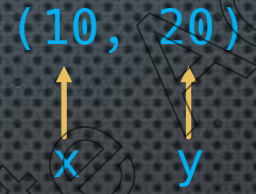
NAMED TUPLES

Tuple as Data Structure

We have seen how we interpreted tuples as data structures

The position of the object contained in the tuple gave it meaning

For example, we can represent a 2D coordinate as:



If `pt` is a position tuple, we can retrieve the `x` and `y` coordinates using:

```
x, y = pt
```

or

```
x = pt[0]
y = pt[1]
```

So, for example, to calculate the distance of `pt` from the origin we could write:

```
dist = math.sqrt(pt[0] ** 2 + pt[1] ** 2)
```

Now this is not very readable, and if someone sees this code they will have to know that `pt[0]` means the x-coordinate and `pt[1]` means the y-coordinate.

This is not very transparent.

Using a Class Instead

At this point, in order to make things clearer for the reader (not the compiler, the reader), we might want to approach this using a class instead.

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

```
pt = Point2D(10, 20)

distance = sqrt(pt.x ** 2 + pt.y ** 2)
```

```
class Stock:
    def __init__(self, symbol, year, month, day, open, high, low, close):
        self.symbol = symbol
        self.year = year
        self.month = month
        self.day = day
        self.open = open
        self.high = high
        self.low = low
        self.close = close
```

Class Approach	Tuple Approach
djia.symbol	djia[0]
djia.open	djia[4]
djia.close	djia[7]
djia.high – djia.low	djia[5] – djia[6]

Extra Stuff

At the very least we should implement the `__repr__` method

→ `Point(x=10, y=20)`

We probably should implement the `__eq__` method too

→ `Point(10, 20) == Point(10, 20)` → `True`

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

    def __repr__(self):
        return f'Point2D(x={self.x}, y={self.y}'

    def __eq__(self, other):
        if isinstance(other, Point2D):
            return self.x == other.x and self.y == other.y
        else:
            return False
```

Named Tuples to the rescue

There are other reasons to seek another approach. I cover some of those in the coding video

Amongst other things, Point2D objects are mutable – something we may not want!

There's a lot to like using tuples to represent simple data structures

The real drawback is that we have to know what the positions mean, and remember this in our code

If we ever need to change the structure of our tuple in our code (like inserting a value that we forgot)

most likely our code will break!

```
eric = ('Idle', 42)
last_name, age = eric
```

```
eric = ('Eric', 'Idle', 42)
last_name, age = eric
```

Broken!!

Class approach:

```
last_name = eric.last_name
age = eric.age
```

Named Tuples to the rescue

So what if we could somehow combine these two approaches, essentially creating tuples where we can, in addition, give meaningful names to the positions?

That's what `namedtuples` essentially do

They subclass `tuple`, and add a layer to assign property names to the positional elements

Located in the `collections` standard library module

```
from collections import namedtuple
```

`namedtuple` is a function which generates a new class → class factory

that new class inherits from `tuple`

but also provides named properties to access elements of the tuple

but an instance of that class is still a tuple

Generating Named Tuple Classes

We have to understand that `namedtuple` is a class factory

When we use it, we are essentially creating a new class, just as if we had used `class` ourselves

`namedtuple` needs a few things to generate this class:

- the class name we want to use
- a sequence of field names (strings) we want to assign, in the order of the elements in the tuple

  field names can be any valid variable name
  except that they cannot start with an underscore

The return value of the call to `namedtuple` will be a class

We need to assign that class to a variable name in our code so we can use it to construct instances

In general, we use the same name as the name of the class that was generated

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

Generating Named Tuple Classes

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

We can create instances of `Point2D` just as we would with any class (since it is a class)

```
pt = Point2D(10, 20)
```

The variable name that we use to assign to the class generated and returned by `namedtuple` is arbitrary

```
Pt2D = namedtuple('Point2D', ['x', 'y'])
pt = Pt2D(10, 20)
```

Generating Named Tuple Classes


