✨ Member-only story

PYTHON PROGRAMMING

Python “Tuple+”: Named Tuples

Tuples are a powerful Python type — but named tuples even more so!

Marcin Kozak · Follow
Published in Towards Data Science · 25 min read · Jan 3, 2024

Named tuples join the strengths of names and tuples. Photo by Ainur Iman on Unsplash

The three most popular Python data types are the list, the dictionary, and the tuple. Lists and dictionaries are mutable, meaning that their elements can be

altered after creation. Tuples, on the other hand, are immutable, so they cannot be changed after creation. If you do need to modify the contents of a tuple, you must create a new instance with the desired changes and assign it to the same variable.

This article focuses on Python named tuples, a specialized type of tuple that combines the power of regular tuples with the added flexibility of named fields. Compared to regular tuples, named tuples can make code simpler, more readable and more maintainable — and even more Pythonic. However, you must be careful, as sometimes using named tuples excessively can inadvertently reduce code readability rather than enhance it.

Read on to learn more!

To understand named tuples, you have to first understand regular Python tuples. If you're not familiar with them, I strongly advise you to first read the following two articles about this data type:

Python Tuple, the Whole Truth, and Only the Truth: Hello, Tuple!
Learn the basics of tuples and of using them
towardsdatascience.com

Python Tuple, The Whole Truth and Only Truth: Let's Dig Deep
Learn the intricacies of tuples
towardsdatascience.com

What's fantastic about named tuples is that they function like regular tuples: everything that works for a regular tuple will work for a named tuple. But that's not all, as named tuples offer additional features — hence the moniker "tuple+". Therefore, I'll assume you're familiar with the key concepts covered in these two articles above, and we'll focus on the advantages of named tuples.

First and foremost, keep in mind that all tuples are immutable. You may find it easy to forget this crucial characteristic when you start adding methods to named tuple definitions. While this is possible, never forget that a named tuple is still a tuple — an immutable data container.

> Never forget that a named tuple is still a tuple — an immutable data container.

No, not every time you use a tuple you should use a named tuple instead. While the previous paragraph may have implied otherwise, the truth is that using named tuples in certain scenarios can not only fail to enhance the code over regular tuples, but can actually decrease its simplicity and readability.

What named tuples are

The term "named tuple" neatly encapsulates this data structure's essence: a tuple with named elements, or more formally, a tuple with named fields — or named attributes.

For medium-sized and long tuples, naming each element might seem pointless. What's the benefit of naming, say, the twenty-eighth or fifty-fifth element of a tuple with 100 fields?

However, for small tuples, the situation is different. When a tuple has two or three elements, naming each element can make sense. Perhaps the most

common scenario for using such small tuples in Python is function returns, like below:

```
def foo(*args, **kwargs) -> tuple[int, str]
    n: int
    st: str
    ...
    return n, st
```

Since this function returns a tuple, you can capture the output in the following ways:

```
output = foo()
output[0] # an integer
output[1] # a string
```

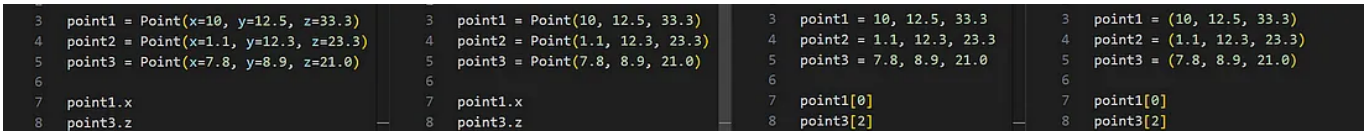
or you can use tuple unpacking:

```
n, st = foo()
# n is an integer
# st is a string
```

Now, imagine that `foo()` returns a tuple with named elements. You could access them like this:

```
output.n
output.st
```

Using named attributes isn't a new concept in Python — in fact, it's one of the things that make Python so readable a programming language. Enough to compare the below examples:



Readability of creation and access of named and regular tuples. Image by author

Starting from left:

- Named tuples with their full specification, meaning they're created with names. You can use names to access each field, but also you can use indexking, like in the case of regular tuples (not shown).
- You can generate an instance of a named tuple without employing attribute names.
- A regular tuple. You cannot use the `Point` name, which itself provides some information about what the tuple is about. You can access the elements using indexing.
- In the third panel, the tuples were constructed without parentheses. It's an issue of personal preference — I prefer omitting parentheses, line in the fourth panel. While sometimes parentheses can be helpful, I don't think they are in this particular example; they add only visual clutter and nothing else.

The above screenshot aimed to illustrate how both regular and named tuples can be used, but we haven't yet delved into how to generate named tuples or the additional features they offer — and yes, they offer much more than just field names. We'll cover all this in this article.

Two flavors of named tuples

Since Python 3.6, we have two flavors of named tuples:

- `collections.namedtuple` : A traditional, untyped named tuple.
- `typing.NamedTuple` : A typed version of `collections.namedtuple` , introduced in Python 3.6 with the addition of type hints.

