Seeing Stars

DAVID BEAZLEY

David Beazley is an open source developer and author of the *Python Essential Reference* (4th Edition, Addison-Wesley, 2009). He is also known as the creator of Swig (http://www.swig.org) and Python Lex-Yacc (http://www.dabeaz.com/ply.html). Beazley is based in Chicago, where he also teaches a variety of Python courses. dave@dabeaz.com

As you read this, Python 3.5 should be hitting the streets with a wide assortment of new features and even some new syntax. "New syntax?" you ask. Why yes. Even though Python has been around for more than 25 years now, it continues to evolve and sprout surprising new features from time to time. In this month's installment, I'm going to look at a seemingly minor part of Python that turns out to be fairly useful—the use of `*` and `**` in function arguments, function argument passing, and data handling.

You Want an Argument?

Traditionally, `*` and `**` have been used to write functions that accept any number of positional or keyword arguments. For example, this function accepts any number of positional arguments, which are passed as a tuple to `args`:

```
>>> def f(*args):
...     print(args)
...
>>> f(1,2,3)
(1, 2, 3)
>>> f(1)
(1,)
>>> f(4,5)
(4, 5)
>>>
```

This function accepts any number of keyword arguments, which are passed to `kwargs` as a dictionary:

```
>>> def g(**kwargs):
...     print(kwargs)
...
>>> g(color='red', size='huge')
{'color': 'red', 'size': 'huge'}
>>> g(xmin=0, xmax=-10, title='Plot')
{'xmin': 0, 'xmax': -10, 'title': 'Plot'}
>>>
```

The `*args` and `**kwargs` can be combined with other arguments and even used together as long as they go at the end of the argument list and the keyword arguments appear last. For example:

```
def h(x, y, *args, **kwargs):
    ...
```

A common use of `*args` and `**kwargs` is in writing code that's meant to be very general purpose. For example, consider this class definition that makes it easy for someone to define simple data structures:

```
class Structure(object):
    _fields = ()
    def __init__(self, *args):
        if len(args) != len(self._fields):
            raise TypeError('Expected %d arguments' % len(self._
fields))
        for name, val in zip(self._fields, args):
            setattr(self, name, val)

# Examples
class Date(Structure):
    _fields = ('year', 'month', 'day')

class Address(Structure):
    _fields = ('hostname', 'port')
```

Sometimes `**kwargs` is used to write functions that take a large number of options that you want specified by keyword only. For example:

```
def config(**options):
    outfile = options['outfile']    # Required argument
    level = options.get('level', 0)  # Optional argument
    ...

config(outfile='output.txt', level=20)    # Ok
config('output.txt', 20)              # Error.
```

Passing Argument

The * and ** syntax are also used to pass data as arguments to functions. For example, suppose you have this function:

```
def f(x, y, z):
    ...
```

If you already have a sequence of arguments or a dictionary of keywords, you can pass them as follows:

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

f(*a)     # Same as f(1, 2, 3)
f(**b)    # Same as f(x=1, y=2, y=3)
```

Both of these conventions can be especially useful when working with data that you have already obtained somehow but that you want transformed into another form. For example, suppose you have a list of tuples and a class definition like this:

```
stocks = [
  ('IBM', 50, 91.25),
  ('HPQ', 75, 37.23),
  ('MSFT', 100, 47.80)
]

class Stock(object):
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price
```

You can easily convert the list into instances using a statement like this:

```
stocks = [Stock(*s) for s in stocks]
```

The use of * also enables some unusual tricks. For example, consider this example of "unzipping" data:

```
>>> a = ['name', 'shares', 'price']
>>> b = ['IBM', 50, 91.25]
>>> # Zip the two sequences into a list of tuples
>>> c = list(zip(a,b))
>>> c = [('name', 'IBM'), ('shares',50), ('price', 91.25)]
>>> # Unzip a list of tuples into separate sequences
>>> d, e = zip(*c)
>>> d
('name', 'shares', 'price')
>>> e
('IBM', 50, 91.25)
>>>
```

Needless to say, that last step with `zip(*c)` might require a bit more study (left as an exercise).

Keyword-Only Arguments

Python 3 introduced an extension to the * syntax that makes it easier to have keyword-only arguments. Specifically, named arguments are allowed to appear after an argument with *. For example:

```
def receive(maxsize, *, block=True):
    ...

msg = receive(1024)                # OK
msg = receive(1024, block=False)   # OK
msg = receive(1024, False)         # Error

def total(*items, initial=0):
    total = initial
    for it in items:
        total += it
    return total

a = total(1,2,3, initial=100)    # a <- 106
```

This ability to have named keyword-only arguments can be a useful way to clean up library code that might otherwise depend on `**kwargs`. For example, the `config()` function from earlier could be rewritten as follows:

```
def config(*, outfile, level=0):
    ...
```

This version will produce better error messages, have a more useful help screen, and involve much less code related to handling the arguments. Keyword-only arguments are good.

Wildcard Unpacking

If you have a tuple, it is easy to unpack into separate variables. For example:

```
address = ('www.python.org', 80)

hostname, port = address       # Unpack
```

This all works well as long as the number of items in the tuple exactly matches the number of variables specified—if not, you get an error. Python 3 allows you to use the * as a wildcard in unpacking. For example:

```
>>> row = ('Elwood', 'Blues', '1060 W Addison', 'Chicago', 'IL',
'60613')
>>> first, last, *rest = row
>>> first
'Elwood'
>>> last
'Blues'
>>> rest
['1060 W Addison', 'Chicago', 'IL', '60613']
>>> first, last, *rest, zipcode = row
>>> first
'Elwood'
>>> last
'Blues'
>>> zipcode
'60613'
>>> rest
['1060 W Addison', 'Chicago', 'IL']
>>>
```

Notice how all of the extra values are simply placed in a list. Wildcard unpacking can be particularly useful if you're working with rows of data of varying length but are only interested in some of the values. For example:

```
rows = [
    (1, 2),
    (3, 4),
    (5, 6, 'x'),
    (7, 8, 'x', 'y'),
    (9, 10)
]

for x, y, *extra in rows:
    ...
```

Unpacking and Argument-Passing Extensions

Python 3.5 extends the capabilities of * and ** in some new and interesting directions. First, you can use both operations more than once when making function calls. For example:

```
def f(a, b, c, d):
    ...

x = (1, 2)
y = (3, 4)
f(*x, *y)       #  Same as f(1, 2, 3, 4)

x = { 'a': 1, 'b': 2}
y = { 'c': 3, 'd': 4}
f(**x, **y)     #  Same as f(a=1, b=2, c=3, d=4)
```

These extensions simplify code that previously had to assemble the arguments by hand. For example, in previous versions of Python, you would have had to write the following:

```
f(*(x+y))
f(*(tuple(x)+tuple(y)))    # Safer version to make sure types
                           # match in +

kwargs = dict(x)           # Make a copy of x
kwargs.update(y)           # Merge in values from y
f(**kwargs)
```

You can also perform unpacking when creating list, tuple, set, and dictionary literals. For example:

```
a = [1, 2]
b = [ *a, 3, 4]        # b = [1, 2, 3, 4]
c = [3, *a, 4]         # c = [3, 1, 2, 4]
d = [3, *a, *a, 4]     # d = [3, 1, 2, 1, 2, 4]

m = { 'x': 1, 'y': 2 }
n = { **m, 'z': 3 }    # n = {'x':1, 'y':2, 'z':3 }
```

In such unpacking, later elements will silently replace earlier elements if there happen to be any duplicates. For example:

```
a = { 'x': 1, 'y': 2 }
b = { 'x': 3, 'z': 4 }
c = { **a, **b }       # c = { 'x':3, 'y':2, 'z':4 }
```

Although these enhancements look minor, they do enable certain kinds of new operations. It is now easy to merge dictionaries as a single expression as shown above. This can extend naturally into operations involving lists of dictionaries and other structures. For example:

```
s1 = [
   {'x': 1, 'y': 2},
   {'x': 3, 'y': 4},
   {'x': 5, 'y': 6}
]

s2 = [
   {'z': 10, 'w': 11 },
   {'z': 12, 'w': 13 },
   {'z': 14, 'w': 15 }
]

merged = [ { **i1, **i2 } for i1, i2 in zip(s1, s2) ]
# merged = [
#    { 'x': 1, 'y': 2, 'z': 10, 'w': 11},
#    { 'x': 3, 'y': 4, 'z': 12, 'w': 13},
#    { 'x': 5, 'y': 6, 'z': 14, 'w': 15}
# ]
```

This change also enables a common dictionary type transformation that I find myself performing with some regularity. For example, suppose you have some raw dictionary data read from a file such as this:

```
rows = [
 {'name': 'AA', 'price': '32.20', 'shares': '100'},
 {'name': 'IBM', 'price': '91.10', 'shares': '50'},
 {'name': 'CAT', 'price': '83.44', 'shares': '150'},
 {'name': 'MSFT', 'price': '51.23', 'shares': '200'},
 {'name': 'GE', 'price': '40.37', 'shares': '95'},
 {'name': 'MSFT', 'price': '65.10', 'shares': '50'},
 {'name': 'IBM', 'price': '70.44', 'shares': '100'}
}
```

Now suppose you wanted to apply a conversion to some of the values (e.g., convert shares to an integer and price to a float). You can do this:

```
conversions = [ ('shares', int), ('price', float) ]
converted = [ {**row, **{name:func(row[name]) for name, func
in conversions}}
           for row in rows ]
```

This does exactly what you want, although I'm willing to concede that it might be too clever for its own good. The alternative is to unwind it to this:

```
converted = []
for row in rows:
   newrow = dict(row)
   for name, func in conversions:
      newrow[name] = func(row[name])
   converted.append(newrow)
```

Needless to say, that's not nearly as clever nor preserving of one's future job security.

More Information

If you're intrigued by some of the new uses of * and **kwargs, more information can be found in various PEPs. For example, PEP 448 describes the generalized unpacking features added to Python 3.5 [1]; PEP 3102 describes keyword-only arguments [2]; and PEP 3132 describes the wildcard unpacking of sequences [3].

These are not the only syntax changes to Python 3.5. In future installments, we'll look at some of the new features added to the language. In the meantime, you might take a look at the "What's New in Python 3.5" document [4].

References

[1] PEP 448: https://www.python.org/dev/peps/pep-0448/.

[2] PEP 3102: https://www.python.org/dev/peps/pep-3102/.

[3] PEP 3132: https://www.python.org/dev/peps/pep-3132/.

[4] https://docs.python.org/dev/whatsnew/3.5.html.