```
class MyClass:
    pass
```

```
MyClassAlias = MyClass
```

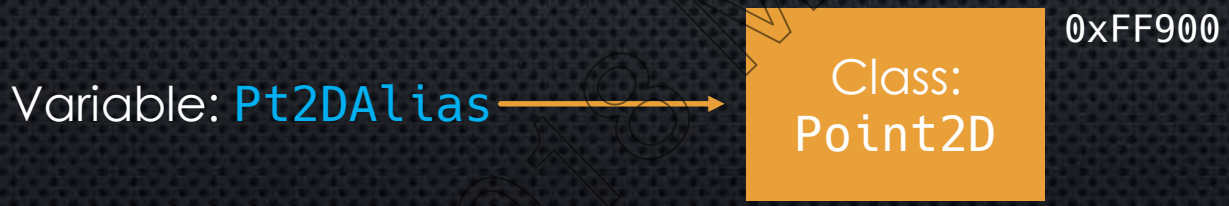
```
instance_1 = MyClass()
```

```
instance_2 = MyClassAlias()
```

instantiates the same class

Similarly

```
Pt2DAlias = namedtuple('Point2D', ['x', 'y'])
```



This is the same concept as aliasing a function, or assigning a lambda function to a variable name!

Generating Named Tuple Classes

There are many ways we can provide the list of field names to the `namedtuple` function

- a list of string
- a tuple of strings
- a single string with the field names separated by whitespace or commas

→ in fact any sequence, just remember that order matters!

```
namedtuple('Point2D', ['x', 'y'])

namedtuple('Point2D', ('x', 'y'))

namedtuple('Point2D', 'x, y')

namedtuple('Point2D', 'x y')
```

Instantiating Named Tuples

After we have created a named tuple class, we can instantiate them just like an ordinary class

In fact, the `__new__` method of the generated class uses the field names we provided as param names

```
Point2D = namedtuple('Point2D', 'x y')
```

We can use positional arguments:

```
pt1 = Point2D(10, 20)        10 → x    20 → y
```

And even keyword arguments:

```
pt2 = Point2D(x=10, y=20)    10 → x    20 → y
```

Accessing Data in a Named Tuple

Since named tuples are also regular tuples, we can still handle them just like any other tuple

- by index
- slice
- iterate

```
Point2D = namedtuple('Point2D', 'x y')

pt1 = Point2D(10, 20)
```

```
isinstance(pt1, tuple) → True
```

```
x, y = pt1

x = pt1[0]

for e in pt1:
    print(e)
```

Accessing Data in a Named Tuple

But now, in addition, we can also access the data using the field names:

```
Point2D = namedtuple('Point2D', 'x y')
pt1 = Point2D(10, 20)
```

```
pt1.x   → 10
pt1.y   → 20
```

Since namedtuple generated classes inherit from tuple

```
class Point2D(tuple):
    …
```

`pt1` **is** a `tuple`, and is therefore immutable

```
pt1.x = 100
```

will not work!

The rename keyword-only argument for namedtuple

Remember that field names for named tuples must be valid identifiers, but cannot start with an underscore

This would not work: `Person = namedtuple('Person', 'name age _ssn')`

`namedtuple` has a keyword-only argument, `rename` (defaults to `False`) that will automatically rename any invalid field name

uses convention: `_{position in list of field names}`

This will now work:

```
Person = namedtuple('Person', 'name age _ssn', rename=True)
```

And the actual field names would be:

```
name    age    _2
```

Introspection

We can easily find out the field names in a named tuple generated class

class property → `_fields`

```
Person = namedtuple('Person', 'name age _ssn', rename=True)

Person._fields     → ('name', 'age', '_2')
```

Introspection

Remember that `namedtuple` is a class factory, i.e. it generates a class

We can actually see what the code for that class is, using the class property `_source`

```
Point2D = namedtuple('Point2D', 'x y')

Point2D._source     →
```

lots of code omitted

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


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

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

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

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

Extracting Named Tuple Values to a Dictionary

Instance method: `_asdict()`

that creates a dictionary of all the named values in the tuple

```
Point2D = namedtuple('Point2D', 'x y')
pt1 = Point2D(10, 20)


pt1._asdict()      → {'x': 10, 'y': 20}
```

Code