Despite what I've said above — that the latter is a typed version of the former — typed named tuples offer more than just types compared to their untyped counterparts, although at a cost. One of the goals of this article is to uncover these distinctions and demonstrate when it's more appropriate to use the `collections` named tuple and when to use the `typing` one.

In what follows, we'll discuss these two types of named tuples using several

- `name` : a string; e.g., `"John Smith"`
- `year` : an integer representing a study year; e.g., `2`
- `gpa` : a float representing a grade point average; e.g. `3.43`

The second tuple will be similar in nature but will hold more comprehensive information about university employees, with the following attributes:

- `name` : a string; e.g., `"John Smith"`
- `dept` : a tuple of two strings representing the faculty department; e.g., `("Computer Science", "Data science")`
- `position` : a string representing the employee's position; e.g., `"assistant professor"`
- `started` : a `datetime` object representing the starting date of the employee's contract; e.g., `2020-08-01`

We'll start with the traditional named tuple type, that is, `collections.namedtuple` . Then we will move to `typing.NamedTuple` , which uses

the former.

collections.namedtuple

Definition

The biggest advantage of `collections.namedtuple` as compared to its `typing` counterpart is the simplicity of its definition:

```
from collections import namedtuple

Student = namedtuple("Student", "name year gpa")
```

You can achieve the same in the following ways:

```
Student = namedtuple("Student", ["name", "year", "gpa"])
Student = namedtuple("Student", "name, year, gpa")
```

We used a list in the former example, but it can be any sequence, such as a list, tuple or even a generator. Remember to use valid Python identifiers for attribute names, which is a typical rule for Python naming; there is an atypical exception, however— names cannot start with the underscore.

Creating an instance is simple:

```
student1 = Student(name="John Smith", year=2, gpa=3.32)
# or shorter, without names
student1 = Student("John Smith", 2, 3.32)
```

While ultimately using a string of names to define field names serves the same purpose as using an iterable of strings, the underlying processes may slightly differ, potentially leading to different performance. Hence, let's benchmark them using the following code (see this article to read about benchmarking using `timeit`):

```
from timeit import repeat

setup = "from collections import namedtuple"

n = 1_000_000

code1 = 'namedtuple("Student", "name year gpa")'
code2 = 'namedtuple("Student", ["name", "year", "gpa"])'

t1 = repeat(code1, setup=setup, number=n)
t2 = repeat(code2, setup=setup, number=n)

print(
    f"{code1}: {round(min(t1), 4)}", "\n",
    f"{code2}: {round(min(t2), 4)}"
)
```

I got the following results:

```
namedtuple("Student", "name year gpa"): 66.2871
namedtuple("Student", ["name", "year", "gpa"]): 69.3802
```

These are the times (in seconds) it takes to create a million named tuples using both ways. As you can see, creating a named tuple using a single string is slightly faster than using a list of strings. The difference is minor, however, and both methods are more or less equally readable. Personally, I prefer the readability of the single-string approach.

Let's create the employee named tuple:

```
Employee = namedtuple("Employee", "name dept position start")
```

This tuple is slightly more complex, but it doesn't fully reflect this complexity yet. The complication arises from the fact that `dept` is itself a tuple, something we haven't encountered yet but will encounter when creating an instance. To address this, let's first define a `Dept` named tuple:

```
Dept = namedtuple("Dept", "faculty department")
```

Now we're ready to create instances:

```
employee1 = Employee(
    "John Smart",
    Dept("Agriculture", "Agronomy"),
    "assistant professor",
    datetime(year=2020,month=9,day=1)
)
```

Let's see its representation[1]:

```
>>> employee1 # doctest: +NORMALIZE_WHITE_SPACE
Employee(
    name='John Smart',
    dept=Dept(faculty='Agriculture', department='Agronomy'),
    position='assistant professor',
    start=datetime.datetime(2020, 9, 1, 0, 0)
)
```

Arguments

Below, we'll discuss arguments you can use when creating a `collections.namedtuple`. Two of them are required while the others are optional — and far less frequently used.

`typename` and `field_names`

These are the two required arguments. We've already used them, when we created the `collections.namedtuple` types above. Let's return to the definition of `Dept`. We could also do this using keywords arguments:

```
Dept = namedtuple(
    typename="Dept",
    field_names="faculty department"
)
```

We have already covered the `field_names` argument. The `typename` argument is quite interesting. For someone using named tuples for the first time, a natural question to ask is, can I use a different identifier for the type and for `typename`?

Yes, you can:

```
>>> from collections import namedtuple
>>> XY = namedtuple("Point", "x y")
>>> XY
<class '__main__.Point'>
```

This works, but:

```
>>> Point
Traceback (most recent call last):
  ...
```

```
NameError: name 'Point' is not defined. Did you mean: 'print'?
>>> XY.__name__
'Point'
```

You don't encounter a different `typename` value from the variable's name very often, at least not in basic programs. However, in more sophisticated code, you might find yourself using it quite frequently. This can be the case, for instance, in metaprogramming.

