merge7

June 14, 2023

0.0.1 Tuples as Data Structures

Tuples are an immutable container type.

They contain a collection of objects. The tuple is a sequence type - this means order matters (and is preserved) and elements can be accessed by index (zero based), slicing, or iteration.

Other common sequence types in Python include lists and strings. Strings, like tuples are immutable, whereas lists are mutable.

Tuples are sometimes presented as immutable lists, but in fact, they could be compared more closely to strings with one major difference: strings are homogeneous sequences, while tuples can be heterogeneous.

A tuple literal is often presented as:

[1]:
```
('a', 10, True)
```

[1]:
```
('a', 10, True)
```

But the parentheses are not what indicate a tuple - it is the commas:

[2]:
```
a = ('a', 10, True)
b = 'b', 20, False
```

[3]:
```
type(a)
```

[3]:
```
tuple
```

[4]:
```
type(b)
```

[4]:
```
tuple
```

Sometimes however, the parentheses are *required* to remove any ambiguity.

For example, consider this function that expects a tuple (or other iterable) as its argument:

[5]:
```
def iterate(t):
    for element in t:
        print(element)
```

If we call the function this way, Python will interpret it as three arguments:

[6]:
```
iterate(1, 2, 3)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-6-868649c3b72b> in <module>()
----> 1 iterate(1, 2, 3)

TypeError: iterate() takes 1 positional argument but 3 were given
```

Instead, we now **have** to use the parentheses to indicate we are packing a tuple:

[7]:
```
iterate((1, 2, 3))
```

```
1
2
3
```

Since tuples are sequence types, we can access items by index:

[8]:
```
a = 'a', 10, True
```

[9]:
```
a[2]
```

[9]:
```
True
```

Or we can even slice them:

[10]:
```
a = 1, 2, 3, 4, 5
a[2:4]
```

[10]:
```
(3, 4)
```

We can iterate over them:

[11]:
```
a = 1, 2, 3, 4, 5
for element in a:
    print(element)
```

```
1
2
3
4
5
```

We can also use unpacking:

[12]:
```
point = 10, 20, 30
```

[13]:
```
x, y, z = point
```

[14]:
```
print(x)
print(y)
print(z)
```

```
10
20
30
```

Tuples are immutable, in the sense that we cannot change the reference of an object in the container and we cannot add or remove objects from the container. This is the same as strings.

[15]:
```
a = 10, 'python', True
```

[16]:
```
a[0] = 20
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-16-90c4006d224a> in <module>()
----> 1 a[0] = 20

TypeError: 'tuple' object does not support item assignment
```

We can however 'extend' tuples, but just as with strings, we are actually just creating a new tuple:

[17]:
```
a = 1, 2, 3
```

[18]:
```
id(a)
```

[18]: 2726988303960

[19]:
```
a = a + (4, 5, 6)
```

[20]:
```
a
```

[20]: (1, 2, 3, 4, 5, 6)

[21]:
```
id(a)
```

[21]: 2726964089000

As you can see we no longer have the same memory address for `a`.

We have to be careful when we think about immutability of tuples. The tuple, as a container is immutable, but the elements contained in the tuple may very well be mutable.

Let's define a simple point class to store the x and y coordinates of a point in 2D space:

[22]:
```
class Point2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f'{self.__class__.__name__}(x={self.x}, y={self.y})'
```

[23]:
```
a = Point2D(0, 0), Point2D(10, 10), Point2D(20, 20)
```

[24]:
```
a
```

[24]:
```
(Point2D(x=0, y=0), Point2D(x=10, y=10), Point2D(x=20, y=20))
```

Although the tuple `a` is immutable, its contained elements are mutable:

So we cannot do this:

[25]:
```
a[0] = Point2D(-10, -10)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-25-e869cf518b45> in <module>()
----> 1 a[0] = Point2D(-10, -10)

TypeError: 'tuple' object does not support item assignment
```

But we can modify the contents of the first element:

[26]:
```
a[0].x = -10
```

[27]:
```
a
```

[27]:
```
(Point2D(x=-10, y=0), Point2D(x=10, y=10), Point2D(x=20, y=20))
```

Tuples as Data Records We can interpret tuples as lightweight data structures where, by convention, the position of the element in the tuple has meaning.

For example, we may elect to represent a point as a tuple, and not use the class approach we just did:

[28]:
```
pt1 = (0, 0)
pt2 = (10, 10)
```

Here, we simply decide that the first position of the tuple represents the x=coordinate while the second element represents the y-coordinate of a point in 2D space.

We could also decide that we are going to represent a city using a tuple, where the first position will the city name, the second position will be the country, and the the third position will be the

population:

[29]:
```
london = 'London', 'UK', 8_780_000
new_york = 'New York', 'USA', 8_500_000
beijing = 'Beijing', 'China', 21_000_000
```

We can even have a list of these tuples:

[30]:
```
cities = london, new_york, beijing
```

We can obtain a list of all the cities in the list using a simple list comprehension and the fact that the city name is the first element (index 0) of each tuple:

[31]:
```
city_names = [t[0] for t in cities]
print(city_names)
```

```
['London', 'New York', 'Beijing']
```

We could even calculate the total population of all these cities.

We start with a simple loop to do this:

[32]:
```
total = 0
for city in cities:
    total += city[2]
print (f'total={total}')
```

```
total=38280000
```

You will note that the reason this worked is because the `cities` list contained **only** city tuples. The list was homogeneous. The tuples on the other hand are heterogeneous.

This is often a key difference between lists and tuples, especially when we consider tuples as data structures. The tuples are heterogeneous, while the list needs to be homogeneous so we can apoply the same calculations to each element of the list.

The above example would break if one of the elements in the `cities` list was an integer for example.

Back to our example calculating the total population. There is a more Pythonic way of doing this.

First we use a comprehension to extract just the population from each city :

[33]:
```
[city[2] for city in cities]
```

[33]:
```
[8780000, 8500000, 21000000]
```

Next we simply sum up the population values:

[34]:
```
sum([city[2] for city in cities])
```

[34]:
```
38280000
```

In fact (and we'll cover this in detail later in this course), we don't even need the square brackets in the sum:

[35]:
```
sum(city[2] for city in cities)
```

[35]:
```
38280000
```

Now, since tuples are sequence types, and hence iterable, we can also use unpacking to extract values from the tuple:

[36]:
```
city, country, population = new_york
```

[37]:
```
print(city)
print(country)
print(population)
```

```
New York
USA
8500000
```

We can also use extended unpacking:

[38]:
```
record = 'DJIA', 2018, 1, 19, 25_987, 26_072, 25_942, 26_072
```

Where the structure is: symbol, year, month, day, open, high low, close

We could then unpack the record using straight unpacking:

[39]:
```
symbol, year, month, day, open_, high, low, close = record
```

[40]:
```
print(symbol)
print(close)
```

```
DJIA
26072
```

But suppose we are only interested in the symbol, year, month, day and close. Then we could use extended unpacking as follows:

[41]:
```
symbol, year, month, day, *others, close = record
```

[42]:
```
print(symbol, year, month, day, close)
```

```
DJIA 2018 1 19 26072
```

[43]:
```
print(others)
```

```
[25987, 26072, 25942]
```

A convetion often used in Python when we are not particularly interested in something, is to use an underscore as a variable name:

[44]:
```
symbol, year, month, day, *_, close = record
```

There's nothing special about the underscore here, it's just a legal variable name (in an interactive Python session, the underscore is actually used to store the results of the last calculation)

[45]:
```
print(_)
```

```
[25987, 26072, 25942]
```

By the way do not write code like this to do the unpacking we just did:

[46]:
```
symbol, year, close = record[0], record[1], record[7]
```

Although this works, it is not very readable code, plus you are packing a new tuple (the right hand side) and then unpacking it into the variables on the left. Much better to do this:

[47]:
```
symbol, year, *_, close = record
```

If you only need to pick a few elements out of the tuple (like in our example where we just wanted the population to sum it up), then by all means access it directly using the index.

But did you know that you can also unpack tuples directly in the loop?

[48]:
```
for element in cities:
    print(element)
```

```
('London', 'UK', 8780000)
('New York', 'USA', 8500000)
('Beijing', 'China', 21000000)
```

As you can see, each element is a tuple, and we can actually unpack it at the same time as the loop this way:

[49]:
```
for city, country, population in cities:
    print(f'city={city}, population={population}')
```

```
city=London, population=8780000
city=New York, population=8500000
city=Beijing, population=21000000
```

This, by the way, is how we can use the `enumerate` function in Python. The enumerate function produces an iterable from another iterable but contains the index number. Values are returned as tuples, where the first position is the index value, and the second position is the value (here we also see how a tuple was used as a data structure). So that tuple can be unpacked as follows:

[50]:
```
for index, value in enumerate(beijing):
    print(f'{index}: {value}')
```

```
0: Beijing
1: China
2: 21000000
```

Of course, since we are not interested in the country in this case, we might write it this way as well:

[51]:
```
for city, _, population in cities:
    print(f'city={city}, population={population}')
```

```
city=London, population=8780000
city=New York, population=8500000
city=Beijing, population=21000000
```

Another frequent application of usign tuples as data structures is for returning multiple values from a function.

[67]:
```
from random import uniform
from math import sqrt

def random_shot(radius):
    '''Generates a random 2D coordinate within
    the bounds [-radius, radius] * [-radius, radius]
    (a square of area 4)
    and also determines if it falls within
    a circle centered at the origin
    with specified radius'''

    random_x = uniform(-radius, radius)
    random_y = uniform(-radius, radius)

    if sqrt(random_x ** 2 + random_y ** 2) <= radius:
        is_in_circle = True
    else:
        is_in_circle = False

    return random_x, random_y, is_in_circle
```

[71]:
```
num_attempts = 1_000_000
count_inside = 0
for i in range(num_attempts):
    *_, is_in_circle = random_shot(1)
    if is_in_circle:
        count_inside += 1

print(f'Pi is approximately: {4 * count_inside / num_attempts}')
```

```
Pi is approximately: 3.14294
```

0.0.2 Named Tuples

The `namedtuple` function in `collections` allows us to create a tuple that also has names attached to each field (aka property). This can be handy to reference data in the tuple structure by name instead of just relying on position.

The `namedtuple` function is basically a class factory that creates a new type of class that uses a tuple as its underlying data storage (in fact, named tuples inherit from `tuple`), but layers in field names to each position and makes a property out of the field name.

The `namedtuple` function creates a **class**, and we then use that class to instantiate our instances of named tuples.

To use the `namedtuple` function we therefore need to select a class **name**, as well as indicate the **property** names, in the order in which they will be stored and accessed in the tuple.

Note that a `namedtuple`, like the regular `tuple` is an **immutable** data structure. (In fact, named tuples inherit from tuples - we'll revisit this in our section on metaclasses)

If you find yourself writing code such as:

[1]:
```
class Point3D:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z
```

Forget it! You seriously need to use named tuples! Not only can you shorten the amount of code you need to write, but you get some additional functionality for "free", such as `__repr__` and `__eq__` that you do not have to implement yourself!

Creating Named Tuples We are going to create a `Point` named tuple that will contain an x-coordinate and a y-coordinate.

[2]:
```
from collections import namedtuple
```

[3]:
```
Point2D = namedtuple('Point2D', ('x', 'y'))
```

Note that we have two different uses of `Point2D` here. The label we are assigning the return value of the call to `namedtuple` and the **name** of the class generated by calling `namedtuple`.

We could also have done the following:

[4]:
```
Pt = namedtuple('Point2D', ('x', 'y'))
```

The `namedtuple` class name is `Point2D`, but the label we `Pt` simply points to that class, so we would then create instances of the `Point2D` class as follows:

[5]:
```
pt1 = Pt(10, 20)
```

And we can see what `pt1` is:

[6]:
```
pt1
```

[6]:
```
Point2D(x=10, y=20)
```

As you can see we have an object of type `Point2D`, and it has two properties, `x` and `y` with respective values `10` and `20`.

The only weird thing here is that we are using `Pt` to generate our instances of the `Point2D` class.

That's why we usually always created `namedtuple` generated classes this way:

[7]:
```
Point2D = namedtuple('Point2D', ('x', 'y'))
```

Then the following makes more sense:

[8]:
```
pt1 = Point2D(10, 20)
```

[9]:
```
pt1
```

[9]:
```
Point2D(x=10, y=20)
```

This is not different than doing this:

[10]:
```
Pt3 = Point3D  # class we defined earlier
```

[11]:
```
pt3 = Pt3(10, 20, 30)
```

[12]:
```
pt3
```

[12]:
```
<__main__.Point3D at 0x27408e1fa90>
```

As you can see above, we used another label `Pt3` as a label that also references the `Point3D` class. It would be weird to do it this way here, and its weird for tuples as well. Of course, you may run into circumstances where you need to do this - just not as a general rule.

Note that all named tuples are honest to goodness **classes**, just as if you had used a `class` definition such as with `Point3D`.

The `namedtuple` function **generates** classes for us - it is a **class factory**.

[13]:
```
type(Point3D)
```

[13]:
```
type
```

[14]:
```
type(Point2D)
```

[14]:
```
type
```

However, `Point2D` is a subclass of `tuple`, while `Point3D` is not:

[15]:
```
isinstance(pt1, tuple)
```

[15]:
```
True
```

[16]:
```
isinstance(pt3, tuple)
```

[16]:
```
False
```

So, when we create an instance of a class, we are in fact calling the `__new__` method with our initial values. It's just a callable that has the **field names** we used to generate our named tuple class as its parameters. This means we can use keyword arguments when instantiating our named tuples!

[17]:
```
pt4 = Point2D(y=20, x=10)
```

[18]:
```
pt4
```

[18]:
```
Point2D(x=10, y=20)
```

What did we get for free using a named tuple vs our own class? First using a named tuple for our 2D point:

[19]:
```
pt2d_1 = Point2D(10, 20)
pt2d_2 = Point2D(10, 20)
```

[20]:
```
pt2d_1
```

[20]:
```
Point2D(x=10, y=20)
```

[21]:
```
pt2d_1 == pt2d_2
```

[21]:
```
True
```

Now using our 3D class:

[22]:
```
pt3d_1 = Point3D(10, 20, 30)
pt3d_2 = Point3D(10, 20, 30)
```

[23]:
```
pt3d_1
```

[23]:
```
<__main__.Point3D at 0x27408e1f9e8>
```

Oh, we probably need to implement the `__repr__` method in our class

[24]:
```
pt3d_1 == pt3d_2
```

[24]:
```
False
```

And we would also need to implement the **eq** method!

Let's do that:

[25]:
```
class Point3D:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z
```

```
    def __repr__(self):
        return f"Point3D(x={self.x}, y={self.y}, z={self.z})"

    def __eq__(self, other):
        if isinstance(other, Point3D):
            return self.x == other.x and self.y == other.y and self.z == other.z
        else:
            return False
```

[26]:
```
pt3d_1 = Point3D(10, 20, 30)
pt3d_2 = Point3D(10, 20, 30)
```

[27]:
```
pt3d_1
```

[27]:
```
Point3D(x=10, y=20, z=30)
```

[28]:
```
pt3d_1 == pt3d_2
```

[28]:
```
True
```

How about finding the largest coordinate in the point?

That's easy for `Point2D` since it is a tuple, but not the case for `Point3D`:

[29]:
```
max(pt2d_1)
```

[29]:
```
20
```

[30]:
```
max(pt3d_1)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-30-e803e2758ff1> in <module>()
----> 1 max(pt3d_1)

TypeError: 'Point3D' object is not iterable
```

How about calculating the dot product of two points (considering them as vectors starting at the origin)?

The formula would be: a.b = a.x * b.x + a.y + b.y + a.z * b.z

For the 3D point we would need to do the following:

[31]:
```
def dot_product_3d(a, b):
    return a.x * b.x + a.y * b.y + a.z + b.z
```

[32]:
```
dot_product_3d(pt3d_1, pt3d_2)
```

[32]:
```
560
```

But for our 2D point, which, remember is a tuple as well, we can write a generic function that would work equally well with a 3D named tuple too:

[33]:
```
def dot_product(a, b):
    return sum(e[0] * e[1] for e in  zip(a, b))
```

Here's a break down of how we implemented the dot product:

First we zip up the components of `a` and `b` to get an iterable of tuples containing the x-coordinates in the 1st element, and the y-coordinates in the second tuple. Our zip will contain as many elements as there are dimensions.

[34]:
```
a = Point2D(1, 2)
b = Point2D(10, 20)
print(a)
print(b)
print(tuple(a))
print(tuple(b))
print(list(zip(a, b)))
```

```
Point2D(x=1, y=2)
Point2D(x=10, y=20)
(1, 2)
(10, 20)
[(1, 10), (2, 20)]
```

Note that if we had more dimensions this would work equally well.

Suppose we had 3 dimensions:

[35]:
```
u = (1, 2, 3)
v = (10, 20, 30)
list(zip(u, v))
```

[35]:
```
[(1, 10), (2, 20), (3, 30)]
```

Then we create a comprehension that multiplies the components together:

[36]:
```
[e[0] * e[1] for e in zip(a, b)]
```

[36]:
```
[10, 40]
```

Then we simply add those up:

[37]:
```
sum([e[0] * e[1] for e in zip(a, b)])
```

[37]:
```
50
```

[38]:
```
dot_product(a, b)
```

[38]:
```
50
```

And if we defined a 4D point named tuple:

[39]:
```
Point4D = namedtuple('Point4D', ['i', 'j', 'k', 'l'])
```

[40]:
```
pt4d_1 = (1, 1, 1, 10)
pt4d_2 = (2, 2, 2, 10)
```

[41]:
```
dot_product(pt4d_1, pt4d_2)
```

[41]:
```
106
```

As you can see we got the correct dot product. We could not have done this using our `Point3D` class!

Other Ways to Specify Field Names There are a number of ways we can specify the field names for the named tuple:

- we can provide a sequence of strings containing each property name
- we can provide a single string with property names separated by whitespace or a comma

[42]:
```
Circle = namedtuple('Circle', ['center_x', 'center_y', 'radius'])
```

[43]:
```
circle_1 = Circle(0, 0, 10)
circle_2 = Circle(center_x=10, center_y=20, radius=100)
```

[44]:
```
circle_1
```

[44]:
```
Circle(center_x=0, center_y=0, radius=10)
```

[45]:
```
circle_2
```

[45]:
```
Circle(center_x=10, center_y=20, radius=100)
```

Or we can do it this way:

[46]:
```
City = namedtuple('City', 'name country population')
```

[47]:
```
new_york = City('New York', 'USA', 8_500_000)
```

[48]:
```
new_york
```

[48]:
```
City(name='New York', country='USA', population=8500000)
```

This would work equally well:

[49]:
```
Stock = namedtuple('Stock', 'symbol, year, month, day, open, high, low, close')
```

[50]:
```
djia = Stock('DJIA', 2018, 1, 25, 26_313, 26_458, 26_260, 26_393)
```

[51]:
```
djia
```

[51]:
```
Stock(symbol='DJIA', year=2018, month=1, day=25, open=26313, high=26458,
low=26260, close=26393)
```

In fact, since whitespace can be used we can even use a multi-line string!

[52]:
```
Stock = namedtuple('Stock', '''symbol
                               year month day
                               open high low close''')
```

[53]:
```
djia = Stock('DJIA', 2018, 1, 25, 26_313, 26_458, 26_260, 26_393)
```

[54]:
```
djia
```

[54]:
```
Stock(symbol='DJIA', year=2018, month=1, day=25, open=26313, high=26458,
low=26260, close=26393)
```

Accessing Items in a Named Tuple The major advantage of named tuples are that, as the name suggests, we can access the properties (fields) of the tuple by name:

[55]:
```
pt1
```

[55]:
```
Point2D(x=10, y=20)
```

[56]:
```
pt1.x
```

[56]:
```
10
```

[57]:
```
circle_1
```

[57]:
```
Circle(center_x=0, center_y=0, radius=10)
```

[58]:
```
circle_1.radius
```

[58]:
```
10
```

Now named tuples *are* tuples, so elements can be accessed by index, unpacked, and iterated.

[59]:
```
circle_1[2]
```

[59]:
```
10
```

[60]:
```
for item in djia:
    print(item)
```

```
DJIA
2018
1
25
26313
26458
26260
26393
```

We can also unpack named tuples just like ordinary tuples:

[61]:
```
pt1
```

[61]:
```
Point2D(x=10, y=20)
```

[62]:
```
x, y = pt1
```

[63]:
```
print(x, y)
```

```
10 20
```

We can also use extended unpacking:

[64]:
```
djia
```

[64]:
```
Stock(symbol='DJIA', year=2018, month=1, day=25, open=26313, high=26458,
low=26260, close=26393)
```

[65]:
```
symbol, *_, close = djia
```

[66]:
```
print(symbol, close)
```

```
DJIA 26393
```

And remember that the _ we use in the unpacking is just a regular variable:

[67]:
```
print(_)
```

```
[2018, 1, 25, 26313, 26458, 26260]
```

The field names for these named tuples can be any valid variable name **except** that they cannot start with an underscore.

For example the following would not be valid:

[68]:
```
Person = namedtuple('Person', ['firstname', 'lastname', '_age', 'ssn'])
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-68-cc651156ccc1> in <module>()
----> 1 Person = namedtuple('Person', ['firstname', 'lastname', '_age', 'ssn'])

D:\Users\fbapt\Anaconda3\envs\deepdive\lib\collections\__init__.py in␣
 ↪namedtuple(typename, field_names, verbose, rename, module)
    409         if name.startswith('_') and not rename:
    410             raise ValueError('Field names cannot start with an␣
 ↪underscore: '
--> 411                              '%r' % name)
    412         if name in seen:
    413             raise ValueError('Encountered duplicate field name: %r' %␣
 ↪name)

ValueError: Field names cannot start with an underscore: '_age'
```

We can also choose to let the `namedtuple` function replace invalid field names automatically for us, by using the keyword argument `rename`. When we set that argument to `True` (it is `False` by default) it will replace the invalid name using the position (index) of the field, preceded by an underscore:

[69]:
```
Person = namedtuple('Person', ['firstname', 'lastname', '_age', 'ssn'],␣
 ↪rename=True)
```

[70]:
```
eric = Person('Eric', 'Idle', 42, 'unknown')
```

[71]:
```
eric
```

[71]:
```
Person(firstname='Eric', lastname='Idle', _2=42, ssn='unknown')
```

As you can see the invalid field name `_y` was replaced by `_1` since it was the second element (i.e. index of `1`)

Named Tuple Internals We can easily find out the fields in a named tuple using the `_fields` property:

[72]:
```
Point2D._fields
```

[72]:
```
('x', 'y')
```

[73]:
```
Stock._fields
```

[73]:
```
('symbol', 'year', 'month', 'day', 'open', 'high', 'low', 'close')
```

There is also a property, `_source` that allows us to see exactly the class that was generated by calling `namedtuple` (remember that `namedtuple` is a class **factory**):

[74]:
```
print(Point2D._source)
```

```
from builtins import property as _property, tuple as _tuple
from operator import itemgetter as _itemgetter
from collections import OrderedDict

class Point2D(tuple):
    'Point2D(x, y)'

    __slots__ = ()

    _fields = ('x', 'y')

    def __new__(_cls, x, y):
        'Create new instance of Point2D(x, y)'
        return _tuple.__new__(_cls, (x, y))

    @classmethod
    def _make(cls, iterable, new=tuple.__new__, len=len):
        'Make a new Point2D object from a sequence or iterable'
        result = new(cls, iterable)
        if len(result) != 2:
            raise TypeError('Expected 2 arguments, got %d' % len(result))
        return result

    def _replace(_self, **kwds):
        'Return a new Point2D object replacing specified fields with new values'
        result = _self._make(map(kwds.pop, ('x', 'y'), _self))
        if kwds:
            raise ValueError('Got unexpected field names: %r' % list(kwds))
        return result

    def __repr__(self):
        'Return a nicely formatted representation string'
        return self.__class__.__name__ + '(x=%r, y=%r)' % self

    def _asdict(self):
        'Return a new OrderedDict which maps field names to their values.'
        return OrderedDict(zip(self._fields, self))

    def __getnewargs__(self):
        'Return self as a plain tuple.  Used by copy and pickle.'
        return tuple(self)

    x = _property(_itemgetter(0), doc='Alias for field number 0')
```

```
    y = _property(_itemgetter(1), doc='Alias for field number 1')
```

And of course this will be slightly different for another named tuple generated class:

[75]:
```
print(Person._source)
```

```
from builtins import property as _property, tuple as _tuple
from operator import itemgetter as _itemgetter
from collections import OrderedDict

class Person(tuple):
    'Person(firstname, lastname, _2, ssn)'

    __slots__ = ()

    _fields = ('firstname', 'lastname', '_2', 'ssn')

    def __new__(_cls, firstname, lastname, _2, ssn):
        'Create new instance of Person(firstname, lastname, _2, ssn)'
        return _tuple.__new__(_cls, (firstname, lastname, _2, ssn))

    @classmethod
    def _make(cls, iterable, new=tuple.__new__, len=len):
        'Make a new Person object from a sequence or iterable'
        result = new(cls, iterable)
        if len(result) != 4:
            raise TypeError('Expected 4 arguments, got %d' % len(result))
        return result

    def _replace(_self, **kwds):
        'Return a new Person object replacing specified fields with new values'
        result = _self._make(map(kwds.pop, ('firstname', 'lastname', '_2',
'ssn'), _self))
        if kwds:
            raise ValueError('Got unexpected field names: %r' % list(kwds))
        return result

    def __repr__(self):
        'Return a nicely formatted representation string'
        return self.__class__.__name__ + '(firstname=%r, lastname=%r, _2=%r,
ssn=%r)' % self

    def _asdict(self):
        'Return a new OrderedDict which maps field names to their values.'
        return OrderedDict(zip(self._fields, self))
```

```
    def __getnewargs__(self):
        'Return self as a plain tuple.  Used by copy and pickle.'
        return tuple(self)

    firstname = _property(_itemgetter(0), doc='Alias for field number 0')

    lastname = _property(_itemgetter(1), doc='Alias for field number 1')

    _2 = _property(_itemgetter(2), doc='Alias for field number 2')

    ssn = _property(_itemgetter(3), doc='Alias for field number 3')
```

Converting Named Tuples to Dictionaries The `namedtuple` generated class also provides us an instance method, `_asdict()` that will create a dictionary from all the fields in the named tuple:

[76]:
```
eric._asdict()
```

[76]:
```
OrderedDict([('firstname', 'Eric'),
             ('lastname', 'Idle'),
             ('_2', 42),
             ('ssn', 'unknown')])
```

Technically, it is an `OrderedDict` which we will cover in later section. Basically an `OrderedDict` is a dictionary that, unlike the standard built-in `Dictionary` is **guaranteed** to preserve the order of the keys.

[**Note** that as of Python 3.6, regular dictionaries **do** preserve the order of the keys, but until just recently it was not **guaranteed** and was bascially an implementation detail.

However, this has now changed!! Guido van Rossum has now agreed that this is no longer an implementation detail, and starting in Python 3.7 dictionary order is guaranteed. Since it is actually already the case in Python 3.6, you can now safely assume this fact - as long as you are running your code under Python 3.6 or higher. Your code will break if you rely on dictionary order prior to 3.6, in that case, still use an `OrderedDict`]

Overhead of Named Tuples At this point you may be wondering whether there's more overhead to using a named tuple vs a regular tuple.

There is, but it is tiny. The field names are stored in the **class**, not every instance of the named tuples. This means that the overhead incurred by the field names for one instance of the named tuple vs 1000 instances is the same. Otherwise, the instances are tuples, so you can access contained objects using indexing, slicing and iteration just as if it were a plain tuple. No overhead there either. Looking up values by name do have some overhead of course, but no more than if you had created a custom class.

0.0.3 Named Tuples - Modifying and Extending

[1]:
```
from collections import namedtuple
```

[2]:
```
Point2D = namedtuple('Point2D', 'x y')
```

The objects generated by `namedtuple` generated classes are **immutable**.

In other words the following will not work:

[3]:
```
origin = Point2D(10,0)
```

[4]:
```
origin.x = 0
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-4-ebd2d3bb5d60> in <module>()
----> 1 origin.x = 0

AttributeError: can't set attribute
```

However, we may want to "change" the value of one of the coordinates of our `origin` variable.

This is just like strings, we have to create a new version of the tuple, and assign it to the same label.

Suppose we want to change the x-coordinate of our `origin` to something else, but retain whatever the y-coordinate was.

We could do it as follows:

[5]:
```
origin = Point2D(0, origin.y)
```

[6]:
```
origin
```

[6]:
```
Point2D(x=0, y=0)
```

Of course this could become quite unwieldy when we have a larger number of properties and we only need to change a single item:

[7]:
```
Stock = namedtuple('Stock', 'symbol year month day open high low close')
```

[8]:
```
djia = Stock('DJIA', 2018, 1, 25, 26_313, 26_458, 26_260, 26_393)
```

To update the `close` property for example, we could write:

[9]:
```
djia = Stock(djia.symbol, djia.year, djia.month, djia.day,
                  djia.open, djia.high, djia.low, 26_394)
```

Now that was quite painful!

We can be a bit more clever about this and use tuple unpacking and argument unpacking as follows:

[10]:
```
*values, _ = djia
```

We didn't care about the `close` price since we are replacing it, hence the underscore variable name.

And we now have everything else in a list:

[11]:
```
values
```

[11]:
```
['DJIA', 2018, 1, 25, 26313, 26458, 26260]
```

And now we are going to use the `*` again, but this time to unpack the list into separate arguments when we call the `Stock` initializer:

[12]:
```
djia = Stock(*values, 26_393)
```

[13]:
```
djia
```

[13]:
```
Stock(symbol='DJIA', year=2018, month=1, day=25, open=26313, high=26458,
low=26260, close=26393)
```

This is much better than our first attempt!

But this approach does not always work, what happens if we want to change a values somewhere in the middle? Or two values?

We cannot do: `*first, month, *last = djia`

That would make no sense whatsoever! (and Python will tell you so!)

Maybe slicing and unpacking can work here...

[14]:
```
djia
```

[14]:
```
Stock(symbol='DJIA', year=2018, month=1, day=25, open=26313, high=26458,
low=26260, close=26393)
```

We could try **slicing**:

[15]:
```
djia[:3]
```

[15]:
```
('DJIA', 2018, 1)
```

[16]:
```
djia[:3] + (26,) + djia[4:]
```

[16]:
```
('DJIA', 2018, 1, 26, 26313, 26458, 26260, 26393)
```

So now we could use this to create a new StockPrice instance:

[17]:
```
djia2 = Stock(*(djia[:3] + (26,) + djia[4:]))
```

[18]:
```
djia2
```

[18]:
```
Stock(symbol='DJIA', year=2018, month=1, day=26, open=26313, high=26458,
low=26260, close=26393)
```

This works, but that's quite cumbersome…

And it gets worse - suppose we want to modify the year and day using this approach:

[19]:
```
djia
```

[19]:
```
Stock(symbol='DJIA', year=2018, month=1, day=25, open=26313, high=26458,
low=26260, close=26393)
```

[20]:
```
values = djia[0:1] + (2019,) + djia[2:3] + (26,) + djia[4:]
```

[21]:
```
values
```

[21]:
```
('DJIA', 2019, 1, 26, 26313, 26458, 26260, 26393)
```

[22]:
```
djia3 = Stock(*values)
```

[23]:
```
djia3
```

[23]:
```
Stock(symbol='DJIA', year=2019, month=1, day=26, open=26313, high=26458,
low=26260, close=26393)
```

Or, if you want to avoid unpacking the `values` into the multiple positional arguments required by the `Stock` constructor, we can make us of the `_make` class method that can use an iterable:

[24]:
```
djia4 = Stock._make(values)
```

[25]:
```
djia4
```

[25]:
```
Stock(symbol='DJIA', year=2019, month=1, day=26, open=26313, high=26458,
low=26260, close=26393)
```

This is really getting too complex.

Fortunately there's a better way!

The namedtuple implementation also provides another instance method called `_replace` which takes keyword-only arguments. That method will make a copy of the current tuple and substitute property values based on the keyword-only arguments passed in.

[26]:
```
djia
```

[26]:
```
Stock(symbol='DJIA', year=2018, month=1, day=25, open=26313, high=26458,
low=26260, close=26393)
```

[27]:
```
id(djia)
```

[27]:
```
2785020879400
```

[28]:
```
djia5 = djia._replace(year=2019, day=26)
```

[29]:
```
djia5
```

[29]:
```
Stock(symbol='DJIA', year=2019, month=1, day=26, open=26313, high=26458,
low=26260, close=26393)
```

[30]:
```
djia
```

[30]:
```
Stock(symbol='DJIA', year=2018, month=1, day=25, open=26313, high=26458,
low=26260, close=26393)
```

[31]:
```
id(djia5)
```

[31]:
```
2785020880480
```

Much better!!

Extending Named Tuples Sometimes we may want to add one or more properties to an existing class without modifying the code for the custom class itself.

Using inheritance is one way to go about it so you may be tempted to do this with named tuples as well, but it's not easy, and there's a cleaner way to do this if all you're after is additional data fields.

Let's say we have a Point class that is for 2D problems:

[32]:
```
Point2D = namedtuple('Point2D', 'x y')
```

We could easily create a 3D point class as follows:

[33]:
```
Point3D = namedtuple('Point3D', 'x y z')
```

But if our named tuple has many fields, such as our `Stock` named tuple that's a little more difficult:

[34]:
```
djia
```

[34]:
```
Stock(symbol='DJIA', year=2018, month=1, day=25, open=26313, high=26458,
low=26260, close=26393)
```

Suppose we want to create a new class, say `StockExt`, it would take some effort:

[35]:
```
StockExt = namedtuple('StockExt',
                      '''symbol year month day open high low
                      close previous_close''')
```

Instead we can leverage that `_fields` property:

[36]:
```
Stock._fields
```

[36]:
```
('symbol', 'year', 'month', 'day', 'open', 'high', 'low', 'close')
```

Remember that the `namedtuple` initializer can handle a list or tuple containing the field names. For example, the one we just retrieved from `_fields`.

Now all we need to do is create a new tuple that contains those fields along with whatever extras we want:

[37]:
```
new_fields = Stock._fields + ('previous_close',)
```

[38]:
```
new_fields
```

[38]:
```
('symbol',
 'year',
 'month',
 'day',
 'open',
 'high',
 'low',
 'close',
 'previous_close')
```

And now we can create our new named tuple this way:

[39]:
```
StockExt = namedtuple('StockExt', Stock._fields + ('previous_close',))
```

[40]:
```
StockExt._fields
```

[40]:
```
('symbol',
 'year',
 'month',
 'day',
 'open',
 'high',
 'low',
 'close',
 'previous_close')
```

If you did not want to use tuple concatenation for some reason, you could also do it using strings:

[41]:
```
' '.join(Stock._fields) + ' previous_close'
```

[41]:
```
'symbol year month day open high low close previous_close'
```

[42]:
```
StockExt = namedtuple('StockExt',
                      ' '.join(Stock._fields) + ' previous_close')
```

[43]:
```
StockExt._fields
```

[43]:
```
('symbol',
 'year',
 'month',
 'day',
 'open',
 'high',
 'low',
 'close',
 'previous_close')
```

Now, with this newly extended class, we may want to take one of the "old" named tuple instance (`djia`) and create the extended version of it using the `StockExt` class.

This is also quite simple to do, since named tuples are tuples, and can therefore be unpacked in the arguments of a function call.

[44]:
```
djia
```

[44]:
```
Stock(symbol='DJIA', year=2018, month=1, day=25, open=26313, high=26458,
low=26260, close=26393)
```

[45]:
```
djia_ext = StockExt(*djia, 25_000)
```

[46]:
```
djia_ext
```

[46]:
```
StockExt(symbol='DJIA', year=2018, month=1, day=25, open=26313, high=26458,
low=26260, close=26393, previous_close=25000)
```

or, we can use the `_make` method:

[47]:
```
djia_ext = StockExt._make(djia + (25_000, ))
```

[48]:
```
djia_ext
```

[48]:
```
StockExt(symbol='DJIA', year=2018, month=1, day=25, open=26313, high=26458,
low=26260, close=26393, previous_close=25000)
```

0.0.4 Named Tuples - DocStrings and Default Values

[1]:
```
from collections import namedtuple
```

Adding DocStrings to Named Tuples This is easy to do, both with the generated class, as well as it's properties.

[2]:
```
Point2D = namedtuple('Point2D', 'x y')
```

[3]:
```
Point2D.__doc__ = 'Represents a 2D Cartesian coordinate'
```

And we can even add docstrings to the properties:

[4]:
```
Point2D.x.__doc__ = 'x-coordinate'
Point2D.y.__doc__ = 'y-coordinate'
```

[5]:
```
help(Point2D)
```

```
Help on class Point2D in module __main__:

class Point2D(builtins.tuple)
 |  Represents a 2D Cartesian coordinate
 |
 |  Method resolution order:
 |      Point2D
 |      builtins.tuple
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __getnewargs__(self)
 |      Return self as a plain tuple.  Used by copy and pickle.
 |
 |  __repr__(self)
 |      Return a nicely formatted representation string
 |
 |  _asdict(self)
 |      Return a new OrderedDict which maps field names to their values.
 |
 |  _replace(_self, **kwds)
 |      Return a new Point2D object replacing specified fields with new values
 |
 |  ----------------------------------------------------------------------
 |  Class methods defined here:
 |
 |  _make(iterable, new=<built-in method __new__ of type object at
0x00000000595CB160>, len=<built-in function len>) from builtins.type
 |      Make a new Point2D object from a sequence or iterable
 |
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |
```

```
|  __new__(_cls, x, y)
|      Create new instance of Point2D(x, y)
|
|  ----------------------------------------------------------------------
|  Data descriptors defined here:
|
|  x
|      x-coordinate
|
|  y
|      y-coordinate
|
|  ----------------------------------------------------------------------
|  Data and other attributes defined here:
|
|  _fields = ('x', 'y')
|
|  _source = "from builtins import property as _property, tupl…_itemget…
|
|  ----------------------------------------------------------------------
|  Methods inherited from builtins.tuple:
|
|  __add__(self, value, /)
|      Return self+value.
|
|  __contains__(self, key, /)
|      Return key in self.
|
|  __eq__(self, value, /)
|      Return self==value.
|
|  __ge__(self, value, /)
|      Return self>=value.
|
|  __getattribute__(self, name, /)
|      Return getattr(self, name).
|
|  __getitem__(self, key, /)
|      Return self[key].
|
|  __gt__(self, value, /)
|      Return self>value.
|
|  __hash__(self, /)
|      Return hash(self).
|
|  __iter__(self, /)
|      Implement iter(self).
```

```
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __len__(self, /)
 |      Return len(self).
 |
 |  __lt__(self, value, /)
 |      Return self<value.
 |
 |  __mul__(self, value, /)
 |      Return self*value.n
 |
 |  __ne__(self, value, /)
 |      Return self!=value.
 |
 |  __rmul__(self, value, /)
 |      Return self*value.
 |
 |  count(…)
 |      T.count(value) -> integer -- return number of occurrences of value
 |
 |  index(…)
 |      T.index(value, [start, [stop]]) -> integer -- return first index of
value.
 |      Raises ValueError if the value is not present.
```

