W02_02_Unpacking

September 12, 2022

1 Unpacking

1.0.1 Tuple unpacking

We have already seen that when we return multiple items from a function, they are packed up into a tuple, and then unpacked by the caller, for example:

[43]:
```
def count_punct(s):
    # notice single-quote!
    return s.count("."), s.count(","), s.count("'")
s = "It was the best of times, it was the blurst of times."
np, nc, nq = count_punct(s)
```

This is actually a special case of a generic mechanism in Python called *unpacking*. It works with tuples and dictionaries.

[44]:
```
print(np, nc, nq)
```

```
1 1 0
```

It doesn't only work in `return`. Here's an example showing how to swap two items using unpacking:

[45]:
```
a = 10
b = 20
a, b = b, a
```

[46]:
```
print(a, b)
```

```
20 10
```

We can use a "wild-card" `*` when unpacking, as follows. This is a common pattern in list-processing - equivalent to `car` and `cons`, for Lisp fans.

[7]:
```
L = [5, 6, 7, 8, 9]
head, *rest = L
print(head)
print(rest)
```

```
5
[6, 7, 8, 9]
```

Notice that head is a single item, the first item or the "head" of the list, whereas `rest` gets everything else, so it's a list. Notice that `*rest` is used when unpacking, but the variable that is created is just called `rest`.

Exercise: suppose `L = [5]`. What values will `head` and `rest` have?

1.0.2 Tuple packing in function arguments

The opposite also exists. This "packing" is the basis of two mechanisms for variable-length argument lists in Python functions, called `*args` and `**kwargs`.

First, notice that - surprisingly? - this works:

[10]:
```python
print(max(4, 5))
print(max(4, 5, 6))
print(max(4, 5, 6, 7))
```

```
5
6
7
```

Here, `max` takes a variable number of arguments. How can we program a function like that?

[48]:
```python
def max(*args): # override the builtin `max`
    # will raise error if len(args) == 0
    result = args[0]
    for arg in args:
        if arg > result:
            result = arg
    return result
```

[32]:
```python
print(max(4, 5))
print(max(4, 5, 6))
print(max(4, 5, 6, 7))
```

```
5
6
7
```

What we have seen is that `*` attached to a function parameter name allows multiple arguments to be packed into that parameter. It becomes a tuple:

[13]:
```python
def type_test(*args):
    print(type(args))
type_test(4, 5, 6)
```

```
<class 'tuple'>
```

1.0.3 Dict packing in function arguments

A similar mechanism is available for keyword arguments. In this case, multiple parameter name-value pairs are packed into a `dict`. Here's a contrived example:

[14]:
```python
def f(**kwargs):
    for k in kwargs:
        if k.startswith("_"):
            print(k, kwargs[k])
f(a=1, b=2, _c=3)
```

```
_c 3
```

This mechanism is used a lot in large libraries like Matplotlib (for plotting). See, e.g., https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.scatter.html.

```python
matplotlib.pyplot.scatter(x, y, s=None, c=None, marker=None, cmap=None, norm=None, vmin=None, v
```

This API already has a lot of arguments, but there are many more argument relating to `Container` which the `scatter` function will not use explicitly, but will only pass on to `Container` sub-functions. So instead of writing `scatter()` with all those extra arguments, they are just anonymised and shortened as `**kwargs`. This is great: if the `Container` API changes, we don't have to update our `scatter` function at all.

By the way, above we also see a bare `*`. That says that everything after `*` can only be passed as keyword arguments. There is a similar `/` also. It marks the end of positional-only arguments, ie arguments which must not be passed as keyword arguments. There is quite a bit of complexity here!

1.0.4 Tuple and dict unpacking at call time

Unpacking can also be useful when *calling* a function. In this example, we have a function which expects individual arguments, but our data is already packed up. We unpack on the fly. This is just one example where Python makes it easy to "plug in" to an API which doesn't quite fit our data.

[24]:
```python
def f(a, b, c, d):
    print(a + b + c + d)
ab = (1, 2)
cd = {"c": 3, "d": 4}
f(*ab, **cd) # unpack on the fly
```

```
10
```

Once again, we can also use the unpacking syntax in other contexts, not just function APIs. This example merges two dictionaries by unpacking them into a new dictionary:

[19]:
```python
d1 = {"a": 1, "b": 2}
d2 = {"a": 7, "c": 3, "d": 4}
d = {**d1, **d2}
```

Exercise: what is the value of `d["a"]`? Think about it first, then confirm by trying it.