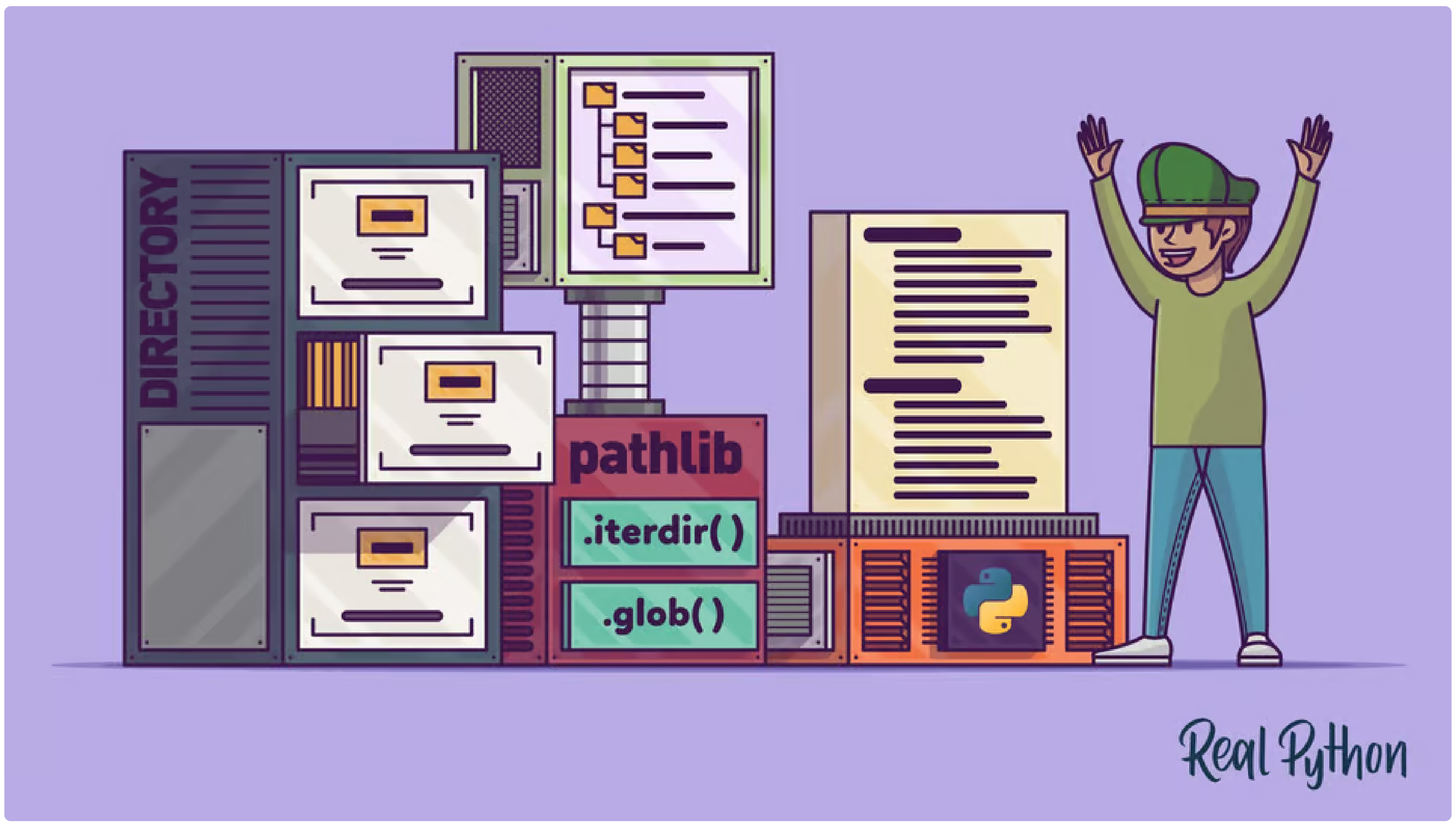


How to Get a List of All Files in a Directory With Python

by Ian Currie Nov 28, 2022 0 Comments
intermediate

Mark as Completed

Tweet Share Email

Table of Contents

Getting a list of all the files and folders in a directory is a natural first step for many **file-related operations** in Python. When looking into it, though, you may be surprised to find various ways to go about it.

