

Write Pythonic and Clean Code With namedtuple

by [Leodanis Pozo Ramos](#) ⌚ May 12, 2021 💬 12 Comments 🏷️ [data-structures](#) [intermediate](#) [python](#)

Mark as Completed

🔖

🔗 Share

📱 Share

✉️ Email

Table of Contents

- [Using namedtuple to Write Pythonic Code](#)
- [Creating Tuple-Like Classes With namedtuple\(\)](#)
 - [Providing Required Arguments to namedtuple\(\)](#)
 - [Using Optional Arguments With namedtuple\(\)](#)
- [Exploring Additional Features of namedtuple Classes](#)
 - [Creating namedtuple Instances From Iterables](#)
 - [Converting namedtuple Instances Into Dictionaries](#)
 - [Replacing Fields in Existing namedtuple Instances](#)
 - [Exploring Additional namedtuple Attributes](#)
- [Writing Pythonic Code With namedtuple](#)
 - [Using Field Names Instead of Indices](#)
 - [Returning Multiple Named Values From Functions](#)
 - [Reducing the Number of Arguments to Functions](#)
 - [Reading Tabular Data From Files and Databases](#)
- [Using namedtuple vs Other Data Structures](#)
 - [namedtuple vs Dictionary](#)
 - [namedtuple vs Data Class](#)
 - [namedtuple vs typing.NamedTuple](#)
- [Subclassing namedtuple Classes](#)
- [Measuring Creation Time: tuple vs namedtuple](#)
- [Conclusion](#)

Gain real-time insights into your Python application's health and performance

[Learn more](#)[Remove ads](#)[Watch Now](#)

This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Writing Clean, Pythonic Code With namedtuple](#)

Python's [collections](#) module provides a [factory function](#) called `namedtuple()`, which is specially designed to make your code more **Pythonic** when you're working with tuples. With `namedtuple()`, you can create [immutable](#) sequence types that allow you to access their values using descriptive field names and the **dot notation** instead of unclear integer indices.

If you have some experience using Python, then you know that writing Pythonic code is a core skill for Python developers. In this tutorial, you'll level up that skill using `namedtuple`.

In this tutorial, you'll learn how to:

- Create `namedtuple` classes using `namedtuple()`
- Identify and take advantage of **cool features** of `namedtuple`
- Use `namedtuple` instances to write **Pythonic code**
- Decide whether to use a `namedtuple` or a **similar data structure**
- **Subclass** a `namedtuple` to provide new features

To get the most out of this tutorial, you need to have a general understanding of Python's philosophy related to [writing Pythonic and readable code](#). You also need to know the basics of working with:

- [Tuples](#)
- [Dictionaries](#)
- [Classes](#) and [object-oriented programming](#)
- [Data classes](#)
- [Type hints](#)

If you don't have all the required knowledge before starting this tutorial, then that's okay! You can stop and review the above resources as needed.

Free Bonus: [Click here to get a Python Cheat Sheet](#) and learn the basics of Python 3, like working with data types, dictionaries, lists, and Python functions.

Using namedtuple to Write Pythonic Code

Python's `namedtuple()` is a [factory function](#) available in [collections](#). It allows you to create `tuple` subclasses with **named fields**. You can access the values in a given named tuple using the **dot notation** and the field names, like in `obj.attr`.

Python's `namedtuple` was created to improve code readability by providing a way to access values using descriptive field names instead of integer indices, which most of the time don't provide any context on what the values are. This feature also makes the code cleaner and more maintainable.

In contrast, using indices to values in a regular tuple can be annoying, difficult to read, and error-prone. This is especially true if the tuple has a lot of fields and is constructed far away from where you're using it.

Note: In this tutorial, you'll find different terms used to refer to Python's `namedtuple`, its factory function, and its instances.

To avoid confusion, here's a summary of how each term is used throughout the tutorial:

Term	Meaning
<code>namedtuple()</code>	The factory function
<code>namedtuple</code> , <code>namedtuple</code> class	The tuple subclass returned by <code>namedtuple()</code>
<code>namedtuple</code> instance, <code>named tuple</code>	An instance of a specific <code>namedtuple</code> class

You'll find these terms used with their corresponding meaning throughout the tutorial.

Besides this main feature of named tuples, you'll find out that they:

- Are **immutable** data structures
- Have a consistent [hash](#) value
- Can work as **dictionary keys**
- Can be stored in [sets](#)
- Have a helpful [docstring](#) based on the type and field names
- Provide a helpful **string representation** that prints the tuple content in a `name=value` format
- Support **indexing**
- Provide additional methods and attributes, such as `._make()`, `._asdict()`, `._fields`, and so on
- Are **backward compatible** with regular tuples
- Have **similar memory consumption** to regular tuples

In general, you can use `namedtuple` instances wherever you need a tuple-like object. Named tuples have the advantage that they provide a way to access their values using field names and the dot notation. This will make your code more Pythonic.

With this brief introduction to `namedtuple` and its general features, you can dive deeper into creating and using them in your code.



Master Real-World Python Skills
With a Community of Experts

Level Up With Unlimited Access to Our Vast Library
of Python Tutorials and Video Lessons

Watch Now »

 [Remove ads](#)

Creating Tuple-Like Classes With `namedtuple()`

You use a `namedtuple()` to create an [immutable](#) and tuple-like data structure with field names. A popular example that you'll find in tutorials about `namedtuple` is to create a class to represent a mathematical [point](#).

Depending on the problem, you probably want to use an immutable data structure to represent a given point. Here's how you can create a two-dimensional point using a regular tuple:

Python



```
>>> # Create a 2D point as a tuple
>>> point = (2, 4)
>>> point
(2, 4)

>>> # Access coordinate x
>>> point[0]
2
>>> # Access coordinate y
>>> point[1]
4

>>> # Try to update a coordinate value
>>> point[0] = 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Here, you create an immutable two-dimensional point using a regular tuple. This code works: You have a point with two coordinates, and you can't modify any of those coordinates. However, is this code readable? Can you tell up front what the 0 and 1 indices mean? To prevent these ambiguities, you can use a namedtuple like this:

Python



```
>>> from collections import namedtuple

>>> # Create a namedtuple type, Point
>>> Point = namedtuple("Point", "x y")
>>> isinstance(Point, tuple)
True

>>> # Instantiate the new type
>>> point = Point(2, 4)
>>> point
Point(x=2, y=4)

>>> # Dot notation to access coordinates
>>> point.x
2
>>> point.y
4

>>> # Indexing to access coordinates
>>> point[0]
2
>>> point[1]
4

>>> # Named tuples are immutable
>>> point.x = 100
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

Now you have a point with two appropriately named fields, x and y. Your point provides a user-friendly and descriptive string representation (Point(x=2, y=4)) by default. It allows you to access the coordinates using the dot notation, which is convenient, readable, and explicit. You can also use indices to access the value of each coordinate.

Note: It's important to note that, while tuples and named tuples are [immutable](#), the values they store don't necessarily have to be immutable.

It's totally legal to create a tuple or a named tuple that holds mutable values:

Python



```
>>> from collections import namedtuple

>>> Person = namedtuple("Person", "name children")
>>> john = Person("John Doe", ["Timmy", "Jimmy"])
>>> john
Person(name='John Doe', children=['Timmy', 'Jimmy'])
>>> id(john.children)
139695902374144

>>> john.children.append("Tina")
>>> john
Person(name='John Doe', children=['Timmy', 'Jimmy', 'Tina'])
>>> id(john.children)
139695902374144

>>> hash(john)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

You can create named tuples that contain mutable objects. You can modify the mutable objects in the underlying tuple. However, this doesn't mean that you're modifying the tuple itself. The tuple will continue holding the same memory references.

Finally, tuples or named tuples with mutable values aren't [hashable](#), as you saw in the above example.

Finally, since `namedtuple` classes are subclasses of `tuple`, they're immutable as well. So if you try to change a value of a coordinate, then you'll get an `AttributeError`.

Providing Required Arguments to `namedtuple()`

As you learned before, `namedtuple()` is a factory function rather than a typical data structure. To create a new `namedtuple`, you need to provide two positional arguments to the function:

1. **typename** provides the class name for the `namedtuple` returned by `namedtuple()`. You need to pass a string with a [valid Python identifier](#) to this argument.
2. **field_names** provides the field names that you'll use to access the values in the tuple. You can provide the field names using:
 - An [iterable](#) of strings, such as `["field1", "field2", ..., "fieldN"]`
 - A string with each field name separated by whitespace, such as `"field1 field2 ... fieldN"`
 - A string with each field name separated by commas, such as `"field1, field2, ..., fieldN"`

To illustrate how to provide `field_names`, here are different ways to create points:

Python



```
>>> from collections import namedtuple

>>> # A list of strings for the field names
>>> Point = namedtuple("Point", ["x", "y"])
>>> Point
<class '__main__.Point'>
>>> Point(2, 4)
Point(x=2, y=4)

>>> # A string with comma-separated field names
>>> Point = namedtuple("Point", "x, y")
>>> Point
<class '__main__.Point'>
>>> Point(4, 8)
Point(x=4, y=8)

>>> # A generator expression for the field names
>>> Point = namedtuple("Point", (field for field in "xy"))
>>> Point
<class '__main__.Point'>
>>> Point(8, 16)
Point(x=8, y=16)
```

In these examples, you first create `Point` using a [list](#) of field names. Then you use a string with comma-separated field names. Finally, you use a generator expression. This last option might look like overkill in this example. However, it's intended to illustrate the flexibility of the process.

Note: If you use an iterable to provide the field names, then you should use a sequence-like iterable because the order of the fields is important to produce reliable results.

Using a set, for example, would work but could produce unexpected results:

Python



```
>>> from collections import namedtuple

>>> Point = namedtuple("Point", {"x", "y"})
>>> Point(2, 4)
Point(y=2, x=4)
```

When you use an unordered iterable to provide the fields to a `namedtuple`, you can get unexpected results. In the above example, the coordinate names are swapped, which might not be right for your use case.

You can use any valid Python identifier for the field names, except for:

- Names starting with an underscore (`_`)
- Python [keywords](#)

If you provide field names that violate either of these conditions, then you get a `ValueError`:

Python



```
>>> from collections import namedtuple

>>> Point = namedtuple("Point", ["x", "_y"])
Traceback (most recent call last):
...
ValueError: Field names cannot start with an underscore: '_y'
```

In this example, the second field name starts with an underscore, so you get a `ValueError` telling you that field names can't start with that character. This is intended to avoid name conflicts with the `namedtuple` methods and attributes.

In the case of `typename`, a question can arise when you look at the examples above: Why do I need to provide the `typename` argument? The answer is that you need a name for the class returned by `namedtuple()`. This is like creating an alias for an existing class:

Python



```
>>> from collections import namedtuple

>>> Point1 = namedtuple("Point", "x y")
>>> Point1
<class '__main__.Point'>

>>> class Point:
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y
...

>>> Point2 = Point
>>> Point2
<class '__main__.Point'>
```

In the first example, you create `Point` using `namedtuple()`. Then you assign this new type to the [global variable](#) `Point1`. In the second example, you create a regular Python class also named `Point`, and then you assign the class to `Point2`. In both cases, the class name is `Point`. `Point1` and `Point2` are aliases for the class at hand.

Finally, you can also create a named tuple using keyword arguments or providing an existing dictionary, like this:

Python



```
>>> from collections import namedtuple

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

>>> Point(x=2, y=4)
Point(x=2, y=4)

>>> Point(**{"x": 4, "y": 8})
Point(x=4, y=8)
```

In the first example, you use keyword arguments to create a `Point` object. In the second example, you use a dictionary whose keys match the fields of `Point`. In this case, you need to perform a **dictionary unpacking**.

[Learn Python »](#)[Remove ads](#)

Using Optional Arguments With `namedtuple()`

Besides the two required arguments, the `namedtuple()` factory function also takes the following optional arguments:

- `rename`
- `defaults`
- `module`

If you set `rename` to `True`, then all the invalid field names are automatically replaced with positional names.

Say your company has an old database application, written in Python, to manage the data about the passengers that travel with the company. You're asked to update the system, and you start creating named tuples to store the data you read from the database.

The application provides a function called `get_column_names()` that returns a list of strings with the column names, and you think you can use that function to create a `namedtuple` class. You end up with the following code:

Python

```
# passenger.py

from collections import namedtuple

from database import get_column_names

Passenger = namedtuple("Passenger", get_column_names())
```

However, when you run the code, you get an [exception traceback](#) like the following:

```
Python Traceback

Traceback (most recent call last):
...
ValueError: Type names and field names cannot be a keyword: 'class'
```

This tells you that the `class` column name isn’t a valid field name for your `namedtuple` class. To prevent this situation, you decide to use `rename`:

```
Python

# passenger.py

# ...

Passenger = namedtuple("Passenger", get_column_names(), rename=True)
```

This causes `namedtuple()` to automatically replace invalid names with positional names. Now suppose you retrieve one row from the database and create your first `Passenger` instance, like this:

```
Python

>>> from passenger import Passenger
>>> from database import get_passenger_by_id

>>> Passenger(get_passenger_by_id("1234"))
Passenger(_0=1234, name='john', _2='Business', _3='John Doe')
```

In this case, `get_passenger_by_id()` is another function available in your hypothetical application. It retrieves the data for a given passenger in a tuple. The final result is that your newly created passenger has three positional field names, and only `name` reflects the original column name. When you dig into the database, you realize that the passengers table has the following columns:

Column	Stores	Replaced?	Reason
<code>_id</code>	A unique identifier for each passenger	Yes	It starts with an underscore.
<code>name</code>	A short name for each passenger	No	It’s a valid Python identifier.
<code>class</code>	The class in which the passenger travels	Yes	It’s a Python keyword.
<code>name</code>	The passenger’s full name	Yes	It’s repeated.

In situations where you create named tuples based on values outside your control, the `rename` option should be set to `True` so the invalid fields get renamed with valid positional names.

The second optional argument to `namedtuple()` is `defaults`. This argument defaults to `None`, which means that the fields won’t have default values. You can set `defaults` to an iterable of values. In this case, `namedtuple()` assigns the values in the `defaults` iterable to the rightmost fields:

```
Python
```



```
>>> from collections import namedtuple

>>> Developer = namedtuple(
...     "Developer",
...     "name level language",
...     defaults=["Junior", "Python"]
... )

>>> Developer("John")
Developer(name='John', level='Junior', language='Python')
```

In this example, the `level` and `language` fields have default values. This makes them optional arguments. Since you don't define a default value for `name`, you need to provide a value when you create the `namedtuple` instance. So, arguments without a default value are required. Note that the default values are applied to the rightmost fields.

The last argument to `namedtuple()` is `module`. If you provide a valid module name to this argument, then the `__module__` attribute of the resulting `namedtuple` is set to that value. This attribute holds the name of the module in which a given function or callable is defined:

Python



```
>>> from collections import namedtuple

>>> Point = namedtuple("Point", "x y", module="custom")
>>> Point
<class 'custom.Point'>
>>> Point.__module__
'custom'
```

In this example, when you access `__module__` on `Point`, you get `'custom'` as a result. This indicates that your `Point` class is defined in your `custom` module.

The [motivation](#) to add the `module` argument to `namedtuple()` in [Python 3.6](#) was to make it possible for named tuples to support [pickling](#) through different [Python implementations](#).



[Become a Python Expert »](#)

[Remove ads](#)

Exploring Additional Features of namedtuple Classes

Besides the methods inherited from `tuple`, such as `.count()` and `.index()`, `namedtuple` classes also provide three additional methods and two attributes. To prevent name conflicts with custom fields, the names of these attributes and methods start with an underscore. In this section, you'll learn about these methods and attributes and how they work.

Creating namedtuple Instances From Iterables

You can use `._make()` to create named tuple instances. The method takes an iterable of values and returns a new named tuple:

Python



```
>>> from collections import namedtuple

>>> Person = namedtuple("Person", "name age height")
>>> Person._make(["Jane", 25, 1.75])
Person(name='Jane', age=25, height=1.75)
```

Here, you first create a `Person` class using `namedtuple()`. Then you call `._make()` with a list of values for each field in the `namedtuple`. Note that `._make()` is a [class method](#) that works as an alternative [class constructor](#) and returns a new named tuple instance.

Finally, `._make()` expects a single iterable as an argument, a `list` in the above example. On the other hand, the `namedtuple` constructor can take positional or keyword arguments, as you already learned.

Converting namedtuple Instances Into Dictionaries

You can convert existing named tuple instances into dictionaries using `._asdict()`. This method returns a new dictionary that uses the field names as keys. The keys of the resulting dictionary are in the same order as the fields in the original `namedtuple`:

Python



```
>>> from collections import namedtuple

>>> Person = namedtuple("Person", "name age height")
>>> jane = Person("Jane", 25, 1.75)
>>> jane._asdict()
{'name': 'Jane', 'age': 25, 'height': 1.75}
```

When you call `._asdict()` on a named tuple, you get a new `dict` object that maps field names to their corresponding values in the original named tuple.

Since [Python 3.8](#), `._asdict()` has returned a regular dictionary. Before that, it returned an `OrderedDict` object:

Python



```
Python 3.7.9 (default, Jan 14 2021, 11:41:20)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from collections import namedtuple

>>> Person = namedtuple("Person", "name age height")
>>> jane = Person("Jane", 25, 1.75)
>>> jane._asdict()
OrderedDict([('name', 'Jane'), ('age', 25), ('height', 1.75)])
```

[Python 3.8 updated `._asdict\(\)`](#) to return a regular dictionary because dictionaries remember the insertion order of their keys in Python 3.6 and up. Note that the order of keys in the resulting dictionary is equivalent to the order of fields in the original named tuple.

Replacing Fields in Existing namedtuple Instances

The last method you'll learn about is `._replace()`. This method takes keyword arguments of the form `field=value` and returns a new `namedtuple` instance updating the values of the selected fields:

Python



```
>>> from collections import namedtuple

>>> Person = namedtuple("Person", "name age height")
>>> jane = Person("Jane", 25, 1.75)

>>> # After Jane's birthday
>>> jane = jane._replace(age=26)
>>> jane
Person(name='Jane', age=26, height=1.75)
```

In this example, you update Jane's age after her birthday. Although the name of `._replace()` might suggest that the method modifies the existing named tuple, that's not what happens in practice. This is because `namedtuple` instances are immutable, so `._replace()` doesn't update `jane` [in place](#).

Exploring Additional namedtuple Attributes

Named tuples also have two additional attributes: `._fields` and `._field_defaults`. The first attribute holds a tuple of strings listing the field names. The second attribute holds a dictionary that maps field names to their respective default values, if any.

In the case of `._fields`, you can use it to introspect your `namedtuple` classes and instances. You can also create new classes from existing ones:

Python



```
>>> from collections import namedtuple

>>> Person = namedtuple("Person", "name age height")

>>> ExtendedPerson = namedtuple(
...     "ExtendedPerson",
...     [*Person._fields, "weight"]
... )

>>> jane = ExtendedPerson("Jane", 26, 1.75, 67)
>>> jane
ExtendedPerson(name='Jane', age=26, height=1.75, weight=67)
>>> jane.weight
67
```

In this example, you create a new `namedtuple` called `ExtendedPerson` with a new field, `weight`. This new type extends your old `Person`. To do that, you access `._fields` on `Person` and unpack it to a new list along with an additional field, `weight`.

You can also use `._fields` to iterate over the fields and the values in a given `namedtuple` instance using Python's [zip\(\)](#):

Python



```
>>> from collections import namedtuple

>>> Person = namedtuple("Person", "name age height weight")
>>> jane = Person("Jane", 26, 1.75, 67)
>>> for field, value in zip(jane._fields, jane):
...     print(field, "->", value)
...
name -> Jane
age -> 26
height -> 1.75
weight -> 67
```

In this example, `zip()` yields tuples of the form `(field, value)`. This way, you can access both elements of the field-value pair in the underlying named tuple. Another way to iterate over fields and values at the same time could be using `._asdict().items()`. Go ahead and give it a try!

With `._field_defaults`, you can introspect `namedtuple` classes and instances to find out what fields provide default values. Having default values makes your fields optional. For example, say your `Person` class should include an additional field to hold the country in which the person lives. Since you're mostly working with people from Canada, you set the appropriate default value for the country field like this:

Python



```
>>> from collections import namedtuple

>>> Person = namedtuple(
...     "Person",
...     "name age height weight country",
...     defaults=["Canada"]
... )

>>> Person._field_defaults
{'country': 'Canada'}
```

With a quick query to `._field_defaults`, you can figure out which fields in a given `namedtuple` provide default values. In this example, any other programmer on your team can see that your `Person` class provides "Canada" as a handy default value for country.

If your `namedtuple` doesn't provide default values, then `._field_defaults` holds an empty dictionary:

Python



```
>>> from collections import namedtuple

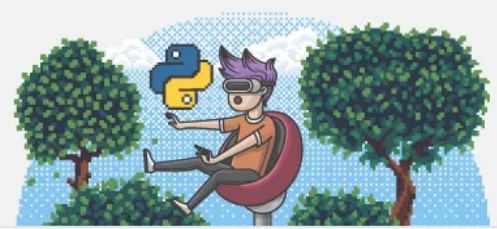
>>> Person = namedtuple("Person", "name age height weight country")
>>> Person._field_defaults
{}
```

If you don't provide a list of default values to `namedtuple()`, then it relies on the default value to defaults, which is `None`. In this case, `._field_defaults` holds an empty dictionary.

Python Dependency Management Pitfalls

A free email class

realpython.com



[Remove ads](#)

Writing Pythonic Code With namedtuple

Arguably, the fundamental use case of named tuples is to help you write more Pythonic code. The `namedtuple()` factory function was created to allow you to write readable, explicit, clean, and maintainable code.

In this section, you'll write a bunch of practical examples that will help you spot good opportunities for using named tuples instead of regular tuples so you can make your code more Pythonic.

Using Field Names Instead of Indices

Say you're creating a painting application and you need to define the pen properties to use according to the user's choice. You've coded the pen's properties in a tuple:

Python



```
>>> pen = (2, "Solid", True)

>>> if pen[0] == 2 and pen[1] == "Solid" and pen[2]:
...     print("Standard pen selected")
...
Standard pen selected
```

This line of code defines a tuple with three values. Can you tell what the meaning of each value is? Maybe you can guess that the second value is related to the line style, but what's the meaning of 2 and `True`?

You could add a nice comment to provide some context for `pen`, in which case you would end up with something like this:

Python



```
>>> # Tuple containing: line weight, line style, and beveled edges
>>> pen = (2, "Solid", True)
```

Cool! Now you know the meaning of each value in the tuple. However, what if you or another programmer is using `pen` far away from this definition? They'd have to go back to the definition just to remember what each value means.

Here's an alternative implementation of `pen` using a `namedtuple`:

Python



```
>>> from collections import namedtuple

>>> Pen = namedtuple("Pen", "width style beveled")
>>> pen = Pen(2, "Solid", True)

>>> if pen.width == 2 and pen.style == "Solid" and pen.beveled:
...     print("Standard pen selected")
...
Standard pen selected
```

Now your code makes it clear that 2 represents the pen's width, "Solid" is the line style, and so on. Anyone reading your code can see and understand that. Your new implementation of pen has two additional lines of code. That's a small amount of work that produces a big win in terms of readability and maintainability.

Returning Multiple Named Values From Functions

Another situation in which you can use a named tuple is when you need to [return multiple values](#) from a given function. In this case, using a named tuple can make your code more readable because the returned values will also provide some context for their content.

For example, Python provides a built-in function called `divmod()` that takes two numbers as arguments and returns a tuple with the **quotient** and **remainder** that result from the integer division of the input numbers:

Python



```
>>> divmod(8, 4)
(2, 0)
```

To remember the meaning of each number, you might need to read the documentation of `divmod()` because the numbers themselves don't provide much information on their individual meaning. The function's name doesn't help very much either.

Here's a function that uses a namedtuple to clarify the meaning of each number that `divmod()` returns:

Python



```
>>> from collections import namedtuple

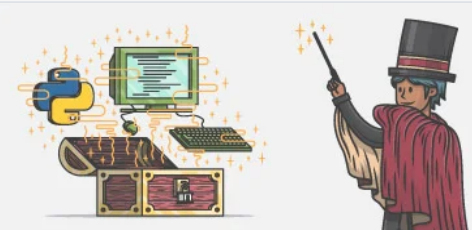
>>> def custom_divmod(a, b):
...     DivMod = namedtuple("DivMod", "quotient remainder")
...     return DivMod(*divmod(a, b))
...

>>> custom_divmod(8, 4)
DivMod(quotient=2, remainder=0)
```

In this example, you add context to each returned value, so any programmer reading your code can immediately understand what each number means.

Improve Your Python with Python Tricks

realpython.com



[Remove ads](#)

Reducing the Number of Arguments to Functions

Reducing the number of arguments a function can take is considered a best programming practice. This makes your function's signature more concise and optimizes your [testing](#) process because of the reduced number of arguments and possible combinations between them.

Again, you should consider using named tuples to approach this use case. Say you’re coding an application to manage your clients’ information. The application uses a database to store clients’ data. To process the data and update the database, you’ve created several functions. One of your high-level functions is `create_user()`, which looks like this:

Python

```
def create_user(db, username, client_name, plan):
    db.add_user(username)
    db.complete_user_profile(username, client_name, plan)
```

This function takes four arguments. The first argument, `db`, represents the database you’re working with. The rest of the arguments are closely related to a given client. This is a great opportunity to reduce the number of arguments to `create_user()` using a named tuple:

Python

```
User = namedtuple("User", "username client_name plan")
user = User("john", "John Doe", "Premium")

def create_user(db, user):
    db.add_user(user.username)
    db.complete_user_profile(
        user.username,
        user.client_name,
        user.plan
    )
```

Now `create_user()` takes only two arguments: `db` and `user`. Inside the function, you use convenient and descriptive field names to provide the arguments to `db.add_user()` and `db.complete_user_profile()`. Your high-level function, `create_user()`, is more focused on the `user`. It’s also easier to test because you just need to provide two arguments to each test.

Reading Tabular Data From Files and Databases

A quite common use case for named tuples is to use them to store database records. You can define `namedtuple` classes using the column names as field names and retrieve the data from the rows in the database to named tuples. You can also do something similar with [CSV files](#).

For example, say you have a CSV file with data regarding the employees of your company, and you want to read that data into a suitable data structure for further processing. Your CSV file looks like this:

CSV

```
name,job,email
"Linda","Technical Lead","linda@example.com"
"Joe","Senior Web Developer","joe@example.com"
"Lara","Project Manager","lara@example.com"
"David","Data Analyst","david@example.com"
"Jane","Senior Python Developer","jane@example.com"
```

You’re thinking of using Python’s [csv module](#) and its [DictReader](#) to process the file, but you have an additional requirement—you need to store the data into an immutable and lightweight data structure. In this case, a `namedtuple` might be a good choice:

Python




```
>>> import csv
>>> from collections import namedtuple

>>> with open("employees.csv", "r") as csv_file:
...     reader = csv.reader(csv_file)
...     Employee = namedtuple("Employee", next(reader), rename=True)
...     for row in reader:
...         employee = Employee(*row)
...         print(employee.name, employee.job, employee.email)
...
Linda Technical Lead linda@example.com
Joe Senior Web Developer joe@example.com
Lara Project Manager lara@example.com
David Data Analyst david@example.com
Jane Senior Python Developer jane@example.com
```

In this example, you first open the `employees.csv` file in a [with statement](#). Then you use `csv.reader()` to get an iterator over the lines in the CSV file. With `namedtuple()`, you create a new `Employee` class. The call to `next()` retrieves the first row of data from `reader`, which contains the CSV file header. This header provides the field names for your `namedtuple`.

Note: When you create a `namedtuple` based on field names out of your control, you should set `.rename` to `True`. This way, you prevent issues with invalid field names, which could be a common situation when you're working with database tables and queries, CSV files, or any other types of tabular data.

Finally, the [for loop](#) creates an `Employee` instance from each row in the CSV file and [prints](#) the list of employees to the screen.

Using namedtuple vs Other Data Structures

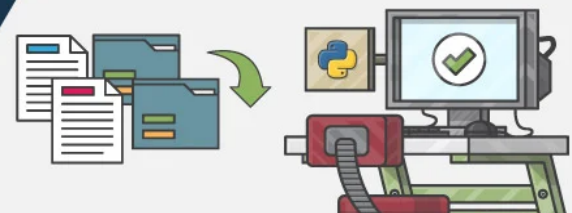
So far, you've learned how to create named tuples to make your code more readable, explicit, and Pythonic. You've also written some examples that help you spot opportunities for using named tuples in your code.

In this section, you'll take a general look at the similarities and differences between `namedtuple` classes and other Python data structures, such as dictionaries, data classes, and typed named tuples. You'll compare named tuples with other data structures regarding the following characteristics:

- Readability
- Mutability
- Memory usage
- Performance

This way, you'll be better prepared to choose the right data structure for your specific use case.

Free PDF Download: Python 3 Cheat Sheet

[Download Now](#)realpython.com

 [Remove ads](#)

namedtuple vs Dictionary

The [dictionary](#) is a fundamental data structure in Python. The language itself is built around dictionaries, so they're everywhere. Since they're so common and useful, you probably use them a lot in your code. But how different are dictionaries and named tuples?

In terms of readability, you can probably say that dictionaries are as readable as named tuples. Even though they don't provide a way to access attributes through the dot notation, the dictionary-style key lookup is quite readable and straightforward:

Python



