A namedtuple is a factory function from Python's collections module that lets you create tuple-like objects with fields accessible by attribute lookup (like point.x) as well as by index (point[0]). It's a simple way to create lightweight, immutable data structures, making your code more readable and self-documenting. 👨‍💻

Think of it as creating a blueprint for a tuple where each position has a name.

Why Use a namedtuple?

Regular tuples are useful, but they can lead to unclear code. For example, if you have a tuple representing a color, rgb = (255, 0, 0), you have to remember that rgb[0] is red, rgb[1] is green, and so on. This can make your code hard to read and maintain.

namedtuple solves this problem by adding meaningful names to the positions. Instead of rgb[0], you could write rgb.red, which is much clearer.

Key advantages:

- **Readability:** Accessing fields by name makes code self-documenting.
- **Memory-Efficient:** They are just as memory-efficient as regular tuples because they don't have per-instance dictionaries, making them much lighter than a standard object or dictionary.
- **Immutable:** Like regular tuples, their values cannot be changed after creation. This is useful for creating constant data structures or keys for dictionaries.

How to Create and Use a namedtuple

Using a namedtuple is a three-step process: **define**, **instantiate**, and **access**.

1. Define the namedtuple Blueprint

First, you import namedtuple from the collections module and then use it to define your new data type.

The syntax is: namedtuple('TypeName', ['field_name1', 'field_name2'])

- 'TypeName': The name of the new tuple subclass you are creating. By convention, this matches the variable name.
- ['field_name1', ...]: A list of strings for the field names. You can also provide a single space-separated string like 'field_name1 field_name2'.

Python

```
from collections import namedtuple

# Define a 'Point' type with 'x' and 'y' fields
Point = namedtuple('Point', ['x', 'y'])

# Define a 'Car' type using a space-separated string for fields
Car = namedtuple('Car', 'make model year color')
```

2. Create an Instance

Once you've defined the type, you can create instances of it just like you would with a regular class.

Python

```
# Create instances of our namedtuples
p1 = Point(10, 20)
my_car = Car('Toyota', 'Corolla', 2021, 'blue')

print(p1)
# Output: Point(x=10, y=20)

print(my_car)
# Output: Car(make='Toyota', model='Corolla', year=2021, color='blue')
```

3. Access Data

This is the main benefit! You can access data by name (using dot notation) or by index (like a regular tuple).

Python

```
# Access by name (more readable)
print(f"The x-coordinate is: {p1.x}")      # Output: The x-coordinate is: 10
print(f"The car model is: {my_car.model}") # Output: The car model is: Corolla

# Access by index (like a regular tuple)
print(f"The y-coordinate is: {p1[1]}")      # Output: The y-coordinate is: 20
print(f"The car's make is: {my_car[0]}")  # Output: The car's make is: Toyota
```

Remember, namedtuple instances are **immutable**. You cannot change a value after creation.

Python

```
# This will raise an AttributeError
# p1.x = 30
```

Useful Helper Methods

namedtuple comes with a few handy helper methods that start with an underscore.

_asdict()

Returns the namedtuple as an OrderedDict (or a regular dict in newer Python versions), which

is great for serialization (e.g., converting to JSON).

Python

```
car_dict = my_car._asdict()
print(car_dict)
# Output: {'make': 'Toyota', 'model': 'Corolla', 'year': 2021, 'color': 'blue'}
```

_replace(**kwargs)

Since namedtuple is immutable, you can't change its fields. The _replace() method returns a **new** instance of the tuple with specified fields updated to new values.

Python

```
# Let's "repaint" the car
repainted_car = my_car._replace(color='red')

print(my_car)       # The original is unchanged
# Output: Car(make='Toyota', model='Corolla', year=2021, color='blue')

print(repainted_car)  # A new instance is created
# Output: Car(make='Toyota', model='Corolla', year=2021, color='red')
```

_fields

A tuple containing the field names of the namedtuple.

Python

```
print(Car._fields)
# Output: ('make', 'model', 'year', 'color')
```

_make(iterable)

Creates a new instance from an existing sequence or iterable.

Python

```
data = ['Ford', 'Mustang', 2022, 'black']
new_car = Car._make(data)

print(new_car)
# Output: Car(make='Ford', model='Mustang', year=2022, color='black')
```

---

namedtuple vs. Other Data Structures

Feature	namedtuple	dict	tuple	dataclass (Python 3.7+)
Mutable?	No	Yes	No	Yes (by default)
Access	By name (.) and index ([])	By key ([])	By index ([])	By name (.)
Memory Usage	Low (same as tuple)	High	Low	Medium (like a class)

Readability	High	High	Low	High
Best For	Lightweight, immutable data records.	Flexible, mutable collections of key-value pairs.	Simple, fixed sequences of items.	More complex mutable (or immutable) data objects, especially with type hints and methods.

While dataclasses are a more modern and flexible alternative, **namedtuple remains an excellent choice** for simple, backward-compatible, and memory-efficient data containers where immutability is a key requirement.