Python built-in functions to know

May 21st, 2019 8:40 am | Comments

Also published on Python Morsels

In every Intro to Python class I teach, there's always at least one "how can we be expected to know all this" question.

It's usually along the lines of either:

1. Python has so many functions in it, what's the best way to remember all these?
2. What's the best way to learn the functions we'll need day-to-day like `enumerate` and `range`?
3. How do you know about all the ways to solve problems in Python? Do you memorize them?

There are dozens of built-in functions and classes, hundreds of tools bundled in Python's standard library, and thousands of third-party libraries on PyPI. There's no way anyone could ever memorize all of these things.

I recommend triaging your knowledge:

1. Things I should memorize such that I know them well
2. Things I should know *about* so I can look them up more effectively later
3. Things I shouldn't bother with at all until/unless I need them one day

We're going to look through the Built-in Functions page in the Python documentation with this approach in mind.

This will be a very long article, so I've linked to 5 sub-sections and 25 specific built-in functions in the next section so you can jump ahead if you're pressed for time or looking for one built-in in particular.

Which built-ins should you know about?

I estimate **most Python developers will only ever need about 30 built-in functions**, but which 30 depends on what you're actually doing with Python.

We're going to take a look at all 71 of Python's built-in functions, in a birds eye view sort of way.

I'll attempt to categorize these built-ins into five categories:

1. **Commonly known built-ins**: most newer Pythonistas get exposure to these built-ins pretty quickly out of necessity
2. **Overlooked by new Pythonistas**: these functions are useful to know about, but they're easy to overlook when you're newer to Python
3. **Learn these later**: these built-ins are generally useful to know about, but you'll find them when/if you need them
4. **Maybe learn these eventually**: these can come in handy, but only in specific circumstances
5. **You likely don't need these**: you're unlikely to need these unless you're doing something fairly specialized

The built-in functions in categories 1 and 2 are the **essential built-ins** that nearly all Python programmers should eventually learn about. The built-ins in categories 3 and 4 are the **specialized built-ins**, which are often very useful but your need for them will vary based on your use for Python. And category 5 are **arcane built-ins**, which might be very handy when you need them but which many Python programmers are likely to never need.

Note for pedantic Pythonistas: I will be referring to all of these built-ins as **functions**, even though 27 of them aren't actually functions.

The commonly known built-in functions (which you likely already know about):

1. print
2. len
3. str
4. int
5. float
6. list
7. tuple
8. dict
9. set
10. range

The built-in functions which are often overlooked by newer Python programmers:

1. sum
2. enumerate
3. zip
4. bool
5. reversed
6. sorted
7. min
8. max
9. any
10. all

There are also 5 commonly overlooked built-ins which I recommend knowing about solely because they make debugging easier:

1. dir
2. vars
3. breakpoint
4. type
5. help

In addition to the 25 built-in functions above, we'll also briefly see the other 46 built-ins in the learn it later maybe learn it eventually and you likely don't need these sections.

The 10 commonly known built-in functions

If you've been writing Python code, these built-ins are likely familiar already.

print

You already know the `print` function. Implementing hello world requires `print`.

You may not know about the various keyword arguments accepted by `print` though:

```
>>> words = ["Welcome", "to", "Python"]
>>> print(words)
['Welcome', 'to', 'Python']
>>> print(*words, end="!\n")
Welcome to Python!
>>> print(*words, sep="\n")
Welcome
to
Python
```

You can look up `print` on your own.

len

In Python, we don't write things like `my_list.length()` or `my_string.length`; instead we strangely (for new Pythonistas at least) say `len(my_list)` and `len(my_string)`.

```
>>> words = ["Welcome", "to", "Python"]
>>> len(words)
3
```

Regardless of whether you like this operator-like `len` function, you're stuck with it so you'll need to get used to it.

str

Unlike many other programming languages, Python doesn't have type coercion so you can't concatenate strings and numbers in Python.

```
>>> version = 3
>>> "Python " + version
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

Python refuses to coerce that `3` integer to a string, so we need to manually do it ourselves, using the built-in `str` function (class technically, but as I said, I'll be calling these all functions):

```
>>> version = 3
>>> "Python " + str(version)
'Python 3'
```

int

Do you have user input and need to convert it to a number? You need the `int` function!

The `int` function can convert strings to integers:

```
>>> program_name = "Python 3"
>>> version_number = program_name.split()[-1]
>>> int(version_number)
3
```

You can also use `int` to truncate a floating point number to an integer:

```
>>> from math import sqrt
>>> sqrt(28)
5.291502622129181
```

```
>>> int(sqrt(28))
5
```

Note that if you need to truncate while dividing, the `//` operator is likely more appropriate (though this works differently with negative numbers): `int(3 / 2) == 3 // 2`.

float

Is the string you're converting to a number not actually an integer? Then you'll want to use `float` instead of `int` for this conversion.

