Appendix B. Special Method Names

> *“ My specialty is being right when other people are wrong. ”*
>
> *— George Bernard Shaw*

B.1. Diving In

Throughout this book, you’ve seen examples of “special methods” — certain “magic” methods that Python invokes when you use certain syntax. Using special methods, your classes can act like sets, like dictionaries, like functions, like iterators, or even like numbers. This appendix serves both as a reference for the special methods we’ve seen already and a brief introduction to some of the more esoteric ones.

B.2. Basics

If you’ve read the introduction to classes, you’ve already seen the most common special method: the `__init__()` method. The majority of classes I write end up needing some initialization. There are also a few other basic special methods that are especially useful for debugging your custom classes.

Notes	You Want…	So You Write…	And Python Calls…
①	to initialize an instance	`x = MyClass()`	`x.__init__()`
②	the “official” representation as a string	`repr(x)`	`x.__repr__()`
③	the “informal” value as a string	`str(x)`	`x.__str__()`
④	the “informal” value as a byte array	`bytes(x)`	`x.__bytes__()`
⑤	the value as a formatted string	`format(x, format_spec)`	`x.__format__(format_spec)`

1. The `__init__()` method is called *after* the instance is created. If you want to control the actual creation process, use the `__new__()` method.
2. By convention, the `__repr__()` method should return a string that is a valid Python expression.
3. The `__str__()` method is also called when you `print(x)`.
4. *New in Python 3*, since the `bytes` type was introduced.
5. By convention, `format_spec` should conform to the Format Specification Mini-Language. `decimal.py` in the Python standard library provides its own `__format__()` method.

B.3. Classes That Act Like Iterators

In the Iterators chapter, you saw how to build an iterator from the ground up using the `__iter__()` and `__next__()` methods.

Notes	You Want…	So You Write…	And Python Calls…
①	to iterate through a sequence	`iter(seq)`	`seq.__iter__()`
②	to get the next value from an iterator	`next(seq)`	`seq.__next__()`
③	to create an iterator in reverse order	`reversed(seq)`	`seq.__reversed__()`

1. The `__iter__()` method is called whenever you create a new iterator. It’s a good place to initialize the iterator with initial values.
2. The `__next__()` method is called whenever you retrieve the next value from an iterator.
3. The `__reversed__()` method is uncommon. It takes an existing sequence and returns an iterator that yields the items in the sequence in reverse order, from last to first.

As you saw in the Iterators chapter, a `for` loop can act on an iterator. In this loop:

```
for x in seq:
    print(x)
```

Python 3 will call `seq.__iter__()` to create an iterator, then call the `__next__()` method on that iterator to get each value of `x`. When the `__next__()` method raises a `StopIteration` exception, the `for` loop ends gracefully.

B.4. Computed Attributes

Notes	You Want…	So You Write…	And Python Calls…
①	to get a computed attribute (unconditionally)	`x.my_property`	`x.__getattribute__('my_property')`
②	to get a computed attribute (fallback)	`x.my_property`	`x.__getattr__('my_property')`
③	to set an attribute	`x.my_property = value`	`x.__setattr__('my_property', value)`
④	to delete an attribute	`del x.my_property`	`x.__delattr__('my_property')`
⑤	to list all attributes and methods	`dir(x)`	`x.__dir__()`

1. If your class defines a `__getattribute__()` method, Python will call it on *every reference to any attribute or method name* (except special method names, since that would cause an unpleasant infinite loop).
2. If your class defines a `__getattr__()` method, Python will call it only after looking for the attribute in all the normal places. If an instance `x` defines an attribute `color`, `x.color` will *not* call `x.__getattr__('color')`; it will simply return the already-defined value of `x.color`.
3. The `__setattr__()` method is called whenever you assign a value to an attribute.
4. The `__delattr__()` method is called whenever you delete an attribute.
5. The `__dir__()` method is useful if you define a `__getattr__()` or `__getattribute__()` method. Normally, calling `dir(x)` would only list the regular attributes and methods. If your `__getattr__()` method handles a `color` attribute dynamically, `dir(x)` would not list `color` as one of the available

attributes. Overriding the `__dir__()` method allows you to list `color` as an available attribute, which is helpful for other people who wish to use your class without digging into the internals of it.

