List destructuring (aka packing and unpacking) All Versions

Examples

Destructuring assignment

In assignments, you can split an Iterable into values using the "unpacking" syntax:

Destructuring as values

```
a, b = (1, 2)
print(a)
# Prints: 1
print(b)
# Prints: 2
```

If you try to unpack more than the length of the iterable, you'll get an error:

```
a, b, c = [1]
# Raises: ValueError: not enough values to unpack (expected 3, got 1)
```

Python 3.x > 3.0

Destructuring as a list

You can unpack a list of unknown length using the following syntax:

```
head, *tail = [1, 2, 3, 4, 5]
```

Here, we extract the first value as a scalar, and the other values as a list:

```
print(head)
# Prints: 1
print(tail)
# Prints: [2, 3, 4, 5]
```

Which is equivalent to:

```
l = [1, 2, 3, 4, 5]
head = l[0]
tail = l[1:]
```

It also works with multiple elements or elements form the end of the list:

```
a, b, *other, z = [1, 2, 3, 4, 5]
print(a, b, z, other)
# Prints: 1 2 5 [3, 4]
```

Ignoring values in destructuring assignments

If you're only interested in a given value, you can use `_` to indicate you aren't interested. Note: this will still set `_`, just most people don't use it as a variable.

```
a, _ = [1, 2]
print(a)
# Prints: 1
a, _, c = (1, 2, 3)
print(a)
# Prints: 1
print(c)
# Prints: 3
```

Python 3.x > 3.0

Ignoring lists in destructuring assignments

Finally, you can ignore many values using the `*_` syntax in the assignment:

```
a, *_ = [1, 2, 3, 4, 5]
print(a)
# Prints: 1
```

which is not really interesting, as you could using indexing on the list instead. Where it gets nice is to keep first and last values in one assignment:

```
a, *_, b = [1, 2, 3, 4, 5]
print(a, b)
# Prints: 1 5
```

or extract several values at once:

```
a, _, b, _, c, *_ = [1, 2, 3, 4, 5, 6]
print(a, b, c)
# Prints: 1 3 5
```

Packing function arguments

In functions, you can define a number of mandatory arguments:

```
def fun1(arg1, arg2, arg3):
    return (arg1,arg2,arg3)
```

which will make the function callable only when the three arguments are given:

```
fun1(1, 2, 3)
```

and you can define the arguments as optional, by using default values:

```
def fun2(arg1='a', arg2='b', arg3='c'):
    return (arg1,arg2,arg3)
```

so you can call the function in many different ways, like:

```
fun2(1)              → (1,b,c)
fun2(1, 2)           → (1,2,c)
fun2(arg2=2, arg3=3) → (a,2,3)
...
```

But you can also use the destructuring syntax to *pack* arguments up, so you can assign variables using a `list` or a `dict`.

Packing a list of arguments

Consider you have a list of values

```
l = [1,2,3]
```

You can call the function with the list of values as an argument using the `*` syntax:

```
fun1(*l)
# Returns: (1,2,3)
fun1(*['w', 't', 'f'])
# Returns: ('w','t','f')
```

But if you do not provide a list which length matches the number of arguments:

```
fun1(*['oops'])
# Raises: TypeError: fun1() missing 2 required positional arguments: 'arg2' and 'arg3'
```

Packing keyword arguments

Now, you can also pack arguments using a dictionary. You can use the `**` operator to tell Python to unpack the `dict` as parameter values:

```
d = {
  'arg1': 1,
  'arg2': 2,
  'arg3': 3
}
fun1(**d)
# Returns: (1, 2, 3)
```

when the function only has positional arguments (the ones without default values) you need the dictionary to be contain of all the expected parameters, and have no extra parameter, or you'll get an error:

```
fun1(**{'arg1':1, 'arg2':2})
# Raises: TypeError: fun1() missing 1 required positional argument: 'arg3'
fun1(**{'arg1':1, 'arg2':2, 'arg3':3, 'arg4':4})
# Raises: TypeError: fun1() got an unexpected keyword argument 'arg4'
```

For functions that have optional arguments, you can pack the arguments as a dictionary the same way:

```
fun2(**d)
# Returns: (1, 2, 3)
```

But there you can omit values, as they will be replaced with the defaults:

```
fun2(**{'arg2': 2})
# Returns: ('a', 2, 'c')
```

And the same as before, you cannot give extra values that are not existing parameters:

```
fun2(**{'arg1':1, 'arg2':2, 'arg3':3, 'arg4':4})
# Raises: TypeError: fun2() got an unexpected keyword argument 'arg4'
```

In real world usage, functions can have both positional and optional arguments, and it works the same:

```
def fun3(arg1, arg2='b', arg3='c')
    return (arg1, arg2, arg3)
```

you can call the function with just an iterable:

```
fun3(*[1])
# Returns: (1, 'b', 'c')
fun3(*[1,2,3])
# Returns: (1, 2, 3)
```

or with just a dictionary:

```
fun3(**{'arg1':1})
# Returns: (1, 'b', 'c')
fun3(**{'arg1':1, 'arg2':2, 'arg3':3})
# Returns: (1, 2, 3)
```

or you can use both in the same call:

```
fun3(*[1,2], **{'arg3':3})
# Returns: (1,2,3)
```

Beware though that you cannot provide multiple values for the same argument:

```
fun3(*[1,2], **{'arg2':42, 'arg3':3})
# Raises: TypeError: fun3() got multiple values for argument 'arg2'
```

Unpacking function arguments

When you want to create a function that can accept any number of arguments, and not enforce the position or the name of the argument at "compile" time, it's possible and here's how:

```
def fun1(*args, **kwargs):
    print(args, kwargs)
```

The *args and **kwargs parameters are special parameters that are set to a tuple and a dict, respectively:

```
fun1(1,2,3)
# Prints: (1, 2, 3) {}
fun1(a=1, b=2, c=3)
# Prints: () {'a': 1, 'b': 2, 'c': 3}
fun1('x', 'y', 'z', a=1, b=2, c=3)
# Prints: ('x', 'y', 'z') {'a': 1, 'b': 2, 'c': 3}
```

If you look at enough Python code, you'll quickly discover that it is widely being used when passing arguments over to another function. For example if you want to extend the string class:

```
class MyString(str):
    def __init__(self, *args, **kwarg):
        print('Constructing MyString')
        super(MyString, self).__init__(*args, **kwarg)
```

Syntax

Parameters

Remarks