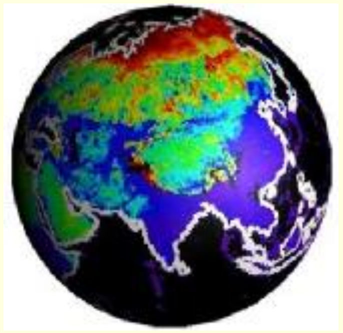


Chapter 12 Tuples

Jihn-Fa Jan, Ph.D.

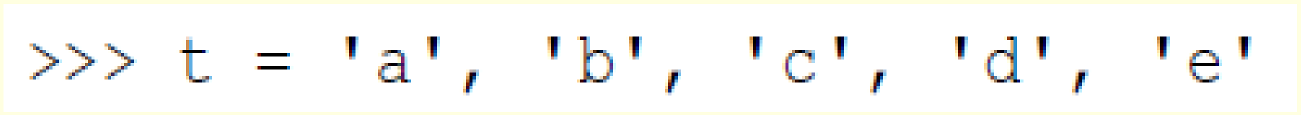
Associate Professor

Department of Land Economics

National Chengchi University

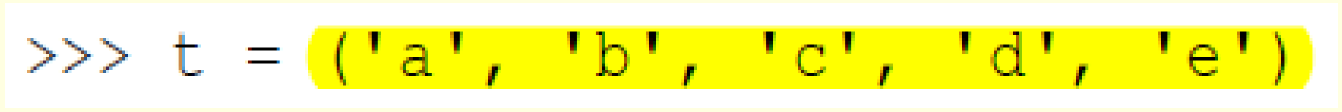
Tuples are immutable

- A tuple is a sequence of values. The values can be any type, and they are indexed by integers, so in that respect tuples are a lot like lists. The important difference is that tuples are immutable.
- Syntactically, a tuple is a comma-separated list of values:



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

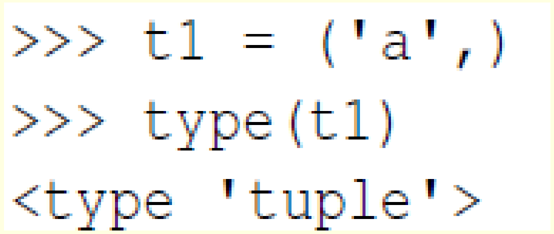
- Although it is not necessary, it is common to enclose tuples in parentheses:



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

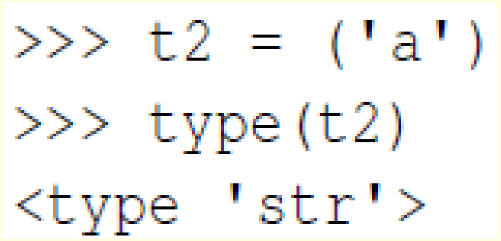
Tuples are immutable (cont'd)

- To create a tuple with a single element, you have to include the final comma:



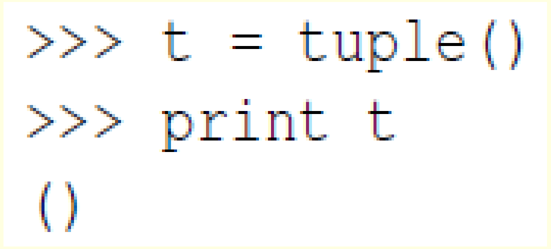
```
>>> t1 = ('a',)
>>> type(t1)
<type 'tuple'>
```

- Without the comma, Python treats ('a') as a string in parentheses:



```
>>> t2 = ('a')
>>> type(t2)
<type 'str'>
```