Nevertheless, a typical approach to creating named tuples using `collections.namedtuple` is to use the same name for the type and the class, so:

```
>>> Point = namedtuple("Point", "x y")
```

This is a preferred approach because it keeps the code simple, avoiding potential confusion that can arise from using a different class name than the type name, as we did in this line:

```
>>> XY = namedtuple("Point", "x y")
```

We're done with the required arguments, that is, `typename` and `field_names`; the others are optional. Although not used very frequently, they increase the usability of `collections.namedtuple`.

`defaults`

We're starting with the most useful optional argument. It works similarly to how default argument values work in function and method definitions, that is, you provide default values to selected field names. What makes the two

ways different is that to provide default values of a named tuple's fields, you don't provide them together. So, instead of `x=10, y=20`, you would do this, as, for instance, `"x y", defaults=[10, 20]` or `["x", y"], defaults=[10, 20]`.

There's more to this method, however. The default value of `defaults` is `None`, meaning that, by default, the fields don't have any default values. If you want to use it, you need to provide a sequence for it.

What if you want to provide default names only for selected field(s), however? Unlike default argument values in function definition, which are assigned positionally, the `defaults` argument employs an "anti-positional" approach. (I invented this term to reflect the way default values are joined with the corresponding fields.) This means that the last default value is assigned to the last field, the second-to-last default value to the second-to-last field, and so on.

This anti-positional approach allows for selective assignment of default values. Not all fields need to have default values, and the order of fields in a named tuple definition can be arranged accordingly. By placing fields with default values on the right and those without defaults on the left, you can effectively utilize the anti-positional approach.

The below examples should explain how the `defaults` iterable works as an argument to `collections.namedtuple`:

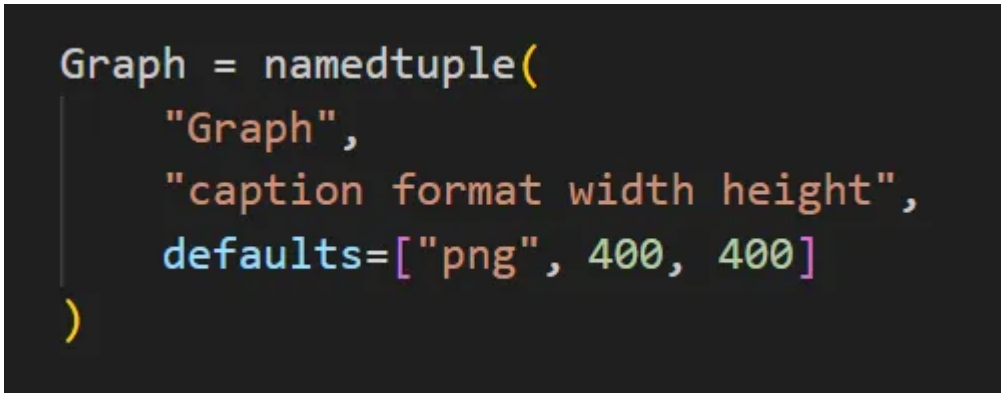
```
>>> Graph = namedtuple(
...     "Graph",
...     "format width height",
...     defaults=["png", 400, 400]
... )
>>> Graph()
Graph(format='png', width=400, height=400)
>>> Graph("tiff")
Graph(format='tiff', width=400, height=400)
>>> Graph("tiff", height=600)
Graph(format='tiff', width=400, height=600)
```

```
>>> Graph(width=600)
Graph(format='png', width=600, height=400)
```

Here, we have three fields, each with a default value:

- `format`, with the default of `"png"`
- `width`, with the default of `400`
- `height`, with the default of `400`

Now imagine we want also to provide a graph's caption, but without a default value. In that case, we must make `caption` the first field in the iterable with field names:



The anti-positional approach of assigning defaults in named tuples. caption has no default, format="png", width=400, height=400. Image by author

That way, `caption` did not get its default value, but we still can create a new instance using positional defaults:

```
>>> Graph = namedtuple(
...     "Graph",
...     "caption format width height",
...     defaults=["png", 400, 400]
... )
>>> Graph()
Traceback (most recent call last):
  ...
TypeError: Graph.__new__() missing 1 required positional argument: 'caption'
>>> Graph("Scatterplot")
Graph(caption='Scatterplot', format='png', width=400, height=400)
```

```
>>> Graph("Scatterplot", "tiff", height=600)
Graph(caption='Scatterplot', format='tiff', width=400, height=600)
>>> Graph(width=600, caption="Scatterplot")
Graph(caption='Scatterplot', format='png', width=600, height=400)
```

Remember that you shouldn't use mutable objects as default values. For one, this is because named tuples are meant to be immutable — although you *can* use mutable objects as tuple fields, and you can even change their values. Nonetheless, using mutable objects as defaults can lead to unexpected behavior. We'll talk about this some other day, as it's a more general rule and applies also to function definitions — but never forget this critical rule.

`rename`

A Boolean argument, `rename` set to `True` means that incorrect field identifiers used for field names will be replaced with positional names. Look:

```
>>> namedtuple("XYZ", "x y _z")
Traceback (most recent call last):
  ...
ValueError: Field names cannot start with an underscore: '_z'
>>> namedtuple("XYZ", "x y _z", rename=True)(1, 2, 3)
XYZ(x=1, y=2, _2=3)
>>> namedtuple("XYZ", "_x y _z", rename=True)(1, 2, 3)
XYZ(_0=1, y=2, _2=3)
```

As you can see, positional arguments translate to names like `_i`, where `i` represents the positional index of the corresponding field. So, you can't start identifiers with the underscore, but when `rename` is set to `True`, fields can start with the underscore — they just won't be used and will be renamed instead. The new names, however, will start with underscores.

Now, my two cents: Be cautious with `rename=True`. I prefer to use names I want, and since I'm choosing them myself, I usually don't see a point in using this option. This doesn't mean `rename=True` is completely useless. It can sometimes be helpful during metaprogramming, but I think that's a rare situation. If you do use it, note that using field names of the tuple created that way (with `rename=True`) would be risky, so it'd be better to use tuple indexing instead. And if so — if you can't use field names and have to use indexing — what's the point of using a named tuple at all?

So, think twice before using `rename=True` when creating a named tuple using `collections.namedtuple`. It's not something you'd typically use in everyday programming.

`module`

This is an advanced argument of `collections.namedtuple`, which you will seldom use. Its default value is `None`. As explained here, the `module` argument was added in Python 3.6, in order to make it possible for `namedtuple` to support pickling using different Python implementations. You can learn more in this GitHub issue.

Creating on the fly

There are two scenarios in which you may want to create a named tuple on the fly. The first and most obvious one is metaprogramming. You can for instance define a named tuple for a specific task based on external input, such as column names from a file from an external source when you don't know the structure of this file beforehand. The second scenario is when you want to create a named tuple only to add field names to a particular tuple, without actually creating a type to be reused later on. As you will see below, creating a `collections.namedtuple` on the fly is very simple.

Above, we defined two named tuple types: `Employee` and `Dept`. You can do the same thing on the fly, meaning you don't assign the named tuple type to a name but create, use, and discard it within the scope.

Let's illustrate this using another (simpler) definition of the `foo()` function:

```
XandY = namedtuple("XandY", "xi yi")

def foo(x, y) -> tuple[str, int]:
    xi: str
    yi: int
    ...
    return XandY(xi, yi)
```

This allows you to access the output tuple using the attribute names `xi` and `yi`. We created the `XandY` type specifically for this very purpose, so we don't need this tuple anywhere else — it's only used inside `foo()`. Do we really need the definition of the `XandY` type?

In such cases, we don't need to define `XandY`; instead, we can create it on the fly:

```
def foo(x, y) -> tuple[str, int]:
    xi: str
    yi: int
    ...
    return namedtuple("XandY", "xi yi")(xi, yi)
```

While this approach may be slightly less clear, it makes the code shorter and avoids creating a data structure solely for a single return statement. A good rule of thumb for creating a data type is to do so when:

- you need to use the type more than once,
- you want to export the type in order to enable the user us it, or
- you need this type for clarity reasons.

Here, our goal was simply to add names to the two fields of the tuple returned by the function. Neither of these two conditions applied, so creating the type on the fly seemed like a good choice.

Another alternative is to create the data type inside the function:

```
def foo(x, y) -> tuple[str, int]:
    xi: str
    yi: int
    ...
    XandY = namedtuple("XandY", "xi yi")
    return XandY(xi, yi)
```

Here, `XandY` is a temporary variable that refers to a `namedtuple` type, defined in the scope of the `foo()` function.

The choice between creating a named tuple on the fly or defining it inside the function depends on the complexity of the data structure: For simple structures, creating on the fly can indeed be the simplest method. As with many decisions in Python, this one is partially based on personal preference — but always pay attention to code clarity.

Adding functionality

You can add additional functionality to a named tuple created using `collections.namedtuple` in two ways, both of which are less readable than doing the same using `typing.NamedTuple` (as discussed later in the article).

The first way is inheritance:

```
class Point(namedtuple("Point", "x y")):
    def distance(self, other: "Point"):
        return (
            (self.x - other.x)**2
```

```
            + (self.y - other.y)**2
        )**.5
```

As you can see, our `Point` class inherits from the very particular `namedtuple` we want to create. Of course, you can name it first:

```
BasePoint = namedtuple("BasePoint", "x y")

class Point(BasePoint):
    def distance(self, other: "NewPoint"):
        return (
            (self.x - other.x)**2
            + (self.y - other.y)**2
        )**.5
```

This is how our `Point` named tuple works with the `.distance()` method:

```
>>> p1 = Point(1, 1)
>>> p2 = Point(2, 3)
>>> p1.distance(p2) == distance(p1, p2)
True
>>> p1.distance(p2)
2.23606797749979
```

The `.distance()` method works as it would in a regular Python class.