Adding Default Values to Named Tuples

Using a Prototype This technique is in the Python docs, and uses the concept of creating a prototype object that has the default values set:

[6]:
```
Vector = namedtuple('Vector', 'x1 y1 x2 y2 origin_x origin_y')
```

[7]:
```
vector_zerooorigin = Vector(x1=None, y1=None, x2=None, y2=None, origin_x=0,␣
↪origin_y=0)
```

[8]:
```
vector_zerooorigin
```

[8]:
```
Vector(x1=None, y1=None, x2=None, y2=None, origin_x=0, origin_y=0)
```

The named tuple `vector_zerooorigin` is now a prototype of a vector with zero origin.

To create new vectors using that origin as a default, we no longer use the `Vector` class, but instead use `_replace` as follows:

[9]:
```
v1 = vector_zerooorigin._replace(x1=1, y1=1, x2=10, y2=10)
```

[10]:
```
v1
```

[10]:
```
Vector(x1=1, y1=1, x2=10, y2=10, origin_x=0, origin_y=0)
```

This certainly works, and can be useful in cases where you may want more than one prototype (e.g. `vector_zerooorigin` and `vector_otherorigin`)

Using `__defaults__` There is an alternative way of doing this. And, in my opinion, a much cleaner alternative.

In Python the default values for a function's parameters are stored as a tuple in the `__defaults__` attribute.

[11]:
```
def func(a, b=20, c=30):
    print(a, b, c)
```

[12]:
```
func.__defaults__
```

[12]:
```
(20, 30)
```

[13]:
```
func(10)
```

```
10 20 30
```

But the `__defaults__` property is writable:

[14]:
```
func.__defaults__ = (200, 300)
```

[15]:
```
func(10)
```

```
10 200 300
```

In this case, the function we are interested in specifying default values for, is the named tuple class constructor, i.e. `__new__`.

So, we will simply need to set `Vector.__new__.__defaults__` to the desired tuple of default values.

The only thing to note is that if you specify less default values (say `m` values) than the total number of arguments (say `n` values, where `m < n`), then the defaults will apply to the **last** `m` values. Think of it as writing out your field names and default values on two lines, and right-aligning them. (If you specify more, then the values at the beginning are effectively ignored)

[16]:
```
Vector.__new__.__defaults__ = (0, 0)
```

Here I am basically setting default values for the last two elements only, i.e. `origin_x` and `origin_y`.

[17]:
```
v1 = Vector(0, 0, 10, 10, -10, -10)
```

[18]:
```
v1
```

[18]:
```
Vector(x1=0, y1=0, x2=10, y2=10, origin_x=-10, origin_y=-10)
```

[19]:
```
v2 = Vector(5, 5, 20, 20)
```

[20]:
```
v2
```

[20]:
```
Vector(x1=5, y1=5, x2=20, y2=20, origin_x=0, origin_y=0)
```

[21]:
```
v3 = Vector(x1=1, y1=1, x2=10, y2=10)
```

[22]:
```
v3
```

[22]:
```
Vector(x1=1, y1=1, x2=10, y2=10, origin_x=0, origin_y=0)
```

An even simpler way to set default values if you want **all** the defaults to be the same:

[23]:
```
Vector.__new__.__defaults__ = (0,) * len(Vector._fields)
```

[24]:
```
v5 = Vector()
```

[25]:
```
v5
```

[25]:
```
Vector(x1=0, y1=0, x2=0, y2=0, origin_x=0, origin_y=0)
```

Of course, the usual admonishment of not using mutable default values holds here as well.

0.0.5 Named Tuples - Application - Alternative to Dictionaries

First an important caveat: all this really only works for dictionaries with **string** keys. Dictionary keys can be other hashable data types, (including tuples, as long as they contain hashable types in turn), and these examples will not work with those types of dictionaries.

[4]:
```
from collections import namedtuple
```

[11]:
```
data_dict = dict(key1=100, key2=200, key3=300)
```

[12]:
```
Data = namedtuple('Data', data_dict.keys())
```

[13]:
```
Data._fields
```

[13]:
```
('key1', 'key2', 'key3')
```

Now we can create an instance of the `Data` named tuple using the data in the `data_dict` dictionary.

We could try the following (bad idea):

[15]:
```
d1 = Data(*data_dict.values())
```

[16]:
```
d1
```

[16]:
```
Data(key1=100, key2=200, key3=300)
```

This looks like it worked.

But consider this second dictionary, where we do not create the keys in the same order:

[35]:
```
data_dict_2 = dict(key1=100, key3=300, key2=200)
```

[36]:
```
d2 = Data(*data_dict_2.values())
```

[37]:
```
d2
```

[37]:
```
Data(key1=100, key2=300, key3=200)
```

Obviously this went terribly wrong!

We cannot guarantee that the order of `values()` will be in the same order as the keys (in our named tuple and in the dictionary).

Instead, we should unpack the dictionary itself, resulting in keyword arguments that will be passed to the `Data` constructor:

[38]:
```
d2 = Data(**data_dict_2)
```

[39]:
```
d2
```

[39]:
```
Data(key1=100, key2=200, key3=300)
```

So, the pattern to create a named tuple out of a single dictionary is straightforward:

For any dictionary `d` we can created a named tuple class and insert the data into it as follows:

1. `Struct = namedtuple('Struct', d.keys())`

2. `data = Struct(**d)`

Because dictionaries now preserve key order, the order of the fields in the named tuple structure will be the same. If you want your fields to be sorted in a different way, just sort the keys when you create the named tuple class. For example, to have keys sorted alphabetically we could do:

[40]:
```
data_dict = dict(first_name='John', last_name='Cleese', age=42, complaint='dead␣
↪parrot')
```

[41]:
```
data_dict.keys()
```

[41]:
```
dict_keys(['first_name', 'last_name', 'age', 'complaint'])
```

[44]:
```
sorted(data_dict.keys())
```

[44]:
```
['age', 'complaint', 'first_name', 'last_name']
```

[45]:
```
Struct = namedtuple('Struct', sorted(data_dict.keys()))
```

[46]:
```
Struct._fields
```

[46]:
```
('age', 'complaint', 'first_name', 'last_name')
```

Of course we can still put in the correct values from the dictionary into the correct slots in the tuple by unpacking the dictionary instead of just the values:

[48]:
```
d1 = Struct(**data_dict)
```

[49]:
```
d1
```

[49]:
```
Struct(age=42, complaint='dead parrot', first_name='John', last_name='Cleese')
```

And of course, since this is now a named tuple we can access the data using the field name:

[50]:
```
d1.complaint
```

[50]:
```
'dead parrot'
```

instead of how we would have done it with the dictionary:

[51]:
```
data_dict['complaint']
```

[51]:
```
'dead parrot'
```

I also want to point out that with dictionaries we often end up with code where the key is stored in some variable and then referenced this way:

[53]:
```
key_name = 'age'
data_dict[key_name]
```

[53]:
```
42
```

We cannot use this approach directly with named tuples however. For example this will not work:

[54]:
```
key_name = 'age'
d1.key_name
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-54-f110bbdbc0a7> in <module>()
      1 key_name = 'age'
----> 2 d1.key_name

AttributeError: 'Struct' object has no attribute 'key_name'
```

However, we can use the `getattr` function that we have seen before:

[57]:
```
key_name = 'age'
getattr(d1, key_name)
```

[57]:
```
42
```

We also have the `get` method on dictionaries that can specify a default value to return if the key does not exist:

[59]:
```
data_dict.get('age', None), data_dict.get('invalid_key', None)
```

[59]:
```
(42, None)
```

And we can do the same with the `getattr` function:

[60]:
```
getattr(d1, 'age', None), getattr(d1, 'invalid_field', None)
```

[60]:
```
(42, None)
```

Now this is not very useful if you are only working with a single instance of a dictionary that has the same set of keys. Kind of pointless really.

You also do not want to create a new named tuple for every instance of a dictionary - that would just be way too much overhead.

But in cases where you have a collection of dictionaries that share a common set of keys, this can be really useful, as long as you are willing to live with the fact that you now have immutable structures.

Let's suppose we have this data list:

[3]:
```
data_list = [
    {'key1': 1, 'key2': 2},
    {'key1': 3, 'key2': 4},
    {'key1': 5, 'key2': 6, 'key3': 7},
    {'key2': 100}
]
```

The first thing to note is that we need to figure out all the possible keys that have been used in the dictionaries in this list.

The easiest way to do this is to extract all the keys of all the dictionaries and then make a `set` out of them, to eliminate duplicate key names:

We could do it this way, using a simple loop:

[79]:
```
keys = set()
for d in data_list:
    for key in d.keys():
        keys.add(key)
```

[80]:
```
keys
```

[80]:
```
{'key1', 'key2', 'key3'}
```

But actually a more efficient way would be to use a comprehension:

[110]:
```
keys = {key for dict_ in data_list for key in dict_.keys()}
```

[111]:
```
keys
```

[111]:
```
{'key1', 'key2', 'key3'}
```

In fact, we can also use the fact that we can union multiple sets (we'll cover this in detail later) by unpacking all the keys and creating a union of them:

[114]:
```
keys = set().union(*(dict_.keys() for dict_ in data_list))
```

[115]:
```
keys
```

[115]:
```
{'key1', 'key2', 'key3'}
```

However you do it, we end up with a set of all the possible keys used in our list of dictionaries.

Now we can go ahead and create a named tuple with all those keys as fields:

[117]:
```
Struct = namedtuple('Struct', keys)
```

[118]:
```
Struct._fields
```

[118]:
```
('key3', 'key2', 'key1')
```

As you can see, sets do not preserve order, so in this case we'll probably sort the keys to create our named tuple:

[119]:
```
Struct = namedtuple('Struct', sorted(keys))
```

[120]:
```
Struct._fields
```

[120]:
```
('key1', 'key2', 'key3')
```

Now, we're also going to provide default values, since not all dictionaries have all the keys in them. In this case I'm going to set the default to `None` if the key is missing:

[121]:
```
Struct.__new__.__defaults__ = (None,) * len(Struct._fields)
```

Now we're ready to load up all these dictionaries into a new list of named tuples:

[122]:
```
tuple_list = [Struct(**dict_) for dict_ in data_list]
```

[123]:
```
tuple_list
```

[123]:
```
[Struct(key1=1, key2=2, key3=None),
 Struct(key1=3, key2=4, key3=None),
 Struct(key1=5, key2=6, key3=7),
 Struct(key1=None, key2=100, key3=None)]
```

So lastly, let's just package this all up neatly into a single function that will take an iterable of dictionaries, or an arbitrary number of dictionaries as positional arguments, and return a list of named tuples:

[5]:
```
def tuplify_dicts(dicts):
    keys = {key for dict_ in dicts for key in dict_.keys()}
    Struct = namedtuple('Struct', keys)
    Struct.__new__.__defaults__ = (None,) * len(Struct._fields)
    return [Struct(**dict_) for dict_ in dicts]
```

[6]:
```
tuplify_dicts(data_list)
```

[6]:
```
[Struct(key1=1, key2=2, key3=None),
 Struct(key1=3, key2=4, key3=None),
 Struct(key1=5, key2=6, key3=7),
 Struct(key1=None, key2=100, key3=None)]
```

Isn't Python wonderful? :-)

0.0.6 Named Tuples - Application - Returning Multiple Values

We already know that we can easily return multiple values from a function by using a tuple:

[21]:
```
from random import randint, random

def random_color():
    red = randint(0, 255)
    green = randint(0,255)
    blue = randint(0, 255)
    alpha = round(random(), 2)
    return red, green, blue, alpha
```

[23]:
```
random_color()
```

[23]:
```
(97, 254, 97, 0.06)
```

So of course, we could call the function this and unpack the results at the same time:

[25]:
```
red, green, blue, alpha = random_color()
```

[26]:
```
print(f'red={red}, green={green}, blue={blue}, alpha={alpha}')
```

```
red=42, green=178, blue=69, alpha=0.7
```

But it might be nicer to use a named tuple:

[27]:
```
from collections import namedtuple
```

[28]:
```
Color = namedtuple('Color', 'red green blue alpha')

def random_color():
    red = randint(0, 255)
    green = randint(0,255)
    blue = randint(0, 255)
    alpha = round(random(), 2)
    return Color(red, green, blue, alpha)
```

[29]:
```
color = random_color()
```

[30]:
```
color.red
```

[30]:
```
5
```

[31]:
```
color
```

[31]:
```
Color(red=5, green=210, blue=143, alpha=0.06)
```