Chapter 8. Core Built-ins and Standard Library Modules

The term *built-in* has more than one meaning in Python. In many contexts, *built-in* means an object directly accessible to Python code without an **import** statement. The section **"Python built-ins"** shows Python's mechanism 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 `file` type, and **"Internal Types"** some other built-in types intrinsic to Python's internal operation. This chapter provides additional coverage of built-in core types in the opening section and covers built-in functions available in the module `builtins` in **"Built-in Functions"**.

Some modules are called "built-in" because they're in the Python standard library (though it takes an **import** statement to use them), as opposed to add-on modules, also known as Python *extensions*.

This chapter covers several built-in core modules: namely, the standard library modules `sys`, `copy`, `collections`, `functools`, `heapq`, `argparse`, and `itertools`. You'll find a discussion of each module *x* in the respective section "The *x* Module."

Chapter 9 covers some string-related built-in core modules (`string`, `codecs`, and `unicodedata`) with the same section-name convention. **Chapter 10** covers `re` in **"Regular Expressions and the re Module"**.

Built-in Types

Table 8-1 provides a brief overview of Python's core built-in types. More details about many of these types, and about operations on their instances, are found throughout Chapter 3. In this section, by "any number" we mean, specifically, "any noncomplex number." Also, many built-ins accept at least some of their parameters in a positional-only way; we use the 3.8+ positional-only marker `/`, covered in "Positional-only marker", to indicate this.

Table 8-1. Python's core built-in types

`bool`	`bool(x=False, /)` Returns **`False`** when *x* evaluates as falsy; returns **`True`** when *x* evaluates as truthy (see "Boolean Values"). `bool` extends `int`: the 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'` and `str(False)` is `'False'`.
`bytearray`	`bytearray(x=b'', /[, codec[, errors]])` Returns a mutable sequence of bytes (`int`s with values from `0` to `255`), supporting the usual methods of mutable sequences, plus the methods of `str`. When *x* is a `str`, you must also pass `codec` and may pass `errors`; the result is just 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 protocol, the read-only buffer of bytes from *x* initializes the instance. Otherwise, *x* must be an iterable yielding `int`s `>=0` and `<256`; e.g., `bytearray([1,2,3,4]) == bytearray(b'\x01\x02\x03\x04')`.

`bytes`	`bytes(`*`x`*`=b'', /[, codec[, errors]])` Returns an immutable sequence of bytes, with the same nonmutating methods and the same initialization behavior as `bytearray`.
`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 `imag` is also a number: 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*. (We cover dictionaries 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. When a key appears more than once in *x*, Python uses the value corresponding to the last occurrence of the key. In other words, when *x* is any iterable yielding pairs, `c = dict(x)` is exactly equivalent to:
	`c = {}`
	`for key, value in x:`
	`    c[key] = value`
	You can also 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: each such extra item 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 frozenset, `frozenset(seq)` returns *seq* itself, just like `seq.copy()` does. See **"Set Operations"**.

`int`	`int(`*`x`*`=0, /, base=10)` Converts any number, or a suitable string, to an `int`. When *`x`* is a number, `int` truncates toward `0`, "dropping" any fractional part. `base` may be present only when *`x`* is a string: then, `base` is the conversion base, between `2` and `36`, with `10` as the default. You can explicitly pass `base` 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`*`, /)` Returns an object *`m`* "viewing" exactly the same underlying memory as *`x`*, which must be an object supporting the **buffer protocol** (for example, an instance of `bytes`, `bytearray`, or `array.array`), with items of *`m`*`.itemsize` bytes each. In the normal case in which *`m`* is "one-dimensional" (we don't cover the complicated case of "multidimensional" `memoryview` instances in this book), `len(`*`m`*`)` is the number of items. You can index *`m`* (returning `int`) or slice it (returning an instance of `memoryview` "viewing" the appropriate subset of the same underlying memory). *`m`* is mutable when *`x`* is (but you can't change *`m`*'s size, so, when you assign to a slice, it must be from an iterable of the same length as the slice). *`m`* is a sequence, thus iterable, and is hashable when *`x`* is hashable and when *`m`*`.itemsize` is one byte. *`m`* supplies several read-only attributes and methods; see the **online docs** for details. Two particularly useful

	methods are `m.tobytes` (returns *m*'s data as an instance of `bytes`) and `m.tolist` (returns *m*'s data as a list of `ints`).
`object`	`object()` Returns a new instance of `object`, the most fundamental type in Python. Instances of type `object` have no functionality: the only use of such instances is as "sentinels"—i.e., objects not equal to any distinct object. For instance, when a function takes an optional argument where **None** is a legitimate value, you can use a sentinel for the argument's default value to indicate that the argument was omitted:

```
MISSING = object()
def check_for_none(obj=MISSING):
    if obj is MISSING:
        return -1
    return 0 if obj is None else 1
```

`set`	`set(`*`seq`*`=(), /)` Returns a new mutable set object with the same items as iterable *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)`. 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 slices 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 8-2** and `__str__` in **Table 4-1**.
`super`	`super()`, `super(cls, obj, /)` Returns a superobject 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. The `super(cls, obj)` syntax is a legacy form from Python 2 that has been retained for compatibility. In new code, you usually call `super()` without arguments, within a method, and Python determines the *`cls`* and *`obj`* by introspection (as `type(self)` and `self`, respectively). See **"Cooperative superclass method calling"**.
`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 the following warning for details). This function is commonly used for debugging; for example, when value *`x`* does not behave as expected, inserting `print(type(x), x)`. It can also be used to dynamically create classes at runtime, as described in **Chapter 4**.

TYPE EQUALITY CHECKING: AVOID IT!

Use `isinstance` (covered in **Table 8-2**), *not* equality comparison of types, to check whether an instance belongs to a particular class in order to support inheritance properly.[1] Using `type(x)` to check for equality or identity to some other type object is known as *type equality checking*. Type equality checking is inappropriate in production Python code, as it interferes with polymorphism. Typically, you 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*** (one of this book's authors is often credited with an early use of this colorful phrase).

When you just *have* to type-check, usually for debugging purposes, use `isinstance` instead. In a broader sense, `isinstance(x, atype)` is also a form of type checking, but it is a lesser evil than `type(x) is atype`. `isinstance` accepts an *x* that is an instance of any subclass of *atype*, or an object that implements protocol *atype*, not just a *direct* instance of *atype* itself. In particular, `isinstance` is fine when you're checking for an abstract base class (see **"Abstract Base Classes"**) or protocol (see **"Protocols"**); this newer idiom is also sometimes known as ***goose typing*** (again, this phrase is credited to one of this book's authors).

Built-in Functions

Table 8-2 covers Python functions (and some types that, in practice, are only used *as if* they were functions) in the module `builtins`, in alphabetical order. Built-ins' names are *not* keywords. This means you *can* bind, in local or global scope, an identifier that's a built-in name, although we recommend avoiding it (see the following warning!). Names bound in local or global scope override names bound in built-in scope, so 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 often tempting to use natural names such as `input`, `list`, or `filter` for your own variables, but *don't do it*: these are names of built-in Python types or functions, and reusing them for your own purposes makes those built-in types and functions inaccessible. 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.

Many built-in functions cannot be called with named arguments, only with positional ones. In **Table 8-2**, we mention cases in which this limitation does not hold; when it does, we also use the **3.8+** positional-only marker `/`, covered in **"Positional-only marker"**.

Table 8-2. Python's core built-in functions

`__import__`	`__import__(`*`module_name`*`[, `*`globals`*`[, `*`locals`*`[, `*`fromlis`* `/)` Deprecated in modern Python; use `importlib.import_mo` covered in **"Module Loading"**.

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

`all`	`all(`*`seq`*`, /)` *`seq`* is an iterable. `all` returns **`False`** when any item of *`se`* falsy; otherwise, `all` returns **`True`**. Like the operators **`and`** **`or`**, covered in **"Short-Circuiting Operators"**, `all` stops evaluating and returns a result as soon as it knows the ar in the case of `all`, this means it stops as soon as a falsy ite reached, but proceeds throughout *`seq`* if all of *`seq`*'s items truthy. Here is a typical toy example of the use of `all`:

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

When *seq* is empty, `all` returns **`True`**.

`any`	`any(`*`seq`*`, /)`
	seq is an iterable. `any` returns **`True`** if any item of *seq* is tr
	otherwise, `any` returns **`False`**. Like the operators **`and`** and
	covered in **"Short-Circuiting Operators"**, `any` stops eval
	and returns a result as soon as it knows the answer; in th
	of `any`, this means it stops as soon as a truthy item is reac
	but proceeds throughout *seq* if all of *seq*'s items are falsy
	is a typical toy example of the use of `any`:
	
	`if any(x<0 for x in the_numbers):`
	`    print('some of the numbers are negative')`
	`else:`
	`    print('none of the numbers are negative')`
	
	When *seq* is empty, `any` returns **`False`**.

`ascii`	`ascii(`*`x`*`, /)` Like `repr`, but escapes non-ASCII characters in the string returns; the result is usually similar to that of `repr`.
`bin`	`bin(`*`x`*`, /)` Returns a binary string representation of integer *x*. E.g., `bin(23)=='0b10111`'.
`breakpoint`	`breakpoint()` Invokes the `pdb` Python debugger. Set `sys.breakpointhoc`

		callable function if you want `breakpoint` to invoke an alt debugger.
`callable`	`callable(`*`obj`*`, /)`	Returns **`True`** when *`obj`* can be called; otherwise, returns An object can be called if it is a function, method, class, o 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 correspon the integer *`code`* in Unicode. See also `ord` later in this tabl
`compile`	`compile(`*`source`*`, `*`filename`*`, `*`mode`*`)`	Compiles a string and returns a code object usable by `exe` `eval`. `compile` raises `SyntaxError` when *`source`* is not syntactically valid Python. When *`source`* is a multiline compound statement, the last character must be `'\n'`. *`mo`* must be `'eval'` when *`source`* is an expression and the re meant for `eval`; otherwise, *`mode`* must be `'exec'` (for a sir multiple-statement string) or `'single'` (for a string conta single statement) when the string is meant for `exec`. *`file`* must be a string, used only in error messages (if an error occurs). See also `eval` later in this table, and **"compile an Code Objects"**. (`compile` also takes the optional argumen `flags`, `dont_inherit`, `optimize`, and 3.11+ `_feature_ver` though these are rarely used; see the **online documentat** for more information on these arguments.)
`delattr`	`delattr(`*`obj`*`, `*`name`*`, /)`	Removes the attribute *name* from *`obj`*. `delattr(`*`obj`*`, '`*`ide`* is like `del` *`obj.ident`*. If *`obj`* has an attribute named *`name`* because its class has it (as is normally the case, for examp *methods* of *`obj`*), you cannot delete that attribute from *`ob`* You may be able to delete that attribute from the *class*, if metaclass lets you. If you can delete the class attribute, *`ob`*

ceases to have the attribute, and so does every other insta 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(` returns a sorted list of names of attributes of *`obj`*, includi ones coming from *`obj`*'s type or by inheritance. See also v later in this table.

`divmod`

`divmod(dividend, divisor, /)`
Divides two numbers and returns a pair whose items are quotient and remainder. See also `__divmod__` in **Table 4-**

`enumerate`

`enumerate(iterable, start=0)`
Returns a new iterator whose items are pairs. For each su pair, the second item is the corresponding item in `iterab` while the first item is an integer: `start, start+1, start` For example, the following snippet loops on a list `L` of int changing `L` in place by halving every even value:

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

`enumerate` is one of the few built-ins callable with named arguments.

`eval`

`eval(expr[, globals[, locals]], /)`
Returns the result of an expression. *`expr`* may be a code o ready for evaluation, or a string; if a string, `eval` gets a co object by internally calling `compile(expr, '<string>', 'eval')`. `eval` evaluates the code object as an expression, the *`globals`* and *`locals`* dictionaries as namespaces (whe they're missing, `eval` uses the current namespace). `eval` (

	execute statements: it only evaluates expressions. Nevert `eval` is dangerous; avoid it unless you know and trust tha comes from a source that you are certain is safe. See also `ast.literal_eval` (covered in **"Standard Input"**), and **"Dynamic Execution and exec"**.
`exec`	`exec(`*`statement`*`[, `*`globals`*`[, `*`locals`*`]], /)` Like `eval`, but applies to any statement and returns **None**. is very dangerous, unless you know and trust that *staten* comes from a source that you are certain is safe. See also **"Statements"** and **"Dynamic Execution and exec"**.

`filter`	`filter(`*`func`*`, `*`seq`*`, /)`
	Returns an iterator of those items of *seq* for which *func* i *func* can be any callable object accepting a single argume **None**. *seq* can be any iterable. When *func* is callable, `filt` calls *func* on each item of *seq*, just like the following gene expression:
	`(item for item in seq if func(item))`
	When *func* is **None**, `filter` tests for truthy items, just like
	`(item for item in seq if item)`

`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(`*`ob`* `'`*`ident`*`')` is like *obj*`.`*ident*. When *default* is present and is not found in *obj*, `getattr` returns *default* instead of ra

	`AttributeError`. See also **“Object attributes and items”** **“Attribute Reference Basics”**.
`globals`	`globals()` Returns the `__dict__` of the calling module (i.e., the dictic used as the global namespace at the point of call). See als `locals` later in this table. (Unlike `locals()`, the `dict` retu by `globals()` is read/write, and updates to that `dict` are equivalent to ordinary name definitions.)
`hasattr`	`hasattr(`*`obj`*`, `*`name`*`, /)` Returns **`False`** when *`obj`* has no attribute *`name`* (i.e., when `getattr(`*`obj`*`, `*`name`*`)` would raise `AttributeError`); other returns `True`. See also **“Attribute Reference Basics”**.
`hash`	`hash(`*`obj`*`, /)` Returns the hash value for *`obj`*. *`obj`* can be a dictionary ke an item in a set, only if *`obj`* can be hashed. All objects tha compare equal must have the same hash value, even if th of different types. If the type of *`obj`* does not define equal comparison, `hash(`*`obj`*`)` normally returns `id(`*`obj`*`)` (see `id` table and `__hash__` in **Table 4-1**).
`help`	`help([`*`obj`*`, /])` When called without an *`obj`* argument, begins an interact help session, which you exit by entering **`quit`**. When *`obj`* given, `help` prints the documentation for *`obj`* and its attri and returns **`None`**. `help` is useful in interactive Python ses to get a quick reference to an object’s functionality.
`hex`	`hex(`*`x`*`, /)` Returns a hex string representation of `int` *`x`*. See also `__h` **Table 4-4**.
`id`	`id(`*`obj`*`, /)` Returns the integer value that is the identity of *`obj`*. The `i`

	obj is unique and constant during *obj*'s lifetime[a] (but ma reused at any later time after *obj* is garbage-collected, so rely on storing or checking `id` values). When a type or cla does not define equality comparison, Python uses `id` to compare and hash instances. For any objects *x* and *y*, ider check *x* **is** *y* is the same as `id(`*x*`)==id(`*y*`)`, but more reada and better performing.
`input`	`input(`*prompt*`='', /)` Writes *prompt* to standard output, reads a line from stanc input, and returns the line (without `\n`) as a `str`. At end-o input raises `EOFError`.
`isinstance`	`isinstance(`*obj*`, `*cls*`, /)` Returns **`True`** when *obj* is an instance of class *cls* (or any subclass of *cls*, or implements protocol or ABC *cls*); othe returns **`False`**. *cls* can be a tuple whose items are classes **3.10+** multiple types joined using the | operator): in this `isinstance` returns **`True`** when *obj* is an instance of any c items of *cls*; otherwise, it returns **`False`**. See also **"Abstra Base Classes"** and **"Protocols"**.
`issubclass`	`issubclass(`*cls1*`, `*cls2*`, /)` Returns **`True`** when *cls1* is a direct or indirect subclass of or defines all the elements of protocol or ABC *cls2*; other returns **`False`**. *cls1* and *cls2* must be classes. *cls2* can al tuple whose items are classes. In this case, `issubclass` re **`True`** when *cls1* is a direct or indirect subclass of any of t items of *cls2*; otherwise, it returns **`False`**. For any class *C*, `issubclass(`*C*`, `*C*`)` returns **`True`**.
`iter`	`iter(`*obj*`, /)`, `iter(`*func*`, `*sentinel*`, /)` Creates and returns an iterator (an object that you can repeatedly pass to the `next` built-in function to get one ite time; see **"Iterators"**). When called with one argument,

`iter(obj)` normally returns `obj.__iter__()`. When *obj* i
sequence without a special method `__iter__`, `iter(obj)`
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**.

`iter` *(cont.)*

When called with two arguments, the first argument mus callable without arguments, and `iter(`*`func`*`, `*`sentinel`*`)` i 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`** *`ob`* actly equivalent to **`for`** *`x`* **`in`** `iter(`*`obj`*`)`; therefore, do *not* explici `iter` in such a **`for`** statement. That would be redundant and, the 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__` met whose body is just **`return`** `self`, as an iterator's class shou

`len`

`len(`*`container`*`, /)`
Returns the number of items in *`container`*, which may be sequence, a mapping, or a set. See also `__len__` in **"Conta methods"**.

`locals`

`locals()`
Returns a dictionary that represents the current local namespace. Treat the returned dictionary as read-only; t to modify it may or may not affect the values of local vari and might raise an exception. See also `globals` and `vars` table.