The above approach is recommended for defining methods for `collections.namedtuple`. There's also a nifty trick I'd like to share, but please don't use it in actual code. It's just a clever trick I found in Luciano Ramalho's *Fluent Python. 2nd edition* book. Knowing such tricks can broaden your Python knowledge — in this case, however, don't confuse knowing with using.

While it works similarly to the above method, the code might look a bit strange:

```
Point = namedtuple("Point", "x y")

def distance(point: Point, other: Point):
    return (
        (point.x - other.x)**2
        + (point.y - other.y)**2
    )**.5

Point.distance = distance
```

And this is it! This code will work the very same way as above, as what you got is the same named tuple as before, just defined in a different way.

By the way, this trick is not limited to named tuples. You can use it to use this method to add a method to a regular Python class. Remember only that the first argument of the function to be converted into a class method needs to represent the instance (`self`). So, after converting the `distance()` function into a `Point` method, the `point` argument becomes the instance, that is, the `self` argument.

typing.NamedTuple

Definition

`NamedTuple` from the `typing` module is a very similar data structure to `collections.namedtuple` . In fact, if you look at the implementation of the former, you will see that it directly calls the latter:

```
def _make_nmtuple(name, types, module, defaults = ()):
    fields = [n for n, t in types]
    types = {n: _type_check(t, f"field {n} annotation must be a type")
             for n, t in types}
    nm_tpl = collections.namedtuple(name, fields,
                                    defaults=defaults, module=module)
    nm_tpl.__annotations__ = nm_tpl.__new__.__annotations__ = types
    return nm_tpl
```

A VS Code screenshot from typing.py module in Python 3.11: A function creating a typing.NamedTuple. Image by author

Why do we need another named tuple structure, then? You already know the answer: `collections.namedtuple` does not enable one to use type hints while `typing.NamedTuple` does. As you will see soon, however, there are more differences between the two.

Let's define the corresponding named tuples to those we created above:

```
from typing import NamedTuple
from datetime import datetime

class Student(NamedTuple):
    name: str
    year: int
    gpa: float

class Dept(NamedTuple):
    faculty: str
    department: str

class Employee(NamedTuple):
    name: str
    dept: Dept
    position: str
    start: datetime
```

In my opinion, these definitions do show us that type hints can be very helpful in understanding the contents of these data containers. One might think that this `class`-based definition requires more lines, and that's entirely true. Nevertheless, those additional lines do come with increased readability.

Instance creation remains identical to the `collections.namedtuple` approach, so we won't repeat that code.

Using default values

Using default values is much simpler than in the case of `collections.namedtuple`:

```
class Graph(NamedTuple):
    format: str = "png"
    width: int = 400
    height: int = 400
```

Again, remember you have to define fields without default values first and those with defaults last:

```
class Graph(NamedTuple):
    caption: str
    format: str = "png"
    width: int = 400
    height: int = 400
```

As was the case with `collections.namedtuple`, you should *not* use mutable objects as default values.

Creating on the fly

Unfortunately, `typing.NamedTuple` doesn't offer a method for creating named tuples on the fly. In this regard, `collections.namedtuple` emerges as the clear winner.

Adding functionality

`typing.NamedTuple` is an improved version of named tuples not only in terms of type hints — but also class definition. You define a new named tuple as a custom class that inherits from `typing.NamedTuple`.

This may look quite similar to the inheritance approach we have taken above to add methods to a named tuple created using `collections.namedtuple`, but there is a significant difference. Before, we inherited from an already-created named-tuple class while in the case of `typing.NamedTuple`, we inherit from this very class.

When defining a `typing.NamedTuple` named tuple, you can define regular class, static and instance methods. Look:

```
from typing import NamedTuple

class Point(NamedTuple):
    x: float
    y: float

    def distance(self, other: Point) -> float:
        return (
            (self.x - other.x)**2
            + (self.y - other.y)**2
        )**.5
```

You can use this class in the very same way as the one created before, using `collections.namedtuple`:

```
>>> p1 = Point(1, 1)
>>> p2 = Point(2, 3)
>>> p1.distance(p2) == distance(p1, p2)
True
>>> round(p1.distance(p2), 3)
2.236
```

Performance

Since `typing.NamedTuple` directly calls `collections.namedtuple`, defining a named tuple using the former class should take more time than defining a named tuple using the latter class. The question is how much more.

