Extended Unpacking

Let's see how we might split a list into it's first element, and "everything else" using slicing:

In [1]:
```
l = [1, 2, 3, 4, 5, 6]
```

In [2]:
```
a = l[0]
b = l[1:]
print(a)
print(b)
```

```
1
[2, 3, 4, 5, 6]
```

We can even use unpacking to simplify this slightly:

In [3]:
```
a, b = l[0], l[1:]
print(a)
print(b)
```

```
1
[2, 3, 4, 5, 6]
```

But we can use the ***** operator to achieve the same result:

In [4]:
```
a, *b = l
print(a)
print(b)
```

```
1
[2, 3, 4, 5, 6]
```

Note that the ***** operator can only appear **once**!

Like standard unpacking, this extended unpacking will work with any iterable.

With tuples:

In [6]:
```
a, *b = -10, 5, 2, 100
print(a)
print(b)
```

```
-10
[5, 2, 100]
```

With strings:

In [7]:
```
a, *b = 'python'
print(a)
print(b)
```

```
p
['y', 't', 'h', 'o', 'n']
```

What about extracting the first, second, last elements and *the rest*.

Again we can use slicing:

In [9]:
```
s = 'python'

a, b, c, d = s[0], s[1], s[2:-1], s[-1]
print(a)
print(b)
print(c)
```
```
p
y
tho
n
```

But we can just as easily do it this way using unpacking:

In [13]:
```
a, b, *c, d = s
print(a)
print(b)
print(c)
```
```
p
y
['t', 'h', 'o']
n
```

As you can see though, **c** is a list of characters, not a string.

It that's a problem we can easily fix it this way:

In [11]:
```
print(c)
c = ''.join(c)
```
```
['t', 'h', 'o']
tho
```

We can also use unpacking on the right hand side of an assignment expression:

In [51]:
```
l1 = [1, 2, 3]
l2 = [4, 5, 6]
l = [*l1, *l2]
```
```
[1, 2, 3, 4, 5, 6]
```

In [53]:
```
l1 = [1, 2, 3]
s = 'ABC'
l = [*l1, *s]
```
```
[1, 2, 3, 'A', 'B', 'C']
```

This unpacking works with unordered types such as sets and dictionaries as well.

The only thing is that it may not be very useful considering there is no particular ordering, so a first or last element has no real useful meaning.

In [15]:

In [16]:
```
for c in s:
```
```
10
3
d
-99
```

As you can see, the order of the elements when we created the set was not retained!

In [54]:
```
s = {10, -99, 3, 'd'}
a, b, *c = s
print(a)
print(b)
```
```
10
3
['d', -99]
```

So unpacking this way is of limited use.

However consider this:

In [55]:
```
s = {10, -99, 3, 'd'}
*a, = s
```
```
[10, 3, 'd', -99]
```

At first blush, this doesn't look terribly exciting - we simply unpacked the set values into a list.

But this is actually quite useful in both sets and dictionaries to combine things (although to be sure, there are alternative ways to do this as well - which we'll cover later in this course)

In [21]:
```
s1 = {1, 2, 3}
```

How can we combine both these sets into a single merged set?

In [22]:
```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-22-1659087814e1> in <module>()
----> 1 s1 + s2

TypeError: unsupported operand type(s) for +: 'set' and 'set'
```

Well, **+** doesn't work...

We could use the built-in method for unioning sets:

In [23]:

```
Help on class set in module builtins:

class set(object)
 |  set() -> new empty set object
 |  set(iterable) -> new set object
 |
 |  Build an unordered collection of unique elements.
 |
 |  Methods defined here:
 |
 |  __and__(self, value, /)
 |      Return self&value.
 |
 |  __contains__(...)
 |      x.__contains__(y) <==> y in x.
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
```

In [25]:

```python
print(s1)
print(s2)
```

```
{1, 2, 3}
{3, 4, 5}
```

Out[25]:

```
{1, 2, 3, 4, 5}
```

What about joining 4 different sets?