```
>>> program_name = "Python 3"
>>> version_number = program_name.split()[-1]
>>> float(version_number)
3.0
>>> pi_digits = '3.14159265358979323846264338327950288419716939375'
>>> len(pi_digits)
50
>>> float(pi_digits)
3.141592653589793
```

You can also use `float` to convert integers to floating point numbers.

In Python 2, we used to use `float` to convert integers to floating point numbers to force float division instead of integer division. "Integer division" isn't a thing anymore in Python 3 (unless you're specifically using the `//` operator), so we don't need `float` for that purpose anymore. So if you ever see `float(x) / y` in your Python 3 code, you can change that to just `x / y`.

list

Want to make a list out of some other iterable?

The `list` function does that:

```
>>> numbers = [2, 1, 3, 5, 8]
>>> squares = (n**2 for n in numbers)
>>> squares
<generator object <genexpr> at 0x7fd52dbd5930>
>>> list_of_squares = list(squares)
>>> list_of_squares
[4, 1, 9, 25, 64]
```

If you know you're working with a list, you could use the `copy` method to make a new copy of a list:

```
>>> copy_of_squares = list_of_squares.copy()
```

But if you don't know what the iterable you're working with is, the `list` function is the more general way to loop over an iterable and copy it:

```
>>> copy_of_squares = list(list_of_squares)
```

You could also use a list comprehension for this, but I wouldn't recommend it.

Note that when you want to make an empty list, using the *list literal syntax* (those `[]` brackets) is recommended:

```
>>> my_list = list()  # Don't do this
>>> my_list = []  # Do this instead
```

Using `[]` is considered more idiomatic since those square brackets (`[]`) actually *look* like a Python list.

tuple

The `tuple` function is pretty much just like the `list` function, except it makes tuples instead:

```
>>> numbers = [2, 1, 3, 4, 7]
>>> tuple(numbers)
(2, 1, 3, 4, 7)
```

If you need a tuple instead of a list, because you're trying to make a hashable collection for use in a dictionary key for example, you'll want to reach for `tuple` over `list`.

dict

The `dict` function makes a new dictionary.

Similar to like `list` and `tuple`, the `dict` function is equivalent to looping over an iterable of key-value pairs and making a dictionary from them.

Given a list of two-item tuples:

```
>>> color_counts = [('red', 2), ('green', 1), ('blue', 3), ('purple', 5)]
```

This:

```
>>> colors = {}
>>> for color, n in color_counts:
...     colors[color] = n
...
>>> colors
{'red': 2, 'green': 1, 'blue' 3, 'purple': 5}
```

Can instead be done with the `dict` function:

```
>>> colors = dict(color_counts)
>>> colors
{'red': 2, 'green': 1, 'blue' 3, 'purple': 5}
```

The `dict` function accepts two types of arguments:

1. **another dictionary** (mapping is the generic term), in which case that dictionary will be copied
2. **a list of key-value tuples** (more correctly, an iterable of two-item iterables), in which case a new dictionary will be constructed from these

So this works as well:

```
>>> colors
{'red': 2, 'green': 1, 'blue' 3, 'purple': 5}
>>> new_dictionary = dict(colors)
>>> new_dictionary
{'red': 2, 'green': 1, 'blue' 3, 'purple': 5}
```

The `dict` function can also accept keyword arguments to make a dictionary with string-based keys:

```
>>> person = dict(name='Trey Hunner', profession='Python Trainer')
>>> person
{'name': 'Trey Hunner', 'profession': 'Python Trainer'}
```

But I very much prefer to use a dictionary literal instead:

```
>>> person = {'name': 'Trey Hunner', 'profession': 'Python Trainer'}
>>> person
{'name': 'Trey Hunner', 'profession': 'Python Trainer'}
```

The dictionary literal syntax is more flexible and a bit faster but most importantly I find that it more clearly conveys the fact that we are creating a dictionary.

Like with `list` and `tuple`, an empty dictionary should be made using the literal syntax as well:

```
>>> my_list = dict()  # Don't do this
>>> my_list = {}  # Do this instead
```

Using `{}` is slightly more CPU efficient, but more importantly it's more idiomatic: it's common to see curly braces (`{}`) used for making dictionaries but `dict` is seen much less frequently.

set

The `set` function makes a new set. It takes an iterable of hashable values (strings, numbers, or other immutable types) and returns a `set`:

```
>>> numbers = [1, 1, 2, 3, 5, 8]
>>> set(numbers)
{1, 2, 3, 5, 8}
```

There's no way to make an empty set with the `{}` set literal syntax (plain `{}` makes a dictionary), so the `set` function is the only way to make an empty set:

```
>>> numbers = set()
>>> numbers
set()
```

Actually that's a lie because we have this:

```
>>> {*()}  # This makes an empty set
set()
```

But that syntax is confusing (it relies on a lesser-used feature of the `*` operator), so I don't recommend it.

range

The `range` function gives us a `range` object, which represents a range of numbers:

```
>>> range(10_000)
range(0, 10000)
>>> range(-1_000_000_000, 1_000_000_000)
range(-1000000000, 1000000000)
```

The resulting range of numbers includes the start number but excludes the stop number (`range(0, 10)` does not include `10`).

The `range` function is useful when you'd like to loop over numbers.

```
>>> for n in range(0, 50, 10):
...     print(n)
...
0
```

```
5 10
6 20
7 30
8 40
```

A common use case is to do an operation `n` times (that's a list comprehension by the way):

```
1 first_five = [get_things() for _ in range(5)]
```

Python 2's `range` function returned a list, which means the expressions above would make very very large lists. Python 3's `range` works like Python 2's `xrange` (though they're a bit different) in that numbers are **computed lazily** as we loop over these `range` objects.

The 10 commonly overlooked built-ins

If you've been programming Python for a bit or if you just taken an introduction to Python class, you probably already knew about the built-in functions above.

I'd now like to show off 10 built-in functions that are very handy to know about, but are more frequently overlooked by new Pythonistas. After this we'll look at 5 built-in functions that you'll likely find handy while debugging.

bool

The `bool` function checks the truthiness of a Python object.

For numbers, **truthiness** is a question of **non-zeroness**:

```
1 >>> bool(5)
2 True
3 >>> bool(-1)
4 True
5 >>> bool(0)
6 False
```

For collections, truthiness is usually a question of **non-emptiness** (whether the collection has a length greater than `0`):

```
1  >>> bool('hello')
2  True
3  >>> bool('')
4  False
5  >>> bool(['a'])
6  True
7  >>> bool([])
8  False
9  >>> bool({})
10 False
11 >>> bool({1: 1, 2: 4, 3: 9})
12 True
13 >>> bool(range(5))
14 True
15 >>> bool(range(0))
16 False
17 >>> bool(None)
18 False
```

Truthiness is kind of a big deal in Python.

Instead of asking questions about the length of a container, many Pythonistas ask questions about truthiness instead:

```
1 # Instead of doing this
2 if len(numbers) == 0:
3     print("The numbers list is empty")
4
5 # Many of us do this
6 if not numbers:
7     print("The numbers list is empty")
```

You likely won't see `bool` used often, but on the occasion that you need to coerce a value to a boolean to ask about its truthiness, you'll want to know about `bool`.

enumerate

Whenever you need to count upward, one number at a time, while looping over an iterable at the same time, the `enumerate` function will come in handy.

That might seem like a very niche task, but it comes up quite often.

For example we might want to keep track of the line number in a file:

```
1 >>> with open('hello.txt', mode='rt') as my_file:
2 ...     for n, line in enumerate(my_file, start=1):
3 ...         print(f"{n:03}", line)
4 ...
5 001 This is the first line of the file
6 002 This is the second line
7 003 This is the last line of the file
```

The `enumerate` function is also very commonly used to keep track of the *index* of items in a sequence.

```
1 def palindromic(sequence):
2     """Return True if the sequence is the same thing in reverse."""
3     for i, item in enumerate(sequence):
4         if item != sequence[-(i+1)]:
5             return False
6     return True
```

Note that you may see newer Pythonistas use `range(len(sequence))` in Python. If you ever see code with `range(len(...))`, you'll almost always want to use `enumerate` instead.