```
>>> from collections import namedtuple

>>> jane = {"name": "Jane", "age": 25, "height": 1.75}
>>> jane["age"]
25

>>> # Equivalent named tuple
>>> Person = namedtuple("Person", "name age height")
>>> jane = Person("Jane", 25, 1.75)
>>> jane.age
25
```

In both examples, you have a total understanding of the code and its intention. The named tuple definition requires two additional lines of code, though: one line to [import](#) the `namedtuple()` factory function and another to define your `namedtuple` class, `Person`.

A big difference between both data structures is that dictionaries are [mutable](#) and named tuples are immutable. This means that you *can* modify dictionaries in place, but you *can't* modify named tuples:

Python



```
>>> from collections import namedtuple

>>> jane = {"name": "Jane", "age": 25, "height": 1.75}
>>> jane["age"] = 26
>>> jane["age"]
26
>>> jane["weight"] = 67
>>> jane
{'name': 'Jane', 'age': 26, 'height': 1.75, 'weight': 67}

>>> # Equivalent named tuple
>>> Person = namedtuple("Person", "name age height")
>>> jane = Person("Jane", 25, 1.75)

>>> jane.age = 26
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute

>>> jane.weight = 67
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Person' object has no attribute 'weight'
```

You can update the value of an existing key in a dictionary, but you can't do something similar in a named tuple. You can add new key-value pairs to existing dictionaries, but you can't add field-value pairs to existing named tuples.

Note: In named tuples, you can use `._replace()` to update the value of a given field, but that method creates and returns a new named tuple instance instead of updating the underlying instance in place.

In general, if you need an immutable data structure to properly solve a given problem, then consider using a named tuple instead of a dictionary so you can meet your requirements.

Regarding memory usage, named tuples are a quite lightweight data structure. Fire up your [code editor or IDE](#) and create the following script:

Python

```
# namedtuple_dict_memory.py

from collections import namedtuple
from pympler import asizeof

Point = namedtuple("Point", "x y z")
point = Point(1, 2, 3)

namedtuple_size = asizeof.sizeof(point)
dict_size = asizeof.sizeof(point._asdict())
gain = 100 - namedtuple_size / dict_size * 100

print(f"namedtuple: {namedtuple_size} bytes ({gain:.2f}% smaller)")
print(f"dict:      {dict_size} bytes")
```

This small script uses `asizeof.sizeof()` from [Pympler](#) to get the memory footprint of a named tuple and its equivalent dictionary.

Note: Pympler is a tool to monitor and analyze the memory behavior of Python objects.

You can install it from [PyPI](#) using `pip` as usual:

Shell



```
$ pip install pympler
```

After you run this command, Pympler will be available in your [Python environment](#) so you can run the above script.

If you [run the script](#) from your command line, then you'll get the following output:

Shell



```
$ python namedtuple_dict_memory.py
namedtuple: 160 bytes (67.74% smaller)
dict:      496 bytes
```

This output confirms that named tuples consume less memory than equivalent dictionaries. So if memory consumption is a restriction for you, then you should consider using a named tuple instead of a dictionary.

Note: When you compare named tuples and dictionaries, the final memory consumption difference will depend on the number of values and their types. With different values, you'll get different results.

Finally, you need to have an idea of how different named tuples and dictionaries are in terms of operations performance. To do that, you'll test [membership](#) and attribute access operations. Get back to your code editor and create the following script:

Python

```
# namedtuple_dict_time.py

from collections import namedtuple
from time import perf_counter

def average_time(structure, test_func):
    time_measurements = []
    for _ in range(1_000_000):
        start = perf_counter()
        test_func(structure)
        end = perf_counter()
        time_measurements.append(end - start)
    return sum(time_measurements) / len(time_measurements) * int(1e9)

def time_dict(dictionary):
    "x" in dictionary
    "missing_key" in dictionary
    2 in dictionary.values()
    "missing_value" in dictionary.values()
    dictionary["y"]

def time_namedtuple(named_tuple):
    "x" in named_tuple._fields
    "missing_field" in named_tuple._fields
    2 in named_tuple
    "missing_value" in named_tuple
    named_tuple.y

Point = namedtuple("Point", "x y z")
point = Point(x=1, y=2, z=3)

namedtuple_time = average_time(point, time_namedtuple)
dict_time = average_time(point._asdict(), time_dict)
gain = dict_time / namedtuple_time

