Is it a class or a function? It's a callable!

Apr 16th, 2019 10:20 am | Comments

If you search course curriculum I've written, you'll often find phrases like "`zip` function", "`enumerate` function", and "`list` function". Those terms are all technically misnomers.

When I use terms like "the `bool` function" and "the `str` function" I'm incorrectly implying that `bool` and `str` are functions. But these **aren't functions: they're classes**!

I'm going to explain why this confusion between classes and functions happens in Python and then explain **why this distinction often doesn't matter**.

Class or function?

When I'm training a new group of Python developers, there's group activity we often do: the class or function game.

In **the class or function game**, we take something that we "call" (using parenthesis: `()`) and we guess whether it's a class or a function.

For example:

- We can call `zip` with a couple iterables and we get another iterable back, so is `zip` a class or a function?
- When we call `len`, are we calling a class or a function?
- What about `int`: when we write `int('4')` are we calling a class or a function?

Python's `zip`, `len`, and `int` are all often guessed to be functions, but **only one of these is really a function**:

```
>>> zip
<class 'zip'>
>>> len
<built-in function len>
>>> int
<class 'int'>
```

While `len` is a function, `zip` and `int` are classes.

The `reversed`, `enumerate`, `range`, and `filter` "functions" also aren't really functions:

```
>>> reversed
<class 'reversed'>
>>> enumerate
<class 'enumerate'>
>>> range
<class 'range'>
>>> filter
<class 'filter'>
```

After playing the class or function game, we always discuss **callables**, and then we discuss the fact that **we often don't care whether something is a class or a function**.

What's a callable?

A **callable** is anything you can *call*, using parenthesis, and possibly passing arguments.

All three of these lines involve callables:

```
>>> something()
>>> x = AnotherThing()
>>> something_else(4, 8, *x)
```

We don't know what `something`, `AnotherThing`, and `something_else` do: but we *know* they're callables.

We have a number of callables in Python:

- Functions are callables
- Classes are callables
- Methods (which are functions that hang off of classes) are callables
- Instances of classes can even be turned into callables

Callables are a pretty important concept in Python.

Classes are callables

Functions are the most obvious callable in Python. Functions can be “called” in every programming language. A *class* being callable is a bit more u

In JavaScript we can make an “instance” of the `Date` class like this:

```
> new Date(2020, 1, 1, 0, 0)
2020-02-01T08:00:00.000Z
```

In JavaScript the class instantiation syntax (the way we create an “instance” of a class) involves the `new` keyword. In Python we don’t have a `new` ke

In Python we can make an “instance” of the `datetime` class (from `datetime`) like this:

```
>>> datetime(2020, 1, 1, 0, 0)
datetime.datetime(2020, 1, 1, 0, 0)
```

In Python, the syntax for **instantiating a new class instance** is the same as the syntax for **calling a function**. There’s no `new` needed: we just call th

When we **call a function**, we get its return value. When we **call a class**, we get an “instance” of that class.

We use the same syntax for constructing objects from classes and for calling functions: this fact is the main reason the word “callable” is such an important part of our Python vocabulary.

Disguising classes as functions

There are many classes-which-look-like-functions among the Python built-ins and in the Python standard library.

I sometimes explain **decorators** (an intermediate-level Python concept) as “functions which accept functions and return functions”.

But that’s not an entirely accurate explanation. There are also **class decorators**: functions which accept classes and return classes. And there are also **decorators which are implemented using classes**: classes which accept functions and return objects.

A better explanation of the term decorators might be “callables which accept callables and return callables” (still not entirely accurate, but good enough for our purposes).

Python’s property decorator seems like a function:

```
>>> class Circle:
...     def __init__(self, radius):
...         self.radius = radius
...     @property
...     def diameter(self):
...         return self.radius * 2
...
>>> c = Circle(5)
>>> c.diameter
10
```

But it’s a class:

```
>>> property
<class 'property'>
```

The `classmethod` and `staticmethod` decorators are *also* classes:

```
>>> classmethod
<class 'classmethod'>
>>> staticmethod
<class 'staticmethod'>
```

What about context managers, like suppress and redirect_stdout from the `contextlib` module? These both use the snake_case naming convention, s

```
>>> from contextlib import suppress
>>> from io import StringIO
>>> with suppress(ValueError):
...     int('hello')
...
>>> with redirect_stdout(StringIO()) as fake_stdout:
...     print('hello!')
...
>>> fake_stdout.getvalue()
'hello!\n'
```

But they're actually **implemented using classes**, despite the `snake_case` naming convention:

```
>>> suppress
<class 'contextlib.suppress'>
>>> redirect_stdout
<class 'contextlib.redirect_stdout'>
```

Decorators and context managers are just two places in Python where you'll often see callables which look like functions but aren't. Whether a **callable** is a class or a function is often **just an implementation detail**.