The distinction between the `__getattr__()` and `__getattribute__()` methods is subtle but important. I can explain it with two examples:

```
class Dynamo:
    def __getattr__(self, key):
        if key == 'color':         ①
            return 'PapayaWhip'
        else:
            raise AttributeError   ②
```

```
>>> dyn = Dynamo()
>>> dyn.color ③
'PapayaWhip'
>>> dyn.color = 'LemonChiffon'
>>> dyn.color ④
'LemonChiffon'
```

① The attribute name is passed into the `__getattr__()` method as a string. If the name is `'color'`, the method returns a value. (In this case, it's just a hard-coded string, but you would normally do some sort of computation and return the result.)

② If the attribute name is unknown, the `__getattr__()` method needs to raise an `AttributeError` exception, otherwise your code will silently fail when accessing undefined attributes. (Technically, if the method doesn't raise an exception or explicitly return a value, it returns `None`, the Python null value. This means that *all* attributes not explicitly defined will be `None`, which is almost certainly not what you want.)

③ The `dyn` instance does not have an attribute named `color`, so the `__getattr__()` method is called to provide a computed value.

④ After explicitly setting `dyn.color`, the `__getattr__()` method will no longer be called to provide a value for `dyn.color`, because `dyn.color` is already defined on the instance.

On the other hand, the `__getattribute__()` method is absolute and unconditional.

```
class SuperDynamo:
    def __getattribute__(self, key):
        if key == 'color':
            return 'PapayaWhip'
        else:
            raise AttributeError
```

```
>>> dyn = SuperDynamo()
>>> dyn.color  ①
'PapayaWhip'
>>> dyn.color = 'LemonChiffon'
>>> dyn.color  ②
'PapayaWhip'
```

① The `__getattribute__()` method is called to provide a value for `dyn.color`.

② Even after explicitly setting `dyn.color`, the `__getattribute__()` method *is still called* to provide a value for `dyn.color`. If present, the `__getattribute__()` method *is called unconditionally* for every attribute and method lookup, even for attributes that you explicitly set after creating an instance.

☞ If your class defines a `__getattribute__()` method, you probably also want to define a `__setattr__()` method and coordinate between them to keep track of attribute values. Otherwise, any attributes you set after creating an instance will disappear into a black hole.

You need to be extra careful with the `__getattribute__()` method, because it is also called when Python looks up a method name on your class.

```
class Rastan:
    def __getattribute__(self, key):
        raise AttributeError           ①
    def swim(self):
        pass
```

```
>>> hero = Rastan()
>>> hero.swim() ②
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in __getattribute__
AttributeError
```

① This class defines a `__getattribute__()` method which always raises an `AttributeError` exception. No attribute or method lookups will succeed.

② When you call `hero.swim()`, Python looks for a `swim()` method in the `Rastan` class. This lookup goes through the `__getattribute__()` method, *because all attribute and method lookups go through the* `__getattribute__()` *method*. In this case, the `__getattribute__()` method raises an `AttributeError` exception, so the method lookup fails, so the method call fails.

B.5. Classes That Act Like Functions

You can make an instance of a class callable — exactly like a function is callable — by defining the `__call__()` method.

Notes	You Want…	So You Write…	And Python Calls…
	to "call" an instance like a function	`my_instance()`	`my_instance.__call__()`

The `zipfile` module uses this to define a class that can decrypt an encrypted zip file with a given password. The zip decryption algorithm requires you to store state during decryption. Defining the decryptor as a class allows you to maintain this state within a single instance of the decryptor class. The state is initialized in the `__init__()` method and updated as the file is decrypted. But since the class is also "callable" like a function, you can pass the instance as the first argument of the `map()` function, like so:

```
# excerpt from zipfile.py
class _ZipDecrypter:
.
.
.
    def __init__(self, pwd):
        self.key0 = 305419896               ①
        self.key1 = 591751049
        self.key2 = 878082192
        for p in pwd:
            self._UpdateKeys(p)

    def __call__(self, c):                  ②
        assert isinstance(c, int)
        k = self.key2 | 2
        c = c ^ (((k * (k^1)) >> 8) & 255)
        self._UpdateKeys(c)
        return c
.
.
.
zd = _ZipDecrypter(pwd)                   ③
bytes = zef_file.read(12)
h = list(map(zd, bytes[0:12]))            ④
```

① The `_ZipDecryptor` class maintains state in the form of three rotating keys, which are later updated in the `_UpdateKeys()` method (not shown here).