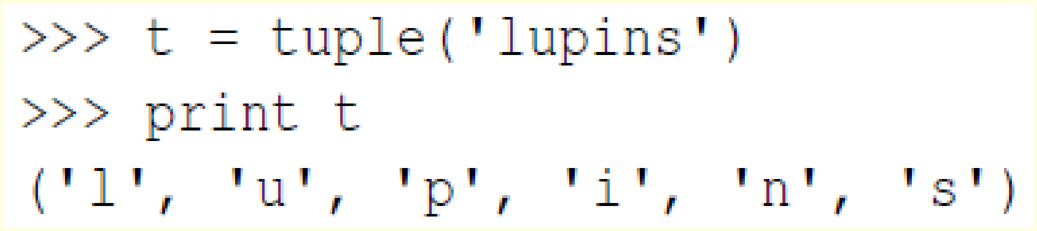
- Creating an empty tuple:



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

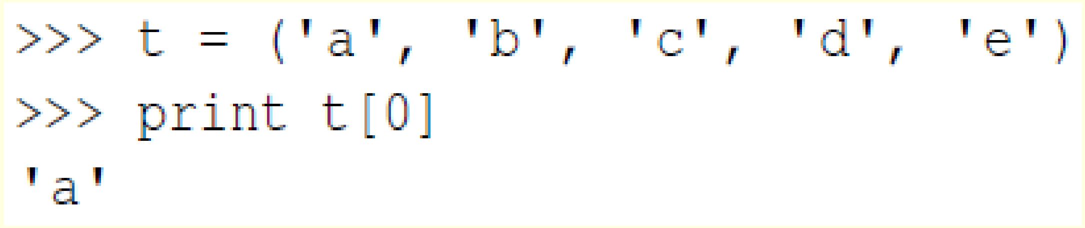
Tuples are immutable (cont'd)

- If the argument is a sequence (string, list or tuple), the result is a tuple with the elements of the sequence:



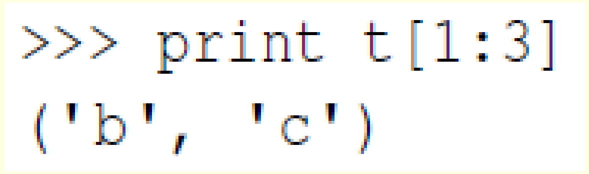
```
>>> t = tuple('lupins')
>>> print t
('l', 'u', 'p', 'i', 'n', 's')
```

- Most list operators also work on tuples. The bracket operator indexes an element:



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

- The slice operator selects a range of elements:



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

Tuples are immutable (cont’d)

- **But if you try to modify one of the elements of the tuple, you get an error:**

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

- **You can’t modify the elements of a tuple, but you can replace one tuple with another:**

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

Tuple assignment

- **It is often useful to swap the values of two variables. With conventional assignments, you have to use a temporary variable. For example, to swap a and b:**

```
>>> temp = a
>>> a = b
>>> b = temp
```

- **This solution is cumbersome; tuple assignment is more elegant:**

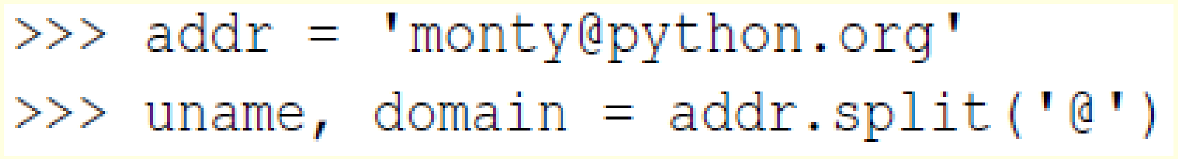
```
>>> a, b = b, a
```

  - All the expressions on the right side are evaluated before any of the assignments.
  - The number of variables on the left and the number of values on the right have to be the same:

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

Tuple assignment (cont'd)

- **More generally, the right side can be any kind of sequence (string, list or tuple). For example, to split an email address into a user name and a domain, you could write:**

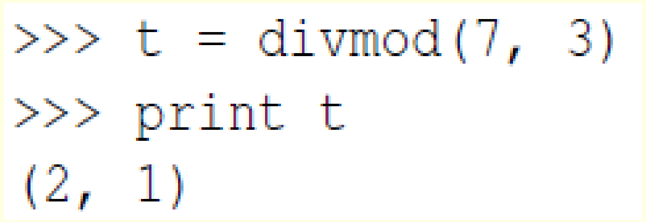


```
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
```

  - The return value from split is a list with two elements; the first element is assigned to uname, the second to domain.

