Extracted from:

Python Brain Teasers

Exercise Your Mind

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Python Brain Teasers

Exercise Your Mind

Miki Tebeka

edited by Margaret Eldridge

Python Brain Teasers

Exercise Your Mind

Miki Tebeka

The Pragmatic Bookshelf

Raleigh, North Carolina

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Margaret Eldridge
Copy Editor: Jennifer Whipple
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-900-7
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—September 2021

Puzzle 27

An Inside Job

inside.py
```python
def add_n(items, n):
    items += range(n)

items = [1]
add_n(items, 3)
print(items)
```

Guess the Output

Try to guess what the output is before moving to the next page.

This code will print: `[1, 0, 1, 2]`

In the *Call Me Maybe* puzzle, we talked about rebinding versus mutation. And most of the time, `items += range(n)` is translated to `items = items + range(n)`, which is rebinding.

There is a special optimization for += in some cases. Here's what the documentation says (my emphasis):

> An augmented assignment expression like x += 1 can be rewritten as x = x + 1 to achieve a similar, but not exactly equal, effect. In the augmented version, x is only evaluated once. Also, *when possible, the actual operation is performed in place, meaning that rather than creating a new object and assigning that to the target, the old object is modified instead.*

A type defines how the + operator behaves with the `__add__` special method and can define `__iadd__` as a special case for +=. The documentation says

> These methods are called to implement the augmented arithmetic assignments (+=, -=, =, @=, /=, //=, %=, *=, <⇐, >>=, &=, ^=, |=). These methods should attempt to do the operation in place (modifying self) and return the result (which could be, but does not have to be, self). If a specific method is not defined, the augmented assignment falls back to the normal methods.

The built-in `list` object defines `__iadd__`, which calls the `extend` method.

What will happen if you change the code inside `add_n` to `items = items + range(n)`? You will get an exception: `TypeError: can only concatenate list (not "range") to list`.

In Python 3 the built-in `range` function returns a `range` object. Even though it *looks* like a `list` (`len`, `[]`, and friends will work), you can't add it to a `list`.

If you want the rebinding code to work, you'll need to write `items = items + list(range(n))` and then the output will be `[1]`.

As a general rule, try not to mutate the object passed to your functions. This style of programming is called *functional* programming. Functional code is easier to test and reason about. Give it a try. It's fun.

Further Reading

Functional Programming on Wikipedia
  en.wikipedia.org/wiki/Functional_programming

Built-in `range` Documentation
  docs.python.org/3/library/functions.html#func-range

“Augmented Assignment Statements” in the Python Reference
docs.python.org/3/reference/simple_stmts.html#augmented-assignment-statements

“Functional Programming HOWTO” in the Python Documentation
docs.python.org/3/howto/functional.html

__iadd__ Documentation
docs.python.org/3/reference/datamodel.html#object.__iadd__

“More on Lists” in the Python Documentation
docs.python.org/3/tutorial/datastructures.html#more-on-lists