item`, just like the generator expression:

```
(item for x in range(times))
```

When `times` is absent, the iterator is unbounded, yielding a potentially infinite number of items, which are all the object `item`. Just like the generator:

```
def repeat_unbounded(item):
    while True: yield item
```

starmap

`starmap(func,iterable)`

Yields `func(*item)` for each `item` in `iterable` (each `item` must be an iterable, normally a tuple), just like the generator:

```
def starmap(func,iterable):
    for item in iterable:
        yield func(*item)
```

takewhile

`takewhile(func,iterable)`

Yields items from `iterable` as long as `func(item)` is true, then stops, just like the generator:

```
def takewhile(func,iterable):
    for item in iterable:
        if func(item):
            yield item
        else:
            break
```

tee

`tee(iterable,n=2)`

Returns a tuple of `n` independent iterators, each yielding items that are the same as those of `iterable`. The returned iterators are independent from each other, but they are *not* independent from `iterable`; avoid altering the object `iterable` in any way, as long as you’re still using any of the returned iterators.

We have shown equivalent generators and genexps for many attributes of `itertools`, but it’s important to remember the sheer speed of `itertools` types. To take a trivial example, consider repeating some action 10 times:

```
for _ in itertools.repeat(0, 10): pass
```

This turns out to be about 10 to 20 percent faster, depending on Python release and platform, than the straightforward alternative:

```
for _ in range(10): pass
```