`map`

`map(`*`func`*`, `*`seq`*`, /)`,
`map(`*`func`*`, /, *`*`seqs`*`)`

map calls *func* on every item of iterable *seq* and returns a
iterator of the results. When you call map with multiple *se*
iterables, *func* must be a callable object that accepts *n*
arguments (where *n* is the number of *seqs* arguments,). m
repeatedly calls *func* with *n* arguments, one correspondi
from each iterable.
For example, map(*func*, *seq*) is just like the generator
expression:

```
(func(item) for item in seq).map(func, seq1, se
```

is just like the generator expression:

```
(func(a, b) for a, b in zip(seq1, seq2))
```

When map's iterable arguments have different lengths, ma
as if the longer ones were truncated (just as zip itself doe

max	max(*seq*, /, *, key=**None**[, default=...]), max(*args, key=**None**[, default=...]) Returns the largest item in the iterable argument *seq*, or largest one of multiple positional arguments *args*. You ca a key argument, with the same semantics covered in **“So list”**. You can also pass a default argument, the value to if *seq* is empty; when you don't pass default, and *seq* is e max raises ValueError. (When you pass key and/or defau must pass either or both as named arguments.)
min	min(*seq*, /, *, key=**None**[, default=...]), min(*args, key=**None**[, default=...]) Returns the smallest item in the iterable argument *seq*, o smallest one of multiple positional arguments *args*. You c pass a key argument, with the same semantics covered in **“Sorting a list”**. You can also pass a default argument, tl

	value to return if *seq* is empty; when you don't pass `defa` and *seq* is empty, `min` raises `ValueError`. (When you pass and/or `default`, you must pass either or both as named arguments.)
`next`	`next(it[, default], /)` Returns the next item from iterator *it*, which advances t next item. When *it* has no more items, `next` returns *defa* or, when you don't pass *default*, raises `StopIteration`.
`oct`	`oct(x, /)` Converts `int` *x* to an octal string. See also `__oct__` in **Tabl**
`open`	`open(file, mode='r', buffering=-1)` Opens or creates a file and returns a new file object. `open` accepts many, many more optional parameters; see **"The Module"** for details. `open` is one of the few built-ins callable with named argu
`ord`	`ord(ch, /)` Returns an `int` between `0` and `sys.maxunicode` (inclusive corresponding to the single-character `str` argument *ch*. S `chr` earlier in this table.
`pow`	`pow(x, y[, z], /)` When *z* is present, `pow(x, y, z)` returns `(x ** y) % z`. Wh missing, `pow(x, y)` returns *x* ** *y*. See also `__pow__` in **Tab** When *x* is an `int` and *y* is a nonnegative `int`, `pow` returns and uses Python's full value range for `int` (though evalua `pow` for large *x* and *y* integer values may take some time). either *x* or *y* is a `float`, or *y* is `< 0`, `pow` returns a `float` (o `complex`, when `x < 0` and `y != int(y)`); in this case, `pow` `OverflowError` if *x* or *y* is too large.
`print`	`print(/, *args, sep=' ', end='\n', file=sys.stdout` `flush=False)`

	Formats with `str`, and emits to stream `file`, each item of (if any), separated by `sep`, with `end` after all of them; then flushes the stream if `flush` is truthy.

`range`	`range([start=0, ]stop[, step=1], /)`
	Returns an iterator of `int`s in arithmetic progression:
	`start, start+step, start+(2*step), ...`
	When `start` is missing, it defaults to `0`. When `step` is miss defaults to `1`. When `step` is `0`, `range` raises `ValueError`. W `step` is `> 0`, the last item is the largest `start+(i*step)` str less than `stop`. When `step` is `< 0`, the last item is the smal `start+(i*step)` strictly greater than `stop`. The iterator is when `start` is greater than or equal to `stop` and `step` is g than `0`, or when `start` is less than or equal to `stop` and st less than `0`. Otherwise, the first item of the iterator is alwa `start`.
	When what you need is a `list` of `int`s in arithmetic progression, call `list(range(...))`.

`repr`	`repr(obj, /)` Returns a complete and unambiguous string representati *`obj`*. When feasible, `repr` returns a string that you could p `eval` in order to create a new object with the same value See also `str` in **Table 8-1** and `__repr__` in **Table 4-1**.
`reversed`	`reversed(seq, /)` Returns a new iterator object that yields the items of *`seq`* must be specifically a sequence, not just any iterable) in r order.
`round`	`round(number, ndigits=0)` Returns a `float` whose value is `int` or `float` `number` roun `ndigits` digits after the decimal point (i.e., the multiple of

	`ndigits` that is closest to `number`). When two such multipl equally close to `number`, `round` returns the *even* multiple. S today's computers represent floating-point numbers in bi not in decimal, most of `round`'s results are not exact, as th **tutorial** in the docs explains in detail. See also **"The deci Module"** and David Goldberg's famous language-indeper **article** on floating-point arithmetic.
`setattr`	`setattr(`*`obj`*`, `*`name`*`, `*`value`*`, /)` Binds *`obj`*'s attribute *`name`* to *`value`*. `setattr(`*`obj`*`, '`*`ident`* *`val`*`)` is like *`obj`*`.`*`ident`*`=`*`val`*. See also `getattr` earlier in thi table, **"Object attributes and items"**, and **"Setting an attribute"**.

`sorted`	`sorted(`*`seq`*`, /, *, key=`**`None`**`, reverse=`**`False`**`)`
	Returns a list with the same items as iterable *`seq`*, in sorte order. Same as:
	<pre>def sorted(seq, /, *, key=None, reverse=False):
	    result = list(seq)
	    result.sort(key, reverse)
	    return result</pre>
	See **"Sorting a list"** for the meaning of the arguments. If want to pass `key` and/or `reverse`, you *must* pass them by

`sum`	`sum(`*`seq`*`, /, start=0)` Returns the sum of the items of iterable *`seq`* (which shoul numbers, and, in particular, cannot be strings) plus the v `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 **"Buil up a string from pieces"**.

