namedtuple — Tuple Subclass with Named Fields

The standard `tuple` uses numerical indexes to access its members.

```
# collections_tuple.py

bob = ('Bob', 30, 'male')
print('Representation:', bob)

jane = ('Jane', 29, 'female')
print('\nField by index:', jane[0])

print('\nFields by index:')
for p in [bob, jane]:
    print('{} is a {} year old {}'.format(*p))
```

This makes `tuples` convenient containers for simple uses.

```
$ python3 collections_tuple.py

Representation: ('Bob', 30, 'male')

Field by index: Jane

Fields by index:
Bob is a 30 year old male
Jane is a 29 year old female
```

In contrast, remembering which index should be used for each value can lead to errors, especially if the `tuple` has a lot of fields and is constructed far from where it is used. A `namedtuple` assigns names, as well as the numerical index, to each member.

Defining

`namedtuple` instances are just as memory efficient as regular tuples because they do not have per-instance dictionaries. Each kind of `namedtuple` is represented by its own class, which is created by using the `namedtuple()` factory function. The arguments are the name of the new class and a string containing the names of the elements.

```
# collections_namedtuple_person.py

import collections

Person = collections.namedtuple('Person', 'name age')

bob = Person(name='Bob', age=30)
print('\nRepresentation:', bob)

jane = Person(name='Jane', age=29)
print('\nField by name:', jane.name)

print('\nFields by index:')
for p in [bob, jane]:
    print('{} is {} years old'.format(*p))
```

As the example illustrates, it is possible to access the fields of the `namedtuple` by name using dotted notation (`obj.attr`) as well as by using the positional indexes of standard tuples.

```
$ python3 collections_namedtuple_person.py


Representation: Person(name='Bob', age=30)
```

```
Representation: Person(name='Bob', age=30)

Field by name: Jane

Fields by index:
Bob is 30 years old
Jane is 29 years old
```

Just like a regular `tuple`, a `namedtuple` is immutable. This restriction allows `tuple` instances to have a consistent hash value, which makes it possible to use them as keys in dictionaries and to be included in sets.

```
# collections_namedtuple_immutable.py

import collections

Person = collections.namedtuple('Person', 'name age')

pat = Person(name='Pat', age=12)
print('\nRepresentation:', pat)

pat.age = 21
```

Trying to change a value through its named attribute results in an `AttributeError`.

```
$ python3 collections_namedtuple_immutable.py


Representation: Person(name='Pat', age=12)
Traceback (most recent call last):
  File "collections_namedtuple_immutable.py", line 17, in
<module>
    pat.age = 21
AttributeError: can't set attribute
```

Invalid Field Names

Field names are invalid if they are repeated or conflict with Python keywords.

```
# collections_namedtuple_bad_fields.py

import collections

try:
    collections.namedtuple('Person', 'name class age')
except ValueError as err:
    print(err)

try:
    collections.namedtuple('Person', 'name age age')
except ValueError as err:
    print(err)
```

As the field names are parsed, invalid values cause `ValueError` exceptions.

```
$ python3 collections_namedtuple_bad_fields.py

Type names and field names cannot be a keyword: 'class'
Encountered duplicate field name: 'age'
```

In situations where a `namedtuple` is created based on values outside the control of the program (such as to represent the rows returned by a database query, where the schema is not known in advance), the `rename` option should be set to `True` so the invalid fields are renamed.

```
# collections_namedtuple_rename.py

import collections

with_class = collections.namedtuple(
```

```
    'Person', 'name class age',
    rename=True)
print(with_class._fields)

two_ages = collections.namedtuple(
    'Person', 'name age age',
    rename=True)
print(two_ages._fields)
```

The new names for renamed fields depend on their index in the tuple, so the field with name `class` becomes `_1` and the duplicate age field is changed to `_2`.

```
$ python3 collections_namedtuple_rename.py

('name', '_1', 'age')
('name', 'age', '_2')
```

Special Attributes

`namedtuple` provides several useful attributes and methods for working with subclasses and instances. All of these built-in properties have names prefixed with an underscore (_), which by convention in most Python programs indicates a private attribute. For `namedtuple`, however, the prefix is intended to protect the name from collision with user-provided attribute names.

The names of the fields passed to `namedtuple` to define the new class are saved in the `_fields` attribute.

```
# collections_namedtuple_fields.py

import collections

Person = collections.namedtuple('Person', 'name age')

bob = Person(name='Bob', age=30)
print('Representation:', bob)
print('Fields:', bob._fields)
```

Although the argument is a single space-separated string, the stored value is the sequence of individual names.

```
$ python3 collections_namedtuple_fields.py

Representation: Person(name='Bob', age=30)
Fields: ('name', 'age')
```

`namedtuple` instances can be converted to `OrderedDict` instances using `_asdict()`.

```
# collections_namedtuple_asdict.py

import collections

Person = collections.namedtuple('Person', 'name age')

bob = Person(name='Bob', age=30)
print('Representation:', bob)
print('As Dictionary:', bob._asdict())
```

The keys of the `OrderedDict` are in the same order as the fields for the `namedtuple`.

```
$ python3 collections_namedtuple_asdict.py

Representation: Person(name='Bob', age=30)
As Dictionary: OrderedDict([('name', 'Bob'), ('age', 30)])
```

The `_replace()` method builds a new instance, replacing the values of some fields in the process.

```
# collections_namedtuple_replace.py

import collections
```

```
Person = collections.namedtuple('Person', 'name age')

bob = Person(name='Bob', age=30)
print('\nBefore:', bob)
bob2 = bob._replace(name='Robert')
print('After:', bob2)
print('Same?:', bob is bob2)
```

Although the name implies it is modifying the existing object, because `namedtuple` instances are immutable the method actually returns a new object.

```
$ python3 collections_namedtuple_replace.py


Before: Person(name='Bob', age=30)
After: Person(name='Robert', age=30)
Same?: False
```

deque — Double-Ended Queue

OrderedDict — Remember the Order Keys are Added to a Dictionary

Quick Links

This page was last updated 2017-01-28.

Navigation

deque — Double-Ended Queue
OrderedDict — Remember the Order Keys are Added to a Dictionary

The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?

This Site

Module Index
Index

Other Writing

- Blog
- The Python Standard Library By Example