Multiple assignment and tuple unpacking improve Python code readability

Mar 7th, 2018 4:30 pm | Comments

Whether I'm teaching new Pythonistas or long-time Python programmers, I frequently find that **Python programmers underutilize multiple assignment**.

Multiple assignment (also known as tuple unpacking or iterable unpacking) allows you to assign multiple variables at the same time in one line of code. This feature often seems simple after you've learned about it, but **it can be tricky to recall multiple assignment when you need it most**.

In this article we'll see what multiple assignment is, we'll take a look at common uses of multiple assignment, and then we'll look at a few uses for multiple assignment that are often overlooked.

Note that in this article I will be using f-strings which are a Python 3.6+ feature. If you're on an older version of Python, you'll need to mentally translate those to use the string `format` method.

How multiple assignment works

I’ll be using the words **multiple assignment**, **tuple unpacking**, and **iterable unpacking** interchangeably in this article. They’re all just different words for the same thing.

Python’s multiple assignment looks like this:

```
>>> x, y = 10, 20
```

Here we’re setting `x` to `10` and `y` to `20`.

What’s happening at a lower level is that we’re creating a tuple of `10, 20` and then looping over that tuple and taking each of the two items we get f

This syntax might make that a bit more clear:

```
>>> (x, y) = (10, 20)
```

Parenthesis are optional around tuples in Python and they’re also optional in multiple assignment (which uses a tuple-like syntax). All of these are e

```
>>> x, y = 10, 20
>>> x, y = (10, 20)
>>> (x, y) = 10, 20
>>> (x, y) = (10, 20)
```

Multiple assignment is often called “tuple unpacking” because it’s frequently used with tuples. But we can use multiple assignment with any iterable, not just tuples. Here we’re using it with a list:

```
>>> x, y = [10, 20]
>>> x
10
>>> y
20
```

And with a string:

```
>>> x, y = 'hi'
>>> x
'h'
>>> y
'i'
```

Anything that can be looped over can be “unpacked” with tuple unpacking / multiple assignment.

Here’s another example to demonstrate that multiple assignment works with any number of items and that it works with variables as well as objects we’ve just created:

```
>>> point = 10, 20, 30
>>> x, y, z = point
>>> print(x, y, z)
10 20 30
>>> (x, y, z) = (z, y, x)
>>> print(x, y, z)
30 20 10
```

Note that on that last line we’re actually swapping variable names, which is something multiple assignment allows us to do easily.

Alright, let’s talk about how multiple assignment can be used.

Unpacking in a for loop

You’ll commonly see multiple assignment used in `for` loops.

Let’s take a dictionary:

```
>>> person_dictionary = {'name': "Trey", 'company': "Truthful Technology LLC"}
```

Instead of looping over our dictionary like this:

```
for item in person_dictionary.items():
    print(f"Key {item[0]} has value {item[1]}")
```

You’ll often see Python programmers use multiple assignment by writing this:

```
for key, value in person_dictionary.items():
    print(f"Key {key} has value {value}")
```

When you write the `for X in Y` line of a for loop, you’re telling Python that it should do an assignment to `X` for each iteration of your loop. Just like in an assignment using the `=` operator, we can use multiple assignment here.

This:

```
for key, value in person_dictionary.items():
    print(f"Key {key} has value {value}")
```

Is essentially the same as this:

```
for item in person_dictionary.items():
    key, value = item
    print(f"Key {key} has value {value}")
```

We’re just not doing an unnecessary extra assignment in the first example.

So multiple assignment is great for unpacking dictionary items into key-value pairs, but it’s helpful in many other places too.

It’s great when paired with the built-in `enumerate` function:

```
for i, line in enumerate(my_file):
    print(f"Line {i}: {line}")
```

And the `zip` function:

```
for color, ratio in zip(colors, ratios):
    print(f"It's {ratio*100}% {color}.")
```

```
for (product, price, color) in zip(products, prices, colors):
    print(f"{product} is {color} and costs ${price:.2f}")
```

If you're unfamiliar with `enumerate` or `zip`, see my article on looping with indexes in Python.

Newer Pythonistas often see multiple assignment in the context of `for` loops and sometimes assume it's tied to loops. Multiple assignment works fo

An alternative to hard coded indexes

It's not uncommon to see hard coded indexes (e.g. `point[0]`, `items[1]`, `vals[-1]`) in code:

```
print(f"The first item is {items[0]} and the last item is {items[-1]}")
```

When you see Python code that uses hard coded indexes there's often a way to **use multiple assignment to make your code more readable**.