It's not really a mistake to refer to `property` or `redirect_stdout` as functions because **they may as well be functions**. We can **call** them, and that's what we care about.

Callable objects

Python's "call" syntax, those `(...)` parenthesis, can **create a class instance** or **call a function**. But this "call" syntax can **also be used to call an object**.

Technically, everything in Python "is an object":

```
>>> isinstance(len, object)
True
>>> isinstance(range, object)
True
>>> isinstance(range(5), object)
True
```

But we often use the term "object" to imply that we're working with an instance of a class (by *instance of a class* I mean "the thing you get back when you call a class").

There's a partial function which lives in the `functools` module, which can "partially evaluate" a function by storing arguments to be used when calling the function later. This is often used to make Python look a bit more like a functional programming language:

```
>>> from functools import partial
>>> just_numbers = partial(filter, str.isdigit)
>>> list(just_numbers(['4', 'hello', '50']))
['4', '50']
```

I said above that Python has “a `partial` function”, which is both true and false.

While the phrase “a `partial` function” makes sense, the `partial` callable **isn’t implemented using a function**.

```
>>> partial
<class '__main__.partial'>
```

The Python core developers *could* have implemented `partial` as a function, like this:

```
def partial(func, *args, **kwargs):
    """Return "partially evaluated" version of given function/arguments."""
    def wrapper(*more_args, **more_kwargs):
        all_kwargs = {**kwargs, **more_kwargs}
        return func(*args, *more_args, **all_kwargs)
    return wrapper
```

But instead they chose to use a class, doing something more like this:

```
class partial:
    """Return "partially evaluated" version of given function/arguments."""
    def __init__(self, func, *args, **kwargs):
        self.func, self.args, self.kwargs = func, args, kwargs
    def __call__(self, *more_args, **more_kwargs):
        all_kwargs = {**self.kwargs, **more_kwargs}
        return self.func(*self.args, *more_args, **all_kwargs)
```

That `__call__` method allows us to *call* `partial` objects. So the `partial` class makes a **callable object**.

Adding a `__call__` method to any class will **make instances of that class callable**. In fact, checking for a `__call__` method is one way to ask the question “is this object callable?”

All functions, classes, and callable objects have a `__call__` method:

```
>>> hasattr(open, '__call__')
True
>>> hasattr(dict, '__call__')
True
>>> hasattr({}, '__call__')
False
```

Though a better way to check for callability than looking for a `__call__` is to use the built-in `callable` function:

```
>>> callable(len)
True
>>> callable(list)
True
>>> callable([])
False
```

In Python, classes, functions, and instances of classes can all be used as “callables”.

The distinction between functions and classes often doesn’t matter

The Python documentation has a page called Built-in Functions. But this Built-in Functions page **isn't actually for built-in functions**: it's for built-in callables.

Of the 69 "built-in functions" listed in the Python Built-In Functions page, **only 42 are actually implemented as functions**: 26 are classes and 1 (h

Of the 26 classes among those built-in "functions", four *were* actually functions in Python 2 (the now-lazy `map`, `filter`, `range`, and `zip`) but have sin

The Python built-ins and the standard library are both full of maybe-functions-maybe-classes.

operator.itemgetter

The `operator` module has lots of callables:

```
>>> from operator import getitem, itemgetter
>>> get_a_and_b = itemgetter('a', 'b')
>>> d = {'a': 1, 'b': 2, 'c': 3}
>>> get_a_and_b(d)
(1, 2)
>>> getitem(d, 'a'), getitem(d, 'b')
(1, 2)
```

Some of these callables (like itemgetter are *callable classes*) while others (like `getitem`) are functions:

```
>>> itemgetter
<class 'operator.itemgetter'>
>>> get_a_and_b
operator.itemgetter('a', 'b')
>>> getitem
<built-in function getitem>
```

The `itemgetter` class *could* have been implemented as "a function that returns a function". Instead it's a class which implements a `__call__` method, so its class instances are callable.