```
1 def palindromic(sequence):
2     """Return True if the sequence is the same thing in reverse."""
3     for i in range(len(sequence)):
4         if sequence[i] != sequence[-(i+1)]:
5             return False
6     return True
```

If `enumerate` is news to you (or if you often use `range(len(...))`), see looping with indexes.

zip

The `zip` function is even more specialized than `enumerate`.

The `zip` function is used for looping over multiple iterables at the same time.

```
1  >>> one_iterable = [2, 1, 3, 4, 7, 11]
2  >>> another_iterable = ['P', 'y', 't', 'h', 'o', 'n']
3  >>> for n, letter in zip(one_iterable, another_iterable):
4  ...     print(letter, n)
5  ...
6  P 2
7  y 1
8  t 3
9  h 4
10 o 7
11 n 11
```

If you ever have to loop over two lists (or any other iterables) at the same time, `zip` is preferred over `enumerate`. The `enumerate` function is handy when you need indexes while looping, but `zip` is great when we care specifically about looping over two iterables at once.

If you're new to `zip`, see looping over multiple iterables at the same time.

Both `enumerate` and `zip` return iterators to us. Iterators are the lazy iterables that power `for` loops.

By the way, if you need to use `zip` on iterables of different lengths, you may want to look up itertools.zip_longest in the Python standard library.

reversed

The `reversed` function, like `enumerate` and `zip`, returns an iterator.

```
1 >>> numbers = [2, 1, 3, 4, 7]
2 >>> reversed(numbers)
3 <list_reverseiterator object at 0x7f3d4452f8d0>
```

The only thing we can do with this iterator is loop over it (but only once):

```
1 >>> reversed_numbers = reversed(numbers)
2 >>> list(reversed_numbers)
3 [7, 4, 3, 1, 2]
4 >>> list(reversed_numbers)
5 []
```

Like `enumerate` and `zip`, `reversed` is a sort of **looping helper function**. You'll pretty much see `reversed` used exclusively in the `for` part of a `for` loop:

```
1 >>> for n in reversed(numbers):
2 ...     print(n)
3 ...
4 7
5 4
6 3
7 1
8 2
```

There are some other ways to reverse Python lists besides the `reversed` function:

```
1 # Slicing syntax
2 for n in numbers[::-1]:
3     print(n)
4
5 # In-place reverse method
6 numbers.reverse()
7 for n in numbers:
8     print(n)
```

But the `reversed` function is **usually the best way to reverse any iterable** in Python.

Unlike the list `reverse` method (e.g. `numbers.reverse()`), `reversed` doesn't mutate the list (it returns an iterator of the reversed items instead).

Unlike the `numbers[::-1]` slice syntax, `reversed(numbers)` doesn't build up a whole new list: the lazy iterator it returns retrieves the next item in reverse as we loop. Also `reversed(numbers)` is a lot more readable than `numbers[::-1]` (which just looks weird if you've never seen that particular use of slicing before).

If we combine the non-copying nature of the `reversed` and `zip` functions, we can rewrite the `palindromic` function (from enumerate above) without taking any extra memory (no copying of lists is done here):

```
1 def palindromic(sequence):
2     """Return True if the sequence is the same thing in reverse."""
```

```
    for n, m in zip(sequence, reversed(sequence)):
        if n != m:
            return False
    return True
```

sum

The `sum` function takes an iterable of numbers and returns the sum of those numbers.

```
>>> sum([2, 1, 3, 4, 7])
17
```

There's not much more to it than that.

Python has lots of helper functions that **do the looping for you**, partly because they pair nicely with generator expressions:

```
>>> numbers = [2, 1, 3, 4, 7, 11, 18]
>>> sum(n**2 for n in numbers)
524
```

min and max

The `min` and `max` functions do what you'd expect: they give you the minimum and maximum items in an iterable.

```
>>> numbers = [2, 1, 3, 4, 7, 11, 18]
>>> min(numbers)
1
>>> max(numbers)
18
```

The `min` and `max` functions compare the items given to them by using the `<` operator. So all values need to be orderable and comparable to each other (fortunately many objects are orderable in Python).

The `min` and `max` functions also accept a `key` function to allow customizing what "minimum" and "maximum" really mean for specific objects.

sorted

The `sorted` function takes any iterable and returns a new list of all the values in that iterable in sorted order.