When you're faced with many ways of doing something, it can be a good indication that there's no one-size-fits-all solution to your problems. Most likely, every solution will have its own advantages and trade-offs. This is the case when it comes to getting a list of the contents of a directory in Python.

In this tutorial, you'll be focusing on the most general-purpose techniques in the `pathlib` module to list items in a directory, but you'll also learn a bit about some alternative tools.

Source Code: Click here to download the free source code, directories, and bonus materials that showcase different ways to list files and folders in a directory with Python.

Before `pathlib` came out in Python 3.4, if you wanted to work with file paths, then you'd use the `os` module. While this was very efficient in terms of performance, you had to handle all the paths as strings.

Handling paths as strings may seem okay at first, but once you start bringing multiple operating systems into the mix, things get more tricky. You also end up with a bunch of code related to string manipulation, which can get very abstracted from what a file path is. Things can get cryptic pretty quickly.

Note: Check out the downloadable materials for some tests that you can run on your machine. The tests will compare the time it takes to return a list of all the items in a directory using methods from the `pathlib` module, the `os` module, and even the future Python 3.12 version of `pathlib`. That new version includes the well-known `walk()` function, which you won't cover in this tutorial.

That's not to say that working with paths as strings isn't feasible—after all, developers managed fine without `pathlib` for many years! The `pathlib` module just takes care of a lot of the tricky stuff and lets you focus on the main logic of your code.

It all begins with creating a `Path` object, which will be different depending on your operating system (OS). On Windows, you'll get a `WindowsPath` object, while Linux and macOS will return `PosixPath`:

Windows Linux + macOS

Python >>>

```
>>> import pathlib
>>> desktop = pathlib.Path("C:/Users/RealPython/Desktop")
>>> desktop
WindowsPath("C:/Users/RealPython/Desktop")
```

With these OS-aware objects, you can take advantage of the many methods and properties available, such as ones to get a list of files and folders.

Note: If you're interested in learning more about `pathlib` and its features, then check out Python 3's pathlib Module: Taming the File System and the `pathlib` documentation.

Now, it's time to dive into listing folder contents. Be aware that there are several ways to do this, and picking the right one will depend on your specific use case.

Getting a List of All Files and Folders in a Directory in Python

Before getting started on listing, you'll want a set of files that matches what you'll encounter in this tutorial. In the supplementary materials, you'll find a folder called *Desktop*. If you plan to follow along, download this folder and navigate to the *parent* folder and start your Python REPL there:

Source Code: Click here to download the free source code, directories, and bonus materials that showcase different ways to list files and folders in a directory with Python.

You could also use your own desktop too. Just start the Python REPL in the parent directory of your desktop, and the examples should work, but you'll have your own files in the output instead.

Note: You'll mainly see `WindowsPath` objects as outputs in this tutorial. If you're following along on Linux or macOS, then you'll see `PosixPath` instead. That'll be the only difference. The code you write is the same on all platforms.

If you only need to list the contents of a given directory, and you don't need to get the contents of each *subdirectory* too, then you can use the `Path` object's `.iterdir()` method. If your aim is to move through directories and subdirectories recursively, then you can jump ahead to the section on recursive listing.

The `.iterdir()` method, when called on a `Path` object, returns a generator that yields `Path` objects representing child items. If you wrap the generator in a `list()` constructor, then you can see your list of files and folders:

Python

```python
>>> import pathlib
>>> desktop = pathlib.Path("Desktop")

>>> # .iterdir() produces a generator
>>> desktop.iterdir()
<generator object Path.iterdir at 0x000001A8A5110740>

>>> # Which you can wrap in a list() constructor to materialize
>>> list(desktop.iterdir())
[WindowsPath('Desktop/Notes'),
 WindowsPath('Desktop/realpython'),
 WindowsPath('Desktop/scripts'),
 WindowsPath('Desktop/todo.txt')]
```

Passing the generator produced by `.iterdir()` to the `list()` constructor provides you with a list of `Path` objects representing all the items in the *Desktop* directory.

As with all generators, you can also use a `for` loop to iterate over each item that the generator yields. This gives you the chance to explore some of the properties of each object:

Python

```python
>>> desktop = pathlib.Path("Desktop")
>>> for item in desktop.iterdir():
...     print(f"{item} - {'dir' if item.is_dir() else 'file'}")
...
Desktop\Notes - dir
Desktop\realpython - dir
Desktop\scripts - dir
Desktop\todo.txt - file
```

Within the `for` loop body, you use an f-string to display some information about each item.