In [57]:

```python
s1 = {1, 2, 3}
s2 = {3, 4, 5}
s3 = {5, 6, 7}
s4 = {7, 8, 9}
print(s1.union(s2).union(s3).union(s4))
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9}
{1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Or we could use unpacking in this way:

In [27]:

Out[27]:

```
{1, 2, 3, 4, 5, 6, 7, 8, 9}
```

What we did here was to unpack each set directly into another set!

The same works for dictionaries - just remember that * for dictionaries unpacks the keys only.

In [29]:

```python
d1 = {'key1': 1, 'key2': 2}
d2 = {'key2': 3, 'key3': 3}
```

Out[29]:

```
['key1', 'key2', 'key2', 'key3']
```

So, is there anything to unpack the key-value pairs for dictionaries instead of just the keys?

Yes - we can use the ** operator:

In [30]:
```
d1 = {'key1': 1, 'key2': 2}
d2 = {'key2': 3, 'key3': 3}

{**d1, **d2}
```

Out[30]:
```
{'key1': 1, 'key2': 3, 'key3': 3}
```

Notice what happened to the value of **key2**. The value for the second occurrence of **key2** was retained (overwritten).

In fact, if we write the unpacking reversing the order of d1 and d2:

In [31]:
```
{**d2, **d1}
```

Out[31]:
```
{'key1': 1, 'key2': 2, 'key3': 3}
```

we see that the value of **key2** is now **2**, since it was the second occurrence.

Of course, we can unpack a dictionary into a dictionary as seen above, but we can mix in our own key-value pairs as well - it is just a dictionary literal after all.

In [32]:
```
{'a': 1, 'b': 2, **d1, **d2, 'c': 3}
```

Out[32]:
```
{'a': 1, 'b': 2, 'c': 3, 'key1': 1, 'key2': 3, 'key3': 3}
```

Again, if we have the same keys, only the "latest" value of the key is retained:

In [33]:
```
{'key1': 100, **d1, **d2, 'key3': 200}
```

Out[33]:
```
{'key1': 1, 'key2': 3, 'key3': 200}
```

Nested Unpacking

Python even supports nested unpacking:

In [36]:
```
a, b, (c, d) = [1, 2, ['X', 'Y']]
print(a)
print(b)
print(c)
print(d)
```

```
1
2
X
Y
```

In fact, since a string is an iterable, we can even write:

In [37]:
```
a, b, (c, d) = [1, 2, 'XY']
print(a)
print(b)
print(c)
```

```
1
2
X
Y
```

We can even write something like this:

In [38]:
```
a, b, (c, d, *e) = [1, 2, 'python']
print(a)
print(b)
print(c)
print(d)
```

```
1
2
p
y
['t', 'h', 'o', 'n']
```

Remember when we said that we can use a * only **once**...

How about this then?

In [39]:
```
a, *b, (c, d, *e) = [1, 2, 3, 'python']
print(a)
print(b)
print(c)
print(d)
```

```
1
[2, 3]
p
y
['t', 'h', 'o', 'n']
```

We can break down what happened here in multiple steps:

In [40]:
```
a, *b, tmp = [1, 2, 3, 'python']
print(a)
print(b)
```

```
1
[2, 3]
python
```

In [41]:
```
c, d, *e = tmp
print(c)
print(d)
```

```
p
y
['t', 'h', 'o', 'n']
```

So putting it together we get our original line of code:

In [68]:
```
a, *b, (c, d, *e) = [1, 2, 3, 'python']
print(a)
print(b)
print(c)
print(d)
```

```
1
[2, 3]
p
y
['t', 'h', 'o', 'n']
```

If we wanted to do the same thing using slicing:

In [1]:
```
l = [1, 2, 3, 'python']
```

Out[1]:
```
(1, [2, 3], 'p', 'y', ['t', 'h', 'o', 'n'])
```

In [2]:
```
l = [1, 2, 3, 'python']
a, b, c, d, e = l[0], l[1:-1], l[-1][0], l[-1][1], list(l[-1][2:])
print(a)
print(b)
print(c)
print(d)
```

```
1
[2, 3]
p
y
['t', 'h', 'o', 'n']
```

Of course, this works for arbitrary lengths and indexable sequence types:

In [3]:
```
l = [1, 2, 3, 4, 'unladen swallow']
a, b, c, d, e = l[0], l[1:-1], l[-1][0], l[-1][1], list(l[-1][2:])
print(a)
print(b)
print(c)
print(d)
```

```
1
[2, 3, 4]
u
n
['l', 'a', 'd', 'e', 'n', ' ', 's', 'w', 'a', 'l', 'l', 'o', 'w']
```

or even:

In [4]:
```
l = [1, 2, 3, 4, ['a', 'b', 'c', 'd']]
a, b, c, d, e = l[0], l[1:-1], l[-1][0], l[-1][1], list(l[-1][2:])
print(a)
print(b)
print(c)
print(d)
```

```
1
[2, 3, 4]
a
b
['c', 'd']
```

In [ ]: