Unpacking in Python: Beyond Parallel Assignment

s Pozo Ramos (https://twitter.com/lpozo78) •
in-python-beyond-parallel-assignment/#disqus_thread)

Int

Unpa that consists of assigning an iterable of values to a tuple (/lists-vs-tuple a single assignment statement. As a complement, the term *packing* can be us a single variable using the iterable unpacking operator, `*`.

Histo ically referred to this kind of operation as *tuple unpacking*. However, since be quite useful and popular, it's been generalized to all kinds of iterab ccurate term would be *iterable unpacking*.

In thi packing is and how we can take advantage of this Python feature to make e, and pythonic.

Additionally, we'll also cover some practical examples of how to use the iterable unpacking feature in the context of assignments operations, `for` loops, function definitions, and function calls.

Packing and Unpacking in Python

Python allows a `tuple` (or `list`) of variables to appear on the left side of an assignment operation. Each variable in the `tuple` can receive one value (or more, if we use the `*` operator) from an iterable on the right side of the assignment.

For historical reasons, Python developers used to call this *tuple unpacking*. However, since this feature has been generalized to all kind of iterable, a more accurate term would be *iterable unpacking* and that's what we'll call it in this tutorial.

Unpacking operations have been quite popular among Python developers because they can make our code more readable, and elegant. Let's take a closer look to unpacking in Python and see how this feature can improve our code.

Unpacking Tuples

In Python, we can put a `tuple` of variables on the left side of an assignment operator (`=`) and a `tuple` of values on the right side. The values on the right will be automatically assigned to the variables on the left according to their position in the `tuple`. This is commonly known as *tuple unpacking* in Python. Check out the following example:

```
>>> (a, b, c) = (1, 2, 3)
>>> a
1
>>> b
2
>>> c
3
```


When we put tuples on both sides of an assignment operator, a tuple unpacking operation takes place. The values on the right are assigned to the variables on the left according to their relative position in each `tuple`. As you can see in the above example, `a` will be `1`, `b` will be `2`, and `c` will be `3`.

To create a `tuple` object, we don't need to use a pair of parentheses `()` as delimiters. This also works for tuple unpacking, so the following syntaxes are equivalent:

```
>>> (a, b, c) = 1, 2, 3
>>> a, b, c = (1, 2, 3)
>>> a, b, c = 1, 2, 3
```

Since all these variations are valid Python syntax, we can use any of them, depending on the situation. Arguably, the last syntax is more commonly used when it comes to unpacking in Python.

When we are unpacking values into variables using tuple unpacking, the number of variables on the left side `tuple` must exactly match the number of values on the right side `tuple`. Otherwise, we'll get a `ValueError`.

For example, in the following code, we use two variables on the left and three values on the right. This will raise a `ValueError` telling us that there are too many values to unpack:

```
>>> a, b = 1, 2, 3
Traceback (most recent call last):
  ...
ValueError: too many values to unpack (expected 2)
```


Note: The only exception to this is when we use the `*` operator to pack several values in one variable as we'll see later on.

On the other hand, if we use more variables than values, then we'll get a `ValueError` but this time the message says that there are not enough values to unpack:

```
>>> a, b, c = 1, 2
Traceback (most recent call last):
  ...
ValueError: not enough values to unpack (expected 3, got 2)
```

If we use a different number of variables and values in a tuple unpacking operation, then we'll get a `ValueError`. That's because Python needs to unambiguously know what value goes into what variable, so it can do the assignment accordingly.

Unpacking Iterables

The tuple unpacking feature got so popular among Python developers that the syntax was extended to work with any iterable object. The only requirement is that the iterable yields exactly one item per variable in the receiving `tuple` (or `list`).

Check out the following examples of how iterable unpacking works in Python:

```
>>> # Unpacking strings
>>> a, b, c = '123'
>>> a
'1'
>>> b
'2'
>>> c
'3'
>>> # Unpacking lists
>>> a, b, c = [1, 2, 3]
>>> a
1
>>> b
2
>>> c
3
>>> # Unpacking generators
>>> gen = (i ** 2 for i in range(3))
>>> a, b, c = gen
>>> a
0
>>> b
1
>>> c
4
>>> # Unpacking dictionaries (keys, values, and items)
>>> my_dict = {'one': 1, 'two':2, 'three': 3}
>>> a, b, c = my_dict  # Unpack keys
>>> a
'one'
>>> b
'two'
>>> c
'three'
>>> a, b, c = my_dict.values()  # Unpack values
>>> a
1
>>> b
2
>>> c
```

```
3
>>> a, b, c = my_dict.items()  # Unpacking key-value pairs
>>> a
('one', 1)
>>> b
('two', 2)
>>> c
('three', 3)
```


When it comes to unpacking in Python, we can use any iterable on the right side of the assignment operator. The left side can be filled with a `tuple` or with a `list` of variables. Check out the following example in which we use a `tuple` on the right side of the assignment statement:

```
>>> [a, b, c] = 1, 2, 3
>>> a
1
>>> b
2
>>> c
3
```

It works the same way if we use the `range()` iterator:

```
>>> x, y, z = range(3)
>>> x
0
>>> y
1
>>> z
2
```

Even though this is a valid Python syntax, it's not commonly used in real code and maybe a little bit confusing for beginner Python developers.

Finally, we can also use `set` (/sets-in-python/) objects in unpacking operations. However, since sets are unordered collection, the order of the assignments can be sort of incoherent and can lead to subtle bugs. Check out the following example:

```
>>> a, b, c = {'a', 'b', 'c'}
>>> a
'c'
>>> b
'b'
>>> c
'a'
```

If we use sets in unpacking operations, then the final order of the assignments can be quite different from what we want and expect. So, it's best to avoid using sets in unpacking operations unless the order of assignment isn't important to our code.

Packing With the * Operator

The `*` *operator* is known, in this context, as the *tuple (or iterable) unpacking operator*. It extends the unpacking functionality to allow us to collect or pack multiple values in a single variable. In the following example, we pack a `tuple` of values into a single variable by using the `*` operator:

```
>>> *a, = 1, 2
>>> a
[1, 2]
```

For this code to work, the left side of the assignment must be a `tuple` (or a `list`). That's why we use a trailing comma. This `tuple` can contain as many variables as we need. However, it can only contain one *starred expression*.

We can form a stared expression using the unpacking operator, `*`, along with a valid Python identifier, just like the `*a` in the above code. The rest of the variables in the left side `tuple` are called *mandatory* variables because they must be filled with concrete values, otherwise, we'll get an error. Here's how this works in practice.

Packing the trailing values in `b`:

```
>>> a, *b = 1, 2, 3
>>> a
1
>>> b
[2, 3]
```

Packing the starting values in `a`:

```
>>> *a, b = 1, 2, 3
>>> a
[1, 2]
>>> b
3
```

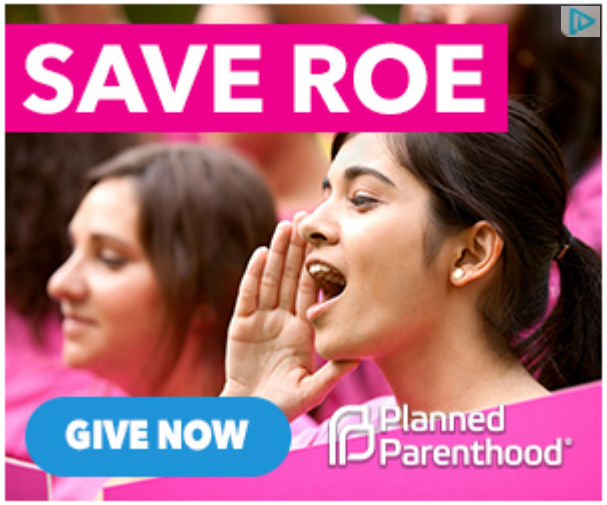
Packing one value in `a` because `b` and `c` are mandatory:

```
>>> *a, b, c = 1, 2, 3
>>> a
[1]
>>> b
2
>>> c
3
```

Packing no values in `a` (`a` defaults to `[]`) because `b`, `c`, and `d` are mandatory:


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

Supplying no value for a mandatory variable (`e`), so an error occurs:




```
>>> *a, b, c, d, e = 1, 2, 3
 ...
ValueError: not enough values to unpack (expected at least 4, got 3)
```

Packing values in a variable with the `*` operator can be handy when we need to collect the elements of a generator in a single variable without using the `list()` function. In the following examples, we use the `*` operator to pack the elements of a generator expression (https://docs.python.org/3/glossary.html#term-generator) and a range (https://docs.python.org/3/library/stdtypes.html#range) object to a individual variable:

```
>>> gen = (2 ** x for x in range(10))
>>> gen
<generator object <genexpr> at 0x7f44613ebcf0>
>>> *g, = gen
>>> g
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
>>> ran = range(10)
>>> *r, = ran
>>> r
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In these examples, the `*` operator packs the elements in `gen`, and `ran` into `g` and `r` respectively. With his syntax, we avoid the need of calling `list()` to create a `list` of values from a `range` object, a generator expression, or a generator function.

Notice that we can't use the unpacking operator, `*`, to pack multiple values into one variable without adding a trailing comma to the variable on the left side of the assignment. So, the following code won't work:

```
>>> *r = range(10)
  File "<input>", line 1
SyntaxError: starred assignment target must be in a list or tuple
```

if we try to use the `*` operator to pack several values into a single variable, then we need to use the singleton `tuple` syntax. For example, to make the above example works, we just need to add a comma after the variable `r`, like in `*r, = range(10)`.

Using Packing and Unpacking in Practice

Packing and unpacking operations can be quite useful in practice. They can make your code clear, readable, and pythonic. Let's take a look at some common use-cases of packing and unpacking in Python.

Assigning in Parallel

One of the most common use-cases of unpacking in Python is what we can call *parallel assignment*. Parallel assignment allows you to assign the values in an iterable to a `tuple` (or `list`) of variables in a single and elegant statement.

For example, let's suppose we have a database about the employees in our company and we need to assign each item in the list to a descriptive variable. If we ignore how iterable unpacking works in Python, we can get ourself writing code like this:

```
>>> employee = ["John Doe", "40", "Software Engineer"]
>>> name = employee[0]
>>> age = employee[1]
>>> job = employee[2]
>>> name
'John Doe'
>>> age
'40'
>>> job
'Software Engineer'
```

Even though this code works, the index handling can be clumsy, hard to type, and confusing. A cleaner, more readable, and pythonic solution can be coded as follows:

```
>>> name, age, job = ["John Doe", "40", "Software Engineer"]
>>> name
'John Doe'
>>> age
40
>>> job
'Software Engineer'
```

Using unpacking in Python, we can solve the problem of the previous example with a single, straightforward, and elegant statement. This tiny change would make our code easier to read and understand for newcomers developers.

Swapping Values Between Variables

Another elegant application of unpacking in Python is swapping values between variables without using a temporary or auxiliary variable. For example, let's suppose we need to swap the values of two variables `a` and `b`. To do this, we can stick to the traditional solution and use a temporary variable to store the value to be swapped as

follows:


```
>>> a = 100
>>> b = 200
>>> temp = a
>>> a = b
>>> b = temp
>>> a
200
>>> b
100
```

This procedure takes three steps and a new temporary variable. If we use unpacking in Python, then we can achieve the same result in a single and concise step:

```
>>> a = 100
>>> b = 200
>>> a, b = b, a
>>> a
200
>>> b
100
```

In statement `a, b = b, a`, we're reassigning `a` to `b` and `b` to `a` in one line of code. This is a lot more readable and straightforward. Also, notice that with this technique, there is no need for a new temporary variable.

Collecting Multiple Values With *

When we're working with some algorithms, there may be situations in which we need to split the values of an iterable or a sequence in chunks of values for further processing. The following example shows how to uses a `list` and slicing operations (https://docs.python.org/3/library/stdtypes.html#common-sequence-operations) to do so:


```
>>> seq = [1, 2, 3, 4]
>>> first, body, last = seq[0], seq[1:3], seq[-1]
>>> first, body, last
(1, [2, 3], 4)
>>> first
1
>>> body
[2, 3]
>>> last
4
```

Even though this code works as we expect, dealing with indices and slices can be a little bit annoying, difficult to read, and confusing for beginners. It has also the drawback of making the code rigid and difficult to maintain. In this situation, the iterable unpacking operator, `*`, and its ability to pack several values in a single variable can be a great tool. Check out this refactoring of the above code:

```
>>> seq = [1, 2, 3, 4]
>>> first, *body, last = seq
>>> first, body, last
(1, [2, 3], 4)
>>> first
1
>>> body
[2, 3]
>>> last
4
```

The line `first, *body, last = seq` makes the magic here. The iterable unpacking operator, `*`, collects the elements in the middle of `seq` in `body`. This makes our code more readable, maintainable, and flexible. You may be thinking, why more flexible? Well, suppose that `seq` changes its length in the road and you still need to collect the middle elements in `body`. In this case, since we're using unpacking in Python, no changes are needed for our code to work. Check out this example:

```
>>> seq = [1, 2, 3, 4, 5, 6]
>>> first, *body, last = seq
>>> first, body, last
(1, [2, 3, 4, 5], 6)
```

If we were using sequence slicing instead of iterable unpacking in Python, then we would need to update our indices and slices to correctly catch the new values.

The use of the `*` operator to pack several values in a single variable can be applied in a variety of configurations, provided that Python can unambiguously determine what element (or elements) to assign to each variable. Take a look at the following examples:

```
>>> *head, a, b = range(5)
>>> head, a, b
([0, 1, 2], 3, 4)
>>> a, *body, b = range(5)
>>> a, body, b
(0, [1, 2, 3], 4)
>>> a, b, *tail = range(5)
>>> a, b, tail
(0, 1, [2, 3, 4])
```

We can move the `*` operator in the `tuple` (or `list`) of variables to collect the values according to our needs. The only condition is that Python can determine to what variable assign each value.

It's important to note that we can't use more than one stared expression in the assignment If we do so, then we'll get a `SyntaxError` as follows:

```
>>> *a, *b = range(5)
  File "<input>", line 1
SyntaxError: two starred expressions in assignment
```

If we use two or more `*` in an assignment expression, then we'll get a `SyntaxError` telling us that two-starred expression were found. This is that way because Python can't unambiguously determine what value (or values) we want to assign to each variable.

Dropping Unneeded Values With *

Another common use-case of the `*` operator is to use it with a dummy variable name to drop some useless or unneeded values. Check out the following example:

```
>>> a, b, *_ = 1, 2, 0, 0, 0, 0
>>> a
1
>>> b
2
>>> _
[0, 0, 0, 0]
```

For a more insightful example of this use-case, suppose we're developing a script that needs to determine the Python version we're using. To do this, we can use the `sys.version_info` attribute (https://docs.python.org/3/library/sys.html#sys.version_info). This attribute returns a tuple containing the five components of the version number: `major`, `minor`, `micro`, `releaselevel`, and `serial`. But we just need `major`, `minor`, and `micro` for our script to work, so we can drop the rest. Here's an example:

```
>>> import sys
>>> sys.version_info
sys.version_info(major=3, minor=8, micro=1, releaselevel='final', serial=0)
>>> mayor, minor, micro, *_ = sys.version_info
>>> mayor, minor, micro
(3, 8, 1)
```

Now, we have three new variables with the information we need. The rest of the information is stored in the dummy variable `_`, which can be ignored by our program. This can make clear to newcomer developers that we don't want to (or need to) use the information stored in `_` cause this character has no apparent meaning.

Note: By default, the underscore character `_` is used by the Python interpreter to store the resulting value of the statements we run in an interactive session. So, in this context, the use of this character to identify dummy variables can be ambiguous.

Returning Tuples in Functions

Python functions can return several values separated by commas. Since we can define `tuple` objects without using parentheses, this kind of operation can be interpreted as returning a `tuple` of values. If we code a function that returns multiple values, then we can perform iterable packing and unpacking operations with the returned values.

Check out the following example in which we define a function to calculate the square and cube of a given number:

```
>>> def powers(number):
...     return number, number ** 2, number ** 3
...
>>> # Packing returned values in a tuple
>>> result = powers(2)
>>> result
(2, 4, 8)
>>> # Unpacking returned values to multiple variables
>>> number, square, cube = powers(2)
>>> number
2
>>> square
4
>>> cube
8
>>> *_, cube = powers(2)
>>> cube
8
```

if we define a function that returns comma-separated values, then we can do any packing or unpacking operation on these values.

Merging Iterables With the * Operator

Another interesting use-case for the unpacking operator, `*`, is the ability to merge several iterables into a final sequence. This functionality works for lists, tuples, and sets. Take a look at the following examples:

```
>>> my_tuple = (1, 2, 3)
>>> (0, *my_tuple, 4)
(0, 1, 2, 3, 4)
>>> my_list = [1, 2, 3]
>>> [0, *my_list, 4]
[0, 1, 2, 3, 4]
>>> my_set = {1, 2, 3}
>>> {0, *my_set, 4}
{0, 1, 2, 3, 4}
>>> [*my_set, *my_list, *my_tuple, *range(1, 4)]
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> my_str = "123"
>>> [*my_set, *my_list, *my_tuple, *range(1, 4), *my_str]
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, '1', '2', '3']
```

We can use the iterable unpacking operator, `*`, when defining sequences to unpack the elements of a subsequence (or iterable) into the final sequence. This will allow us to create sequences on the fly from other existing sequences without calling methods like `append()`, `insert()`, and so on.

The last two examples show that this is also a more readable and efficient way to concatenate iterables. Instead of writing `list(my_set) + my_list + list(my_tuple) + list(range(1, 4)) + list(my_str)` we just write `[*my_set, *my_list, *my_tuple, *range(1, 4), *my_str]`.

Unpacking Dictionaries With the ** Operator

In the context of unpacking in Python, the `**` operator is called the dictionary unpacking operator (https://docs.python.org/3/whatsnew/3.5.html#pep-448-additional-unpacking-generalizations). The use of this operator was extended by PEP 448 (https://www.python.org/dev/peps/pep-0448). Now, we can use it in function

calls, in comprehensions and generator expressions, and in displays (https://docs.python.org/3/reference/expressions.html#dictionary-displays).

A basic use-case for the dictionary unpacking operator is to merge multiple dictionaries (/python-dictionary-tutorial/) into one final dictionary with a single expression. Let's see how this works:

```
>>> numbers = {"one": 1, "two": 2, "three": 3}
>>> letters = {"a": "A", "b": "B", "c": "C"}
>>> combination = {**numbers, **letters}
>>> combination
{'one': 1, 'two': 2, 'three': 3, 'a': 'A', 'b': 'B', 'c': 'C'}
```

If we use the dictionary unpacking operator inside a dictionary display, then we can unpack dictionaries and combine them to create a final dictionary that includes the key-value pairs of the original dictionaries, just like we did in the above code.

An important point to note is that, if the dictionaries we're trying to merge have repeated or common keys, then the values of the right-most dictionary will override the values of the left-most dictionary. Here's an example:

```
>>> letters = {"a": "A", "b": "B", "c": "C"}
>>> vowels = {"a": "a", "e": "e", "i": "i", "o": "o", "u": "u"}
>>> {**letters, **vowels}
{'a': 'a', 'b': 'B', 'c': 'C', 'e': 'e', 'i': 'i', 'o': 'o', 'u': 'u'}
```

Since the `a` key is present in both dictionaries, the value that prevail comes from `vowels`, which is the right-most dictionary. This happens because Python starts adding the key-value pairs from left to right. If, in the process, Python finds keys that already exit, then the interpreter updates that keys with the new value. That's why the value

of the `a` key is lowercased in the above example.

Unpacking in For-Loops

We can also use iterable unpacking in the context of `for` loops. When we run a `for` loop, the loop assigns one item of its iterable to the target variable in every iteration. If the item to be assigned is an iterable, then we can use a `tuple` of target variables. The loop will unpack the iterable at hand into the `tuple` of target variables.

As an example, let's suppose we have a file containing data about the sales of a company as follows:

Product	Price	Sold Units
Pencil	0.25	1500
Notebook	1.30	550
Eraser	0.75	1000
...	...	...

From this table, we can build a `list` of two-elements tuples. Each `tuple` will contain the name of the product, the price, and the sold units. With this information, we want to calculate the income of each product. To do this, we can use a `for` loop like this:

```
>>> sales = [("Pencil", 0.22, 1500), ("Notebook", 1.30, 550), ("Eraser", 0.75, 1000)]
>>> for item in sales:
...     print(f"Income for {item[0]} is: {item[1] * item[2]}")
...
Income for Pencil is: 330.0
Income for Notebook is: 715.0
Income for Eraser is: 750.0
```

This code works as expected. However, we're using indices to get access to individual elements of each `tuple`. This can be difficult to read and to understand by newcomer developers.

Let's take a look at an alternative implementation using unpacking in Python:

```
>>> for product, price, sold_units in sales:
...     print(f"Income for {product} is: {price * sold_units}")
...
Income for Pencil is: 330.0
Income for Notebook is: 715.0
Income for Eraser is: 750.0
```

We're now using iterable unpacking in our `for` loop. This makes our code way more readable and maintainable because we're using descriptive names to identify the elements of each `tuple`. This tiny change will allow a newcomer developer to quickly understand the logic behind the code.

It's also possible to use the `*` operator in a `for` loop to pack several items in a single target variable:

```
>>> for first, *rest in [(1, 2, 3), (4, 5, 6, 7)]:
...     print("First:", first)
...     print("Rest:", rest)
...
First: 1
Rest: [2, 3]
First: 4
Rest: [5, 6, 7]
```

In this `for` loop, we're catching the first element of each sequence in `first`. Then the `*` operator catches a `list` of values in its target variable `rest`.

Finally, the structure of the target variables must agree with the structure of the iterable. Otherwise, we'll get an error. Take a look at the following example:


```
>>> data = [((1, 2), 2), ((2, 3), 3)]
>>> for (a, b), c in data:
...     print(a, b, c)
...
1 2 2
2 3 3
>>> for a, b, c in data:
...     print(a, b, c)
...
Traceback (most recent call last):
  ...
ValueError: not enough values to unpack (expected 3, got 2)
```

In the first loop, the structure of the target variables, `(a, b), c`, agrees with the structure of the items in the iterable, `((1, 2), 2)`. In this case, the loop works as expected. In contrast, the second loop uses a structure of target variables that don't agree with the structure of the items in the iterable, so the loop fails and raises a `ValueError`.

Packing and Unpacking in Functions

We can also use Python's packing and unpacking features when defining and calling functions. This is a quite useful and popular use-case of packing and unpacking in Python.

In this section, we'll cover the basics of how to use packing and unpacking in Python functions either in the function definition or in the function call.

Note: For a more insightful and detailed material on these topics, check out Variable-Length Arguments in Python with `*args` and `**kwargs` (/variable-length-arguments-in-python-with-args-and-kwargs/).

Defining Functions With * and **

We can use the `*` and `**` operators in the signature of Python functions. This will allow us to call the function with a variable number of positional arguments (`*`) or with a variable number of keyword arguments, or both. Let's consider the following function:

```
>>> def func(required, *args, **kwargs):
...     print(required)
...     print(args)
...     print(kwargs)
...
>>> func("Welcome to...", 1, 2, 3, site='StackAbuse.com')
Welcome to...
(1, 2, 3)
{'site': 'StackAbuse.com'}
```

The above function requires at least one argument called `required` . It can accept a variable number of positional and keyword arguments as well. In this case, the `*` operator collects or packs extra positional arguments in a tuple called `args` and the `**` operator collects or packs extra keyword arguments in a dictionary called `kwargs` . Both, `args` and `kwargs` , are optional and automatically default to `()` and `{}` respectively.

Even though the names `args` and `kwargs` are widely used by the Python community, they're not a requirement for these techniques to work. The syntax just requires `*` or `**` followed by a valid identifier. So, if you can give meaningful names to these arguments, then do it. That will certainly improve your code's readability.

Calling Functions With * and **

When calling functions, we can also benefit from the use of the `*` and `**` operator to unpack collections of arguments into separate positional or keyword arguments respectively. This is the inverse of using `*` and `**` in the signature of a function. In the signature, the operators mean **collect or pack** a variable number of arguments in one identifier. In the call, they mean **unpack** an iterable into several arguments.

Here's a basic example of how this works:

```
>>> def func(welcome, to, site):
...     print(welcome, to, site)
...
>>> func(*["Welcome", "to"], **{"site": 'StackAbuse.com'})
Welcome to StackAbuse.com
```

Here, the `*` operator unpacks sequences like `["Welcome", "to"]` into positional arguments. Similarly, the `**` operator unpacks dictionaries into arguments whose names match the keys of the unpacked dictionary.

We can also combine this technique and the one covered in the previous section to write quite flexible functions. Here's an example:

```
>>> def func(required, *args, **kwargs):
...     print(required)
...     print(args)
...     print(kwargs)
...
>>> func("Welcome to...", *(1, 2, 3), **{"site": 'StackAbuse.com'})
Welcome to...
(1, 2, 3)
{'site': 'StackAbuse.com'}
```

The use of the `*` and `**` operators, when defining and calling Python functions, will give them extra capabilities and make them more flexible and powerful.

Conclusion

Iterable unpacking turns out to be a pretty useful and popular feature in Python. This feature allows us to unpack an iterable into several variables. On the other hand, packing consists of catching several values into one variable using the unpacking operator, `*`.

In this tutorial, we've learned how to use iterable unpacking in Python to write more readable, maintainable, and pythonic code.

With this knowledge, we are now able to use iterable unpacking in Python to solve common problems like parallel assignment and swapping values between variables. We're also able to use this Python feature in other structures like `for` loops, function calls, and function definitions.

python (/tag/python/)

(https://twitter.com/share?
=Unpacking%20in%20Python%3A%20Beyond%20Parallel%20Assignment&url=https://stackabuse.com/unpacking-
ython-beyond-parallel-assignment/)

(https://www.facebook.com/sharer/sharer.php?u=https://stackabuse.com/unpacking-in-python-beyond-parallel-
gnment/)

(https://www.linkedin.com/shareArticle?mini=true%26url=https://stackabuse.com/unpacking-in-python-beyond-
llel-assignment/%26source=https://stackabuse.com)

(/author/leodanis/)

About Leodanis Pozo Ramos (/author/leodanis/)

Holguín, Cuba Twitter (https://twitter.com/lpozo78)

Website (https://leodanispozo.netlify.app)

Leodanis is an industrial engineer who loves Python and software development. He is a self-taught Python programmer with 5+ years of experience building desktop applications with PyQt.

‹ Previous Post (/one-hot-encoding-in-python-with-pandas-and-scikit-learn/)

Next Post › (/the-proxy-design-pattern-in-java/)

Follow Us

(https://twitter.com/StackAbuse) (https://www.facebook.com/stackabuse) (https://stackabuse.com/rss/)

Data Visualization in Python

(https://gum.co/data-visualization-in-python)

Ad

Want a remote job?

Recent Posts

How to Use the crontab Command in Unix (/how-to-use-the-crontab-command-in-unix/)

Introduction to Data Visualization in Python with Pandas (/introduction-to-data-visualization-in-python-with-pandas/)

Set Up Gated Checkin for Spring Boot Projects with Github and Jenkins (/set-up-gated-checkin-for-spring-boot-projects-with-github-and-jenkins/)

Tags

ai (/tag/ai/) algorithms (/tag/algorithms/) amqp (/tag/amqp/) angular (/tag/angular/) announcements (/tag/announcements/) apache (/tag/apache/) apache commons (/tag/apache-commons/) api (/tag/api/) arduino (/tag/arduino/) artificial intelligence (/tag/artificial-intelligence/)

Follow Us

(https://twitter.com/StackAbuse) (https://www.facebook.com/stackabuse) (https://stackabuse.com/rss/)