In the second set of curly braces (`{}`) in the f-string, you use a conditional expression to print *dir* if the item is a directory, or *file* if it isn't. To get this information, you use the `.is_dir()` method.

Putting a `Path` object in an f-string automatically casts the object to a string, which is why you no longer have the `WindowsPath` or `PosixPath` annotation.

Iterating over the object deliberately with a `for` loop like this can be very handy for filtering by either files or directories, as in the following example:

Python

```python
>>> desktop = pathlib.Path("Desktop")
>>> for item in desktop.iterdir():
...     if item.is_file():
...         print(item)
...
Desktop\todo.txt
```

Here, you use a conditional statement and the `.is_file()` method to only print the item if it's a file.

You can also place generators into comprehensions, which can make for very concise code:

Python

```python
>>> desktop = pathlib.Path("Desktop")
>>> [item for item in desktop.iterdir() if item.is_dir()]
[WindowsPath('Desktop/Notes'),
 WindowsPath('Desktop/realpython'),
 WindowsPath('Desktop/scripts')]
```

Here, you're filtering the resulting list by using a conditional expression inside the comprehension to check if the item is a directory.

But what if you need all the files and directories in the subdirectories of your folder too? You can adapt `.iterdir()` as a recursive function, as you'll do later in the tutorial, but you may be better off using `.rglob()`, which you'll get into next.

Recursively Listing With `.rglob()`

Directories are often compared with trees because of their **recursive** nature. In trees, the main trunk splits off into various main branches. Each main branch splits off into further sub-branches. Each sub-branch branches off itself too, and so on. Likewise, directories contain subdirectories, which contain subdirectories, which contain more subdirectories, on and on.

To recursively list the items in a directory means to list not only the directory's contents, but also the contents of the subdirectories, their subdirectories, and so on.

With `pathlib`, it's surprisingly easy to recurse through a directory. You can use `.rglob()` to return absolutely everything:

Python >>>

```python
>>> import pathlib
>>> desktop = pathlib.Path("Desktop")

>>> # .rglob() produces a generator too
>>> desktop.rglob("*")
<generator object Path.glob at 0x000001A8A50E2F00>

>>> # Which you can wrap in a list() constructor to materialize
>>> list(desktop.rglob("*"))
[WindowsPath('Desktop/Notes'),
 WindowsPath('Desktop/realpython'),
 WindowsPath('Desktop/scripts'),
 WindowsPath('Desktop/todo.txt'),
 WindowsPath('Desktop/Notes/hash-tables.md'),
 WindowsPath('Desktop/realpython/iterate-dict.md'),
 WindowsPath('Desktop/realpython/tictactoe.md'),
 WindowsPath('Desktop/scripts/rename_files.py'),
 WindowsPath('Desktop/scripts/request.py')]
```

The `.rglob()` method with `"*"` as an argument produces a generator that yields all the files and folders from the `Path` object recursively.

But what's with the asterisk argument to `.rglob()`? In the next section, you'll look into glob patterns and see how you can do more than just list all the items in a directory.

Using a Python Glob Pattern for Conditional Listing

Sometimes you don't want *all* the files. There are times when you just want one type of file or directory, or perhaps all the items with a certain pattern of characters in their name.

A method related to `.rglob()` is the `.glob()` method. Both of these methods make use of glob patterns. A glob pattern represents a collection of paths. Glob patterns make use of wildcard characters to match on certain criteria. For example, the single asterisk `*` matches everything in the directory.

There are many different glob patterns that you can take advantage of. Check out the following selection of glob patterns for some ideas:

Glob Pattern	Matches
`*`	Every item
`*.txt`	Every item ending in `.txt`, such as `notes.txt` or `hello.txt`
`??????`	Every item whose name is six characters long, such as `01.txt`, `A-01.c`, or `.zshrc`

Glob Pattern	Matches
`A*`	Every item that starts with the character *A*, such as `Album`, `A.txt`, or `AppData`
`[abc][abc][abc]`	Every item whose name is three characters long but only composed of the characters *a*, *b*, and *c*, such as `abc`, `aaa`, or `cba`

With these patterns, you can flexibly match many different types of files. Check out the documentation on `fnmatch`, which is the underlying module governing the behavior of `.glob()`, to get a feel for the other patterns that you can use in Python.