`vars`	`vars([obj, ]/)` When called with no argument, `vars` returns a dictionary all variables that are bound in the current scope (like `loc` covered earlier in this table). Treat this dictionary as reac `vars(obj)` returns a dictionary with all attributes curren bound in *obj*, similar to `dir`, covered earlier in this table. dictionary may be modifiable or not, depending on the ty *obj*.
`zip`	`zip(seq, /, *seqs, strict=False)` Returns an iterator of tuples, where the *n*th tuple contain *n*th item from each of the argument iterables. You must c with at least one (positional) argument, and all positional arguments must be iterable. `zip` returns an iterator with many items as the shortest iterable, ignoring trailing item the other iterable objects. **3.10+** When the iterables have different lengths and `strict` is **`True`**, `zip` raises `ValueErr` once it reaches the end of the shortest iterable. See also `m` earlier in this table and `zip_longest` in Table 8-10.

a Otherwise arbitrary; often, an implementation detail, *obj*'s address in memory

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 8-3 covers the most frequently used attributes of `sys`. 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, which we do not cover in this book; see the online docs for details on this module.

Table 8-3. Functions and attributes of the sys module

`argv`	The list of command-line arguments passed to the main script. `argv[0]` is the name of the main script,[a] or `'-c'` if the command line used the **`-c`** option. See **"The argparse Module"** for one good way to use `sys.argv`.
`audit`	`audit(`*`event`*`, /, *`*`args`*`)` Raises an *audit event* whose name is `str` *`event`* and whose arguments are *`args`*. The rationale for Python's audit system is laid out in exhaustive detail in **PEP 578**; Python itself raises the large variety of events listed in the **online docs**. To *listen* for events, call `sys.addaudithook(`*`hook`*`)`, where *`hook`* is a callable whose arguments are a `str`, the event's name, followed by arbitrary positional arguments. For more details, see the **docs**.
`builtin_ module_names`	A tuple of `str`s: the names 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 when *`value`* is **`None`**; otherwise, it saves *`value`* in the built-in variable whose name is _ (an underscore) and displays it via `repr`:
	
	`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`	When **`True`**, Python does not write a bytecode file (with extension *.pyc*) to disk when it imports a source file (with extension *.py*).
`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) get displayed and/or logged. The original value is available as `sys.__excepthook__`.
`exception`	`exception()` 3.11+ When called within an **`except`** clause, returns the current exception instance (equivalent to `sys.exc_info()[1]`).
`exc_info`	`exc_info()` When the current thread is handling an exception, `exc_info` returns a tuple with three items: the class, object, and traceback for the exception. When the 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"**). If you must hold a reference to an exception *e*, clear *e*'s traceback: `e.__traceback__=`**`None`**.[b]
`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; 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.

`getrecursionlimit`	`getrecursionlimit()` Returns the current limit on the depth of Python's call stack. See also **"Recursion"** and `setrecursionlimit` later in this table.
`getrefcount`	`getrefcount(`*`obj`*`, /)` Returns the reference count of *`obj`*. Reference counts are covered in **"Garbage Collection"**.
`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`).
`maxsize`	The 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; currently, always `1114111` (`0x10FFFF`). The version of the Unicode database used by Python is in `unicodedata.unidata_version`.
`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'`. For Linux, check `sys.platform.startswith('linux')`, for portability among Linux versions. See also the online docs for the module **platform**, which we don't cover in this book.
`ps1`, `ps2`	`ps1` and `ps2` specify the primary and secondary interpreter prompt strings, initially `>>>` and `...`, respectively. These `sys` attributes exist only in interactive interpreter sessions. If you bind either attribute to a non-`str` 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 f'[{self.p}]>>> '
...
>>> class Ps2(object):
...     def __str__(self):
...         return f'[{sys.ps1.p}]... '
...
>>> sys.ps1, sys.ps2 = Ps1(), 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 helps you check, during testing and debugging, that your program degrades gracefully, rather than abruptly crashing with a `RecursionError`, under situations of almost runaway recursion. See also **"Recursion"** and `getrecursionlimit` earlier 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 function `input` reads. The original values are available as `__stdin__`, `__stdout__`, and `__stderr__`. We cover file objects in **"The io Module"**.

`tracebacklimit`	The maximum number of levels of traceback displayed for unhandled exceptions. By default, this attribute is not defined (i.e., there is no limit). When `sys.tracebacklimit` is `<= 0`, Python prints only the exception type and value, without a traceback.
`version`	A string that describes the Python version, build number and date, and C compiler used. Use `sys.version` only for logging or interactive output; to perform version comparisons, use `sys.version_info`.
`version_info`	A `namedtuple` of the `major`, `minor`, `micro`, `releaselevel`, and `serial` fields of the running Python version. For example, in the first post-beta release of Python 3.10, `sys.version_info` was `sys.version_info(major=3, minor=10, micro=0, releaselevel='final', serial=0)`, equivalent to the tuple `(3, 10, 0, 'final', 0)`. This form is defined to be directly comparable between versions; to see if the current version running is greater than or equal to, say, 3.8, you can test `sys.version_info[:3] >= (3, 8, 0)`. (Do *not* do string comparisons of the *string* `sys.version`, since the string `"3.10"` would compare as less than `"3.9"`!)

a It could, of course, also be a path to the script, and/or a symbolic link to it, if that's what you gave Python.

b One of the book's authors had this very problem when memoizing return values and exceptions raised in `pyparsing`: the cached exception tracebacks held many object references and interfered with garbage collection. The solution was to clear the tracebacks of the exceptions before putting them in the cache.