② The class defines a `__call__()` method, which makes class instances callable like functions. In this case, the `__call__()` method decrypts a single byte of the zip file, then updates the rotating keys based on the byte that was decrypted.

③ `zd` is an instance of the `_ZipDecryptor` class. The `pwd` variable is passed to the `__init__()` method, where it is stored and used to update the rotating keys for the first time.

④ Given the first 12 bytes of a zip file, decrypt them by mapping the bytes to `zd`, in effect "calling" `zd` 12 times, which invokes the `__call__()` method 12 times, which updates its internal state and returns a resulting byte 12 times.

B.6. Classes That Act Like Sets

If your class acts as a container for a set of values — that is, if it makes sense to ask whether your class "contains" a value — then it should probably define the following special methods that make it act like a set.

Notes	You Want…	So You Write…	And Python Calls…
	the number of items	`len(s)`	`s.__len__()`
	to know whether it contains a specific value	`x in s`	`s.__contains__(x)`

The `cgi` module uses these methods in its `FieldStorage` class, which represents all of the form fields or query parameters submitted to a dynamic web page.

```
# A script which responds to http://example.com/search?q=cgi
import cgi
fs = cgi.FieldStorage()
if 'q' in fs:                                               ①
  do_search()

# An excerpt from cgi.py that explains how that works
class FieldStorage:
.
.
.
    def __contains__(self, key):                            ②
        if self.list is None:
            raise TypeError('not indexable')
        return any(item.name == key for item in self.list)  ③

    def __len__(self):                                      ④
        return len(self.keys())                             ⑤
```

① Once you create an instance of the `cgi.FieldStorage` class, you can use the "`in`" operator to check whether a particular parameter was included in the query string.

② The `__contains__()` method is the magic that makes this work. When you say `if 'q' in fs`, Python looks for the `__contains__()` method on the `fs` object, which is defined in `cgi.py`. The value `'q'` is passed into the `__contains__()` method as the `key` argument.

③ The `any()` function takes a generator expression and returns `True` if the generator spits out any items. The `any()` function is smart enough to stop as soon as the first match is found.

④ The same `FieldStorage` class also supports returning its length, so you can say `len(fs)` and it will call the `__len__()` method on the `FieldStorage` class to return the number of query parameters that it identified.

⑤ The `self.keys()` method checks whether `self.list is None`, so the `__len__` method doesn't need to duplicate this error checking.

B.7. Classes That Act Like Dictionaries

Extending the previous section a bit, you can define classes that not only respond to the "`in`" operator and the `len()` function, but they act like full-blown dictionaries, returning values based on keys.

Notes	You Want…	So You Write…	And Python Calls…
	to get a value by its key	`x[key]`	`x.__getitem__(key)`
	to set a value by its key	`x[key] = value`	`x.__setitem__(key, value)`
	to delete a key-value pair	`del x[key]`	`x.__delitem__(key)`
	to provide a default value for missing keys	`x[nonexistent_key]`	`x.__missing__(nonexistent_key)`

The `FieldStorage` class from the `cgi` module also defines these special methods, which means you can do things like this:

```
# A script which responds to http://example.com/search?q=cgi
import cgi
fs = cgi.FieldStorage()
if 'q' in fs:
  do_search(fs['q'])                              ①
```

```
# An excerpt from cgi.py that shows how it works
class FieldStorage:
.
.
.
    def __getitem__(self, key):                    ②
        if self.list is None:
            raise TypeError('not indexable')
        found = []
        for item in self.list:
            if item.name == key: found.append(item)
        if not found:
            raise KeyError(key)
        if len(found) == 1:
            return found[0]
        else:
            return found
```

① The `fs` object is an instance of `cgi.FieldStorage`, but you can still evaluate expressions like `fs['q']`.