Note that on Windows, glob patterns are case-insensitive, because paths are case-insensitive in general. On Unix-like systems like Linux and macOS, glob patterns are case-sensitive.

Conditional Listing Using `.glob()`

The `.glob()` method of a `Path` object behaves in much the same way as `.rglob()`. If you pass the `"*"` argument, then you'll get a list of items in the directory, but *without recursion*:

Python

```
>>> import pathlib
>>> desktop = pathlib.Path("Desktop")

>>> # .glob() produces a generator too
>>> desktop.glob("*")
<generator object Path.glob at 0x000001A8A50E2F00>

>>> # Which you can wrap in a list() constructor to materialize
>>> list(desktop.glob("*"))
[WindowsPath('Desktop/Notes'),
 WindowsPath('Desktop/realpython'),
 WindowsPath('Desktop/scripts'),
 WindowsPath('Desktop/todo.txt')]
```

Using the `.glob()` method with the `"*"` glob pattern on a `Path` object produces a generator that yields all the items in the directory that's represented by the `Path` object, without going into the subdirectories. In this way, it produces the same result as `.iterdir()`, and you can use the resulting generator in a `for` loop or a comprehension, just as you would with `iterdir()`.

But as you already learned, what really sets the glob methods apart are the different patterns that you can use to match only certain paths. If you only wanted paths that ended with `.txt`, for example, then you could do the following:

Python

```
>>> desktop = pathlib.Path("Desktop")
>>> list(desktop.glob("*.txt"))
[WindowsPath('Desktop/todo.txt')]
```

Since this directory only has one text file, you get a list with just one item. If you wanted to get only items that start with *real*, for example, then you could use the following glob pattern:

Python

```
>>> list(desktop.glob("real*"))
[WindowsPath('Desktop/realpython')]
```

This example also only produces one item, because only one item's name starts with the characters `real`. Remember that on Unix-like systems, glob patterns are case-sensitive.

Note: The *name* here is referred to as the last part of the path, not the other parts of the path, which in this case would start with `Desktop`.

You can also get the contents of a subdirectory by including its name, a forward slash (`/`), and an asterisk. This type of pattern will yield everything inside the target directory:

```
Python >>>
>>> list(desktop.glob("realpython/*"))
[WindowsPath('Desktop/realpython/iterate-dict.md'),
 WindowsPath('Desktop/realpython/tictactoe.md')]
```

In this example, using the `"realpython/*"` pattern yields all the files within the `realpython` directory. It'll give you the same result as creating a path object representing the `Desktop/realpython` path and calling `.glob("*")` on it.

Next up, you'll look a bit further into filtering with `.rglob()` and learn how it differs from `.glob()`.

Conditional Listing Using `.rglob()`

Just the same as with the `.glob()` method, you can adjust the glob pattern of `.rglob()` to give you only a certain file extension, except that `.rglob()` will always search recursively:

```
Python >>>
>>> list(desktop.rglob("*.md"))
[WindowsPath('Desktop/Notes/hash-tables.md'),
 WindowsPath('Desktop/realpython/iterate-dict.md'),
 WindowsPath('Desktop/realpython/tictactoe.md')]
```

By adding `.md` to the glob pattern, now `.rglob()` produces only `.md` files across different directories and subdirectories.

You can actually use `.glob()` and get it to behave in the same way as `.rglob()` by adjusting the glob pattern passed as an argument:

```
Python >>>
>>> list(desktop.glob("**/*.md"))
[WindowsPath('Desktop/Notes/hash-tables.md'),
 WindowsPath('Desktop/realpython/iterate-dict.md'),
 WindowsPath('Desktop/realpython/tictactoe.md')]
```

In this example, you can see that the call to `.glob("**/*.md")` is equivalent to `.rglob(*.md)`. Likewise, a call to `.glob("**/*")` is equivalent to `.rglob("*")`.

The `.rglob()` method is a slightly more explicit version of calling `.glob()` with a recursive pattern, so it's probably better practice to use the more explicit version instead of using recursive patterns with the normal `.glob()`.

Advanced Matching With the Glob Methods

One of the potential drawbacks with the glob methods is that you can only select files based on glob patterns. If you want to do more advanced matching or filter on the attributes of the item, then you need to reach for something extra.

