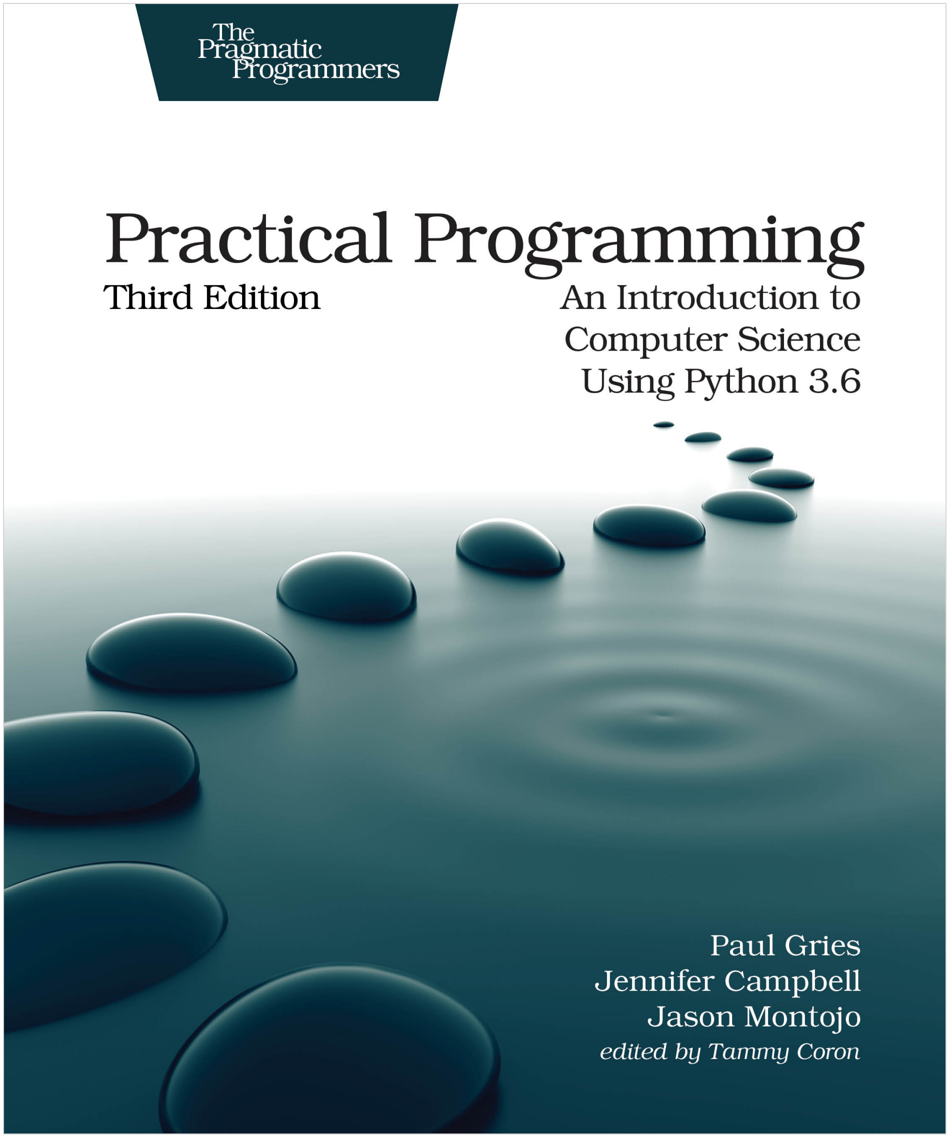

The Pragmatic Programmers

Practical Programming

Third Edition

An Introduction to Computer Science Using Python 3.6

Paul Gries
Jennifer Campbell
Jason Montojo
edited by Tammy Coron

Python treats many data collections as ordered sequences. The most common are lists (mutable), tuples (immutable), and strings (immutable). You learned in Processing Items in a List, on page ? how to iterate through a sequence item by item in a loop. Loops over sequences are so essential to modern idiomatic (or "Pythonic," as further explained in *Pythonic Programming [Zin21]*) programming that the language provides a family of iteration tools that hide low-level loop bookkeeping and help you write clear, efficient programs.