```
>>> numbers = [1, 8, 2, 13, 5, 3, 1]
>>> words = ["python", "is", "lovely"]
>>> sorted(words)
['is', 'lovely', 'python']
>>> sorted(numbers, reverse=True)
[13, 8, 5, 3, 2, 1, 1]
```

The `sorted` function, like `min` and `max`, compares the items given to it by using the `<` operator, so all values given to it need so to be orderable.

The `sorted` function also allows customization of its sorting via a `key` function (just like `min` and `max`).

By the way, if you're curious about `sorted` versus the `list.sort` method, Florian Dahlitz wrote an article comparing the two.

any and all

The `any` and `all` functions can be paired with a generator expression to determine whether *any* or *all* items in an iterable **match a given condition**.

Our `palindromic` function from earlier checked whether *all* items were equal to their corresponding item in the reversed sequence (is the first value equal to the last, second to the second from last, etc.).

We could rewrite `palindromic` using `all` like this:

```
def palindromic(sequence):
    """Return True if the sequence is the same thing in reverse."""
    return all(
        n == m
        for n, m in zip(sequence, reversed(sequence))
    )
```

Negating the condition and the return value from `all` would allow us to use `any` equivalently (though this is more confusing in this example):

```
def palindromic(sequence):
    """Return True if the sequence is the same thing in reverse."""
    return not any(
        n != m
        for n, m in zip(sequence, reversed(sequence))
    )
```

If the `any` and `all` functions are new to you, you may want to read my article on them: Checking Whether All Items Match a Condition in Python.

The 5 built-ins for debugging

The following 5 functions will be useful for debugging and troubleshooting code.

breakpoint

Need to pause the execution of your code and drop into a Python command prompt? You need `breakpoint`!

Calling the `breakpoint` function will drop you into pdb, the Python debugger. There are many tutorials and talks out there on PDB: here's a short one and here's a long one.

This built-in function was added in Python 3.7. On older versions of Python you can use `import pdb ; pdb.set_trace()` instead.

dir

The `dir` function can be used for two things:

1. Seeing a list of all your local variables
2. Seeing a list of all attributes on a particular object

Here we can see that our local variables, right after starting a new Python shell and then after creating a new variable `x`:

```
>>> dir()
['__annotations__', '__doc__', '__name__', '__package__']
>>> x = [1, 2, 3, 4]
>>> dir()
['__annotations__', '__doc__', '__name__', '__package__', 'x']
```

If we pass that `x` list into `dir` we can see all the attributes it has:

```
>>> dir(x)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__n
```

We can see the typical list methods, `append`, `pop`, `remove`, and more as well as many dunder methods for operator overloading.

vars

The `vars` function is sort of a mashup of two related things: checking `locals()` and testing the `__dict__` attribute of objects.

When `vars` is called with no arguments, it's equivalent to calling the `locals()` built-in function (which shows a dictionary of all local variables and their values).

```
>>> vars()
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__spec__': None, '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>}
```

When it's called with an argument, it accesses the `__dict__` attribute on that object (which on many objects represents a dictionary of all instance attributes).

```
>>> from itertools import chain
>>> vars(chain)
mappingproxy({'__getattribute__': <slot wrapper '__getattribute__' of 'itertools.chain' objects>, '__iter__': <slot wrapper '__iter__' of 'itertools.chain' objects>, '__next__': <slot wrapper '__next__' of 'itertools.chain' objects>, '__new__': <built-in method __new__ of type object at 0x5611ee76fac0>, 'frc
```

If you ever try to use `my_object.__dict__`, you can use `vars` instead.

I usually reach for `dir` just before using `vars`.

type

The `type` function will tell you the type of the object you pass to it.

The type of a class instance is the class itself:

```
>>> x = [1, 2, 3]
>>> type(x)
<class 'list'>
```

The type of a class is its metaclass, which is usually `type`:

```
>>> type(list)
<class 'type'>
>>> type(type(x))
<class 'type'>
```

If you ever see someone reach for `__class__`, know that they could reach for the higher-level `type` function instead:

```
>>> x.__class__
<class 'list'>
>>> type(x)
<class 'list'>
```

The `type` function is sometimes helpful in actual code (especially object-oriented code with inheritance and custom string representations), but it's also useful when debugging.

Note that when *type checking*, the `isinstance` function is usually used instead of `type` (also note that we tend not to type check in Python because we prefer to practice duck typing).

help

If you're in an interactive Python shell (the Python REPL as I usually call it), maybe debugging code using `breakpoint`, and you'd like to know how a certain object, method, or attribute works, the `help` function will come in handy.

Realistically, you'll likely resort to getting help from your favorite search engine more often than using `help`. But if you're already in a Python REPL, it's quicker to call `help(list.insert)` than it would be to look up the `list.insert` method documentation in Google.

Learn these later

There are quite a few built-in functions you'll likely want *eventually*, but you may not need *right now*.

I'm going to mention 14 more built-in functions which are handy to know about, but not worth learning until you actually need to use them.

open

Need to read from a file or write to a file in Python? You need the `open` function!

Don't work with files directly? Then you likely don't need the `open` function!

You might think it's odd that I've put `open` in this section because working with files is so common. While most programmers will read or write to files using `open` at some point, some Python programmers, such as Django developers, may not use the `open` function very much (if at all).

Once you need to work with files, you'll learn about `open`. Until then, don't worry about it.

By the way, you might want to look into pathlib (which is in the Python standard library) as an alternative to using `open`. I love the `pathlib` module so much I've considered teaching files in Python by mentioning `pathlib` first and the built-in `open` function later.

input

The `input` function prompts the user for input, waits for them to hit the Enter key, and then returns the text they typed.

Reading from standard input (which is what the `input` function does) is one way to get inputs into your Python program, but there are so many other ways too! You could accept command-line arguments, read from a configuration file, read from a database, and much more.

You'll learn this once you need to prompt the user of a command-line program for input. Until then, you won't need it. And if you've been writing Python for a while and don't know about this function, you may simply never need it.

repr

Need the programmer-readable representation of an object? You need the `repr` function!

All Python objects have two different string representations: `str` and `repr`. For most objects, the `str` and `repr` representations are the same:

```
>>> str(4), repr(4)
('4', '4')
>>> str([]), repr([])
('[]', '[]')
```

But for some objects, they're different:

```
>>> str('hello'), repr("hello")
('hello', "'hello'")
>>> from datetime import date
>>> str(date(2020, 1, 1)), repr(date(2020, 1, 1))
('2020-01-01', 'datetime.date(2020, 1, 1)')
```

The string representation we see at the Python REPL uses `repr`, while the `print` function relies on `str`:

```
>>> date(2020, 1, 1)
datetime.date(2020, 1, 1)
>>> "hello!"
'hello!'
>>> print(date(2020, 1, 1))
2020-01-01
>>> print("hello!")
hello!
```

You'll see `repr` used when logging, handling exceptions, and implementing dunder methods.

super

If you create classes in Python, you'll likely need to use `super`. The `super` function is pretty much essential whenever you're inheriting from another Python class.

Many Python users rarely create classes. Creating classes isn't an *essential* part of Python, though many types of programming require it. For example, you can't really use the Django web framework without creating classes.

If you don't already know about `super`, you'll end up learning this if and when you need it.

property

The `property` function is a decorator and a descriptor (only click those weird terms if you're extra curious) and it'll likely seem somewhat magical when you first learn about it.

This decorator allows us to create an attribute which will always seem to contain the return value of a particular function call. It's easiest to understand with an example.

Here's a class that uses `property`:

```
class Circle:

    def __init__(self, radius=1):
        self.radius = radius

    @property
    def diameter(self):
        return self.radius * 2
```

Here's an access of that `diameter` attribute on a `Circle` object:

```
>>> circle = Circle()
>>> circle.diameter
2
>>> circle.radius = 5
>>> circle.diameter
10
```

If you're doing object-oriented Python programming (you're making classes a whole bunch), you'll likely want to learn about `property` at some point. Unlike other object-oriented programming languages, **we use properties instead of getter methods and setter methods**.

For more on using properties, see making an auto-updating attribute and customizing what happens when you assign an attribute.

issubclass and isinstance

The `issubclass` function checks whether a class is a subclass of one or more other classes.

```
>>> issubclass(int, bool)
False
>>> issubclass(bool, int)
True
>>> issubclass(bool, object)
True
```

The `isinstance` function checks whether an object is an instance of one or more classes.