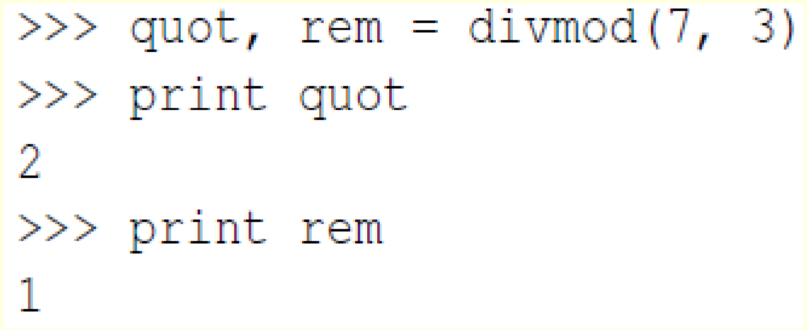
Tuples as return values

- Strictly speaking, a function can only return one value, but if the value is a tuple, the effect is the same as returning multiple values.
- To compute the quotient and remainder for dividing two integers, the built-in function divmod takes two arguments and returns a tuple of two values, the quotient and remainder. You can store the result as tuple.



```
>>> t = divmod(7, 3)
>>> print t
(2, 1)
```

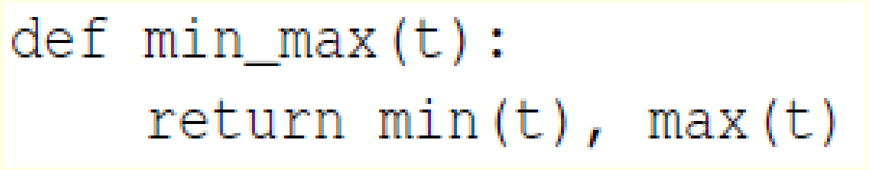
- Or use tuple assignment to store the elements separately:



```
>>> quot, rem = divmod(7, 3)
>>> print quot
2
>>> print rem
1
```

Tuples as return values (cont’d)

- **An example of a function that returns a tuple:**



```
def min_max(t):
    return min(t), max(t)
```

  - max and min are built-in functions that find the largest and smallest elements of a sequence. min_max computes both and returns a tuple of two values.

Variable-length argument tuples

- **Functions can take a variable number of arguments. A parameter name that begins with * gathers arguments into a tuple. For example, printall takes any number of arguments and prints them:**

```
def printall(*args):
    print args
```

- **The gather parameter can have any name you like, but args is conventional. Here's how the function works:**

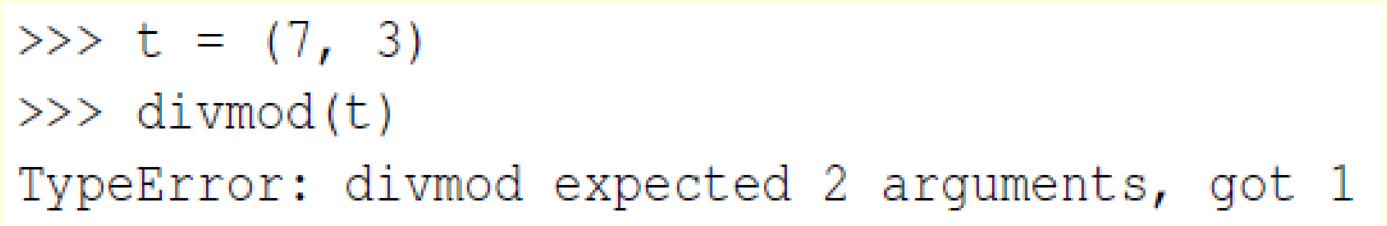
```
>>> printall(1, 2.0, '3')
(1, 2.0, '3')
```

- **You can combine the gather operator with required and positional arguments:**

```
def pointless(required, optional=0, *args):
    print required, optional, args
```

