==============================================

Namedtuple in Collections Module

==============================================

Index

-------------------------------------

=>What is tuple

=>Indexing in Tuple

=>Why we namedtuple()

=>Syntax for creating namedtuple

=>Arguments to namedtuple()

=>Converting namedtuple Instances Into Dictionaries

=>Converting Dictionaries into namedtuple Instances

=>Replacing Fields in Existing namedtuple Instances

=>Adding Additional namedtuple Attributes

=>Attributes of namedtuple

=>Displaying the Field Names and Field Values by using zip()

=>DataClasses(Mutable and immutable) VS namedtuple

=>Programming Examples

---------------------------------------------------------------------

=>Python’s collections module provides a factory function called namedtuple()

=>namedtuple() designed to make our code more Pythonic when you’re working with tuples.

=>With namedtuple(), we 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.

=>Hence the advantage of namedtuple is that we can access their values using field names and the dot notation.

----------------------------------------------------------

By using Normal tuple object

----------------------------------------------------------

>>> # 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:

=========================================================================

Now Lets start how to use namedtuple

Syntax:- varname=namedtuple(typename, field\_names)

we can't create an empty named tuple.

=========================================================================

>>> from collections import namedtuple

>>> # Create a namedtuple type, Point

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

>>> issubclass(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---------AttributeError: can't set attribute

-------------------

Explanation

------------------

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

==========================================================================================

Let starts Deep discussion about

------------------------------------------------------------------------------------------

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:

--------------------------------------------------------------------------------

>>> 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'])

=>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.

=================================================================================

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:

Syntax:- varname=namedtuple(typename, field\_names)

=>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.

=>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:

-------------------------------------------------------------------

>>> 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)

(OR)

>>> # A string with comma-separated field names

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

>>> Point

<class '\_\_main\_\_.Point'>

>>> Point(4, 8)

Point(x=4, y=8)

OR

>>> # A generator expression for the field names

>>> Point = namedtuple("Point", (field for field in "xy")) # but not (field for field in "x,y")

>>> 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.

==============================================================================

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:

>>> 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}

----------------------------------------------------------------------------------

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:

>>> 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)

----------------------------------------------------------------------------------------------

Exploring Additional namedtuple Attributes

----------------------------------------------------------------------------------------------

Named tuples also have one attribute: .\_fields . This attribute holds a tuple of strings listing the field names.

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

>>> 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

-------------------------------------------------------------------------------------------

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

------------------------------------------------------------------------------------------

>>> 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)

Output

------------

name -> Jane

age -> 26

height -> 1.75

weight -> 67

-------------------------------------------------------------------------------------------

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:

>>> 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'

========================================x=================================================

https://realpython.com/python-namedtuple/

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:

>>> 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:

>>> 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 an object of 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 an .\_\_iter\_\_() special method to Person like this:

>>> 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.

===========================================================================================

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:

>>> 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.