```
>>> isinstance(True, str)
False
>>> isinstance(True, bool)
True
>>> isinstance(True, int)
True
>>> isinstance(True, object)
True
```

You can think of `isinstance` as delegating to `issubclass`:

```
>>> issubclass(type(True), str)
False
>>> issubclass(type(True), bool)
True
>>> issubclass(type(True), int)
True
>>> issubclass(type(True), object)
True
```

If you're overloading operators (e.g. customizing what the `+` operator does on your class) you might need to use `isinstance`, but in general we try to avoid strong type checking in Python so we don't see these much.

In Python we usually prefer duck typing over type checking. These functions actually do a bit more than the strong type checking I noted above (the behavior of both can be customized) so it's actually possible to practice a sort of `isinstance`-powered duck typing with abstract base classes like collections.abc.Iterable. But this isn't seen much either (partly because we tend to practice exception-handling and EAFP a bit more than condition-checking and LBYL in Python).

The last two paragraphs were filled with confusing jargon that I may explain more thoroughly in a future serious of articles if there's enough interest.

hasattr, getattr, setattr, and delattr

Need to work with an attribute on an object but the attribute name is dynamic? You need `hasattr`, `getattr`, `setattr`, and `delattr`.

Say we have some `thing` object we want to check for a particular value on:

```
>>> class Thing: pass
...
>>> thing = Thing()
```

The `hasattr` function allows us to check whether the object *has* a certain attribute (note that `hasattr` has some quirks, though most have been ironed out in Python 3):

```
>>> hasattr(thing, 'x')
False
>>> thing.x = 4
>>> hasattr(thing, 'x')
True
```

The `getattr` function allows us to retrieve the value of that attribute (with an optional default if the attribute doesn't exist):

```
>>> getattr(thing, 'x')
4
>>> getattr(thing, 'x', 0)
4
>>> getattr(thing, 'y', 0)
0
```

The `setattr` function allows for setting the value:

```
>>> setattr(thing, 'x', 5)
>>> thing.x
5
```

And `delattr` deletes the attribute:

```
>>> delattr(thing, 'x')
>>> thing.x
Traceback (most recent call last):
```

```
4   File "<stdin>", line 1, in <module>
5 AttributeError: 'Thing' object has no attribute 'x'
```


These functions allow for a specific flavor of metaprogramming and you likely won't see them often.

classmethod and staticmethod

The `classmethod` and `staticmethod` decorators are somewhat magical in the same way the `property` decorator is somewhat magical.

If you have a method that should be callable on either an instance or a class, you want the `classmethod` decorator. Factory methods (alternative constructors) are a common use case for this:

```
1  class RomanNumeral:
2
3      """A Roman numeral, represented as a string and numerically."""
4
5      def __init__(self, number):
6          self.value = number
7
8      @classmethod
9      def from_string(cls, string):
10         return cls(roman_to_int(string))  # function doesn't exist yet
```

It's a bit harder to come up with a good use for `staticmethod`, since you can pretty much always use a module-level function instead of a static method.

```
1  class RomanNumeral:
2
3      """A Roman numeral, represented as a string and numerically."""
4
5      SYMBOLS = {'M': 1000, 'D': 500, 'C': 100, 'L': 50, 'X': 10, 'V': 5, 'I': 1}
6
7      def __init__(self, number):
8          self.value = number
9
10     @classmethod
11     def from_string(cls, string):
12         return cls(cls.roman_to_int(string))
13
14     @staticmethod
15     def roman_to_int(numeral):
16         total = 0
17         for symbol, next_symbol in zip_longest(numeral, numeral[1:]):
18             value = RomanNumeral.SYMBOLS[symbol]
19             next_value = RomanNumeral.SYMBOLS.get(next_symbol, 0)
20             if value < next_value:
21                 value = -value
22             total += value
23         return total
```

The above `roman_to_int` function doesn't require access to the instance *or* the class, so it doesn't even need to be a `@classmethod`. There's no actual need to make this function a `staticmethod` (instead of a `classmethod`): `staticmethod` is just more restrictive to signal the fact that we're not reliant on the class our function lives on.

I find that learning these causes folks to *think* they need them when they often don't. You can go looking for these if you really need them eventually.

next

The `next` function returns the *next* item in an iterator.

Here's a very quick summary of iterators you'll likely run into includes:

- `enumerate` objects
- `zip` objects
- the return value of the `reversed` function
- files (the thing you get back from the `open` function)
- `csv.reader` objects
- generator expressions
- generator functions

You can think of `next` as a way to manually loop over an iterator to get a single item and then break.

