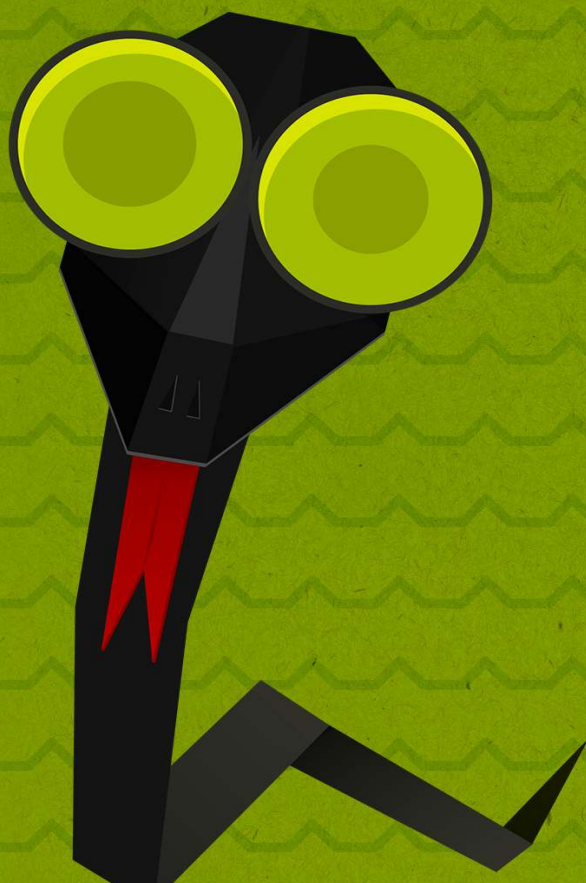
3RD EDITION

JULIEN DANJOU

THE HACKER'S GUIDE TO PYTHON



ISBN 978-1-329-98474-5

Contents

1 Starting your project **1**

1.1 Python versions 1

1.2 Project layout 2

1.3 Version numbering 5

1.4 Coding style & automated checks 7

1.5 Interview with Joshua Harlow 11

2 Modules and libraries **16**

2.1 The import system 16

2.2 Standard libraries 21

2.3 External libraries 23

2.4 Frameworks 26

2.5 Interview with Doug Hellmann 27

3 Managing API changes **36**

3.1 Interview with Christophe de Vienne 41

4 The trap of timezones **46**

5 Documentation **49**

5.1 Getting started with Sphinx and reST 51

5.2 Sphinx modules 52

5.3 Extending Sphinx 56

6 Distribution **59**

6.1 A bit of history 59

6.2 Packaging with *pbr* 62

6.3 The *Wheel* format 64

6.4 Package installation 66

6.5 Sharing your work with the world 68

6.6 Interview with Nick Coghlan 72

6.7 Entry points 75

6.7.1 Visualising entry points 75

6.7.2 Using console scripts 77

6.7.3 Using plugins and drivers 80

7 Virtual environments **84**

8 Unit testing **89**

8.1 The basics 89

8.2 Fixtures 98

8.3 Mocking 99

8.4 Scenarios 105

8.5 Test streaming and parallelism 109

8.6 Coverage 114
8.7 Using virtualenv with tox 118
8.8 Testing policy 123
8.9 Interview with Robert Collins 124

9 Methods and decorators **128**

9.1 Creating decorators 128
9.2 How methods work in Python 135
9.3 Static methods 138
9.4 Class method 139
9.5 Abstract methods 141
9.6 Mixing static, class, and abstract methods 142
9.7 The truth about `super` 145