print(f"namedtuple: {namedtuple_time:.2f} ns ({gain:.2f}x faster)")
print(f"dict:      {dict_time:.2f} ns")
```

This script times operations common to both dictionaries and named tuples, such as membership tests and attribute access. Running the script on your current system displays an output similar to the following:

Shell



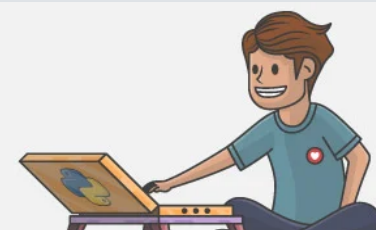
```
$ namedtuple_dict_time.py
namedtuple: 527.26 ns (1.36x faster)
dict:      717.71 ns
```

This output shows that operations on named tuples are slightly faster than similar operations on dictionaries.

5 Thoughts on Mastering Python

A free email class for Python developers

realpython.com



[Remove ads](#)

namedtuple vs Data Class

[Python 3.7](#) came with a new cool feature: [data classes](#). According to [PEP 557](#), data classes are similar to named tuples, but they're mutable:

Data Classes can be thought of as “mutable namedtuples with defaults.” ([Source](#))

However, it'd be more accurate to say that data classes are like mutable named tuples with [type hints](#). The “defaults” part isn't a difference at all because named tuples can also have default values for their fields. So, at first glance, the main differences are mutability and type hints.

To create a data class, you need to import the `dataclass()` decorator from [dataclasses](#). Then you can define your data classes using the regular class definition syntax:

Python



```
>>> from dataclasses import dataclass

>>> @dataclass
... class Person:
...     name: str
...     age: int
...     height: float
...     weight: float
...     country: str = "Canada"
...

>>> jane = Person("Jane", 25, 1.75, 67)
>>> jane
Person(name='Jane', age=25, height=1.75, weight=67, country='Canada')
>>> jane.name
'Jane'
>>> jane.name = "Jane Doe"
>>> jane.name
'Jane Doe'
```

In terms of readability, there are no significant differences between data classes and named tuples. They provide similar string representations, and you can access their attributes using the dot notation.

Mutability-wise, data classes are mutable by definition, so you can change the value of their attributes when needed. However, they have an ace up their sleeve. You can set the `dataclass()` decorator's `frozen` argument to `True` and make them immutable:

Python



```
>>> from dataclasses import dataclass

>>> @dataclass(frozen=True)
... class Person:
...     name: str
...     age: int
...     height: float
...     weight: float
...     country: str = "Canada"
...

>>> jane = Person("Jane", 25, 1.75, 67)
>>> jane.name = "Jane Doe"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 4, in __setattr__
dataclasses.FrozenInstanceError: cannot assign to field 'name'
```

If you set `frozen` to `True` in the call to `dataclass()`, then you make the data class immutable. In this case, when you try to update Jane's name, you get a [FrozenInstanceError](#).

Another subtle difference between named tuples and data classes is that the latter aren't iterable by default. Stick to the Jane example and try to iterate over her data:

Python



```
>>> for field in jane:
...     print(field)
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'Person' object is not iterable
```

If you try to iterate over a bare-bones data class, then you get a `TypeError`. This is common to regular classes. Fortunately, there are ways to work around it. For example, you can add a [special method](#), `__iter__()`, to `Person` like this:

Python



```
>>> from dataclasses import astuple, dataclass

>>> @dataclass
... class Person:
...     name: str
...     age: int
...     height: float
...     weight: float
...     country: str = "Canada"
...     def __iter__(self):
...         return iter(astuple(self))
...

>>> for field in Person("Jane", 25, 1.75, 67):
...     print(field)
...
Jane
25
1.75
67
Canada
```

Here, you first import `astuple()` from `dataclasses`. This function converts the data class into a tuple. Then you pass the resulting tuple to `iter()` so you can build and return an [iterator](#) from `__iter__()`. With this addition, you can start iterating over Jane's data.

Regarding memory consumption, named tuples are more lightweight than data classes. You can confirm this by creating and running a small script similar to the one you saw in the above section. To view the complete script, expand the box below.

Script to Compare Memory Usage: namedtuple vs Data Class

Show/Hide

Here are the results of running your script:

Shell



```
$ python namedtuple_dataclass_memory.py
namedtuple: 160 bytes (61.54% smaller)
data class: 416 bytes
```

Unlike namedtuple classes, data classes keep a per-instance `__dict__` to store [writable instance attributes](#). This contributes to a bigger memory footprint.

Next, you can expand the section below to see an example of code that compares namedtuple classes and data classes in terms of their performance on attribute access.

Script to Compare Performance: namedtuple vs Data Class

Show/Hide


In terms of performance, here are the results:


Shell




```
$ python namedtuple_dataclass_time.py
namedtuple: 274.32 ns (1.08x faster)
data class: 295.37 ns
```

The performance difference is minimal, so you can say that both data structures have equivalent performance when it comes to attribute access operations.





[Remove ads](#)

namedtuple VS typing.NamedTuple

Python 3.5 introduced a [provisional](#) module called [typing](#) to support function type annotations or [type hints](#). This module provides [NamedTuple](#), which is a typed version of namedtuple. With NamedTuple, you can create namedtuple classes with type hints. Following with the Person example, you can create an equivalent typed named tuple like this:

Python

```
>>> from typing import NamedTuple

>>> class Person(NamedTuple):
...     name: str
...     age: int
...     height: float
...     weight: float
...     country: str = "Canada"
...

>>> issubclass(Person, tuple)
True
>>> jane = Person("Jane", 25, 1.75, 67)
>>> jane.name
'Jane'
>>> jane.name = "Jane Doe"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

With NamedTuple, you can create tuple subclasses that support type hints and attribute access through the dot notation. Since the resulting class is a tuple subclass, it’s immutable as well.

A subtle detail to notice in the above example is that NamedTuple subclasses look even more similar to data classes than named tuples.

When it comes to memory consumption, both namedtuple and NamedTuple instances use the same amount of memory. You can expand the box below to view a script that compares memory usage between the two.

Script to Compare Memory Usage: namedtuple VS typing.NamedTuple

Show/Hide

This time, the script that compares memory usage produces the following output:

Shell

```
$ python typed_namedtuple_memory.py
namedtuple:      160 bytes
typing.NamedTuple: 160 bytes
```

In this case, both instances consume the same amount of memory, so there’s no winner this time.