② `fs['q']` invokes the `__getitem__()` method with the `key` parameter set to `'q'`. It then looks up in its internally maintained list of query parameters (`self.list`) for an item whose `.name` matches the given key.

B.8. Classes That Act Like Numbers

Using the appropriate special methods, you can define your own classes that act like numbers. That is, you can add them, subtract them, and perform other mathematical operations on them. This is how fractions are implemented — the `Fraction` class implements these special methods, then you can do things like this:

```
>>> from fractions import Fraction
>>> x = Fraction(1, 3)
>>> x / 3
Fraction(1, 9)
```

Here is the comprehensive list of special methods you need to implement a number-like class.

Notes	You Want…	So You Write…	And Python Calls…
	addition	`x + y`	`x.__add__(y)`
	subtraction	`x - y`	`x.__sub__(y)`
	multiplication	`x * y`	`x.__mul__(y)`
	division	`x / y`	`x.__truediv__(y)`
	floor division	`x // y`	`x.__floordiv__(y)`
	modulo (remainder)	`x % y`	`x.__mod__(y)`
	floor division & modulo	`divmod(x, y)`	`x.__divmod__(y)`
	raise to power	`x ** y`	`x.__pow__(y)`
	left bit-shift	`x << y`	`x.__lshift__(y)`
	right bit-shift	`x >> y`	`x.__rshift__(y)`
	bitwise `and`	`x & y`	`x.__and__(y)`
	bitwise `xor`	`x ^ y`	`x.__xor__(y)`

	bitwise `or`	`x | y`	`x.__or__(y)`

That's all well and good if `x` is an instance of a class that implements those methods. But what if it doesn't implement one of them? Or worse, what if it implements it, but it can't handle certain kinds of arguments? For example:

```
>>> from fractions import Fraction
>>> x = Fraction(1, 3)
>>> 1 / x
Fraction(3, 1)
```

This is *not* a case of taking a `Fraction` and dividing it by an integer (as in the previous example). That case was straightforward: `x / 3` calls `x.__truediv__(3)`, and the `__truediv__()` method of the `Fraction` class handles all the math. But integers don't "know" how to do arithmetic operations with fractions. So why does this example work?

There is a second set of arithmetic special methods with *reflected operands*. Given an arithmetic operation that takes two operands (*e.g.* `x / y`), there are two ways to go about it:

1. Tell `x` to divide itself by `y`, or
2. Tell `y` to divide itself into `x`

The set of special methods above take the first approach: given `x / y`, they provide a way for `x` to say "I know how to divide myself by `y`." The following set of special methods tackle the second approach: they provide a way for `y` to say "I know how to be the denominator and divide myself into `x`."

Notes	You Want…	So You Write…	And Python Calls…

	addition	`x + y`	`y.__radd__(x)`
	subtraction	`x - y`	`y.__rsub__(x)`
	multiplication	`x * y`	`y.__rmul__(x)`
	division	`x / y`	`y.__rtruediv__(x)`
	floor division	`x // y`	`y.__rfloordiv__(x)`
	modulo (remainder)	`x % y`	`y.__rmod__(x)`
	floor division & modulo	`divmod(x, y)`	`y.__rdivmod__(x)`
	raise to power	`x ** y`	`y.__rpow__(x)`
	left bit-shift	`x << y`	`y.__rlshift__(x)`
	right bit-shift	`x >> y`	`y.__rrshift__(x)`
	bitwise `and`	`x & y`	`y.__rand__(x)`
	bitwise `xor`	`x ^ y`	`y.__rxor__(x)`
	bitwise `or`	`x | y`	`y.__ror__(x)`

But wait! There's more! If you're doing "in-place" operations, like `x /= 3`, there are even more special methods you can define.