10 Functional programming **151**

10.1 Generators 152
10.2 List comprehensions 158
10.3 Functional functions functioning 159

11 The AST **170**

11.1 Extending flake8 with AST checks 174
11.2 Hy 181
11.3 Interview with Paul Tagliamonte 183

12 Performances and optimizations **189**

12.1 Data structures 189

12.2 Profiling 191

12.3 Ordered list and bisect 198

12.4 Namedtuple and slots 200

12.5 Memoization 207

12.6 PyPy 209

12.7 Achieving zero copy with the buffer protocol 211

12.8 Interview with Victor Stinner 218

13 Scaling and architecture **221**

13.1 A note on multi-threading 221

13.2 Multiprocessing vs multithreading 224

13.3 Asynchronous and event-driven architecture 226

13.4 Service-oriented architecture 229

14 RDBMS and ORM **234**

14.1 Streaming data with Flask and PostgreSQL 238

14.2 Interview with Dimitri Fontaine 245

15 Python 3 support strategies **256**

15.1 Language and standard library 258

15.2 External libraries 261

15.3 Using six 262

16 Write less, code more **266**

16.1 Single dispatcher 266

16.2 Context managers 272

17 Beyond this book **276**

10 Functional programming

Functional programming might not be the first thing you think of when you think of Python, but the support is there, and it's quite extensive. Many Python developers don't seem to realize this, though, which is a shame: with few exceptions, functional programming allows you to write more concise and efficient code.

When you write code using functional style, your functions are designed not to have side effects: they take an input and produce an output without keeping state or modifying anything not reflected in the return value. Functions that follow this ideal are referred to as *purely functional*:

A non-pure function

```
def remove_last_item(mylist):
    """Removes the last item from a list."""
    mylist.pop(-1)  # This modifies mylist
```

A pure function

```
def butlast(mylist):
    """Like butlast in Lisp; returns the list without the last element."""
    return mylist[:-1]  # This returns a copy of mylist
```

The practical advantages of functional programming include:

- **Formal provability**; admittedly, this is a pure theoretical advantage, nobody is going to mathematically prove a Python program.

- **Modularity**; writing functionally forces a certain degree of separation in solving your problems and eases reuse in other contexts.
- **Brevity.** Functional programming is often less verbose than other paradigms.
- **Concurrency.** Purely functional functions are thread-safe and can run concurrently. While it's not yet the case in Python, some functional languages do this automatically, which can be a big help if you ever need to scale your application.
- **Testability.** It's a simple matter to test a functional program: all you need is a set of inputs and an expected set of outputs. They are idempotent.

Tip

If you want to get serious about functional programming, take my advice: take a break from Python and learn Lisp. I know it might sound strange to talk about Lisp in a Python book, but playing with Lisp for several years is what taught me how to "think functional." You simply won't develop the thought processes necessary to make full use of functional programming if all your experience comes from imperative and object-oriented programming. Lisp isn't *purely* functional itself, but there's more focus on functional programming than you'll find in Python.

10.1 Generators

A generator is an object that returns a value on each call of its `next()` method until it raises `StopIteration`. They were first introduced in PEP 255 and offer an easy way to create objects that implement the iterator protocol.

All you have to do to create a generator is write a normal Python function that contains a `yield` statement. Python will detect the use of `yield` and tag the function as a generator. When the function's execution reaches a `yield` statement, it returns a value as with a `return` statement, but with one notable difference: the interpreter

will save a stack reference, which will be used to resume the function's execution the next time `next` is called.

Creating a generator

```
>> def mygenerator():
...     yield 1
...     yield 2
...     yield 'a'
...
>>> mygenerator()
<generator object mygenerator at 0x10d77fa50>
>>> g = mygenerator()
>>> next(g)
1
>>> next(g)
2
>>> next(g)
'a'
>>> next(g)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

You can check whether a function is a generator or not yourself by using `inspect.isgeneratorfunction`:

```
>>> import inspect
>>> def mygenerator():
...     yield 1
...
>>> inspect.isgeneratorfunction(mygenerator)
True
```

```
>>> inspect.isgeneratorfunction(sum)
False
```

Reading the source code of `inspect.isgeneratorfunction` gives us some insight into the tagging mentioned earlier:

Source code of inspect.isgeneratorfunction

```
def isgeneratorfunction(object):
    """Return true if the object is a user-defined generator function.

    Generator function objects provides same attributes as functions.