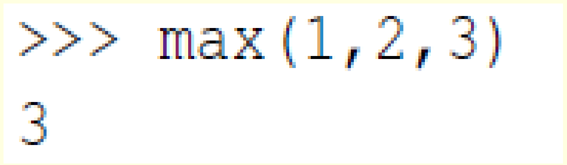
Variable-length argument tuples (cont'd)

- 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. For example, divmod takes exactly two arguments; it doesn't work with a tuple:



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

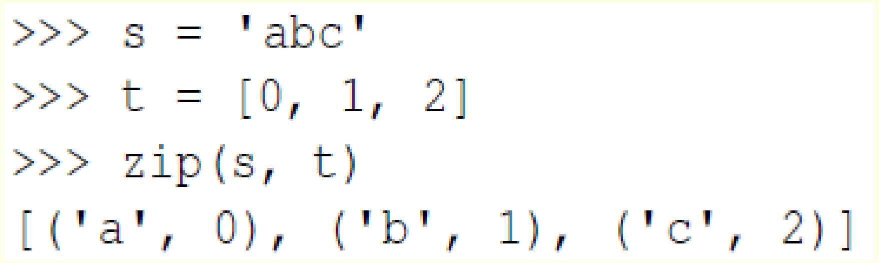
But if you scatter the tuple, it works:



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

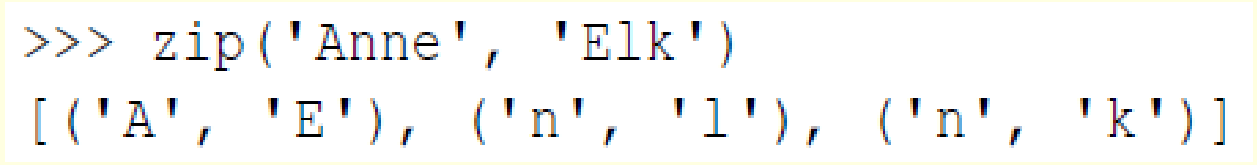
Lists and tuples

- zip is a built-in function that takes two or more sequences and "zips" them into a list1 of tuples where each tuple contains one element from each sequence. For example:



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

- If the sequences are not the same length, the result has the length of the shorter one.



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

Lists and tuples (cont'd)

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

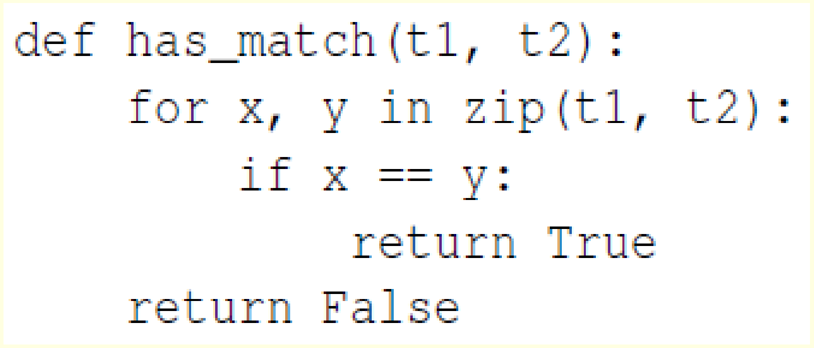
```
t = [('a', 0), ('b', 1), ('c', 2)]
for letter, number in t:
    print number, letter
```

Each time through the loop, Python selects the next tuple in the list and assigns the elements to letter and number. The output of this loop is:

```
0 a
1 b
2 c
```

Lists and tuples (cont'd)

