Python For-Loops

Table of contents

A **for-loop** is a loop that repeats code for every member of a group (an *iterator*), in order:

```
for <name> in <iterator>:
    <code>
```

Iterating over strings, lists, and dictionaries

A for-loop iterates a string's characters, a list's elements, or a dictionary's *keys* (in each case, in order):

```
>>> for x in "abcd":
...     print(x)
a
b
c
d

>>> for x in ['a', 2, 'c']:
...     print(x)
a
```

```
2
c

>>> for key in {"red": 1, "blue": 2}:
...     print(key)
red
blue
```

The looping name retains its final value after the loop exits.

Nested for-loops

Python disallows empty loop bodies — use `pass` (the empty instruction, which has no effect) as a placeholder when a loop body needs no code yet:

```
>>> for d in "01":
...     for a in "xy":
...         pass
...     print(d + a)
0y
1y
```

Accumulator pattern

An **accumulator** is a variable a loop uses to build up an aggregate value across iterations:

```
>>> acc = ""
>>> for x in "abcd":
...     acc = acc + x
>>> acc
'abcd'
```

`range`

`range([start], stop[, step])` builds an iterator of numbers (square brackets denote optional arguments), similar to list slicing:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(2, 7))
[2, 3, 4, 5, 6]
>>> list(range(2, 7, 3))
[2, 5]
>>> list(range(7, 2, -1))
[7, 6, 5, 4, 3]
```

Commonly used to walk through a list by index: `for k in range(len(xs)):`.

`enumerate`

`enumerate(xs)` pairs each element of `xs` with its index, roughly `enumerate([x0, ..., xn]) == [(0, x0), ..., (n, xn)]` (it actually returns an iterable of tuples, not a list):

```
>>> for k, x in enumerate(["a", "b", "c"]):
...     print(k, x)
0 a
1 b
2 c
```

For-loops vs. while-loops

Every for-loop can be rewritten with only a while-loop:

```
>>> for x in xs:
...     ...
```

```
>>> k = 0
>>> while k < len(xs):
...     x = xs[k]
...     ...
...     k += 1
```

The reverse isn't always true — a while-loop whose number of repetitions isn't known in advance (e.g. repeatedly prompting a user until they enter valid input) can't be rewritten as a for-loop.

Out-of-place vs. in-place

Building a new accumulator (e.g. a fresh list) without touching the original input is an **out-of-place** computation. An **in-place** change instead mutates the input directly — covered with lists[1] and the object-oriented part of the course.

See python-comprehensions[2] for a more concise way to write many accumulator-style for-loops.

[1] courses/csse1001/python-lists.pdf
[2] courses/csse1001/python-comprehensions.pdf