Here's some code that has three hard coded indexes:

```
def reformat_date(mdy_date_string):
    """Reformat MM/DD/YYYY string into YYYY-MM-DD string."""
    date = mdy_date_string.split('/')
    return f"{date[2]}-{date[0]}-{date[1]}"
```

We can make this code much more readable by using multiple assignment to assign separate month, day, and year variables:

```
def reformat_date(mdy_date_string):
    """Reformat MM/DD/YYYY string into YYYY-MM-DD string."""
    month, day, year = mdy_date_string.split('/')
    return f"{year}-{month}-{day}"
```

Whenever you see hard coded indexes in your code, stop to consider whether you could use multiple assignment to make your code more readable.

Multiple assignment is very strict

Multiple assignment is actually fairly strict when it comes to unpacking the iterable we give to it.

If we try to unpack a larger iterable into a smaller number of variables, we'll get an error:

```
>>> x, y = (10, 20, 30)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack (expected 2)
```

If we try to unpack a smaller iterable into a larger number of variables, we'll also get an error:

```
>>> x, y, z = (10, 20)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: not enough values to unpack (expected 3, got 2)
```

This strictness is pretty great. If we're working with an item that has a different size than we expected, the multiple assignment will fail loudly and we'll hopefully now know about a bug in our program that we weren't yet aware of.

Let's look at an example. Imagine that we have a short command line program that parses command-line arguments in a rudimentary way, like this:

```
import sys

new_file = sys.argv[1]
old_file = sys.argv[2]
print(f"Copying {new_file} to {old_file}")
```

Our program is supposed to accept 2 arguments, like this:

```
$ my_program.py file1.txt file2.txt
Copying file1.txt to file2.txt
```

But if someone called our program with three arguments, they will not see an error:

```
$ my_program.py file1.txt file2.txt file3.txt
Copying file1.txt to file2.txt
```

There's no error because we're not validating that we've received exactly 2 arguments.

If we use multiple assignment instead of hard coded indexes, the assignment will verify that we receive exactly the expected number of arguments:

```
import sys

_, new_file, old_file = sys.argv
print(f"Copying {new_file} to {old_file}")
```

Note: we're using the variable name _ to note that we don't care about `sys.argv[0]` (the name of our program). Using _ for variables you don't care about is just a convention.

An alternative to slicing

So multiple assignment can be used for avoiding hard coded indexes and it can be used to ensure we're strict about the size of the tuples/iterables we're working with.

Multiple assignment can be used to replace hard coded slices too!

Slicing is a handy way to grab a specific portion of the items in lists and other sequences.

Here are some slices that are "hard coded" in that they only use numeric indexes:

```
all_after_first = items[1:]
all_but_last_two = items[:-2]
items_with_ends_removed = items[1:-1]
```

Whenever you see slices that don't use any variables in their slice indexes, you can often use multiple assignment instead. To do this we have to talk about a feature that I haven't mentioned yet: the `*` operator.

In Python 3.0, the `*` operator was added to the multiple assignment syntax, allowing us to capture remaining items after an unpacking into a list:

```
>>> numbers = [1, 2, 3, 4, 5, 6]
>>> first, *rest = numbers
>>> rest
[2, 3, 4, 5, 6]
>>> first
1
```

The * operator allows us to replace hard coded slices near the ends of sequences.

These two lines are equivalent:

```
>>> beginning, last = numbers[:-1], numbers[-1]
>>> *beginning, last = numbers
```

These two lines are equivalent also:

```
>>> head, middle, tail = numbers[0], numbers[1:-1], numbers[-1]
>>> head, *middle, tail = numbers
```

With the * operator and multiple assignment you can replace things like this:

```
main(sys.argv[0], sys.argv[1:])
```

With more descriptive code, like this:

```
program_name, *arguments = sys.argv
main(program_name, arguments)
```

So if you see hard coded slice indexes in your code, consider whether you could use multiple assignment to clarify what those slices really represent.

Deep unpacking

This next feature is something that long-time Python programmers often overlook. It doesn't come up quite as often as the other uses for multiple assignment that I've discussed, but it can be very handy to know about when you do need it.

We've seen multiple assignment for unpacking tuples and other iterables. We haven't yet seen that this is can be done *deeply*.