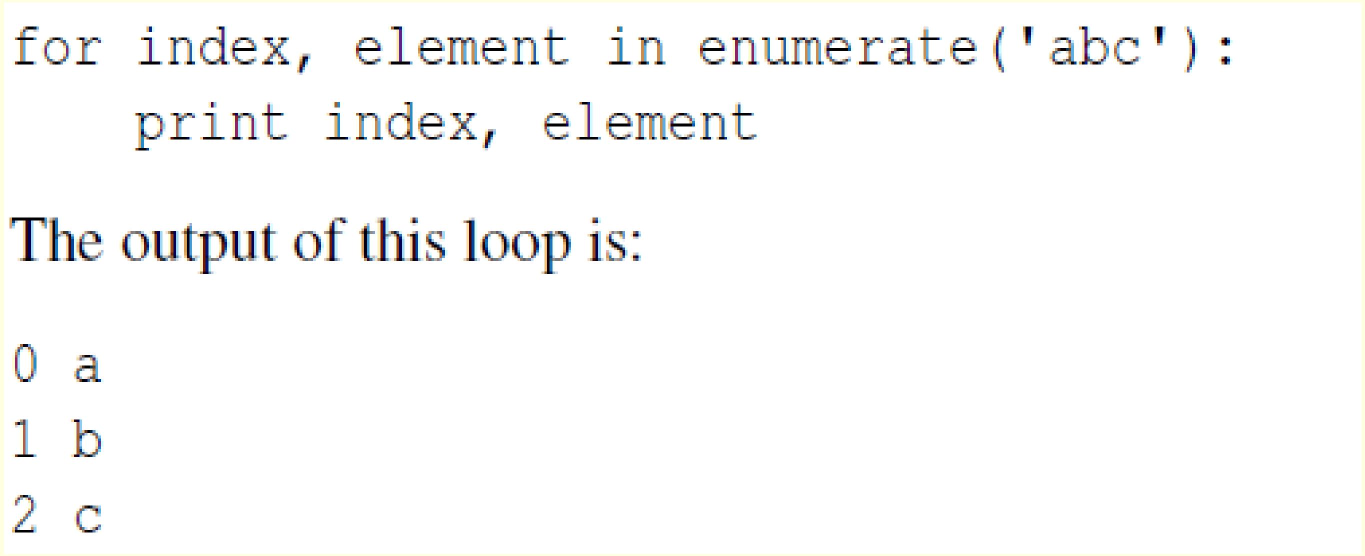
- If you combine zip, for and tuple assignment, you get a useful idiom for traversing two (or more) sequences at the same time. For example, has_match takes two sequences, t1 and t2, and returns True if there is an index i such that t1[i] == t2[i]:



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

Lists and tuples (cont'd)

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



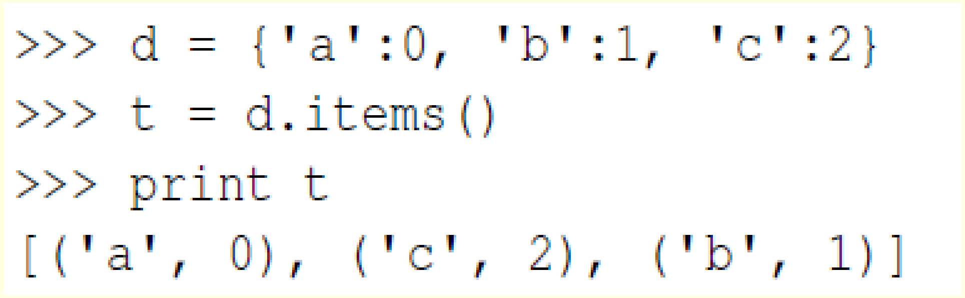
```
for index, element in enumerate('abc'):
    print index, element
```

The output of this loop is:

```
0 a
1 b
2 c
```

Dictionaries and tuples

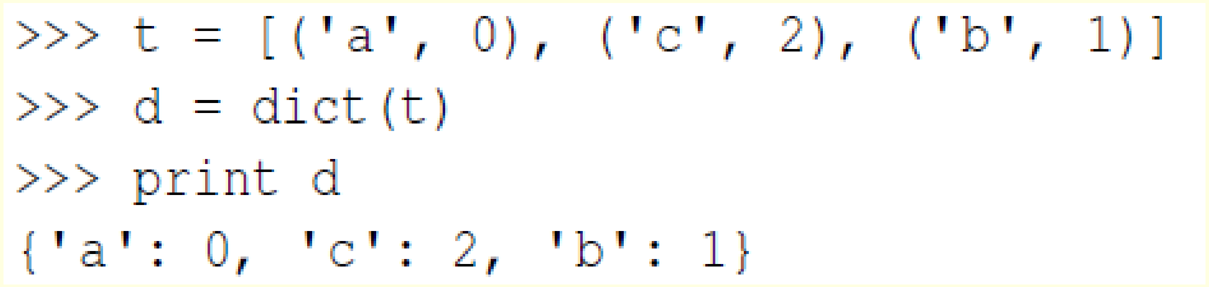
- **Dictionaries have a method called items that returns a list of tuples, where each tuple is a key-value pair.**



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

  - The items are in no particular order.

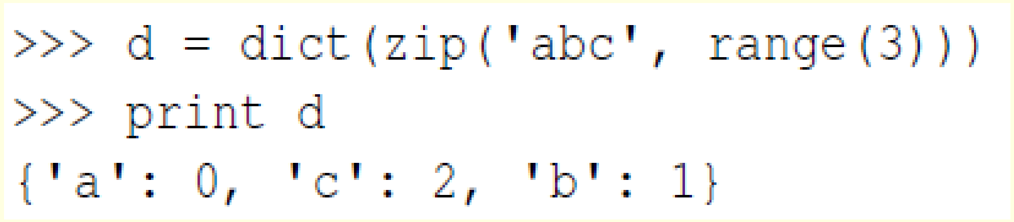
- **Conversely, you can use a list of tuples to initialize a new dictionary:**



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

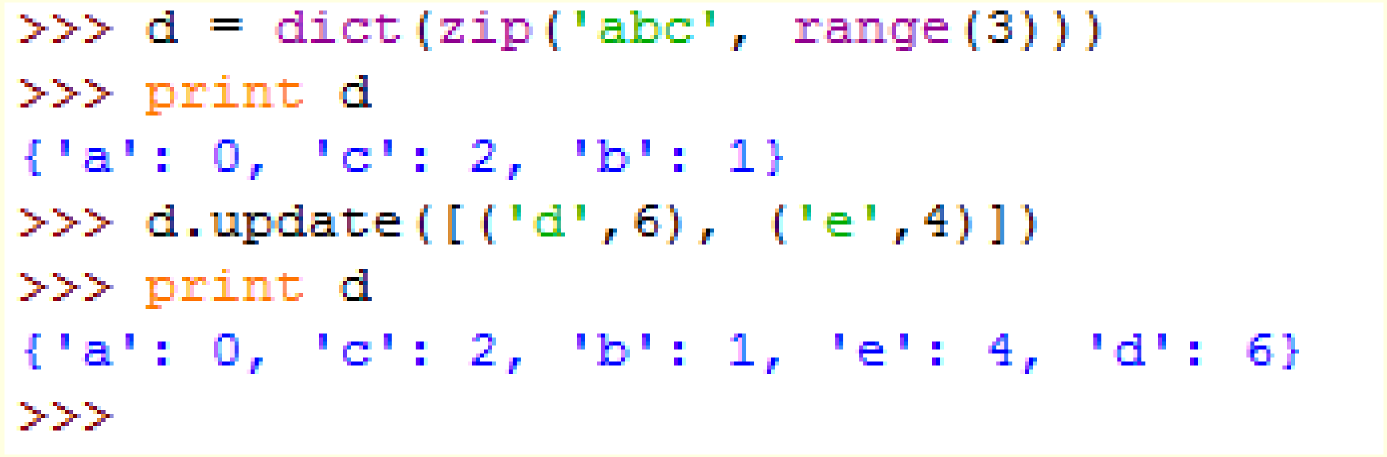
Dictionaries and tuples (cont'd)

- Combining dict with zip yields a concise way to create a dictionary:



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

- The dictionary method update also takes a list of tuples and adds them, as key-value pairs, to an existing dictionary.



```
>>> d = dict(zip('abc', range(3)))
>>> print d
{'a': 0, 'c': 2, 'b': 1}
>>> d.update([('d',6), ('e',4)])
>>> print d
{'a': 0, 'c': 2, 'b': 1, 'e': 4, 'd': 6}
>>>
```