To run more complex matching and filtering, you can follow at least three strategies. You can use:

1. A `for` loop with a conditional check
2. A comprehension with a conditional expression
3. The built-in `filter()` function

Here's how:

Python >>>

```
>>> import pathlib
>>> desktop = pathlib.Path("Desktop")

>>> # Using a for loop
>>> for item in desktop.rglob("*"):
...     if item.is_file():
...         print(item)
...
Desktop\todo.txt
Desktop\Notes\hash-tables.md
Desktop\realpython\iterate-dict.md
Desktop\realpython\tictactoe.md
Desktop\scripts\rename_files.py
Desktop\scripts\request.py

>>> # Using a comprehension
>>> [item for item in desktop.rglob("*") if item.is_file()]
[WindowsPath('Desktop/todo.txt'),
 WindowsPath('Desktop/Notes/hash-tables.md'),
 WindowsPath('Desktop/realpython/iterate-dict.md'),
 WindowsPath('Desktop/realpython/tictactoe.md'),
 WindowsPath('Desktop/scripts/rename_files.py'),
 WindowsPath('Desktop/scripts/request.py')]

>>> # Using the filter() function
>>> list(filter(lambda item: item.is_file(), desktop.rglob("*")))
[WindowsPath('Desktop/todo.txt'),
 WindowsPath('Desktop/Notes/hash-tables.md'),
 WindowsPath('Desktop/realpython/iterate-dict.md'),
 WindowsPath('Desktop/realpython/tictactoe.md'),
 WindowsPath('Desktop/scripts/rename_files.py'),
 WindowsPath('Desktop/scripts/request.py')]
```

In these examples, you've first called the `.rglob()` method with the `"*"` pattern to get all the items recursively. This produces all the items in the directory and its subdirectories. Then you use the three different approaches listed above to filter out the items that aren't files. Note that in the case of `filter()`, you've used a lambda function.

The glob methods are extremely versatile, but for large directory trees, they can be a bit slow. In the next section, you'll be examining an example in which reaching for more controlled iteration with `.iterdir()` may be a better choice.

Opting Out of Listing Junk Directories

Say, for example, that you wanted to find all the files on your system, but you have various subdirectories that have lots and lots of subdirectories and files. Some of the largest subdirectories are temporary files that you aren't interested in.

For example, examine this directory tree that has junk directories—lots of them! In reality, this full directory tree is 1,850 lines long. Wherever you see an ellipsis (`...`), that means that there are hundreds of junk files at that location:

Large directory with junk subdirectories Show/Hide

The issue here is that you have junk directories. The junk directories are sometimes called `temp`, sometimes `temporary files`, and sometimes `logs`. What makes it worse is that they're everywhere and can be at any level of nesting. The good news is that you don't have to list them, as you'll learn next.

Using `.rglob()` to Filter Whole Directories

If you use `.rglob()`, you can just filter out the items once they're produced by `.rglob()`. To properly discard paths that are in a junk directory, you can check if any of the elements in the path match with any of the elements in a list of directories to skip:

```python
>>> SKIP_DIRS = ["temp", "temporary_files", "logs"]
```

Here, you're defining `SKIP_DIRS` as a list that contains the strings of the paths that you want to exclude.

A call to `.rglob()` with a bare asterisk as an argument will produce all the items, even those in the directories that you aren't interested in. Because you have to go through all the items, there's a potential issue if you only look at the name of a path:

```text
large_dir/documents/notes/temp/2/0.txt
```

Since the *name* is just `0.txt`, it wouldn't match any items in `SKIP_DIRS`. You'd need to check the whole path for the blocked name.

You can get all the elements in the path with the `.parts` attribute, which contains a tuple of all the elements in the path:

```python
>>> import pathlib
>>> temp_file = pathlib.Path("large_dir/documents/notes/temp/2/0.txt")
>>> temp_file.parts
('large_dir', 'documents', 'notes', 'temp', '2', '0.txt')
```

Then, all you need to do is to check if any element in the `.parts` tuple is in the list of directories to skip.