    See help(isfunction) for attributes listing."""
    return bool((isfunction(object) or ismethod(object)) and
                object.func_code.co_flags & CO_GENERATOR)
```

Python 3 provides another useful function, `inspect.getgeneratorstate`:

```
>>> import inspect
>>> def mygenerator():
...     yield 1
...
>>> gen = mygenerator()
>>> gen
<generator object mygenerator at 0x7f94b44fec30>
>>> inspect.getgeneratorstate(gen)
'GEN_CREATED'
>>> next(gen)
1
>>> inspect.getgeneratorstate(gen)
'GEN_SUSPENDED'
>>> next(gen)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> inspect.getgeneratorstate(gen)
'GEN_CLOSED'
```

This function gives us the current state of a generator, allowing us to determine whether it's waiting to be run for the first time (`GEN_CREATED`), currently being executed by the interpreter (`GEN_RUNNING`), waiting to be resumed by a call to `next()` (`GEN_SUSPENDED`), or finished running (`GEN_CLOSED`).

In Python, generators are built by keeping a reference of the stack when a function *yield* something, resuming this stack when needed, i.e. when a call to `next()` is executed again.

When you iterate over any kind of data, the obvious approach is to build the entire list first, which is often wasteful in terms of memory consumption. Say we want to find the first number between 1 and 10,000,000 that's equal to 50,000. Sounds easy, doesn't it? Let's make this a challenge. We'll run Python with a memory constraint of 128 MB:

```
$ ulimit -v 131072
$ python
>>> a = list(range(10000000))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
MemoryError
```

Uh-oh. Turns out we can't build a list of ten million items with only 128MB of memory!

Warning

In Python 3, `range()` returns a generator; to get a generator in Python 2, you have to use `xrange()` instead. (This function doesn't exist in Python 3, since it's redundant.)

Let's try using a generator instead:

```
$ python
>>> for value in xrange(10000000):
...     if value == 50000:
...             print(”Found it”)
...             break
...
Found it
```

This time, our program executes without issue. The `range()` function returns an iterable object that dynamically generates our list of integers. Better still, since we were only interested in the 50,000th number, the generator only had to generate 50,000 numbers.

Generators allow you to handle large data sets with minimal consumption of memory and processing cycles by generating values on-the-fly. Whenever you need to work with a huge number of values, generators can help ensure you handle them efficiently.

`yield` also has a less commonly used feature: it can return a value like a function call. This allows us to pass a value to a generator by calling its `send()` method:

Example 10.1 `yield` returning a value

```
def shorten(string_list):
    length = len(string_list[0])
    for s in string_list:
        length = yield s[:length]
```

```
mystringlist = ['loremipsum', 'dolorsit', 'ametfoobar']
shortstringlist = shorten(mystringlist)
result = []
try:
    s = next(shortstringlist)
    result.append(s)
    while True:
        number_of_vowels = len(filter(lambda letter: letter in 'aeiou', s))
        # Truncate the next string depending
        # on the number of vowels in the previous one
        s = shortstringlist.send(number_of_vowels)
        result.append(s)
except StopIteration:
    pass
```

In this example, we've written a function called `shorten` that takes a list of strings and returns a list consisting of those same strings, only truncated. The length of each string is determined by the number of vowels in the previous string: "loremipsum" has four vowels, so the second value returned by the generator will be the first four letters of "dolorsit"; "dolo" has only two vowels, so "ametfoobar" will be truncated to its first two letters ("am"). The generator then stops and raises `StopIteration`. Our generator thus returns:

```
['loremipsum', 'dolo', 'am']
```

Using `yield` and `send()` in this fashion allows Python generators to function like **coroutines** seen in Lua and other languages.

Tip

PEP 289 introduced generator expressions, making it possible to build one-line generators using a syntax similar to list comprehension:

```
>>> (x.upper() for x in ['hello', 'world'])
<generator object <genexpr> at 0x7ffab3832fa0>
>>> gen = (x.upper() for x in ['hello', 'world'])
>>> list(gen)
['HELLO', 'WORLD']
```

10.2 List comprehensions

List comprehension, or *listcomp* for short, allows you to define a list's contents inline with its declaration:

Without list comprehension

