Tuples

Chapter 12

Tuple

- Pronounced as “tuhple” or “too-ple”
- Comes from sequence names *single*, *double*, *triple*, *quadruple*, *quintuple*...

- A sequence of values of any type
- Indexed by integers
- A lot like lists, except that tuples are **immutable**.

Creating a Tuple

- Syntactically, a tuple is a comma-separated list of values:

```
>>> t = ' a' , ' b' , ' c' , ' d' , ' e'
```

- Not necessarily, but often comes in brackets:

```
>>> t = (' a' , ' b' , ' c' , ' d' , ' e' )
```

```
>>> t1 = 'a',          # type?
>>> t2 = ('a')
```

Creating a Tuple (cont.)

```
>>> t = tuple()
>>> t
()

>>> t = tuple('lupins' )   # any sequence goes here
>>> t
(' l' , ' u' , ' p' , ' i' , ' n' , ' s' )
```

Tuple Operators

- Most list operators also work with tuples.

```
>>> t = ('a' , 'b' , 'c' , 'd' , 'e' )
>>> t[0]
'a'

>>> t[1:3]
('b' , 'c' )
```

Tuple are Immutable

```
>>> t[0] = 'A'
TypeError: object doesn't support item assignment

>>> t = ('A',) + t[1:]
>>> t
('A' , 'b' , 'c' , 'd' , 'e' )
```

Relational Operators

- Work with tuples and other sequences. Starts by comparing the first element from each sequence. If they are equal, goes on to the next element, and so on, until finds elements that differ. Subsequent elements are not considered (even if they are really big).

```
>>> (0, 1, 2) < (0, 3, 4)
True

>>> (0, 1, 2000000) < (0, 3, 4)
True
```

Tuple Assignment

- Swapping the values of two variables:

```
a = 1
b = 2

a = b
b = a
```

Tuple Assignment (cont.)

- Swapping requires a temporary variable:

```
a = 1
b = 2

temp = a      # saves old value before updating
a = b
b = temp
```

- Is it possible to do this without creating a temporary variable?

Tuple Assignment (cont.2)

- Swapping using **tuple assignment** is more elegant:

```
a = 1
b = 2

>>> a, b = b, a

>>> a, b = 1, 2, 3
ValueError : too many values to unpack

>>> addr = 'info@sdu.edu.kz'
>>> uname,domain = addr.split('@')   # returns a list
```

Tuples as Return Values

```
>>> t = divmod(7, 3)    # applies both // and %
>>> t
(2, 1)

>>> quot, rem = divmod(7, 3)

def min_max(t):
    return min(t), max(t)   # return two values as one tuple
```

Variable-length Argument Tuples

- Functions can take a variable number of arguments. A parameter name that begins with * **gathers** arguments into a tuple.

```
def printall(*args):     # args is a conventional name
    print(args)

>>> printall(1,2.0,'3') # the function accepts any sequence
(1, 2.0, '3')
```

Variable-length Argument Tuples (cont.)

- The complement of gather is **scatter**. If you have a sequence of values and you want to pass it to a function as multiple arguments, you can use the * operator.

```
>>> t = (7, 3)
>>> divmod(t)
TypeError: divmod expected 2 arguments, got 1

>>> divmod(*t)    # tuple is scattered
(2, 1)

>>> max(1, 2, 3)
3
```

Lists and Tuples

```
>>> s = 'abc'
>>> t = [0, 1, 2]
>>> zip(s, t)    # takes two or more sequences
<zip object at 0x7f7d0a9e7c48>

>>> for pair in zip(s, t):
... print(pair)
...
('a' , 0)
('b' , 1)
('c' , 2)
```

Lists and Tuples (cont.)

- A zip object is a kind of **iterator**, which is any object that iterates through a sequence. Iterators are similar to lists in some ways, but unlike lists, you can't use an index to select an element from an iterator.

- To use list operations:

```
>>> list(zip(s, t))
[('a' , 0), ('b' , 1), ('c' , 2)]
```

- Length matters:

```
>>> list(zip('Anne', 'Elk'))
[('A' , 'E' ), ('n' , 'l' ), ('n' , 'k' )]
```

Lists and Tuples (cont.2)

- You can use tuple assignment in a `for` loop to traverse a list of tuples:

```
t = [('a', 0), ('b', 1), ('c', 2)]   # a list of tuples

for letter,number in t:
    print(number,letter)
```

Lists and Tuples (cont.3)

- `has_match` takes two sequences, `t1` and `t2`, and returns `True` if there is an index `i` such that `t1[i] == t2[i]`:

```
def has_match(t1, t2):
    for x,y in zip(t1, t2):
        if x == y:
            return True
    return False
```

Lists and Tuples (cont.4)

- If you need to traverse the elements of a sequence and their indices, you can use the built-in function `enumerate`:

```
>>> for index,element in enumerate('abc' ):
    print(index,element)

0 a
1 b
2 c
```

Dictionaries and Tuples

```
>>> d = {'a':0, 'b':1, 'c':2}
>>> t = d.items()
>>> t
dict_items([('c', 2), ('a', 0), ('b', 1)])

>>> for key,value in d.items():
... print(key, value)
...
c 2
a 0
b 1
```

Dictionaries and Tuples (cont.)

- Initializing a dictionary from a list of tuples

```
>>> t = [('a', 0), ('c', 2), ('b', 1)]
>>> d = dict(t)
>>> d
{'a' : 0, 'c' : 2, 'b' : 1}

>>> d = dict(zip('abc', range(3)))
>>> d
{'a' : 0, 'c' : 2, 'b' : 1}
```

Dictionaries and Tuples (cont.2)

- Tuple as dictionary key:

```
directory[last, first] = number

for last,first in directory:
    print(first, last, directory[last,first])
```

Sequences of Sequences

- Lists are more common than tuples, mostly because they are mutable. But there are a few cases where you might prefer tuples:

1. In some contexts, like a `return` statement, it is syntactically simpler to create a tuple than a list.

2. If you want to use a sequence as a dictionary key, you have to use an immutable type like a tuple or string.

3. If you are passing a sequence as an argument to a function, using tuples reduces the potential for unexpected behavior due to aliasing.