I'd say that the following multiple assignment is *shallow* because it unpacks one level deep:

```
>>> color, point = ("red", (1, 2, 3))
>>> color
'red'
>>> point
(1, 2, 3)
```

And I'd say that this multiple assignment is *deep* because it unpacks the previous `point` tuple further into `x`, `y`, and `z` variables:

```
>>> color, (x, y, z) = ("red", (1, 2, 3))
>>> color
'red'
```

```
>>> x
1
>>> y
2
```

If it seems confusing what’s going on above, maybe using parenthesis consistently on both sides of this assignment will help clarify things:

```
>>> (color, (x, y, z)) = ("red", (1, 2, 3))
```

We’re unpacking one level deep to get two objects, but then we take the second object and unpack it also to get 3 more objects. Then we assign our
ur
new variables (`color`, `x`, `y`, and `z`).

Take these two lists:

```
start_points = [(1, 2), (3, 4), (5, 6)]
end_points = [(-1, -2), (-3, 4), (-6, -5)]
```

Here’s an example of code that works with these lists by using shallow unpacking:

```
for start, end in zip(start_points, end_points):
    if start[0] == -end[0] and start[1] == -end[1]:
        print(f"Point {start[0]},{start[1]} was negated.")
```

And here’s the same thing with deeper unpacking:

```
for (x1, y1), (x2, y2) in zip(start_points, end_points):
    if x1 == -x2 and y1 == -y2:
        print(f"Point {x1},{y1} was negated.")
```

Note that in this second case, it’s much more clear what type of objects we’re working with. The deep unpacking makes it apparent that we’re receiving two 2-itemed tuples each time we loop.

Deep unpacking often comes up when nesting looping utilities that each provide multiple items. For example, you may see deep multiple assignments when using `enumerate` and `zip` together:

```
items = [1, 2, 3, 4, 2, 1]
for i, (first, last) in enumerate(zip(items, reversed(items))):
    if first != last:
        raise ValueError(f"Item {i} doesn't match: {first} != {last}")
```

I said before that multiple assignment is strict about the size of our iterables as we unpack them. With deep unpacking we can also be **strict about the shape of our iterables**.

This works:

```
>>> points = ((1, 2), (-1, -2))
>>> points[0][0] == -points[1][0] and points[0][1] == -points[1][1]
True
```

But this buggy code works too:

```
>>> points = ((1, 2, 4), (-1, -2, 3), (6, 4, 5))
>>> points[0][0] == -points[1][0] and points[0][1] == -points[1][1]
True
```

Whereas this works:

```
>>> points = ((1, 2), (-1, -2))
>>> (x1, y1), (x2, y2) = points
>>> x1 == -x2 and y1 == -y2
True
```

But this does not:

```
>>> points = ((1, 2, 4), (-1, -2, 3), (6, 4, 5))
>>> (x1, y1), (x2, y2) = points
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack (expected 2)
```

With multiple assignment we're assigning variables while also making particular assertions about the size and shape of our iterables. Multiple assignment will help you clarify your code to both humans (for **better code readability**) and to computers (for **improved code correctness**).

Using a list-like syntax

I noted before that multiple assignment uses a tuple-like syntax, but it works on any iterable. That tuple-like syntax is the reason it's commonly called "tuple unpacking" even though it might be more clear to say "iterable unpacking".

I didn't mention before that multiple assignment also works with **a list-like syntax**.

Here's a multiple assignment with a list-like syntax:

```
>>> [x, y, z] = 1, 2, 3
>>> x
1
```

This might seem really strange. What's the point of allowing both list-like and tuple-like syntaxes?

I use this feature rarely, but I find it helpful for **code clarity** in specific circumstances.

Let's say I have code that used to look like this:

```
def most_common(items):
    return Counter(items).most_common(1)[0][0]
```

And our well-intentioned coworker has decided to use deep multiple assignment to refactor our code to this:

```
def most_common(items):
    (value, times_seen), = Counter(items).most_common(1)
    return value
```

See that trailing comma on the left-hand side of the assignment? It's easy to miss and it makes this code look sort of weird. What is that comma even doing in this code?

That trailing comma is there to make a single item tuple. We're doing deep unpacking here.