Iterators

Generator functions are functions which return iterators when called (generators are iterators):

```
def count(n=0):
    """Generator that counts upward forever."""
    while True:
        yield n
        n += 1
```

And iterator classes are classes which return iterators when called:

```
class count:
    """Iterator that counts upward forever."""
    def __init__(self, n=0):
        self.n = n
    def __iter__(self):
        return self
    def __next__(self):
        n = self.n
        self.n += 1
        return n
```

Iterators can be defined using functions or using classes: whichever you choose is an implementation detail.

The sorted “key function”

The built-in sorted function has an optional `key` argument, which is called to get “comparison keys” for sorting (`min` and `max` have a similar `key` argu

This `key` argument can be a function:

```
>>> def digit_count(s): return len(s.replace('_', ''))
...
>>> numbers = ['400', '2_020', '800_000']
>>> sorted(numbers, key=digit_count)
['400', '2_020', '800_000']
```

But it can also be a class:

```
>>> numbers = ['400', '2_020', '800_000']
>>> sorted(numbers, key=int)
['400', '2_020', '800_000']
```

The Python documentation says “key specifies a function of one argument…”. That’s not *technically* correct because key can be any callable, not just a function. But **we often use the words “function” and “callable” interchangeably** in Python, and that’s okay.

The defaultdict “factory function”

The defaultdict class in the `collections` module accepts a “factory” callable, which is used to generate default values for missing dictionary items.

Usually we use a class as a `defaultdict` factory:

```
>>> from collections import defaultdict
>>> counts = defaultdict(int)
>>> counts['snakes']
0
>>> things = defaultdict(list)
>>> things['newer'].append('Python 3')
>>> things['newer']
['Python 3']
```

But `defaultdict` can also accept a function (or any other callable):

```
>>> import random
>>> colors = ['blue', 'yellow', 'purple', 'green']
>>> favorite_colors = defaultdict(lambda: random.choice(colors))
>>> favorite_colors['Kevin']
'yellow'
>>> favorite_colors['Stacy']
'green'
>>> probabilities = defaultdict(random.random)
>>> probabilities['having fun']
0.6714530824158086
>>> probabilities['seeing a snake']
0.07703364911089605
```

Pretty much anywhere a “callable” is accepted in Python, a function, a class, or some other callable object will work just fine.

Think in terms of “callables” not “classes” or “functions”

In the Python Morsels exercises I send out every week, I often ask learners to make a “callable”. Often I’ll say something like “this week I’d like yo

I say “callable” because I want an iterator back, but I really don’t care whether the callable created is a **generator function**, an **iterator class**, or a **f** hese things are *callables* which return the right type that I’m testing for (an iterator). It’s up to you, the implementor of this callable, to determine how yo

We practice **duck typing** in Python: **if it looks like a duck and quacks like a duck, it’s a duck**. Because of duck typing we tend to use general ter tors are generators, dictionaries are mappings, and functions are callables.

If something looks like a callable and quacks (or rather, calls) like a callable, it’s a callable. Likewise, if something looks like a function and quacks **n if it’s actually implemented using a class or a callable object**!

Callables accept arguments and return something useful to the caller. When we *call* classes we get instances of that class back. When we *call* functi distinction between a class and a function is **rarely important from the perspective of the caller**.

When talking about passing functions or class objects around, try to think in terms of *callables*. **What happens when you call something** is often r

More importantly though, if someone mislabels a function as a class or a class as a function, **don’t correct them unless the distinction is actually** ble: the distinction between these two can often be disregarded.

Want some practice with callables?

You don’t learn by putting more information into your head. You learn through recall, that is trying to retrieve information for your head.

If you’d like to get some practice with the `__call__` method, if you’d like to make your own iterable/iterator-returning callables, or if you just want to practice working with “callables”, I have a Python Morsels exercise for you.

Python Morsels is a weekly Python skill-building service. I send one exercise every week and the first 5 are free.

If you sign up for Python Morsels using the below form, I’ll send you one callable-related exercise of your choosing (choose using the selection below).

Which Python exercise would you like right now?

