1.2. Unpacking Elements from Iterables of Arbitrary

Length

Problem

You need to unpack N elements from an iterable, but the iterable may be longer than N

elements, causing a “too many values to unpack” exception.

Solution

Python “star expressions” can be used to address this problem. For example, suppose

you run a course and decide at the end of the semester that you’re going to drop the first

and last homework grades, and only average the rest of them. If there are only four

assignments, maybe you simply unpack all four, but what if there are 24? A star expression

makes it easy:

**def** drop\_first\_last(grades):

first, \*middle, last = grades

**return** avg(middle)

As another use case, suppose you have user records that consist of a name and email

address, followed by an arbitrary number of phone numbers. You could unpack the

records like this:

**>>>** record = ('Dave', 'dave@example.com', '773-555-1212', '847-555-1212')

**>>>** name, email, \*phone\_numbers = user\_record

**>>>** name

'Dave'

**>>>** email

'dave@example.com'

**>>>** phone\_numbers

['773-555-1212', '847-555-1212']

>>>

It’s worth noting that the phone\_numbers variable will always be a list, regardless of how

many phone numbers are unpacked (including none). Thus, any code that uses

phone\_numbers won’t have to account for the possibility that it might not be a list or

perform any kind of additional type checking.

The starred variable can also be the first one in the list. For example, say you have a

sequence of values representing your company’s sales figures for the last eight quarters.

If you want to see how the most recent quarter stacks up to the average of the first seven,

you could do something like this:

\*trailing\_qtrs, current\_qtr = sales\_record

trailing\_avg = sum(trailing\_qtrs) / len(trailing\_qtrs)

**return** avg\_comparison(trailing\_avg, current\_qtr)

Here’s a view of the operation from the Python interpreter:

Discussion

Extended iterable unpacking is tailor-made for unpacking iterables of unknown or arbitrary

length. Oftentimes, these iterables have some known component or pattern in

their construction (e.g. “everything after element 1 is a phone number”), and star unpacking

lets the developer leverage those patterns easily instead of performing acrobatics

to get at the relevant elements in the iterable.

It is worth noting that the star syntax can be especially useful when iterating over a

sequence of tuples of varying length. For example, perhaps a sequence of tagged tuples:

records = [

('foo', 1, 2),

('bar', 'hello'),

('foo', 3, 4),

]

**def** do\_foo(x, y):

**print**('foo', x, y)

**def** do\_bar(s):

**print**('bar', s)

**for** tag, \*args **in** records:

**if** tag == 'foo':

do\_foo(\*args)

**elif** tag == 'bar':

do\_bar(\*args)

Star unpacking can also be useful when combined with certain kinds of string processing

operations, such as splitting. For example:

**>>>** line = 'nobody:\*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false'

**>>>** uname, \*fields, homedir, sh = line.split(':')

**>>>** uname

'nobody'

**>>>** homedir

'/var/empty'

**>>>** sh

'/usr/bin/false'

>>>

Sometimes you might want to unpack values and throw them away. You can’t just specify

a bare \* when unpacking, but you could use a common throwaway variable name, such

as \_ or ign (ignored). For example:

**>>>** record = ('ACME', 50, 123.45, (12, 18, 2012))

**>>>** name, \*\_, (\*\_, year) = record

**>>>** name

'ACME'

**>>>** year

2012

>>>

There is a certain similarity between star unpacking and list-processing features of various

functional languages. For example, if you have a list, you can easily split it into head

and tail components like this:

**>>>** items = [1, 10, 7, 4, 5, 9]

**>>>** head, \*tail = items

**>>>** head

1

**>>>** tail

[10, 7, 4, 5, 9]

>>>

One could imagine writing functions that perform such splitting in order to carry out

some kind of clever recursive algorithm. For example:

**>>> def** sum(items):

**...** head, \*tail = items

**... return** head + sum(tail) **if** tail **else** head

**...**

**>>>** sum(items)

36

>>>

However, be aware that recursion really isn’t a strong Python feature due to the inherent

recursion limit. Thus, this last example might be nothing more than an academic curiosity

in practice.