To study this, I ran benchmarks based on `timeit` experiments. In each run, I created the three named tuples (`Student`, `Dept` and `Employee`) and compared the execution time of `100_000` runs of this. This is the code:

```
from timeit import repeat

setup1 = "from collections import namedtuple"
setup2 = "from typing import NamedTuple"

n = 100_000

code1 = '''
Student = namedtuple("Student", "name age gpa")
Dept = namedtuple("Dept", "faculty department")
Employee = namedtuple("Employee", "name dept position start")
'''

code2 = '''
from typing import NamedTuple
from datetime import datetime

class Student(NamedTuple):
    name: str
    year: int
    gpa: float

class Dept(NamedTuple):
    faculty: str
    department: str

class Employee(NamedTuple):
    name: str
    dept: Dept
    position: str
    start: datetime
'''

t1 = repeat(code1, setup=setup1, number=n)
t2 = repeat(code2, setup=setup2, number=n)

print(
    "\n",
    f"code1: {round(min(t1), 2)}",
    "\n",
    f"code2: {round(min(t2), 2)}"
)
```

As expected, named tuples from the `typing` module showed significantly slower performance (`28.72` sec against `16.28` sec), meaning that they needed almost two times more to create the three named tuples as shown in the code above than the regular named tuples.

As for memory usage, instances of both named tuples use the very same memory. In our case, the following instance needs `440` bytes:

```
Employee(
    "John Smart",
    Dept("Agriculture", "Agronomy"),
    "assistant professor",
    datetime(year=2020,month=9,day=1)
)
```

You can measure this using the `pympler` package, with the help of the `pympler.asizeof.asizeof()` function:

```
>>> from pympler.asizeof import asizeof
>>> john = Employee(
...     "John Smart",
...     Dept("Agriculture", "Agronomy"),
...     "assistant professor",
...     datetime(year=2020,month=9,day=1)
... )
>>> asizeof(john)
440
```

By the way, a regular tuple:

```
(
    "John Smart",
    ("Agriculture", "Agronomy"),
    "assistant professor",
```

```
    datetime(year=2020,month=9,day=1)
)
```

needs the very same `440` bytes of memory!

The corresponding dictionary, however, obtained using the `_asdict()` method (we'll discuss this and other named tuple methods below) of either named tuple, uses `784` bytes:

```
>>> john_dict = john._asdict()
>>> john_dict # doctest: +NORMALIZE_WHITESPACE
{'name': 'John Smart',
 'dept': Dept(faculty='Agriculture',
              department='Agronomy'),
 'position': 'assistant professor',
 'start': datetime.datetime(2020, 9, 1, 0, 0)}
>>> asizeof(john_dict)
784
```

This shows that tuples are cheaper than the corresponding dictionaries, at least in terms of memory use.

Named tuple methods

This section describes the methods that both sorts of named tuples offers. That is, when you create a named tuple type (class), it offers a method you can use, but also instances created from this class will have its methods. All these methods are the same for both `collections.namedtuple` and `typing.NamedTuple`, so in examples, I will only use the former.

Create an instance using _make()

You can use the `_make()` method to create a new instance using an iterable. This method will work on both the type itself:

```
>>> Point = namedtuple("Point", "x y")
>>> values = [1.1, 2.2]
>>> point = Point._make(values)
>>> point
Point(x=1.1, y=2.2)
```

and its instances:

```
>>> point2 = point._make([3.3, 4.4])
>>> point2
Point(x=3.3, y=4.4)
```

The method does not really offer new functionality, only an alternative way of creating instances using an iterable. You can achieve the same using the typical constructor, using iterable unpacking:

```
>>> Point(*[3.3, 4.4])
Point(x=3.3, y=4.4)
```

You can also use a dictionary:

```
>>> Point(**{'x': 3.3, 'y': 4.4})
Point(x=3.3, y=4.4)
```

Convert a named tuple to a dictionary using _asdict()

This method will work only for named tuple instances, as — as the name suggests — it creates the corresponding dictionary from a named tuple:

```
>>> point._asdict()
{'x': 1.1, 'y': 2.2}
```

The only comment this needs is that the order of fields in the resulting dictionary will be the same as in the original named tuple.

Replace fields with _replace()

This is yet another way of creating a new instance from an existing instance (not the type itself). You can use it if you want to change only some values of an existing instance:

```
>>> point._replace(y=6.6)
Point(x=1.1, y=6.6)
```

Remember that the resulting named tuple is a brand new instance, with some of the fields the same and the others changed; above, `y` was changed.

You can, however, use `_replace()` to change values of all fields:

```
>>> point._replace(x=5.5, y=6.6)
Point(x=5.5, y=6.6)
```

The order of fields in the call to `_replace()` does not matter:

```
>>> point._replace(y=6.6, x=5.5)
Point(x=5.5, y=6.6)
```

It may be tempting to think that since `_replace()` only changes one or more field values, it should be faster than the regular constructor. You couldn't be more wrong! Let's benchmark the two methods of creating new instances:

```
from timeit import repeat

setup = """
from collections import namedtuple
Point = namedtuple("Point", "x y")
point = Point(x=1.1, y=2.2)
"""

n = 1_000_000

code1 = "Point(x=1.1, y=3.3)"
code2 = "point._replace(b=11)"

t1 = repeat(code1, setup=setup, number=n)
t2 = repeat(code2, setup=setup, number=n)

print(
    f"{code1}: {round(min(t1), 4)}", "\n",
    f"{code2}: {round(min(t2), 4)}"
)
```

On my machine (Windows 11, WSL 1, 4 physical and 8 logical cores), I got the following results:

```
Point(x=1.1, y=3.3): 0.2965
point._replace(y=3.3): 0.539
```

As you see, the regular constructor was significantly — almost twice — faster. You would get similar results when changing only one field from a named tuple with many fields.