You can check if any two iterables have an item in common by taking advantage of sets. If you cast one of the iterables to a set, then you can use the `.isdisjoint()` method to determine whether they have any elements in common:

```python
>>> {"documents", "notes", "find_me.txt"}.isdisjoint({"temp", "temporary"})
True

>>> {"documents", "temp", "find_me.txt"}.isdisjoint({"temp", "temporary"})
False
```

If the two sets have no elements in common, then `.isdisjoint()` returns `True`. If the two sets have at least one element in common, then `.isdisjoint()` returns `False`. You can incorporate this check into a `for` loop that goes over all the items returned by `.rglob("*")`:

```python
>>> SKIP_DIRS = ["temp", "temporary_files", "logs"]
>>> large_dir = pathlib.Path("large_dir")

>>> # With a for loop
>>> for item in large_dir.rglob("*"):
...     if set(item.parts).isdisjoint(SKIP_DIRS):
...         print(item)
...
large_dir\documents
large_dir\documents\0.txt
large_dir\documents\1.txt
large_dir\documents\2.txt
large_dir\documents\3.txt
large_dir\documents\4.txt
large_dir\documents\notes
large_dir\documents\tools
large_dir\documents\notes\0.txt
large_dir\documents\notes\find_me.txt
large_dir\documents\tools\33.txt
large_dir\documents\tools\34.txt
large_dir\documents\tools\36.txt
large_dir\documents\tools\37.txt
large_dir\documents\tools\real_python.txt
```

In this example, you print all the items in `large_dir` that aren't in any of the junk directories.

To check if the path is in one of the unwanted folders, you cast `item.parts` to a set and use `.isdisjoint()` to check if `SKIP_DIRS` and `.parts` *don't* have any items in common. If that's the case, then the item gets printed.

You can also accomplish the same effect with `filter()` and comprehensions, as below:

Python >>>

```python
>>> # With a comprehension
>>> [
...     item
...     for item in large_dir.rglob("*")
...     if set(item.parts).isdisjoint(SKIP_DIRS)
... ]

>>> # With filter()
>>> list(
...     filter(
...         lambda item: set(item.parts).isdisjoint(SKIP_DIRS),
...         large_dir.rglob("*")
...     )
... )
```

These methods are already getting a bit cryptic and hard to follow, though. Not only that, but they aren't very efficient, because the `.rglob()` generator has to produce *all* the items so that the matching operation can discard that result.

You can definitely filter out whole folders with `.rglob()`, but you can't get away from the fact that the resulting generator will yield *all the items* and then filter out the undesirable ones, one by one. This can make the glob methods very slow, depending on your use case. That's why you might opt for a recursive `.iterdir()` function, which you'll explore next.

Creating a Recursive `.iterdir()` Function

In the example of junk directories, you ideally want the ability to *opt out* of iterating over all the files in a given subdirectory if they match one of the names in `SKIP_DIRS`:

Python

```python
# skip_dirs.py

import pathlib

SKIP_DIRS = ["temp", "temporary_files", "logs"]

def get_all_items(root: pathlib.Path, exclude=SKIP_DIRS):
    for item in root.iterdir():
        if item.name in exclude:
            continue
        yield item
        if item.is_dir():
            yield from get_all_items(item)
```

In this module, you define a list of strings, `SKIP_DIRS`, that contains the names of directories that you'd like to ignore. Then you define a generator function that uses `.iterdir()` to go over each item.

The generator function uses the type annotation `: pathlib.Path` after the first argument to indicate that you can't just pass in a string that represents a path. The argument needs to be a `Path` object.

If the item name is in the `exclude` list, then you just move on to the next item, skipping the whole subdirectory tree in one go.

If the item isn't in the list, then you yield the item, and if it's a directory, you invoke the function again on that directory. That is, within the function body, the function conditionally invokes the same function again. This is a hallmark of a recursive function.

This recursive function efficiently yields all the files and directories that you want, excluding all that you aren't interested in:

```python
>>> import pathlib
>>> import skip_dirs
>>> large_dir = pathlib.Path("large_dir")

>>> list(skip_dirs.get_all_items(large_dir))
[WindowsPath('large_dir/documents'),
 WindowsPath('large_dir/documents/0.txt'),
 WindowsPath('large_dir/documents/1.txt'),
 WindowsPath('large_dir/documents/2.txt'),
 WindowsPath('large_dir/documents/3.txt'),
 WindowsPath('large_dir/documents/4.txt'),
 WindowsPath('large_dir/documents/notes'),
 WindowsPath('large_dir/documents/notes/0.txt'),
 WindowsPath('large_dir/documents/notes/find_me.txt'),
 WindowsPath('large_dir/documents/tools'),
 WindowsPath('large_dir/documents/tools/33.txt'),
 WindowsPath('large_dir/documents/tools/34.txt'),
 WindowsPath('large_dir/documents/tools/36.txt'),
 WindowsPath('large_dir/documents/tools/37.txt'),
 WindowsPath('large_dir/documents/tools/real_python.txt')]
```

Crucially, you've managed to opt out of having to examine all the files in the undesired directories. Once your generator identifies that the directory is in the `SKIP_DIRS` list, it just skips the whole thing.

