# 2 Iteration

**Key terms: comprehension** `PEP`

**Reading: Mary Rose Cook's A practical introduction to functional programming**[5]

**Reading: Peter J. Denning's "Can Generative AI Bots Be Trusted?"**[6]

**Exercise: Write a program that uses a list comprehension to print out all those numbers under one million, the sum of whose digits equals seven.**

"Go To Statement Considered Harmful"[7] is the popular name of Dijkstra's argument for structured programming (and an origin of the "considered harmful" meme[8]). Python has no "Go To", but unecessary statefulness still causes problems. We can avoid this problem by using iterables instead of counting, and by using function calls instead of loops. Both help to make programs safer, shorter, more reliable, and more testable.

First, we review three flavors of iteration. Explicit counting iteration, like in the C language, means proceeding step by step, relying on a value to track progress through a container. Explicit for-each iteration, like in the Java language, means relying on the machine's knowledge of the size of a collection to review its contents. Compared to the counting approach, eliminating references to numbers and counter objects reduces complexity and potential errors. Implicit iteration (which is actually functional programming like Lisp or Haskell) means alluding to a loop inside an expression. Compared to the explicit approaches, avoiding line orientation increases expressiveness and avoids "spaghetti". Note that the wonderful `*`, or "splat" operator, unpacks arguments in place; it makes program grammar easier, providing a means of switching between references and referents. The essential difference is that the comprehension style is stateless, which sidesteps many potential bugs.

Counting iteration (don't do this):

```
i = 0
while i < 100:
 print (i)
 i += 1
```

For-each iteration (use sparingly):

```
for i in range(100):
 print(i)
```

Implicit iteration (prefer this):

```
print(*range(100),sep='\n')

print('\n'.join(map(str,range(100))))
```

Comprehensions, introduced following PEP 202 in 2000, are another great functional convenience, inline descriptions of anonymous collections. Every element in a comprehension corresponds to an element in source data which is filtered and mapped. The concept is the same as set-builder notation, e.g. $\{i \in \mathbb{I} : 0 \leq i < 100\}$, which in Python could be `[i for i in range(100)]`. Specifying exactly the input conditions (the source material and how it is filtered) and the output conditions (what you want to know about those elements) is very empowering because then the programmer doesn't have to worry about state within a loop.

```
# Demonstrate the collection by comprehension of the current non-magic names:

names_set = {name:len(name) for name in dir() if not name.startswith('__')}
names_list = [name for name in dir() if not name.startswith('__')]
```

Next week, we'll look at functional programming in more detail.

[5] `https://maryrosecook.com/blog/post/a-practical-introduction-to-functional-programming`
[6] `https://fog.ccsf.edu/~abrick/reading/trusted.pdf`
[7] `https://homepages.cwi.nl/~storm/teaching/reader/Dijkstra68.pdf`
[8] `https://en.wikipedia.org/wiki/Considered_harmful`