```
1  >>> numbers = [2, 1, 3, 4, 7, 11]
2  >>> squares = (n**2 for n in numbers)
3  >>> next(squares)
4  4
5  >>> for n in squares:
6  ...     break
7  ...
8  >>> n
9  1
10 >>> next(squares)
11 9
```

Maybe learn these eventually

We've already covered nearly half of the built-in functions.

The rest of Python's built-in functions definitely aren't useless, but they're a bit more special-purposed.

The 15 built-ins I'm mentioning in this section are things you may eventually need to learn, but it's also very possible you'll never reach for these in your own code.

- **iter**: get an iterator from an iterable: this function powers `for` loops and it can be very useful when you're making helper functions for looping lazily
- **callable**: return `True` if the argument is a callable (I talked about this a bit in my article functions and callables)
- **filter** and **map**: as discussed in map and filter in Python, I recommend using generator expressions instead of `map` and `filter`
- **id**, **locals**, and **globals**: these are great tools for teaching Python and you may have already seen them, but you won't see these much in real Python code
- **round**: you'll look this up if you need to round a number
- **divmod**: this function does a floor division (`//`) and a modulo operation (`%`) at the same time
- **bin**, **oct**, and **hex**: if you need to display a number as a string in binary, octal, or hexadecimal form, you'll want these functions
- **abs**: when you need the absolute value of a number, you'll look this up
- **hash**: dictionaries and sets rely on the `hash` function to test for hashability, but you likely won't need it unless you're implementing a clever de-duplication algorithm
- **object**: this function (yes it's a class) is useful for making unique default values and sentinel values, if you ever need those

You're unlikely to need all the above built-ins, but if you write Python code for long enough you're likely to see nearly all of them.

You likely don't need these

You're unlikely to need these built-ins. There are sometimes really appropriate uses for a few of these, but you'll likely be able to get away with never learning about these.

- **ord** and **chr**: these are fun for teaching ASCII tables and unicode code points, but I've never really found a use for them in my own code
- **exec** and **eval**: for evaluating a string as if it was code
- **compile**: this is related to `exec` and `eval`
- **slice**: if you're implementing `__getitem__` to make a custom sequence, you may need this (some Python Morsels exercises require this actually), but unless you make your own custom sequence you'll likely never see `slice`
- **bytes**, **bytearray**, and **memoryview**: if you're working with bytes often, you'll reach for some of these (just ignore them until then)
- **ascii**: like `repr` but returns an ASCII-only representation of an object; I haven't needed this in my code yet
- **frozenset**: like `set`, but it's immutable (and hashable!); very neat but not something I've needed often
- **aiter** and **anext**: if you're deep into asynchronous programming in Python, you may reach for these to work with asynchronous iterators (just ignore them until then)
- **__import__**: this function isn't really meant to be used by you, use importlib instead
- **format**: this calls the `__format__` method, which is used for string formatting; you usually don't need to call this function directly
- **pow**: the exponentiation operator (`**`) usually supplants this… unless you're doing modulo-math (maybe you're implementing RSA encryption from scratch…?)
- **complex**: if you didn't know that `4j+3` is valid Python code, you likely don't need the `complex` function

There's always more to learn

There are 71 built-in functions in Python (technically only 44 of them are actually functions).

When you're newer in your Python journey, I recommend focusing on **only 25 of these built-in functions** in your own code:

1. The 10 commonly known built-ins
2. The 10 built-ins that are often overlooked
3. The 5 debugging functions

After that there are 14 more built-ins which you'll probably learn later (depending on the style of programming you do).

Then come the 15 built-ins which you may or may not ever end up needing in your own code. Some people love these built-ins and some people never use them: as you get more specific in your coding needs, you'll likely find yourself reaching for considerably more niche tools.

After that I mentioned the last 17 built-ins which you'll likely never need (again, very much depending on how you use Python).

You don't need to learn all the Python built-in functions today. Take it slow: focus on those first 25 important built-ins and then work your way into learning about others if and when you eventually need them.

Posted by Trey Hunner May 21st, 2019 8:40 am python

« Is it a class or a function? It's a callable! Loop Better: a deeper look at iteration in Python »

Comments

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

Beyond Intro to Python

Need to fill in **gaps in your Python knowledge**? I have just the thing.

Intro to Python courses often skip over certain **fundamental Python concepts**. I send emails meant help you internalize those concepts without wasting time.

This isn't an Intro to Python course. It's **Python concepts *beyond* Intro to Python**. Sign up below to get started.

email@domain.com Sign up

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