```
>>> x = []
>>> for i in (1, 2, 3):
...     x.append(i)
...
>>> x
[1, 2, 3]
```

With list comprehension

```
>>> x = [i for i in (1, 2, 3)]
>>> x
[1, 2, 3]
```

You can use multiple `for` statements together and use `if` statements to filter out items:

```
x = [word.capitalize()
     for line in ("hello world?", "world!", "or not")
     for word in line.split()
     if not word.startswith("or")]
>>> x
['Hello', 'World?', 'World!', 'Not']
```

Using list comprehension rather than *for* loops is a neat way to quickly define lists. Since we're still talking about functional programming, it's worth noting that lists built through list comprehension can't rely on the program's state. [1] This generally makes them more concise and easier to read than lists made without list comprehension.

Note

There's also syntax for building dictionaries or sets in the same fashion:

```
>>> {x:x.upper() for x in ['hello', 'world']}
{'world': 'WORLD', 'hello': 'HELLO'}
>>> {x.upper() for x in ['hello', 'world']}
set(['WORLD', 'HELLO'])
```

Note that this only works in Python 2.7 and onward.

10.3 Functional functions functioning

Python includes a number of tools for functional programming. These built-in functions cover the basics:

- `map(function, iterable)` applies `function` to each item in `iterable` and returns either a list in Python 2 or an iterable `map` object in Python 3:

[1] Technically they *can,* but that's really not how they're supposed to work.

map usage in Python 3

```
>>> map(lambda x: x + "bzz!", ["I think", "I'm good"])
<map object at 0x7fe7101abdd0>
>>> list(map(lambda x: x + "bzz!", ["I think", "I'm good"]))
['I thinkbzz!', "I'm goodbzz!"]
```

- `filter(function or None, iterable)` filters the items in `iterable` based on the result returned by `function`, and returns either a list in Python 2, or better, an iterable `filter` object in Python 3:

Example 10.2 *filter* usage in Python 3

```
>>> filter(lambda x: x.startswith("I "), ["I think", "I'm good"])
<filter object at 0x7f9a0d636dd0>
>>> list(filter(lambda x: x.startswith("I "), ["I think", "I'm good"]))
['I think']
```

Tip

You can write a function equivalent to `filter` or `map` using generators and list comprehension:

Equivalent of map using list comprehension

```
>>> (x + "bzz!" for x in ["I think", "I'm good"])
<generator object <genexpr> at 0x7f9a0d697dc0>
>>> [x + "bzz!" for x in ["I think", "I'm good"]]
['I thinkbzz!', "I'm goodbzz!"]
```

Equivalent of filter using list comprehension

```
>>> (x for x in ["I think", "I'm good"] if x.startswith("I "))
<generator object <genexpr> at 0x7f9a0d697dc0>
>>> [x for x in ["I think", "I'm good"] if x.startswith("I ")]
['I think']
```

Using generators like this in Python 2 will give you an iterable object rather than a list, just like the `map` and `filter` functions in Python 3.

- `enumerate(iterable[, start])` returns an iterable `enumerate` object that yields a sequence of tuples, each consisting of an integer index (starting with `start`, if provided) and the corresponding item in `iterable`. It's useful when you need to write code that refers to array indexes. For example, instead of writing this:

```
i = 0
while i < len(mylist):
    print("Item %d: %s" % (i, mylist[i]))
    i += 1
```

You could write this:

```
for i, item in enumerate(mylist):
    print("Item %d: %s" % (i, item))
```

- `sorted(iterable, key=None, reverse=False)` returns a sorted version of `iterable`. The `key` argument allows you to provide a function that returns the value to sort on.
- `any(iterable)` and `all(iterable)` both return a boolean depending on the values returned by `iterable`. These functions are equivalent to:

```
def all(iterable):
    for x in iterable:
        if not x:
            return False
    return True
```