Notes	**You Want…**	**So You Write…**	**And Python Calls…**
	in-place addition	`x += y`	`x.__iadd__(y)`
	in-place subtraction	`x -= y`	`x.__isub__(y)`
	in-place multiplication	`x *= y`	`x.__imul__(y)`
	in-place division	`x /= y`	`x.__itruediv__(y)`
	in-place floor division	`x //= y`	`x.__ifloordiv__(y)`
	in-place modulo	`x %= y`	`x.__imod__(y)`

	in-place raise to power	`x **= y`	`x.__ipow__(y)`
	in-place left bit-shift	`x <<= y`	`x.__ilshift__(y)`
	in-place right bit-shift	`x >>= y`	`x.__irshift__(y)`
	in-place bitwise `and`	`x &= y`	`x.__iand__(y)`
	in-place bitwise `xor`	`x ^= y`	`x.__ixor__(y)`
	in-place bitwise `or`	`x |= y`	`x.__ior__(y)`

Note: for the most part, the in-place operation methods are not required. If you don't define an in-place method for a particular operation, Python will try the methods. For example, to execute the expression `x /= y`, Python will:

1. Try calling `x.__itruediv__(y)`. If this method is defined and returns a value other than `NotImplemented`, we're done.
2. Try calling `x.__truediv__(y)`. If this method is defined and returns a value other than `NotImplemented`, the old value of `x` is discarded and replaced with the return value, just as if you had done `x = x / y` instead.
3. Try calling `y.__rtruediv__(x)`. If this method is defined and returns a value other than `NotImplemented`, the old value of `x` is discarded and replaced with the return value.

So you only need to define in-place methods like the `__itruediv__()` method if you want to do some special optimization for in-place operands. Otherwise Python will essentially reformulate the in-place operand to use a regular operand + a variable assignment.

There are also a few "unary" mathematical operations you can perform on number-like objects by themselves.

Notes	You Want…	So You Write…	And Python Calls…
	negative number	`-x`	`x.__neg__()`
	positive number	`+x`	`x.__pos__()`
	absolute value	`abs(x)`	`x.__abs__()`

	inverse	`~x`	`x.__invert__()`
	complex number	`complex(x)`	`x.__complex__()`
	integer	`int(x)`	`x.__int__()`
	floating point number	`float(x)`	`x.__float__()`
	number rounded to nearest integer	`round(x)`	`x.__round__()`
	number rounded to nearest `n` digits	`round(x, n)`	`x.__round__(n)`
	smallest integer `>= x`	`math.ceil(x)`	`x.__ceil__()`
	largest integer `<= x`	`math.floor(x)`	`x.__floor__()`
	truncate `x` to nearest integer toward 0	`math.trunc(x)`	`x.__trunc__()`
PEP 357	number as a list index	`a_list[x]`	`a_list[x.__index__()]`

B.9. Classes That Can Be Compared

I broke this section out from the previous one because comparisons are not strictly the purview of numbers. Many datatypes can be compared — strings, lists, even dictionaries. If you're creating your own class and it makes sense to compare your objects to other objects, you can use the following special methods to implement comparisons.

Notes	You Want…	So You Write…	And Python Calls…
	equality	`x == y`	`x.__eq__(y)`
	inequality	`x != y`	`x.__ne__(y)`
	less than	`x < y`	`x.__lt__(y)`
	less than or equal to	`x <= y`	`x.__le__(y)`
	greater than	`x > y`	`x.__gt__(y)`

	greater than or equal to	`x >= y`	`x.__ge__(y)`
	truth value in a boolean context	`if x:`	`x.__bool__()`

☞ If you define a `__lt__()` method but no `__gt__()` method, Python will use the `__lt__()` method with operands swapped. However, Python will not combine methods. For example, if you define a `__lt__()` method and a `__eq__()` method and try to test whether `x <= y`, Python will not call `__lt__()` and `__eq__()` in sequence. It will only call the `__le__()` method.

B.10. Classes That Can Be Serialized

Python supports serializing and unserializing arbitrary objects. (Most Python references call this process "pickling" and "unpickling.") This can be useful for saving state to a file and restoring it later. All of the native datatypes support pickling already. If you create a custom class that you want to be able to pickle, read up on the pickle protocol to see when and how the following special methods are called.