Here's another way we could write the same code:

```
def most_common(items):
    ((value, times_seen),) = Counter(items).most_common(1)
    return value
```

This might make that deep unpacking a little more obvious but I'd prefer to see this instead:

```
def most_common(items):
    [(value, times_seen)] = Counter(items).most_common(1)
    return value
```

The list-syntax in our assignment makes it more clear that we're unpacking a one-item iterable and then unpacking that single item into `value` and t

When I see this, I also think *I bet we're unpacking a single-item list*. And that is in fact what we're doing. We're using a Counter object from the co nter objects allows us to limit the length of the list returned to us. We're limiting the list we're getting back to just a single item.

When you're unpacking structures that often hold lots of values (like lists) and structures that often hold a very specific number of values (like tuples) you may decide that your code appears more *semantically accurate* if you use a list-like syntax when unpacking those list-like structures.

If you'd like you might even decide to adopt a convention of always using a list-like syntax when unpacking list-like structures (frequently the case when using `*` in multiple assignment):

```
>>> [first, *rest] = numbers
```

I don't usually use this convention myself, mostly because I'm just not in the habit of using it. But if you find it helpful, you might consider using this convention in your own code.

When using multiple assignment in your code, consider when and where a list-like syntax might make your code more descriptive and more clear. This can sometimes improve readability.

Don't forget about multiple assignment

Multiple assignment can improve both the readability of your code and the correctness of your code. It can make your code **more descriptive** while also making implicit assertions about the **size and shape** of the iterables you're unpacking.

The use for multiple assignment that I often see forgotten is its ability to **replace hard coded indexes**, including **replacing hard coded slices** (using the `*` syntax). It's also common to overlook the fact that multiple assignment works *deeply* and can be used with both a *tuple-like* syntax and a *list-like* syntax.

It's tricky to recognize and remember all the cases that multiple assignment can come in handy. Please feel free to use this article as your personal reference guide to multiple assignment.

Get practice with multiple assignment

You don't learn by reading articles like this one, **you learn by writing code**.

To get practice writing some readable code using tuple unpacking, sign up for Python Morsels using the form below. If you sign up to Python Morsels using this form, I'll immediately send you an exercise that involves tuple unpacking.

Your email | Get my tuple unpacking exercise

I won't share you info with others (see the Python Morsels Privacy Policy for details).
This form is reCAPTCHA protected (Google Privacy Policy & TOS)

Privacy - Terms

Posted by Trey Hunner Mar 7th, 2018 4:30 pm favorite, python, readability, tuples

Tweet

« Python: range is not an iterator! Keyword (Named) Arguments in Python: How to Use Them »

Comments

ALSO ON TREY'S BLOG

How to make an iterator in Python

3 years ago • 11 comments

I wrote an article sometime ago on the iterator protocol that powers Python's for …

No really, pathlib is great

2 years ago • 8 comments

I recently published an article about Python's pathlib module and how I …

Why you should be using pathlib

2 years ago • 27 comments

When I discovered Python's new pathlib module a few years ago, I initially wrote …

Ov co

2 ye

List one in F

13 Comments Trey's Blog Disqus' Privacy Policy 1 Login

Recommend 18 Tweet Share Sort by Best

Hi! My name is Trey Hunner.

I help Python teams **write better Python code** through Python team training.

I also help individuals **level-up their Python skills** with weekly Python skill-building.

Python Team Training

Write Pythonic code

Privacy - Terms

Write more Pythonic code ↑

I send out 1 Python exercise every week through a Python skill-building service called Python Morsels.

If you'd like to **improve your Python skills every week**, sign up!

email@domain.com

Sign me up for Python Morsels!

You can find the Privacy Policy here.
reCAPTCHA protected (Google Privacy Policy & TOS)

The best way to improve your skills is to **write more code**, but it's time consuming to figure out what code to write. I've made a Python skill-building service to help solve this problem.

Each week you'll get an exercise that'll help you dive deeper into Python and carefully **reflect on your own coding style**. The first 4 exercises are free.

Sign up below for **four free exercises**!

Your email Sign up

See the Python Morsels Privacy Policy.
This form is reCAPTCHA protected (see Google Privacy Policy & Terms of Service)

Favorite Posts

- Python List Comprehensions
- How to Loop With Indexes in Python
- Check Whether All Items Match a Condition in Python
- Keyword (Named) Arguments in Python: How to Use Them
- Tuple unpacking improves Python code readability
- The Idiomatic Way to Merge Dictionaries in Python
- The Iterator Protocol: How for Loops Work in Python
- Craft Your Python Like Poetry
- Python: range is not an iterator!
- Counting Things in Python: A History

Privacy - Terms

Follow @treyhunner