Disassembling both calls confirms the above observation:

```
>>> import dis
>>> dis.dis("Point(x=1.1, y=3.3)")
  0           0 RESUME                   0
<BLANKLINE>
  1           2 PUSH_NULL
              4 LOAD_NAME                0 (Point)
              6 LOAD_CONST               0 (1.1)
              8 LOAD_CONST               1 (3.3)
             10 KW_NAMES                 2
             12 PRECALL                  2
             16 CALL                     2
             26 RETURN_VALUE
>>> dis.dis("point._replace(y=3.3)")
  0           0 RESUME                   0
<BLANKLINE>
  1           2 LOAD_NAME                0 (point)
              4 LOAD_METHOD              1 (_replace)
             26 LOAD_CONST               0 (3.3)
             28 KW_NAMES                 1
             30 PRECALL                  1
             34 CALL                     1
             44 RETURN_VALUE
```

On the one hand, the latter bytecode is shorter, but this is not what makes the difference. The first disassembly shows that in the first call, the interpreter *directly* invokes the `Point` constructor. The fields gets values as constants, which is a relatively straightforward and efficient approach.

On the other hand, to use the `_replace()` method, the interpreter first locates the object in memory and then invokes the method on the object, passing the provided keyword arguments. This process involves additional steps compared to directly creating a new object. Creating a copy of an existing instance is a more complex operation than creating a new instance from scratch.

I'd say, then — don't overuse the `_replace()` method.

Named tuple methods: public or private?

There's one thing with named tuples I consider strange and even non-Pythonic.

```
>>> from collections import namedtuple
>>> Example = namedtuple("Example", "x y")
>>> [attr for attr in dir(Example) if not attr.startswith("__")]
['_asdict', '_field_defaults', '_fields', '_fields_defaults', '_make',
 '_replace', 'count', 'index', 'x', 'y']
>>>
```

Named tuple attributes. Screenshot from Python session. Image by author.

According to the standard rules of Python coding style, methods starting with the underscore — for named tuples these are `_asdict()`, `_make()` and `_replace()` — should be considered private. Although private methods are technically *not* protected, it's generally considered best practice to avoid using them. This is a general rule — but there's at least one exception: named tuples.

While these "private" namedtuple methods are not meant to remain hidden altogether, they are actually intended to be used as part of the standard API of named tuples: they are *not* private at all. So, what's the deal with their underscore-prefixed names? Why do these public methods seem to imply that they're private?

Only one plausible explanation comes to my mind. By having these methods start with underscores, we're free to use the following field names in our namedtuples: `asdict`, `make`, and `replace` (not that these seem to be typical field names in named tuples). As we've already mentioned, field names cannot start with an underscore. These two things seem to be interconnected, or at least strongly correlated: don't start field names with an underscore, and the built-in namedtuple methods start with an underscore.

Even though this seems to make sense, at least within the context of the above paragraph, it's still quite atypical for Python, even unidiomatic. I can't recall any other standard-library class with this kind of inconsistency (though my memory could be failing me).

Sure, external libraries might get away with inconsistencies with the standards for Python coding. I myself did this not once. For example, in the `perftester` package, I used function arguments `Number` and `Repeat` instead of `number` and `repeat`, in order to allow custom functions to take arguments named `number` and `repeat`. In the `tracemem` package, I made a more unconventional decision: I added `tracemem` objects to the `builtins` globals. (For more information on this topic, refer to this article on Python globals and this one on `tracemem`.)

Nevertheless, the standard library should be stricter than external packages. Its code should be good and idiomatic. Reality, unfortunately, is a different story and we do know that the standard library does not offer exemplary Python code, free of the tiniest mistakes.

Minor errors are minor errors, but idiomatic code is still idiomatic code. I believe — and hope to be right in this! — that this theoretically unidiomatic naming convention used for named tuple methods aims to disseminate an important message, hidden between the lines:

> *Python has its idioms and its good style. Strive for using idiomatic and stylistic language. At the same time, using theoretically unidiomatic code when there are legitimate reasons for doing so should* not *be considered unidiomatic unless there are better or simpler ways to achieve the same effect.*

If I'm right then maybe the two unidiomatic decisions I mentioned earlier also could be considered idiomatic?

Conclusion

Named tuples. When I think of them, the first thing that comes to my mind is the tuple. And this is a correct connotation — because named tuples are above all tuples.