Since `namedtuple` classes and `NamedTuple` subclasses are both subclasses of `tuple`, they have a lot in common. In this case, you can time membership tests for fields and values. You can also time attribute access with the dot notation. Expand the box below to view a script that compares the performance of both `namedtuple` and `NamedTuple`.

Script to Compare Performance: `namedtuple` VS `typing.NamedTuple`

Show/Hide

Here are the results:

Shell

```
$ python typed_namedtuple_time.py
namedtuple:      503.34 ns
typing.NamedTuple: 509.91 ns
```

In this case, you can say that both data structures behave almost the same in terms of performance. Other than that, using `NamedTuple` to create your named tuples can make your code even more explicit because you can add type information to the fields. You can also provide default values, add new functionality, and write [docstrings](#) for your typed named tuples.

In this section, you’ve learned a lot about `namedtuple` and other similar data structures and classes. Here’s a table that summarizes how `namedtuple` compares to the data structures covered in this section:

	<code>dict</code>	Data Class	<code>NamedTuple</code>
Readability	Similar	Equal	Equal
Immutability	No	No by default, yes if using <code>@dataclass(frozen=True)</code>	Yes
Memory usage	Higher	Higher	Equal
Performance	Slower	Similar	Similar
Iterability	Yes	No by default, yes if providing <code>__iter__()</code>	Yes

With this summary, you’ll be able to choose the data structure that best fits your current needs. Additionally, you should consider that data classes and `NamedTuple` allow you to add type hints, which is currently quite a desirable feature in Python code.

Subclassing namedtuple Classes

Since `namedtuple` classes are regular Python classes, you can subclass them if you need to provide additional functionalities, a docstring, a user-friendly string representation, and so on.

For example, storing the age of a person in an object isn’t considered a best practice. So you probably want to store the birth date and compute the age when needed:

Python

```
>>> from collections import namedtuple
>>> from datetime import date

>>> BasePerson = namedtuple(
...     "BasePerson",
...     "name birthdate country",
...     defaults=["Canada"]
... )

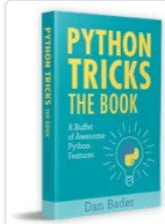
>>> class Person(BasePerson):
...     """A namedtuple subclass to hold a person's data."""
...     __slots__ = ()
...     def __repr__(self):
...         return f"Name: {self.name}, age: {self.age} years old."
...     @property
...     def age(self):
...         return (date.today() - self.birthdate).days // 365
...

>>> Person.__doc__
"A namedtuple subclass to hold a person's data."

>>> jane = Person("Jane", date(1996, 3, 5))
>>> jane.age
25
>>> jane
Name: Jane, age: 25 years old.
```

Person inherits from BasePerson, which is a namedtuple class. In the subclass definition, you first add a docstring to describe what the class does. Then you set `__slots__` to an empty tuple, which prevents the automatic creation of a per-instance `__dict__`. This keeps your BasePerson subclass memory efficient.

You also add a custom `__repr__()` to provide a nice string representation for the class. Finally, you add a [property](#) to compute the person's age using [datetime](#).



“I don’t even feel like I’ve scratched the surface of what I can do with Python”

Write More Pythonic Code »

 [Remove ads](#)

Measuring Creation Time: tuple VS namedtuple

So far, you’ve compared namedtuple classes with other data structures according to several features. In this section, you’ll take a general look at how regular tuples and named tuples compare in terms of **creation time**.

Say you have an application that creates a ton of tuples dynamically. You decide to make your code more Pythonic and maintainable using named tuples. Once you’ve updated all your [codebase](#) to use named tuples, you run the application and notice some performance issues. After some tests, you conclude that the issues could be related to creating named tuples dynamically.

Here’s a script that measures the average time required to create several tuples and named tuples dynamically:

Python

```
# tuple_namedtuple_time.py

from collections import namedtuple
from time import perf_counter

def average_time(test_func):
    time_measurements = []
    for _ in range(1_000):
        start = perf_counter()
        test_func()
        end = perf_counter()
        time_measurements.append(end - start)
    return sum(time_measurements) / len(time_measurements) * int(1e9)

def time_tuple():
    tuple([1] * 1000)

fields = [f"a{n}" for n in range(1000)]
TestNamedTuple = namedtuple("TestNamedTuple", fields)

def time_namedtuple():
    TestNamedTuple(*([1] * 1000))

namedtuple_time = average_time(time_namedtuple)
tuple_time = average_time(time_tuple)
gain = namedtuple_time / tuple_time

print(f"tuple:      {tuple_time:.2f} ns ({gain:.2f}x faster)")
print(f"namedtuple: {namedtuple_time:.2f} ns")
```

In this script, you calculate the average time it takes to create several tuples and their equivalent named tuples. If you run the script from your command line, then you'll get an output similar to the following:

Shell



```
$ python tuple_namedtuple_time.py
tuple:      7075.82 ns (3.36x faster)
namedtuple: 23773.67 ns
```

When you look at this output, you can see that creating `tuple` objects dynamically is a lot faster than creating similar named tuples. In some situations, such as working with large databases, the additional time required to create a named tuple can seriously affect your application's performance, so keep an eye on this if your code creates a lot of tuples dynamically.

Conclusion

Writing [Pythonic](#) code is an in-demand skill in the Python development space. Pythonic code is readable, explicit, clean, maintainable, and takes advantage of Python idioms and best practices. In this tutorial, you learned about creating `namedtuple` classes and instances and how they can help you improve the quality of your Python code.

In this tutorial, you learned:

- How to create and use `namedtuple` classes and instances
- How to take advantage of cool `namedtuple` **features**
- When to use `namedtuple` instances to write **Pythonic code**
- When to use a `namedtuple` instead of a similar **data structure**
- How to **subclass a namedtuple** to add new features

With this knowledge, you can deeply improve the quality of your existing and future code. If you frequently use tuples, then consider turning them into named tuples whenever it makes sense. Doing so will make your code much more readable and Pythonic.

Mark as Completed