Being Pythonic

Adjective "Pythonic" refers to Python code that follows the language's idioms, philosophy, and design principles, rather than just using Python's syntax to write code that resembles another language. It emphasizes readability, simplicity, and the use of built-in features effectively to write clear, efficient, and maintainable programs.

Core Idioms

Consider the following task: given a list or tuple of words from a social media post, extract a list of hashtags (items that begin with #) and return them as "normal" words without the "hash." For example, the input ['#Mary', 'had', 'a', 'little', '#lamb'] should produce the output ['Mary', 'lamb'].

You can build the result incrementally in a loop. Note: as tempting as it is to compare the first character of an item with '#' (i.e., item[0] == '#'), that's unsafe: some items on the list might be empty strings, and indexing would raise an error. The function startswith handles empty strings correctly.

```python
data = ['#Mary', 'had', 'a', 'little', '#lamb']

result = [] # Initialize an empty list
for item in data:
    if item.startswith('#'): # Check the condition
        result.append(item[1:]) # Strip the leading '#' and append

print(result) # ['Mary', 'lamb']
```

This style is known as *imperative programming*: it works at a lower level of abstraction (Programs and Programming, on page ?) and emphasizes how a result is produced, often focusing on implementation details as much as on the outcome itself. By contrast, *comprehensions* are a higher-level, *declarative programming* construct: they describe the transformation from inputs to outputs and typically abstract away the underlying implementation.

Comprehensions

The hashtag extraction task is a special case of a common *Filter–Map–Reduce* pattern in data processing (you can read more about the pattern in *Functional Programming Patterns in Scala and Clojure [Bev13]*). The pattern typically involves three steps: filtering (selecting items that meet a condition), mapping (transforming each selected item), and reducing (aggregating results).

Design Patterns

A *design pattern* is a general, reusable solution to a recurring problem in software design. It is not a complete solution but a blueprint that must be adapted and implemented in code. Read more about design patterns in *Design Patterns: Elements of Reusable Object-Oriented Software [GHJV95]*.

Python supports the Filter–Map–Reduce pattern with three types of comprehensions: list, set, and dictionary. The two general *list comprehension* forms are:

```
[«expr» for «var» in «data»]
[«expr» for «var» in «data» if «cond»]
```

In the first form, `expr` is evaluated for every `var` in `data`, and the results are collected into a list. If `data` is ordered (a tuple, a list, or a string), the result preserves that order. As an example, consider computing the lengths of each item in `data`:

```
data = ['#Mary', 'had', 'a', 'little', '#lamb']

result = [len(item) for item in data]
print(result) # [5, 3, 1, 6, 5]
```

In the second form, `expr` is evaluated only for the items that satisfy the condition `cond`. All other items are filtered out. In the following example, the parts of the comprehension that correspond to the filter, map, and reduce actions, are placed on separate lines for clarity.

```
result = [
    item[1:]                # map
    for item in data
    if item.startswith('#') # filter
]                           # reduce (implicit list construction)
print(result) # ['Mary', 'lamb']
```

How to "Comprehend" Comprehensions

Read a comprehension aloud by starting at the keyword for: "for every item in data, if item.startswith('#')…" ("filter").

Go back to the beginning and prepend "calculate" before the expression: "…calculate item[1:]" ("map").

Add "…and assemble the results" ("reduce").

This mirrors the execution order and often makes complex comprehensions easier to understand.

If you omit the if clause in a comprehension, there is no filtering: every item is included. The expression at the front can also be trivial (just the loop variable) or even a constant. You can use the latter form to count values that satisfy the condition:

```
result1 = [item for item in data if item.startswith('#')]
print(result1) # ['#Mary', '#lamb']
result2 = [1 for item in data if item.startswith('#')]
print(result2) # [1, 1]
print(len(result2)) # 2
```

Last but not least, if the expression is just a variable and there is no condition, the comprehension makes a copy of the original sequence, serving as an expensive equivalent of data[:].

```
result = [item for item in data]
print(result == data[:]) # True
```

A *set comprehension* uses curly braces {} instead of square brackets [] and produces a set. Think of it as essentially an application of set to a list comprehension:

```
result = {item[1:] for item in data if item.startswith('#')}
print(result) # {'Mary', 'lamb'}
```

Storing hashtags in a set dramatically improves performance of membership tests because Python sets use hash tables internally (see Hash Tables and Why They Matter, on page ?).

```
present = 'Mary' in result
print(present) # True
```

Quite expectedly, a *dictionary comprehension* produces a dictionary. As such, for every dictionary item, it needs a key and a value, separated by a colon:

```
{«key_expr»: «value_expr» for «var» in «data»}
{«key_expr»: «value_expr» for «var» in «data» if «cond»}
```

Now, you can precompute a mapping from hashtags to their lengths. To facilitate lookups, convert the keys to lowercase (or any other standard form):

```
result = {item[1:].lower() : len(item[1:])
          for item in data if item.startswith('#')}
print(result) # {'mary': 4, 'lamb': 4}
```

If a hashtag happens in data more than once, the duplicates will be merged during the dictionary construction.

One limitation of comprehensions is that each produces only one output collection. If you want to split items into two groups (those that meet a condition and those that don't), you must have two comprehensions with complementary conditions, iterating over the same data twice.

```
hashtags  = {item[1:].lower() : len(item[1:])
             for item in data if     item.startswith('#')}
justwords = {item.lower() : len(item)
             for item in data if not item.startswith('#')}
```

This approach is inefficient, particularly for large datasets. A single-pass for loop is a better alternative:

```
hashtags = {}
justwords = {}
for item in data:
    if item.startswith('#'):
        hashtags[item[1:].lower()] = len(item[1:])
    else:
        justwords[item.lower()] = len(item)
```

Generators

Comprehensions create the entire output collection in memory, even if you need it piecewise, item by item.

Suppose you want the average length of the words in data. A straightforward approach involves building a list of lengths and then calculating the average:

```
lengths = [len(item) for item in data]
average = sum(lengths) / len(data)
print(average) # 4.0
```

Note that the list lengths exists only to be immediately reduced to a single number by summation. If data is large, so is lengths, even though the built-in function sum needs items one at a time, not all at once. Having a comprehension-like expression that produces items as needed would make this code more memory-efficient.

Enter generators. A *generator expression* yields ("generates") values on demand. It looks like a list comprehension, but it is enclosed in parentheses ():

```
lengths = (len(item) for item in data)
print(type(lengths)) # <class 'generator'>
```

Generator expressions yield items lazily (on demand), which saves memory for large datasets

The value returned by a generator expression is an object of class generator. A generator uses *lazy evaluation*: it doesn't return the results themselves but a "promise" to produce them later. That "promise" can be fulfilled explicitly by calling the built-in function next (beware that each call to next consumes the next generated value and may exhaust the generator before it is otherwise used):

```
>>> data = ['#Mary', 'had', 'a', 'little', '#lamb']
>>> lengths = (len(item) for item in data)
>>> next(lengths)
5
>>> next(lengths)
3
>>> next(lengths)
1
>>> next(lengths)
6
>>> next(lengths)
5
>>> next(lengths)
Traceback (most recent call last):
  File "<python-input-7>", line 1, in <module>
    next(lengths)
    ~~~~^^^^^^^^^
StopIteration
```

At the end of the sequence, the generator raises a StopIteration exception.

Alternatively, pass the generator to an aggregating function, such as sum:

```
average = sum(lengths) / len(data)
print(average) # 4.0
```

Generator expressions may be slightly slower than plain list comprehensions, and they can't be reused: they must be re-created to iterate again. However, for large datasets, they are often essential, making the difference between feasible and infeasible list processing.