LEARN PYTHON PROGRAMMING

Destructuring in Python

Phil Best
5 min read · Jul 4

In this week's post we're going to look at a very interesting and versatile tool called destructuring. Destructuring (also called unpacking) is where we take a collection, like a list or a tuple, and we break it up into individual values. This is quite useful, as it enables us to do things like destructuring assignments, where we assign values to several variables at once from a single collection.

Standard Destructuring Assignments

Python, like many programming languages, allows us to assign more than one variable at a time on a single line. We just have to provide the same number of values on both sides of the assignment. For example:

```
x, y = 5, 11
```

Here we assign the value `5` to `x`, and `11` to `y`. The values are assigned entirely based on order, so if we switch the variable order, or the order of the values we intend to assign, we will end up with different results.

How this works is fairly straightforward, but what's less obvious is that this is an example of destructuring. What if I wrote it like this?

```
x, y = (5, 11)
```

One thing that a lot of newer Python programmers in particular don't realise is that parentheses have nothing at all to do with tuples. In fact, it's the commas which tell Python something is a tuple: we just add brackets for readability in a lot of cases. In some instances the brackets are actually necessary, in order to isolate the tuple from the syntax around it, such as when we put a tuple inside a list. However, in this case the brackets are still not part of the tuple syntax. Adding brackets around a single number doesn't turn it into a tuple.

With all that in mind, we actually end up destructuring a tuple in the example above, splitting the tuple `(5, 11)` into its component values, so that we can bind those values to the two variable names.

We're not limited to just tuples here. We can also destructure a list, for example, or even a set. You're unlikely to want to perform a destructuring assignment with a set, however, since the order is not guaranteed, and we therefore end up with variables we don't really know the values of.

If we try to destructure a collection with more or fewer values than we provide variables, we end up with a `ValueError`.

```
ValueError: not enough values to unpack (expected 4, got 3)
```

Destructuring in `for` loops

A couple of months back, we wrote a snippet post on `enumerate`, which is a vital component in writing good, Pythonic loops. In case you're not familiar, the syntax for `enumerate` looks like this:

```python
example_list = ["A", "B", "C"]

for counter, letter in enumerate(example_list):
    print(counter, letter)

# 0 A
# 1 B
# 2 C
```

`enumerate` takes an iterable collection as an argument and returns an enumerate object containing a tuple for each item in the collection. Each tuple contains a counter value, which increments with each iteration, along with a value from the provided collection.

As you can see in the example above, we provide two variable names when creating our `for` loop: `counter` and `letter`. This is actually a very common example of destructuring. For each tuple in the enumerate object, the first value gets assigned to `counter`, and the second value is assigned to `letter`.

Once again, there isn't any magic going on here with the variable names, and the assignment is entirely based on the order of the values. If we were to switch the position of `counter` and `letter`, we'd end up with some confusing variable names.

It's possible to do this type of destructuring with as many values as we like, and this is not limited to the `enumerate` function. For example, we could do something like this:

```
people = [
    ("Bob", 42, "Mechanic"),
    ("James", 24, "Artist"),
    ("Harry", 32, "Lecturer")
]

for name, age, profession in people:
    print(f"Name: {name}, Age: {age}, Profession: {profession}")
```

Here we break apart each tuple in the `people` list, assigning the values to `name` , `age` , and `profession` respectively.

This is a lot better than something like the code below, where we rely on indices instead of good descriptive names:

```
for person in people:
    print(f"Name: {person[0]}, Age: {person[1]}, Profession: {persor
```

Raymond Hettinger — one of the core Python developers — said said something in one of his talks that really stuck with me. He said that in Python, you should basically never be referring to items by their index: there is nearly always a better way.

Destructuring is a good example of one of those better ways.

Ignoring Values

So, what do we do if we have a collection of values and we don't want to assign all of them? We can use an `_` in place of a variable name.

For example, if we take one of the tuples from the the `people` list above, and we only care about the name and profession, we can do the following:

```
person = ("Bob", 42, "Mechanic")
name, _, profession = person

print(name, profession)  # Bob Mechanic
```

This would also work just as well inside a loop, and can also be used when we don't care about *any* of the values. An example might be when using `range` to ensure a set number of iterations.

```
for _ in range(10):
    <do something>
```

Using `*` to Collect Values

In some situations, we might want to isolate one or two values in a collection, and then keep the other items together. We featured an example of a situation like this our post on list rotation.

In Python, we can use the `*` operator to collect leftover values when performing a destructuring assignment. For example, we might have a list of numbers, and we want to grab the first number, and then assign the remaining numbers to a second variable:

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

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

Here, the first value (`1`) is assigned to `head` , while the rest of the numbers end up in a new list called `tail` .

We can also do this the other way around, creating a new list with everything but the last value, and assigning the last value to its own variable.

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

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

This is interesting, but unless we need to preserve the original list, we already have the `pop` method for stuff like this. However, we can do something else with this syntax. We can can assign *any* number of variables, and then gather up the remainder. We might grab the first and last items, and then gather up the middle, for example:

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

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

Alternatively, we might want to grab the first three items and then bundle up the rest:

```
first, second, third, *rest = [1, 2, 3, 4, 5]
```

There are tonnes of possibilities.

Wrapping Up

I hope you learnt something new from this short post on destructuring in Python!

This isn't actually the end of the story, as there are also ways to pack and unpack collections using `*` and `**`, but I think that content deserves a post of its own to do the topic justice.

If you're learning Python and you find this kind of content interesting, be sure to follow us on Twitter or sign up to our mailing list to stay up to date with all out content. There's a form at the bottom of the page if you're interested.

We also just did a big update to our Complete Python Course, so check that out if you're interested in getting to an advanced level in Python. We have a 30 day money back guarantee, so you really have nothing to lose by giving it a try. We'd love to have you!

Phil Best

I'm a freelance developer, mostly working in web development. I also create course content for Teclado!

Read More

Learn Python Programming

Introduction to Object-Oriented Programming in Python →

Dictionary Merge and Update Operators in Python 3.9 →

Working with Python virtual environments: the complete guide →

→ See all 144 posts

PYTHON SNIPPETS

A Closer Look at Python's Print Function

Print is one of the first functions we encounter when learning Python, but it can do a lot more than say "Hello, world!". Learn about print's other options and master this common function!

Phil Best
2 min read · Jul 8

Like what you see? Enter your e-mail to hear when new posts come out!