The copy Module

As discussed in **"Assignment Statements"**, assignments in Python do not *copy* the righthand-side object being assigned. Rather, assignments *add references* to the RHS object. When you want a *copy* of object *x*, ask *x* for a copy of itself, or 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, this book's authors prefer 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 those returned by `list(x)` for a list *x* and `copy.copy(x)` for any *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); for these cases, the `copy` module also supplies a `deepcopy` function. These functions are discussed further in **Table 8-4**.

Table 8-4. `copy` module functions

`copy`	`copy(x)` Creates and returns a shallow copy of *x*, for *x* of many types (modules, files, frames, and other internal types, however, are not supported). When *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 (but not necessarily **acyclic**) graph of references. Be aware that to reproduce the graph's exact shape, when references to the same object are met more than once during the walk, you must *not* make distinct copies; rather, you must use *references* to the same copied object. 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 each object already copied to the new object that is its copy. `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 also need to obtain a correspondence map between the identities of originals and copies (the final state of `memo` will then be just such a mapping).

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 ABCs covered in **"Abstract Base Classes"**. Since Python 3.4, the ABCs have been in `collections.abc`; for backward compatibility they could still be accessed directly in `collections` itself until Python 3.9, but this functionality was removed in 3.10.

ChainMap

`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` affect only the very first mapping in `c`. To further explain, you could approximate this as follows:

```
class ChainMap(collections.abc.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 a `MutableMapping` requires. See the **online 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 a key has been seen (although values are allowed to be `<= 0`); it's roughly equivalent to types that other languages call "bag" or "multi-set" types. 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`, with a 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`. Note that removing an entry from `c` (with `del`) may *not* decrement the counter, but `subtract` (described in the following table) does:

```
>>> del c['foo']
>>> c['o']
```

```
4
```

```
>>> c.subtract('foo')
>>> c['o']
```

```
2
```

In addition to `dict` methods, *c* supports the extra methods detailed in **Table 8-5**.

Table 8-5. Methods of a `Counter` instance `c`

`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=None, /, **kwds)` Like `c.update(iterable)` "in reverse"—that is, *subtracting* counts rather than *adding* them. Resulting counts in *c* can be `<= 0`.

total	*c*.total() **3.10+** Returns the sum of all the individual counts. Equivalent to sum(*c*.values()).

Counter objects support common arithmetic operators, such as +, -, &, and | for addition, subtraction, union, and intersection. See the **online docs** for more details and a collection of useful recipes on how to use Counter.

OrderedDict

OrderedDict is a subclass of dict with additional methods to access and manipulate items with respect to their insertion order. *o*.popitem() removes and returns the item at the most recently inserted key; *o*.move_to_end(*key*, last=**True**) moves the item with key *key* to the end (when last is **True**, the default) or to the start (when last is **False**). 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. Since Python 3.7, dict insertion order is guaranteed to be maintained: many uses that previously required OrderedDict can now just use ordinary Python dicts. A significant difference remaining between the two is that OrderedDict's test for equality with other OrderedDicts is order sensitive, while dict's equality test is not. See the **online 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=None, *a, **k):
        super().__init__(*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 usually pass it an extra first argument (before any other arguments, positional and/or named, if any, to pass on to plain `dict`). The extra first argument becomes the initial value of `default_factory`; you can also access and rebind `default_factory` later, though doing so is infrequent in normal Python code.

All behavior of `defaultdict` is essentially as implied by this Python equivalent (except `str` and `repr`, which return strings different from those they would 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, for example, 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.[2]

KEYDEFAULTDICT

A variation on `defaultdict` that is *not* found in the `collections` module is a `defaultdict` whose `default_factory` takes the key as an initialization argument. This example shows how you can implement this for yourself:

```
class keydefaultdict(dict):
    def __init__(self, default_factory=None, *a, **k):
        super().__init__(*a, **k)
        self.default_factory = default_factory
    def __missing__(self, key):
        if self.default_factory is None:
            raise KeyError(key)
        self[key] = self.default_factory(key)
        return self[key]
```

The `dict` class supports the `__missing__` method for subclasses to implement custom behavior when a key is accessed that is not yet in the `dict`. In this example, we implement `__missing__` to call the default factory method with the new key, and add it to the `dict`. You can use `keydefaultdict` rather than `defaultdict` when the `default_factory` requires an argument (most often, this happens when the default factory is a class that takes an identifier constructor argument).

---

deque

`deque` is a sequence type whose instances are "double-ended queues" (additions and removals at either end are fast and thread-safe). A `deque` instance *d* is a mutable sequence, with an optional maximum length, and can be indexed and iterated on (however, *d* cannot be sliced; it can only be indexed one item at a time, whether for access, rebinding, or deletion). If a deque instance *d* has a maximum length, when items are added to either side of *d* so that *d*'s length exceeds that maximum, items are silently dropped from the other side.

`deque` is especially useful for implementing first-in, first-out (FIFO) queues.[3] `deque` is also good for maintaining "the latest *N* things seen," also known in some other languages as a *ring buffer*.

Table 8-6 lists the methods the `deque` type supplies.

Table 8-6. `deque` methods

`deque`	`deque(`*`seq`*`=(), /, maxlen=`**`None`**`)` The initial items of *d* are those of *`seq`*, in the same order. *`d`*`.maxlen` is a read-only attribute: when its value is **`None`**, *d* has no maximum length; when an `int`, it must be `>=0`. *d*'s maximum length is *`d`*`.maxlen`.
`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(`*`iterable`*`, /)` 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).

AVOID INDEXING OR SLICING A DEQUE

`deque` is primarily intended for cases that access, add, and remove items from either the `deque`'s start or end. While indexing or slicing into a `deque` is possible, it may only have `O(n)` performance (vs `O(1)` for `list`) when accessing an inner value using `deque[i]` form. If you must access inner values, consider using a `list` instead.

The functools Module

The `functools` module supplies functions and types supporting functional programming in Python, listed in **Table 8-7**.

Table 8-7. Functions and attributes of the `functools` module

cached_property	`cached_property(func)` 3.8+ A caching version of the `property` decor Evaluating the property the first time caches t returned value, so that subsequent calls can r the cached value instead of repeating the proj calculation. `cached_property` uses a threadin to ensure that the property calculation is performed only once, even in a multithreadec environment.[a]

`lru_cache`, `cache`	`lru_cache(max_size=128, typed=False)`, `cache()` A *memoizing* decorator suitable for decoratin function whose arguments are all hashable, a to the function a cache storing the last `max_si` results (`max_size` should be a power of 2, or **N** to have the cache keep all previous results); w you call the decorated function again with arguments that are in the cache, it immediate returns the previously cached result, bypassin underlying function's body code. When `typed` **True**, arguments that compare equal but have different types, such as 23 and 23.0, are cache separately. **3.9+** If setting `max_size` to **None**, u `cache` instead. For more details and examples the **online docs**. **3.8+** `lru_cache` may also be as a decorator with no ().
`partial`	`partial(`*`func`*`, /, *`*`a`*`, **`*`k`*`)` Returns a callable *p* that is just like *func* (whi any callable), but with some positional and/or named parameters already bound to the valu given in *a* and *k*. In other words, *p* is a *partial application* of *func*, often also known (with debatable correctness, but colorfully, in hono mathematician Haskell Curry) as a *currying* o to the given arguments. For example, say that 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 eve driven programming for some GUIs and networking applications. `partial` returns a callable with the attributes (the wrapped function), `args` (the `tuple` of prebound positional arguments), and `keyworc` (the `dict` of prebound named arguments, or

`reduce`

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

Applies *func* to the items of *seq*, from left to r to reduce the iterable to a single value. *func* n be callable with two arguments. `reduce` calls on the first two items of *seq*, then on the resu the first call and the third item, and so on, anc returns the result of the last such call. When is present, `reduce` uses it before *seq*'s first iter any. When *init* is missing, *seq* must be noner When *init* is missing and *seq* has only one it `reduce` returns *seq*`[0]`. Similarly, when *init* i present and *seq* is empty, `reduce` returns *init* `reduce` is thus roughly equivalent to:

```
def reduce_equiv(func, seq, init=None
    seq = iter(seq)
    if init is None:
```

	<pre> init = next(seq)
	    for item in seq:
	        init = func(init, item)
	    return init</pre> An example use of `reduce` is to compute the product of a sequence of numbers: <pre>prod=reduce(operator.mul, seq, 1)</pre>

`singledispatch`, `singledispatchmethod`	Function decorators to support multiple implementations of a method with differing t for their first argument. See the **online docs** f detailed description.
`total_ordering`	A class decorator suitable for decorating class that supply at least one inequality compariso method, such as `__lt__`, and, ideally, also sup `__eq__`. Based on the class's existing methods, class decorator `total_ordering` adds to the cl all other inequality comparison methods that aren't implemented in the class itself or any o superclasses, removing the need for you to ad boilerplate code for them.
`wraps`	`wraps(wrapped)` A decorator suitable for decorating functions wrap another function, `wrapped` (often nested functions within another decorator). `wraps` co the `__name__`, `__doc__`, and `__module__` attrib of `wrapped` on the decorated function, thus improving the behavior of the built-in functio `help`, and of doctests, covered in **"The doctes Module"**.

[a] In Python versions 3.8 to 3.11, `cached_property` is implemented using a class-le lock. As such, it synchronizes for all instances of the class or any subclass, not j the current instance. Thus, `cached_property` can reduce performance in a mult threaded environment, and is *not* recommended.

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 functions listed in **Table 8-8**.

Table 8-8. Functions of the `heapq` module

`heapify`	`heapify(alist, /)` Permutes `list` *`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 *`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`*, never the *`item`* just 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`* (or less than *n* if *`seq`* has fewer than *n* items); like `sorted(`*`seq`*`, reverse=True)[:`*`n`*`]`, but faster when *n* is "small enough"[a] compared to `len(`*`seq`*`)`. You may also specify a (named or positional) `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`* (or less than *n* if *`seq`* has fewer than *n* items); like `sorted(`*`seq`*`)[:`*`n`*`]`, but faster when *n* is "small enough" compared to `len(`*`seq`*`)`. You may also specify a (named or positional) `key=` argument, like you can for `sorted`.

[a] To find out how specific values of *n* and `len(`*`seq`*`)` affect the timing of `nlargest`, `nsmallest`, and `sorted` on your specific Python version and machine, use `timeit`, covered in "The timeit module".

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***[4] (which used to be crucial to optimize sorting in ancient versions of Python, before the `key=` functionality was introduced).

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

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*.

Sort
: Call `heapq` functions on *A*, typically starting with `heapq.heapify(A)`.[5]

Undecorate
: When you extract an item from *A*, typically by calling `heapq.heappop(A)`, return just the last item of the resulting tuple (which was 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 sequence of operations can be 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 for `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 or within the script, *command-line arguments* (including *command-line options*, which by convention are arguments starting with one or two dash characters). In Python, you can access the arguments as `sys.argv`, an attribute of the module `sys` holding those arguments as a list of strings (`sys.argv[0]` is the name or path by which the user started your program; the arguments are in the 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. `argparse` provides one class, which has the following signature:

`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 always pass is `description=`, a string summarizing the purpose of your program.

Given an instance *ap* of `ArgumentParser`, prepare it with 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 arguments and options as attributes.

`add_argument` has a mandatory first argument: 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 can have both a short name (dash, then a 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. Table 8-9 lists the most commonly used ones.

Table 8-9. Common named arguments to `add_argument`

`action`	What the parser does with this argument. Default: `'store'`, which stores the argument's value in the namespace (at the name given by `dest`, described later in this table). Also useful: `'store_true'` and `'store_false'`, making an option into a `bool` (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 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 leading dashes, if any.
`help`	A `str` describing the argument, for help messages.
`nargs`	The number of command-line arguments used by this logical argument. Default: `1`, stored in the namespace. Can be an `int > 0` (uses that many arguments, stores them as a list), `'?'` (`1` or none, in which case it uses `default`), `'*'` (`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 the 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 8-10 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 **online docs** for more `itertools` attributes, including *combinatorial* generators for permutations, combinations, and Cartesian products, as well as a useful taxonomy of `itertools` attributes.

The online docs also offer recipes describing 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`*.[6]

Table 8-10. Functions and attributes of the `itertools` module

`accumulate`	`accumulate(`*`seq`*`, `*`func`*`, /[, initial=`*`init`*`])` Similar to `functools.reduce(`*`func`*`, `*`seq`*`)`, but returns ar

	of all the intermediate computed values, not just the fin 3.8+ You can also pass an initial value *init*, which wor same way as in functools.reduce (see Table 8-7).
chain	chain(*iterables*) Yields items from the first argument, then items from th argument, and so on, until the end of the last argument. just like the generator expression: `(it for iterable in iterables for it in iterab`
chain.from_ iterable	chain.from_iterable(*iterables*, /) Yields items from the iterables in the argument, in orde the genexp: `(it for iterable in iterables for it in iterab`
compress	compress(*data*, *conditions*, /) Yields each item from *data* corresponding to a true item *conditions*, just like the genexp: `(it for it, cond in zip(data, conditions) if c`

count	count(start=0, step=1)
	Yields consecutive integers starting from *start*, just like
	generator:
	`def count(start=0, step=1):`
	`    while True:`