```
def any(iterable):
    for x in iterable:
        if x:
            return True
    return False
```

These functions are useful for checking whether any or all of the values in an iterable satisfy a given condition:

```
mylist = [0, 1, 3, -1]
if all(map(lambda x: x > 0, mylist)):
    print("All items are greater than 0")
if any(map(lambda x: x > 0, mylist)):
    print("At least one item is greater than 0")
```

- `zip(iter1 [,iter2 [...]])` takes multiple sequences and combines them into tuples. It's useful when you need to combine a list of keys and a list of values into a dict. Like the other functions described above, it returns a list in Python 2 and an iterable in Python 3:

```
>>> keys = ["foobar", "barzz", "ba!"]
>>> map(len, keys)
<map object at 0x7fc1686100d0>
>>> zip(keys, map(len, keys))
<zip object at 0x7fc16860d440>
>>> list(zip(keys, map(len, keys)))
[('foobar', 6), ('barzz', 5), ('ba!', 3)]
>>> dict(zip(keys, map(len, keys)))
{'foobar': 6, 'barzz': 5, 'ba!': 3}
```

You might have noticed by now how the return types differ between Python 2 and Python 3. Most of Python's purely functional built-in functions return a list rather than an iterable in Python 2, making them less memory-efficient than their Python 3.x equivalents. If you're planning to write code using these functions, keep in mind that you'll get the most benefit out of them in Python 3. If you're stuck to Python 2, don't despair yet: the `itertools` module from the standard library provides an iterator based version of many of these functions (`itertools.izip`, `itertools.imap`, `itertools.ifilter`, etc).

There's still one important tool missing from this list, however. One common task when working with lists is finding the first item that satisfies a specific condition. This is usually accomplished with a function like this:

```
def first_positive_number(numbers):
    for n in numbers:
        if n > 0:
            return n
```

We can also write this in functional style:

```
def first(predicate, items):
    for item in items:
```

```
        if predicate(item):
            return item


first(lambda x: x > 0, [-1, 0, 1, 2])
```

Or more concisely:

```
# Less efficient
list(filter(lambda x: x > 0, [-1, 0, 1, 2]))[0] ❶
# Efficient but for Python 3
next(filter(lambda x: x > 0, [-1, 0, 1, 2]))
# Efficient but for Python 2
next(itertools.ifilter(lambda x: x > 0, [-1, 0, 1, 2]))
```

❶ Note that this may raise an IndexError if no items satisfy the condition, causing `list(filter())` to return an empty list.

For simple case you can also rely on `next()`:

```
>>> a = range(10)
>>> next(x for x in a if x > 3)
4
```

This will raise StopIteration if a condition can never be satisfied, so in that case the 2nd argument of `next()` can be used:

```
>>> a = range(10)
>>> next((x for x in a if x > 10), 'default')
'default'
```

Instead of writing this same function in every program you make, you can include the small but very useful Python package **first**:

Example 10.3 Using `first`

```
>>> from first import first
>>> first([0, False, None, [], (), 42])
42
>>> first([-1, 0, 1, 2])
-1
>>> first([-1, 0, 1, 2], key=lambda x: x > 0)
1
```

The `key` argument can be used to provide a function which receives each item as an argument and returns a boolean indicating whether it satisfies the condition.

You'll notice that we've used `lambda` in a good portion of the examples so far in this chapter. `lambda` was actually added to Python in the first place to facilitate functional programming functions such as `map()` and `filter()`, which otherwise would have required writing an entirely new function every time you wanted to check a different condition:

```
import operator
from first import first


def greater_than_zero(number):
    return number > 0


first([-1, 0, 1, 2], key=greater_than_zero)
```

This code works identically to the previous example, but it's a good deal more cumbersome: if we wanted to get the first number in the sequence that's greater than, say, 42, then we'd need to `def` an appropriate function rather than defining it in-line with our call to `first`.

But despite its usefulness in helping us avoid situations like this, `lambda` still has

its problems. First and most obviously, we can't pass a `key` function using `lambda` if it would require more than a single line of code. In this event, we're back to the cumbersome pattern of writing new function definitions for each `key` we need. Or are we?

`functools.partial` is our first step towards replacing `lambda` with a more flexible alternative. It allows us to create a wrapper function with a twist: rather than changing the behavior of a function, it instead changes the *arguments* it receives:

```
from functools import partial
from first import first


def greater_than(number, min=0):
    return number > min


first([-1, 0, 1, 2], key=partial(greater_than, min=42))
```

Our new `greater_than` function works just like the old `greater_than_zero` by default, but now we can specify the value we want to compare our numbers to. In this case, we pass `functools.partial` our function and the value we want for `min`, and we get back a new function that has `min` set to 42, just like we want. In other words, we can write a function and use `functools.partial` to customize what it does to our needs in any given situation.

This is still a couple lines more than we strictly need in this case, though. All we're doing in this example is comparing two numbers; what if Python had built-in functions for these kinds of comparisons? As it turns out, the **operator** module has just what we're looking for:

```
import operator
from functools import partial
from first import first
```

```
first([-1, 0, 1, 2], key=partial(operator.le, 0))
```

Here we see that `functools.partial` also works with positional arguments. In this case, `operator.le(a, b)` takes two numbers and returns whether the first is less than or equal to the second: the 0 we pass to `functools.partial` gets sent to `a`, and the argument passed to the function returned by `functools.partial` gets sent to `b`. So this works identically to our initial example, without using `lambda` or defining any additional functions.

Note

`functools.partial` is typically useful in replacement of lambda, and is to be considered as a superior alternative. `lambda` is to be considered an anomaly in Python language [a], due to its limited body size of one line long single expression. On the other hand, `functools.partial` is built as a nice wrapper around the original function.

[a] And was once even planned to be removed in Python 3, but finally escaped from its fate.

The **itertools** module in the Python Standard Library also provides a bunch of useful functions that you'll want to keep in mind. I've seen too many programmers end up writing their own versions of these functions even though Python itself provides them out-of-the-box:

- `chain(*iterables)` iterates over multiple iterables one after each other without building an intermediate list of all items.
- `combinations(iterable, r)` generates all combination of length `r` from the given `iterable`.
- `compress(data, selectors)` applies a boolean mask from `selectors` to `data` and returns only the values from `data` where the corresponding element of `selectors` is true.

- `count(start, step)` generates an endless sequence of values, starting from `start` and incrementing by `step` with each call.
- `cycle(iterable)` loops repeatedly over the values in `iterable`.
- `dropwhile(predicate, iterable)` filters elements of an iterable starting from the beginning until `predicate` is false.
- `groupby(iterable, keyfunc)` creates an iterator grouping items by the result returned by the *keyfunc* function.
- `permutations(iterable[, r])` returns successive `r`-length permutations of the items in `iterable`.
- `product(*iterables)` returns an iterable of the cartesian product of `iterables` without using a nested `for` loop.
- `takewhile(predicate, iterable)` returns elements of an iterable starting from the beginning until `predicate` is false.

These functions are particularly useful in conjunction with the *operator* module. When used together, *itertools* and *operator* can handle most situations that programmers typically rely on `lambda` for:

Example 10.4 Using the `operator` module with `itertools.groupby`

```
>>> import itertools
>>> a = [{'foo': 'bar'}, {'foo': 'bar', 'x': 42}, {'foo': 'baz', 'y': 43}]
>>> import operator
>>> list(itertools.groupby(a, operator.itemgetter('foo')))
[('bar', <itertools._grouper object at 0xb000d0>), ('baz', <itertools. ↩
   _grouper object at 0xb00110>)]
>>> [(key, list(group)) for key, group in itertools.groupby(a, operator. ↩
   itemgetter('foo'))]
```

```
[('bar', [{'foo': 'bar'}, {'x': 42, 'foo': 'bar'}]), ('baz', [{'y': 43, ' ↩
   foo': 'baz'}])]
```

In this case, we could have also written `lambda x:x['foo']`, but using `operator` lets us avoid having to use `lambda` at all.