BASICS

If you've seen the Introduction to Python presentation, 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.

CLASSES THAT ACT LIKE ITERATORS

In the Iterators section, 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	`iter(seq)`	`seq.__iter__()`

	sequence		
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.

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.

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.

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)`

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`

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:

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:...

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:

...

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)`
		`x ^= y`	`x.__ixor__(y)`

	in-place bitwise xor		
	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*__()]

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.

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	pickle.dump(x, *file*, *protocol_version*)	x.__*reduce_ex*__(*protocol_versi*

	protocol)		
*	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.

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)`

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. Refer to an 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.