So, in this case, using `.iterdir()` is going to be far more efficient than the equivalent glob methods.

In fact, you'll find that `.iterdir()` is generally more efficient than the glob methods if you need to filter on anything more complex than can be achieved with a glob pattern. However, if all you need to do is to get a list of all the `.txt` files recursively, then the glob methods will be faster.

Check out the downloadable materials for some tests that demonstrate the relative speed of different ways to list files in Python:

Source Code: Click here to download the free source code, directories, and bonus materials that showcase different ways to list files and folders in a directory with Python.

With that information under your belt, you'll be ready to select the best way to list the files and folders that you need!

Conclusion

In this tutorial, you've explored the `.glob()`, `.rglob()`, and `.iterdir()` methods from the Python `pathlib` module to get all the files and folders in a given directory into a list. You've covered listing the files and folders that are **direct descendants** of the directory, and you've also looked at **recursive listing**.

In general, you've seen that if you just need a basic list of the items in the directory, without recursion, then `.iterdir()` is the cleanest method to use, thanks to its descriptive name. It's also more efficient at this job. If, however, you need a recursive list, then you're best to go with `.rglob()`, which will be faster than an equivalent recursive `.iterdir()`.

You've also examined one example in which using `.iterdir()` to list recursively can produce a huge performance benefit—when you have junk folders that you want to opt out of iterating over.

In the downloadable materials, you'll find various implementations of methods to get a basic list of files from both the `pathlib` and `os` modules, along with a couple scripts that time them all against one another:

Source Code: Click here to download the free source code, directories, and bonus materials that showcase different ways to list files and folders in a directory with Python.

Check them out, modify them, and share your findings in the comments!

Mark as Completed

Python Tricks

Get a short & sweet **Python Trick** delivered to your inbox every couple of days. No spam ever. Unsubscribe any time. Curated by the Real Python team.

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```

Email Address

Send Me Python Tricks »

About **Ian Currie**

Ian is a Python nerd who uses it for everything from tinkering to helping people and companies manage their day-to-day and develop their businesses.

» More about Ian

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

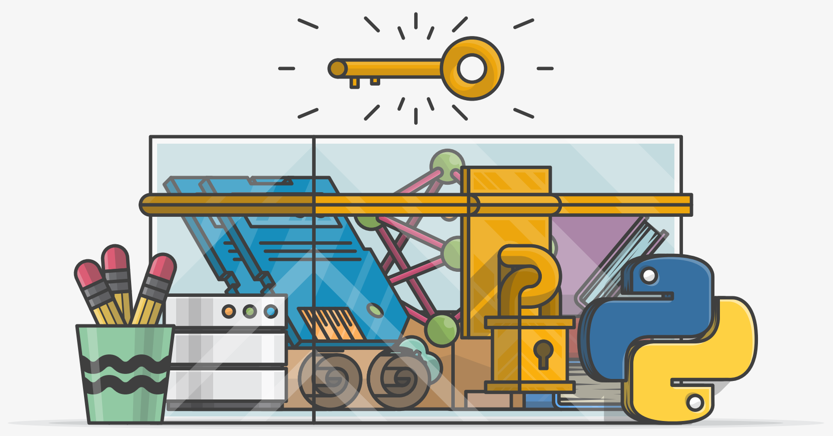
Aldren

Geir Arne

Kate

Leodanis

Master Real-World Python Skills
With Unlimited Access to Real Python



Join us and get access to thousands of tutorials, hands-on video courses, and a community of expert Pythonistas:

Level Up Your Python Skills »

What Do You Think?

Rate this article:

Tweet Share in Share Email

What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

Commenting Tips: The most useful comments are those written with the goal of learning from or helping out other students. Get tips for asking good questions and get answers to common questions in our support portal.

Looking for a real-time conversation? Visit the Real Python Community Chat or join the next "Office Hours" Live Q&A Session. Happy Pythoning!

Keep Learning

Related Tutorial Categories: intermediate