Dictionaries and tuples (cont’d)

- Combining items, tuple assignment and for, you get the idiom for traversing the keys and values of a dictionary:

```
for key, val in d.items():
    print val, key
```

  The output of this loop is:

```
0 a
2 c
1 b
```

Dictionaries and tuples (cont'd)

- **It is common to use tuples as keys in dictionaries (primarily because you can't use lists). For example, a telephone directory might map from last-name, first-name pairs to telephone numbers. Assuming that we have defined last, first and number, we could write:**

  ```
  directory[last,first] = number
  ```

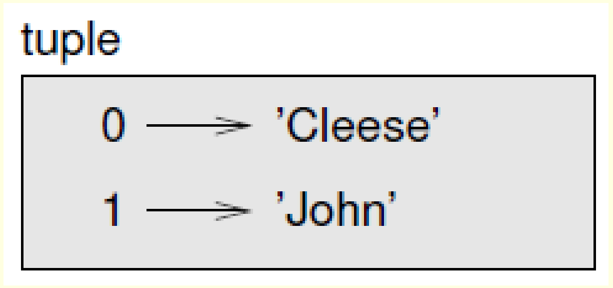
  The expression in brackets is a tuple. We could use tuple assignment to traverse this dictionary.

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

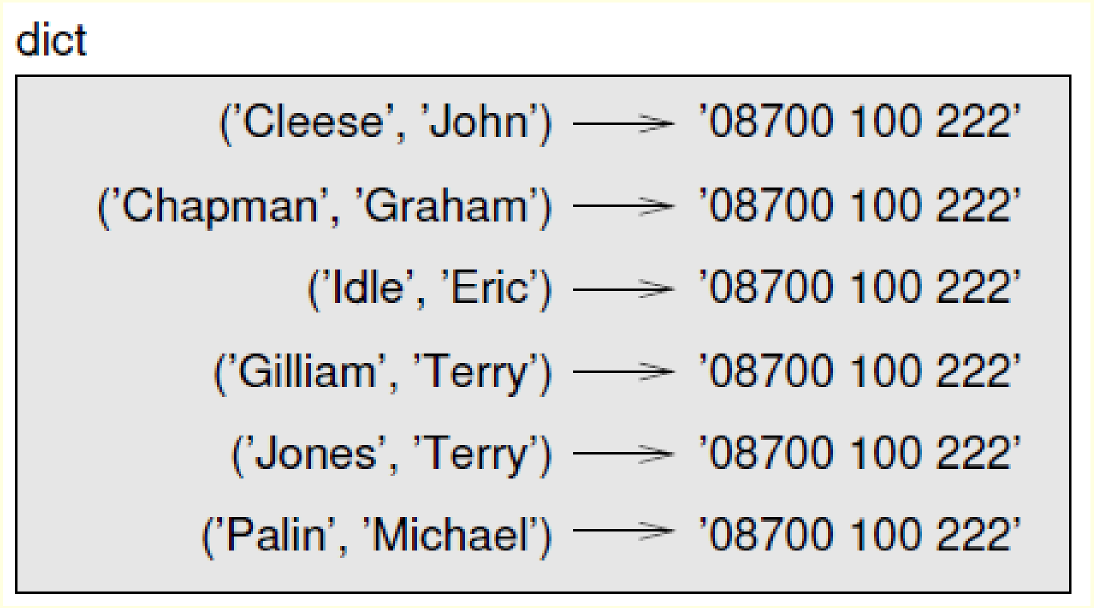
  - This loop traverses the keys in directory, which are tuples. It assigns the elements of each tuple to last and first, then prints the name and corresponding telephone number.

