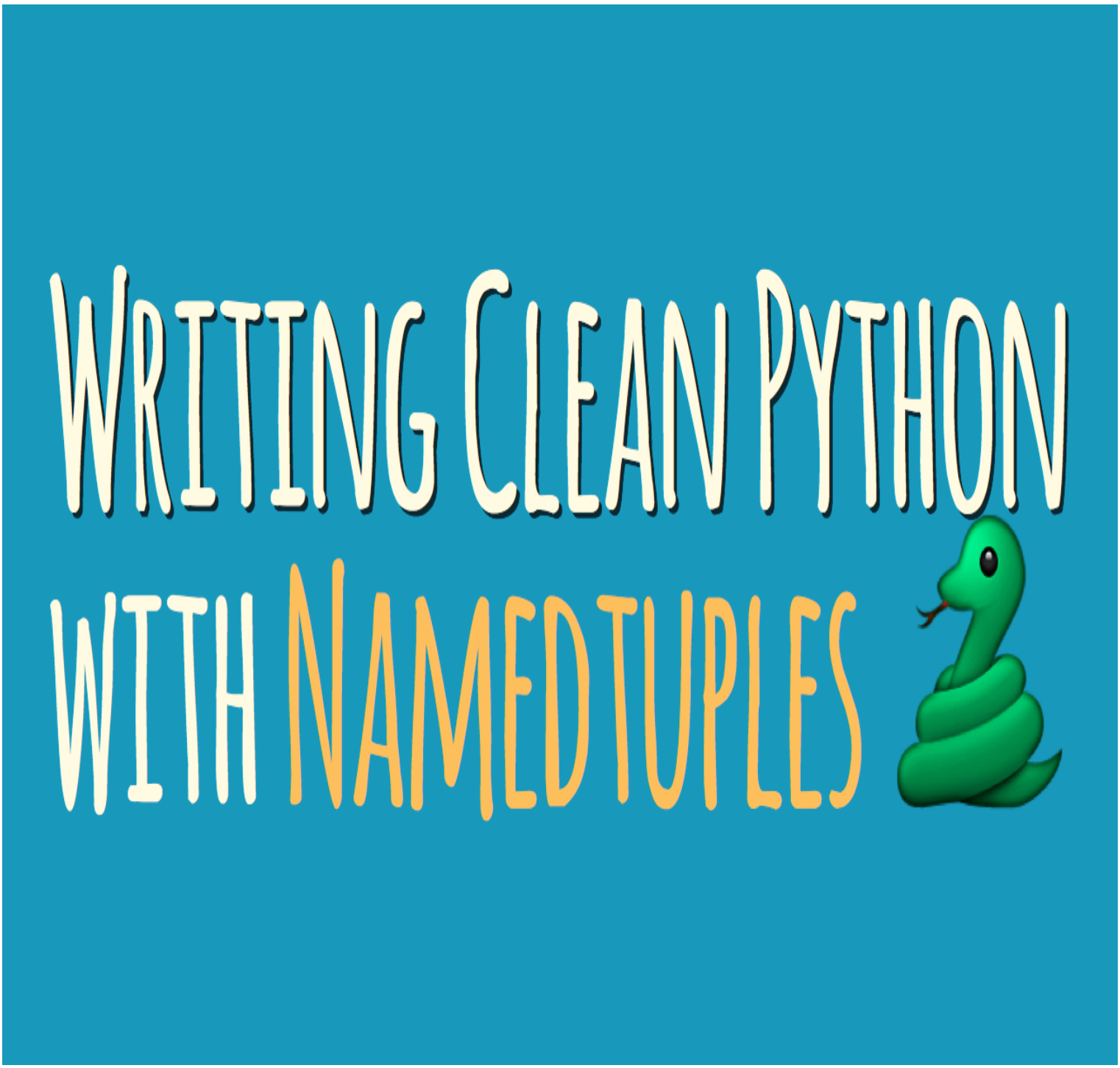
Writing Clean Python With Namedtuples

by Dan Bader — Get free updates of new posts here (https://www.getdrip.com/forms/80014959/submissions/new).

Python comes with a specialized “namedtuple” container type that doesn’t seem to get the attention it deserves. It’s one of these amazing features in Python that’s hidden in plain sight.



Namedtuples can be a great alternative to defining a class manually and they have some other interesting features that I want to introduce you to in this article.

Now, what’s a namedtuple and what makes it so special? A good way to think about namedtuples is to view them as an extension of the built-in tuple data type.

Python's tuples are a simple data structure for grouping arbitrary objects. Tuples are also immutable—they cannot be modified once they've been created.

```
>>> tup = ('hello', object(), 42)
>>> tup
('hello', <object object at 0x105e76b70>, 42)
>>> tup[2]
42
>>> tup[2] = 23
TypeError: "'tuple' object does not support item assignment"
```

A downside of plain tuples is that the data you store in them can only be pulled out by accessing it through integer indexes. You can't give names to individual properties stored in a tuple. This can impact code readability.

Also, a tuple is always an ad-hoc structure. It's hard to ensure that two tuples have the same number of fields and the same properties stored on them. This makes it easy to introduce "slip-of-the-mind" bugs by mixing up the field order.

Namedtuples to the Rescue

Namedtuples aim to solve these two problems.

First of all, namedtuples are immutable just like regular tuples. Once you store something in them you can't modify it.

Besides that, namedtuples are, well…*named tuples*. Each object stored in them can be accessed through a unique (human-readable) identifier. This frees you from having to remember integer indexes, or resorting to workarounds like defining integer constants as mnemonics for your indexes.