But definitely, they are not only tuples — named tuples are also named. This is the second connotation coming to my mind. And this is these names that

make the difference.

This difference can be considered at two levels. First, named tuples differ from tuples because their fields are named and so work as named attributes. Second, names make a difference in terms of usefulness: regular tuples are powerful, but named tuples can be even more so.

Named tuples aren't only milk and honey, though. Yes, they are useful thanks to named fields, but not always will such names be useful. At some point, they can add confusion rather than help. If a tuple has two or three, maybe even five fields — yes, naming them can be super helpful. But what about tuples with ten elements? Or twenty? A hundred?

Imagine a definition of such a big named tuple. This will definitely make the code longer, unless you will use a trick I mentioned to use an iterable to create field names. In some scenarios, it still can make sense, but is there any difference between `my_tuple[27]` and `my_tuple.i27`? I don't think so — at the very least, there is no advantage in using the latter over the former. Indexing is simpler to automatize, however, so in fact, it can be a better choice than names for big containers. (Of course, you can use indexing also for named tuples.) This doesn't mean that it's never useful to create big named tuples — but if you have an idea of doing so, think twice.

Let's summarize the comparison of both flavors of named tuples:

Feature/Aspect	collections.namedtuple	typing.NamedTuple
Syntax	More concise. Uses a factory function.	More explicit. Uses the standard class syntax.
Mutability	Immutable.	Immutable.
Creating on the fly	Supported.	Not supported.
Default values	Supports default values, but in a little less readable way.	Supports default values in a simple and readable way.
Inheritance	Limited: you can inherit from a namedtuple type, not from namedtuple itself.	Supports inheritance.
Type hints	No built-in support for type hints.	Built-in support for type hints.
Adding functionality	Limited, by adding methods to the child class inheriting from a particular namedtuple class.	Supports implementing methods.
Use cases	Lightweight data containers.	More feature-rich data containers.
Readability	Concise and easy to read for simple cases.	Explicit class syntax may enhance readability.

Comparison of collections.namedtuple and typing.NamedTuple. A screenshot from Google Sheets. Image by author

That's a lot to consider. Say you want to use a named tuple in Python, and so you need to choose between `collections.namedtuple` and `typing.NamedTuple`. Which one should you choose?

Certainly, this should depend on the specific requirements of the project and your and — maybe even more importantly — your team's coding style preferences. Other than that, consider the following thoughts when making your choice:

Use `collections.namedtuple` when at least one of the following point holds:

- You need a simple, lightweight, and immutable data structure without the need advanced features; when simplicity comes to play, `collections.namedtuple` is easier to use and more concise than `typing.NamedTuple`.
- The code can do without type hints.
- You need to create the type on the fly.
- You have to ensure backward compatibility before Python 3.5.

For `typing.NamedTuple`, the list will be shorter — but the truth is, it is `typing.NamedTuple` that should be preferred in most typical situations. Use them when:

- You need to use type hints or to add additional functionality, by implementing methods.
- You need explicit and readable code, especially when you use default values in named tuple definition.

> Never forget that named tuples are immutable.

Never forget that named tuples are immutable. It might seem like a simple thing to remember, but I've caught myself forgetting about that a few times when I started implementing custom methods for my named tuples. It's just so tempting to modify the instance, like in a regular Python class. Hence keep in mind: a named tuple is still a tuple, and any methods you want to implement should return something, not modify the instance. The latter is impossible, precisely because of immutability.

> A named tuple is still a tuple, and any methods you want to implement should return something, not modify the instance.

While you can use mutable objects as tuple elements, you *can't use mutable default values*. In fact, avoid using mutable objects in named tuples altogether. Isn't it a bit illogical to use a mutable object in an immutable container?

Therefore, when you need a mutable data container, named tuples are *not* your choice. You can consider using a classical Python class or a data class (via `dataclasses.dataclass`), available in Python 3.7+.

> Avoid using mutable objects in named tuples altogether. Isn't it a bit illogical to use a mutable object in an immutable container?

I hope I succeeded to convince you that named tuples offer a powerful Python data structure. Myself, I use them mainly to create small data types, consisting of several fields. It's a memory- and time-efficient data structure that is simpler to use than regular tuples, thanks to one of the biggest Python advantages: named attributes of its objects.

Footnotes

[1] In many code blocks, I use `doctest` testing, in order to assure that the examples are up-to-date and work correctly. You can read more about `doctest` in its documentation and in the following article:

Python Documentation Testing with doctest: The Easy Way

doctest allows for documentation, unit and integration testing, and test-driven development.

towardsdatascience.com

Thank you for reading. If you enjoyed this article, you may also enjoy other articles I wrote; you will see them here. If you want to join Medium, please use my referral link below:

Join Medium with my referral link - Marcin Kozak

As a Medium member, a portion of your membership fee goes to writers you read, and you get full access to every story...

medium.com

Python Programming Tuples Data Type Hands On Tutorials