```
        yield start
        start += step
```

`count` returns an unending iterator, so use it carefully, a
ensuring you explicitly terminate any loop over it.

`cycle`

`cycle(`*`iterable`*`, /)`

Yields each item of *`iterable`*, endlessly repeating items
beginning each time it reaches the end, just like the gen

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

`cycle` returns an unending iterator, so use it carefully, a
ensuring you explicitly terminate any loop over it.

`dropwhile`

`dropwhile(`*`func`*`, `*`iterable`*`, /)`

Drops the `0+` leading items of *`iterable`* for which *`func`* i
then yields each remaining 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
```

filterfalse	`filterfalse(`*func*`,` *iterable*`, /)`
	Yields those items of *iterable* for which *func* is false, ju
	genexp:
	`(it for it in iterable if not func(it))`
	func can be any callable accepting a single argument, or
	When *func* is **None**, `filterfalse` yields false items, just l
	genexp:
	`(it for it in iterable if not it)`

groupby	`groupby(`*iterable*`, /, key=`**None**`)`
	iterable normally needs to be already sorted according
	(**None**, as usual, standing for the identity function, **lambd**
	groupby yields pairs (*k*, *g*), each pair representing a *gro*
	adjacent items from *iterable* having the same value *k* f
	key(*item*); each *g* is an iterator yielding the items in the
	When the groupby object advances, previous iterators *g*
	invalid (so, if a group of items needs to be processed late
	better store somewhere a `list` "snapshot" of it, `list(`*g*`)`
	Another way of looking at the groups groupby yields is t
	terminates as soon as *key*(*item*) changes (which is why
	normally call groupby only on an *iterable* that's alread
	by *key*).
	For example, suppose that, given a `set` of lowercase wor
	want a `dict` that maps each initial to the longest word h
	initial (with "ties" broken arbitrarily). We could write:
	`import itertools as it`
	`import operator`

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

islice	`islice(`*`iterable`*`[, `*`start`*`], `*`stop`*`[, `*`step`*`], /)` Yields items of *`iterable`* (skipping the first *`start`* ones, b `0`) up to but not including *`stop`*, advancing by steps of *`st`* (default `1`) at a time. All arguments must be nonnegative (or **None**), and *`step`* must be > `0`. Apart from checks and 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)
```

pairwise	`pairwise(`*`seq`*`, /)` **3.10+** Yields pairs of items in *`seq`*, with overlap (for exa `pairwise('ABCD')` will yield `'AB'`, `'BC'`, and `'CD'`). Equi the iterator returned from `zip(`*`seq`*`, `*`seq`*`[1:])`.
repeat	`repeat(`*`item`*`, /[, times])` Repeatedly yields *`item`*, just like the genexp:

```python
(item for _ in range(times))
```

When `times` is absent, the iterator is unbounded, yieldir potentially infinite number of items, each of which is th *item*, just like the generator:

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

`starmap`	`starmap(func, iterable, /)`

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

```python
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 trut finishes, just like the generator:

```python
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 that are the same as those of *`iterable`*. The returned ite independent from each other, but they are *not* independ *`iterable`*; avoid altering the object *`iterable`* in any way as you're still using any of the returned iterators.
`zip_longest`	`zip_longest(*`*`iterables`*`, /, fillvalue=`**`None`**`)` Yields tuples with one corresponding item from each of *`iterables`*; stops when the longest of the *`iterables`* is ex behaving as if each of the others was "padded" to that sa length with references to `fillvalue`. If **`None`** is a value th be valid in one or more of the *iterables* (such that it coul confused with **`None`** values used for padding), you can us Python Ellipsis (`...`) or a sentinel object `FILL=object()` `fillvalue`.

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

```
for _ in itertools.repeat(None, 10): pass
```

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

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

1 I.e., according to the **Liskov substitution principle**, a core notion of object-oriented programming.

2 When first introduced, `defaultdict(int)` was commonly used to maintain counts of items. Since `Counter` is now part of the `collections` module, use `Counter` instead of `defaultdict(int)` for the specific task of counting items.

3 For last-in, first-out (LIFO) queues, aka “stacks,” a `list`, with its `append` and `pop` methods, is perfectly sufficient.

4 Also known as the ***Schwartzian transform***.

5 This step is not *quite* a full “sort,” but it looks close enough to call it one, at least if you squint.

6 Some experts recommend `from itertools import *`, but the authors of this book disagree.