Notes	You Want…	So You Write…	And Python Calls…
	a custom object copy	`copy.copy(x)`	`x.__copy__()`
	a custom object deepcopy	`copy.deepcopy(x)`	`x.__deepcopy__()`
*	to get an object's state before pickling	`pickle.dump(x, file)`	`x.__getstate__()`
*	to serialize an object	`pickle.dump(x, file)`	`x.__reduce__()`
*	to serialize an object (new pickling protocol)	`pickle.dump(x, file, protocol_version)`	`x.__reduce_ex__(protocol_version)`
*	control over how an object is created during unpickling	`x = pickle.load(file)`	`x.__getnewargs__()`
*	to restore an object's state after unpickling	`x = pickle.load(file)`	`x.__setstate__()`

* To recreate a serialized object, Python needs to create a new object that looks like the serialized object, then set the values of all the attributes on the new object. The `__getnewargs__()` method controls how the object is created, then the `__setstate__()` method controls how the attribute values are restored.

B.11. Classes That Can Be Used in a `with` Block

A `with` block defines a runtime context; you “enter” the context when you execute the `with` statement, and you “exit” the context after you execute the last statement in the block.

Notes	You Want…	So You Write…	And Python Calls…
	do something special when entering a `with` block	`with x:`	`x.__enter__()`
	do something special when leaving a `with` block	`with x:`	`x.__exit__(exc_type, exc_value, traceback)`

This is how the `with file` idiom works.

```
# excerpt from io.py:
def _checkClosed(self, msg=None):
    '''Internal: raise an ValueError if file is closed
    '''
    if self.closed:
        raise ValueError('I/O operation on closed file.'
                         if msg is None else msg)
def __enter__(self):
    '''Context management protocol.  Returns self.'''
    self._checkClosed()                                 ①
    return self                                         ②

def __exit__(self, *args):
    '''Context management protocol.  Calls close()'''
    self.close()                                        ③
```

① The file object defines both an `__enter__()` and an `__exit__()` method. The `__enter__()` method checks that the file is open; if it's not, the `_checkClosed()` method raises an exception.
② The `__enter__()` method should almost always return `self` — this is the object that the `with` block will use to dispatch properties and methods.
③ After the `with` block, the file object automatically closes. How? In the `__exit__()` method, it calls `self.close()`.

☞ The `__exit__()` method will always be called, even if an exception is raised inside the `with` block. In fact, if an exception is raised, the exception information will be passed to the `__exit__()` method. See With Statement Context Managers for more details.

For more on context managers, see Closing Files Automatically and Redirecting Standard Output.

B.12. Really Esoteric Stuff

If you know what you're doing, you can gain almost complete control over how classes are compared, how attributes are defined, and what kinds of classes are considered subclasses of your class.

Notes	You Want…	So You Write…	And Python Calls…
	a class constructor	`x = MyClass()`	`x.__new__()`
*	a class destructor	`del x`	`x.__del__()`
	only a specific set of attributes to be defined		`x.__slots__()`
	a custom hash value	`hash(x)`	`x.__hash__()`
	to get a property's value	`x.color`	`type(x).__dict__['color'].__get__(x, type(x))`
	to set a property's value	`x.color = 'PapayaWhip'`	`type(x).__dict__['color'].__set__(x, 'PapayaWhip')`
	to delete a property	`del x.color`	`type(x).__dict__['color'].__del__(x)`
	to control whether an object is an instance of your class	`isinstance(x, MyClass)`	`MyClass.__instancecheck__(x)`
	to control whether a class is a subclass of your class	`issubclass(C, MyClass)`	`MyClass.__subclasscheck__(C)`
	to control whether a class is a subclass of your abstract base class	`issubclass(C, MyABC)`	`MyABC.__subclasshook__(C)`

* Exactly when Python calls the `__del__()` special method is incredibly complicated. To fully understand it, you need to know how Python keeps track of objects in memory. Here's a good article on Python garbage collection and class destructors. You should also read about weak references, the `weakref` module, and probably the `gc` module for good measure.

B.13. Further Reading

Modules mentioned in this appendix:

- `zipfile` module
- `cgi` module
- `collections` module
- `math` module
- `pickle` module
- `copy` module
- `abc` ("Abstract Base Classes") module

Other light reading:

- Format Specification Mini-Language
- Python data model
- Built-in types
- PEP 357: Allowing Any Object to be Used for Slicing
- PEP 3119: Introducing Abstract Base Classes