Here's what a namedtuple looks like:

```
>>> from collections import namedtuple
>>> Car = namedtuple('Car' , 'color mileage')
```

To use namedtuples you need to import the `collections` module. They were added to the standard library in Python 2.6. In the above example we defined a simple "Car" data type with two fields: "color" and "mileage".

You might find the syntax a little weird here—Why are we passing the fields as a string encoding them `"color mileage"`?

The answer is that namedtuple's factory function calls `split()` on the field names string, so this is really just a shorthand to say the following:

```
>>> 'color mileage'.split()
['color', 'mileage']
>>> Car = namedtuple('Car', ['color', 'mileage'])
```

Of course you can also pass a list with string field names directly if you prefer how that looks. The advantage of using a proper list is that it's easier to reformat this code if you need to split it across multiple lines:

```
>>> Car = namedtuple('Car', [
...     'color',
...     'mileage',
... ])
```

However you decide, you can now create new "car" objects with the `Car` factory function. It behaves as if you had defined a `Car` class manually and given it a constructor accepting a "color" and a "mileage" value:

```
>>> my_car = Car('red', 3812.4)
>>> my_car.color
'red'
>>> my_car.mileage
3812.4
```

Tuple unpacking and the *-operator for function argument unpacking (https://www.youtube.com/watch?v=YWY4BZi_o28) also work as expected:

```
>>> color, mileage = my_car
>>> print(color, mileage)
red 3812.4
>>> print(*my_car)
red 3812.4
```

Besides accessing the values stored in a namedtuple by their identifiers, you can still access them by their index. That way namedtuples can be used as a drop-in replacement for regular tuples:

```
>>> my_car[0]
'red'
>>> tuple(my_car)
('red', 3812.4)
```

You'll even get a nice string representation for free, which saves some typing and redundancy:

```
>>> my_car
Car(color='red' , mileage=3812.4)
```

Like tuples, namedtuples are immutable. When you try to overwrite one of their fields you'll get an `AttributeError` exception:

```
>>> my_car.color = 'blue'
AttributeError: "can't set attribute"
```

Namedtuple objects are implemented as regular Python classes internally. When it comes to memory usage they are also "better" than regular classes and just as memory efficient as regular tuples.

A good way to view them is to think that *namedtuples are a memory-efficient shortcut to defining an immutable class in Python manually*.

Subclassing Namedtuples

Because they are built on top of regular classes you can even add methods to a namedtuple's class. For example, you can extend the class like any other class and add methods and new properties to it that way. Here's an example:

```
>>> Car = namedtuple('Car', 'color mileage')
>>> class MyCarWithMethods(Car):
...     def hexcolor(self):
...         if self.color == 'red':
...            return '#ff0000'
...         else:
...             return '#000000'
```

We can now create `MyCarWithMethods` objects and call their `hexcolor()` method, just as expected:

```
>>> c = MyCarWithMethods('red', 1234)
>>> c.hexcolor()
'#ff0000'
```

However, this might be a little clunky. It might be worth doing if you want a class with immutable properties. But it's also easy to shoot yourself in the foot here.

For example, adding a new *immutable* field is tricky because of how namedtuples are structured internally. The easiest way to create hierarchies of namedtuples is to use the base tuple's `._fields` property:

```
>>> Car = namedtuple('Car', 'color mileage')
>>> ElectricCar = namedtuple(
...     'ElectricCar', Car._fields + ('charge',))
```

This gives the desired result:

```
>>> ElectricCar('red', 1234, 45.0)
ElectricCar(color='red', mileage=1234, charge=45.0)
```

Built-in Helper Methods

Besides the `_fields` property each namedtuple instance also provides a few more helper methods you might find useful. Their names all start with an underscore character (_) which usually signals that a method or property is "private" and not part of the stable public interface of a class or module.

With namedtuples the underscore naming convention has a different meaning though: These helper methods and properties *are* part of namedtuple's public interface. The helpers were named that way to avoid naming collisions with user-defined tuple fields. So go ahead and use them if you need them!

I want to show you a few scenarios where the namedtuple helper methods might come in handy. Let's start with the `_asdict()` helper. It returns the contents of a namedtuple as a dictionary:

```
>>> my_car._asdict()
OrderedDict([('color', 'red'), ('mileage', 3812.4)])
```

This is great for avoiding typos when generating JSON-output, for example:

```
>>> json.dumps(my_car._asdict())
'{"color": "red", "mileage": 3812.4}'
```

Another useful helper is the `_replace()` function. It creates a (shallow) copy of a tuple and allows you to selectively replace some of its fields:

```
>>> my_car._replace(color='blue')
Car(color='blue', mileage=3812.4)
```

Lastly, the `_make()` classmethod can be used to create new instances of a namedtuple from a sequence or iterable:

```
>>> Car._make(['red', 999])
Car(color='red', mileage=999)
```

When to Use Namedtuples

Namedtuples can be an easy way to clean up your code and to make it more readable by enforcing a better structure for your data.

I find, for example, that going from ad-hoc data types like dictionaries with a fixed format to namedtuples helps me express my intentions more clearly. Often when I attempt this refactoring I magically come up with a better solution for the problem I'm facing.

Using namedtuples over unstructured tuples and dicts can also make my coworkers' lives easier because they make the data being passed around "self-documenting" (to a degree).

On the other hand I try not to use namedtuples for their own sake if they don't help me write "cleaner", more readable, and more maintainable code. Too much of a good thing can be a bad thing.

However if you use them with care, namedtuples can undoubtedly make your Python code better and more expressive.

Things to Remember

- `collection.namedtuple` is a memory-efficient shortcut to defining an immutable class in Python manually.
- Namedtuples can help clean up your code by enforcing an easier to understand structure on your data.
- Namedtuples provide a few useful helper methods that all start with an _ underscore—but are part of the public interface. It's okay to use them.