Dictionaries and tuples (cont'd)

- **There are two ways to represent tuples in a state diagram.**
  - A simple tuple

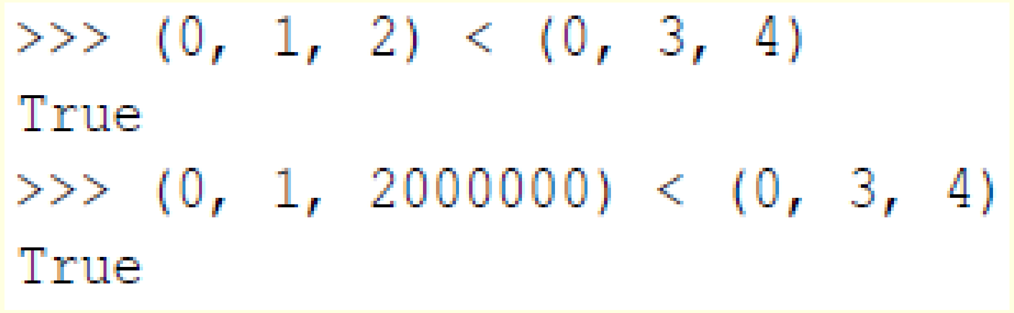


  - A telephone directory


dict
('Cleese', 'John') → '08700 100 222'
('Chapman', 'Graham') → '08700 100 222'
('Idle', 'Eric') → '08700 100 222'
('Gilliam', 'Terry') → '08700 100 222'
('Jones', 'Terry') → '08700 100 222'
('Palin', 'Michael') → '08700 100 222'

Comparing tuples

- The comparison operators work with tuples and other sequences; Python starts by comparing the first element from each sequence. If they are equal, it goes on to the next elements, and so on, until it 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
```

Comparing tuples (cont'd)

- **The sort function works the same way. It sorts primarily by first element, but in the case of a tie, it sorts by second element, and so on.**
- **This feature lends itself to a pattern called DSU for:**
  - Decorate a sequence by building a list of tuples with one or more sort keys preceding the elements from the sequence,
  - Sort the list of tuples, and
  - Undecorate by extracting the sorted elements of the sequence.

Comparing tuples (cont'd)

- **For example, suppose you have a list of words and you want to sort them from longest to shortest:**

```
def sort_by_length(words):
    t = []
    for word in words:
       t.append((len(word), word))

    t.sort(reverse=True)

    res = []
    for length, word in t:
        res.append(word)
    return res
```

  - The first loop builds a list of tuples, where each tuple is a word preceded by its length.
  - sort compares the first element, length, first, and only considers the second element to break ties. The keyword argument reverse=True tells sort to go in decreasing order.
  - The second loop traverses the list of tuples and builds a list of words in descending order of length.

Sequences of sequences

- **In many contexts, the different kinds of sequences (strings, lists and tuples) can be used interchangeably. So how and why do you choose one over the others?**
- **To start with the obvious, strings are more limited than other sequences because the elements have to be characters. They are also immutable. If you need the ability to change the characters in a string (as opposed to creating a new string), you might want to use a list of characters instead.**
- **Lists are more common than tuples, mostly because they are mutable. But there are a few cases where you might prefer tuples:**
  - In some contexts, like a return statement, it is syntactically simpler to create a tuple than a list. In other contexts, you might prefer a list.
  - If you want to use a sequence as a dictionary key, you have to use an immutable type like a tuple or string.
  - If you are passing a sequence as an argument to a function, using tuples reduces the potential for unexpected behavior due to aliasing.

Sequences of sequences (cont’d)

- Because tuples are immutable, they don’t provide methods like sort and reverse, which modify existing lists.
- But Python provides the built-in functions sorted and reversed, which take any sequence as a parameter and return a new list with the same elements in a different order.

Debugging

- Lists, dictionaries and tuples are known generically as data structures; in this chapter we are starting to see compound data structures, like lists of tuples, and dictionaries that contain tuples as keys and lists as values. Compound data structures are useful, but they are prone to what I call shape errors; that is, errors caused when a data structure has the wrong type, size or composition. For example, if you are expecting a list with one integer and I give you a plain old integer (not in a list), it won’t work.
- To help debug these kinds of errors, the author has written a module called structshape that provides a function, also called structshape, that takes any kind of data structure as an argument and returns a string that summarizes its shape. You can download it from hinkpython.com/code/structshape.py