○ Novice exercise with a bonus involving callables
○ Making a callable which returns an iterable
○ Making an iterator-returning callable
○ Advanced exercise with bonuses showing the downside of `__call__`

Your email address | Get my Python Morsels exercise

I won’t share you info with others (see the Python Morsels Privacy Policy for details).
This form is reCAPTCHA protected (Google Privacy Policy & TOS)

Posted by Trey Hunner Apr 16th, 2019 10:20 am python

Tweet

« The problem with inheriting from dict and list in Python Python built-ins worth learning »

Comments

ALSO ON **TREY'S BLOG**

Overusing list comprehensions …
2 years ago • 15 comments
List comprehensions are one of my favorite features in Python. I love list …

How to have a great first PyCon
3 years ago • 1 comment
Update (2019): I originally left the sprints out of this article completely! To …

Keyword (Named) Arguments in …
3 years ago • 19 comments
Keyword arguments are one of those Python features that often seems a little …

Py ite
3 ye
Aft Py ask

4 Comments **Trey's Blog** 🔒 **Disqus' Privacy Policy** 1 **Login**

♡ **Recommend** 4 Tweet Share **Sort by Best**

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Graham Wideman • 2 years ago

As usual, thanks for the thoughtful post. But.... I think you would do an even greater service to clarify this: "A callable is anything you can call, using parenthesis, and possibly passing arguments." This seems to imply that "callable" is synonymous with (or at least a 1:1 correspondence with) using parentheses suffix. But the two concepts (callable; uses parentheses suffix) are separate. Prime examples: Properties do not require using parentheses, and "callable(somobj.someprop)" returns true. Decorators. These do not require using parentheses, yet they behave like functions, and "callable(somedecorator)" returns true. I'd even argue that the fact that decorator syntax doesn't use parens presents a speedbump to learning that a decorator behaves as function (that takes a function and returns a function), given the predominance of docs that imply "function calls always involve parens".

3 ^ | ⌄ • Reply • Share ›

Graham Wideman • 2 years ago

An additional way to undermine over-focus on the distinction between callables that happen to be functions, objects or classes: First, a class is also an object in its own right, so if objects are callable (can have a dunder-call method), so must be classes. Second, functions are also objects, as can be demonstrated by:

```
def myfunc():
...
myfunc.a = 'something'
print(myfunc.a)
```

And since functions are objects, they too must be callable :-).

In short, objects, classes and functions are all examples of the same basic thing. We talk about them specifically as 'classes', or 'objects' or 'functions' when we want to emphasize a specific aspect of their behavior, for which they might be specialized. And when we want the behavior of handing execution to that thing, possibly with some arguments, and expecting a result returned, (ie: "call the thing"), Python usually requires us to use parens syntax, but sometimes not.

2 ^ | ˅ • Reply • Share ›

Adrian • 2 years ago

What a great piece, thanks for the info!

2 ^ | ˅ • Reply • Share ›

Nitin Cherian • a year ago

Hi! My name is Trey Hunner.

I help Python teams **write better Python code** through Python team training.

I also help individuals **level-up their Python skills** with weekly Python skill-building.

Python Team Training

Write Pythonic code

Write more Pythonic code ↑

I send out 1 Python exercise every week through a Python skill-building service called Python Morsels.

If you'd like to **improve your Python skills every week**, sign up!

email@domain.com

Sign me up for Python Morsels!

You can find the Privacy Policy here.
reCAPTCHA protected (Google Privacy Policy & TOS)

The best way to improve your skills is to **write more code**, but it's time consuming to figure out what code to write. I've made a Python skill-building service to help solve this problem.

Each week you'll get an exercise that'll help you dive deeper into Python and carefully **reflect on your own coding style**. The first 4 exercises are free.

Sign up below for **four free exercises**!

Your email [Sign up]

See the Python Morsels Privacy Policy.
This form is reCAPTCHA protected (see Google Privacy Policy & Terms of Service)

Favorite Posts

- Python List Comprehensions
- How to Loop With Indexes in Python
- Check Whether All Items Match a Condition in Python
- Keyword (Named) Arguments in Python: How to Use Them
- Tuple unpacking improves Python code readability
- The Idiomatic Way to Merge Dictionaries in Python
- The Iterator Protocol: How for Loops Work in Python
- Craft Your Python Like Poetry
- Python: range is not an iterator!
- Counting Things in Python: A History

Privacy - Terms

Follow @treyhunner