Lecture 2.2 : Tuples

Introduction

- A *tuple* is essentially an *immutable* list.
- Like a list, a tuple is a sequenced collection type. Like a list it can accommodate a collection of arbitrary types.
- Like a string, a tuple is immutable.

Tuple creation

- We create a tuple using the comma operator and, though not strictly necessary, we typically surround the tuple with round brackets to make it obvious it's a tuple.

```python
t = (4, 5, 6)
print(t)
```

```
(4, 5, 6)
```

Immutability

- What does immutable mean in the context of a tuple? It means that once constructed the *top-level contents* of a tuple cannot be modified.

```python
print(t[0])
# We cannot change the top-level contents of a tuple
t[0] += 1
```

```
4



---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [2], in <module>
      1 print(t[0])
      2 # We cannot change the top-level contents of a tuple
----> 3 t[0] += 1

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

- Is this changing the contents of tuple? No. It is creating a *new* tuple from the contents of two existing tuples (it just so happens we assign the name of an existing tuple to the new one):

```python
t = ('a', 'b', 'c')
t += ('d', 'e', 'f')
print(t)
```

```
('a', 'b', 'c', 'd', 'e', 'f')
```

- Why do we say that the *top-level* contents of a tuple cannot be changed rather than simply saying that the contents of a tuple cannot be changed? We refer specifically to the top-level contents in order to make clear that although the contents are immutable, the contents of the contents of a tuple are not necessarily immutable.

```python
t = (['a', 'b', 'c'], ['cat', 'dog'])
t[1].append('fish')
print(t)
```

```
(['a', 'b', 'c'], ['cat', 'dog', 'fish'])
```

Named tuples

- A special case of a tuple is a *named tuple*. Related data can be grouped together as a set of attribute-value pairs to form a named tuple
- Suppose for example that we wish to model a car. A car has several attributes including a make, model and age. We can use a named tuple as follows to represent a single `Car` data type that has each of these attributes:

```python
from collections import namedtuple

# Create a new data type that is named tuple
Car = namedtuple('Car', ['make', 'model', 'age'])
car1 = Car('Opel', 'Astra', 3)
car2 = Car('Mazda', 'MX5', 7)
print(f'{car1.make} {car1.model} {car1.age}')
print(f'{car2.make} {car2.model} {car2.age}')
```

```
Opel Astra 3
Mazda MX5 7
```

Uses of tuples

- Because they are immutable we often use tuples to store constants or values that we do not want our program to ever change.
- As we'll see later, when a function has multiple values to return to its caller, it will typically return them in a tuple

- Only immutables can be serve as dictionary keys and, being immutable, tuples can be used in this context.
- We can take advantage of tuples and multiple assignment to swap two values without using a temporary variable:

```python
a = 3
b = 7
print(f'a={a}, b={b}')
(b, a) = (a, b)
print(f'a={a}, b={b}')
```

```
a=3, b=7
a=7, b=3
```

Tuple methods

- Apart from in those respects listed above, tuples behave similarly to lists and support the same indexing, slicing, concatenation, iteration etc. operations.

```python
help(tuple)
```

```
Help on class tuple in module builtins:

class tuple(object)
 |  tuple(iterable=(), /)
 |
 |  Built-in immutable sequence.
 |
 |  If no argument is given, the constructor returns an empty tuple.
 |  If iterable is specified the tuple is initialized from iterable's items.
 |
 |  If the argument is a tuple, the return value is the same object.
 |
 |  Built-in subclasses:
 |      asyncgen_hooks
 |      UnraisableHookArgs
 |
 |  Methods defined here:
 |
 |  __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].
 |
 |  __getnewargs__(self, /)
 |  
 |  __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.
 |  
 |  __ne__(self, value, /)
 |      Return self!=value.
 |  
 |  __repr__(self, /)
 |      Return repr(self).
 |  
 |  __rmul__(self, value, /)
 |      Return value*self.
 |  
 |  count(self, value, /)
 |      Return number of occurrences of value.
 |  
 |  index(self, value, start=0, stop=9223372036854775807, /)
 |      Return first index of value.
 |      
 |      Raises ValueError if the value is not present.
 |  
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |  
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
```