

Introduction to Python

Built-In Sequences: Tuples and Dictionaries

Topics

- 1) Tuples
- 2) Tuple unpacking
- 3) List of tuples, List of lists(2D lists)
- 4) enumerate()
- 5) Dictionaries
- 6) Iterating over a dictionary

Tuples

Tuples are in many ways similar to lists, but they are defined with parentheses rather than square brackets:

```
In[1]: t = (1, 2, 3)
```

They can also be defined without any brackets at all:

```
In[2]: t = 1, 2, 3
        print(t, type(t))
(1, 2, 3) tuple
```

Tuples

Like the lists, tuples have a length, and individual elements can be extracted using square-bracket indexing. Slicing is also supported.

```
In[1]: t = (1, 2, 3)
```

```
In [2]: len(t)
```

```
Out [2]: 3
```

```
In [3]: t[0] # access a tuple is the same as a list!
```

```
Out [3]: 1
```

```
In [4]: t[0:2] # slicing is the same
```

```
Out [4]: (1, 2)
```

Tuples

Unlike lists, tuples are *immutable*: this means that once they are created, their size and contents cannot be changed:

```
In [4]: t[0] = 1    # error!
```

```
In [5]: t.append(10) # error!
```

Tuple Unpacking

Tuple unpacking is an assignment feature that assigns right hand side of values into left hand side. In packing, we put values into a new tuple while in unpacking we extract those values into a single variable.

This is packing values into a variable.

```
In [1]: student = ("Mike Smith", 3.2)
```

This is unpacking values from a variable.

```
In [2]: name, gpa = student
```

```
In [3]: print(name, gpa)
```

```
Mike Smith 3.2
```

Tuple Unpacking

Tuple unpacking can be used to do parallel assignment.

```
In [1]: a = 1  
        b = 2  
        a, b = b, a # parallel assignment  
        print(a, b)
```

2 1

Note: In Java, you would need a temporary variable for this.

list(), tuple()

Converting between sequences can be done using the appropriate constructors: list(), tuple().

```
In [1]: t = [1, 2, 3]
        tuple(t)
```

```
Out [1]: (1, 2, 3)
```

```
In [2]: s = (1, 2, 3)
        list(s)
```

```
Out [2]: [1, 2, 3]
```


List of Tuples

We can form a list of tuples(or of lists) and iterate over them by using nested loop, [] or tuple unpacking.

Using nested loop:

```
In[1]: lst = [("Mike", 3.2), ("Sarah", 3.6), ("Jack", 2.8)]  
        for student in lst:  
            for data in student:  
                print(data)
```

Mike

3.2

Sarah

3.6

Jack

2.8

List of Tuples

We can form a list of tuples(or of lists) and iterate over them by using nested loop, [] or tuple unpacking.

Using []:

```
In[1]: lst = [("Mike", 3.2), ("Sarah", 3.6), ("Jack", 2.8)]  
        for student in lst:  
            print(student[0], student[1])
```

Mike 3.2

Sarah 3.6

Jack 2.8

List of Tuples

We can form a list of tuples(or of lists) and iterate over them by using nested loop, [] or tuple unpacking.

Using tuple unpacking:

```
In[1]: lst = [("Mike", 3.2), ("Sarah", 3.6), ("Jack", 2.8)]  
        for name, gpa in lst: # tuple unpacking  
            print(name, gpa)
```

Mike 3.2

Sarah 3.6

Jack 2.8

List of Lists(2D lists)

We can have a list of lists.

```
In[1]: lst = [[1, 2, 3],  
              [4, 5, 6],  
              [7, 8, 9]]
```

```
In[2]: lst[0]
```

```
Out [2]: [1, 2, 3]
```

```
In[3]: lst[1][2]
```

```
Out [3]: 6
```

```
In[4]: lst[0][0]
```

```
Out [4]: 1
```

```
In[5]: lst[2][1]
```

```
Out [5]: 8
```

enumerate()

It is useful to have access to the index of elements when iterating over a list or 2D list. The `enumerate()` function adds a index to elements of an iterable and returns it.

```
In [1]: lst = ['bread', 'milk', 'ham']
```

```
In [2]: for item in enumerate(lst):  
        print(item)
```

```
(0, 'bread')
```

```
(1, 'milk')
```

```
(2, 'ham')
```

enumerate()

Unpacking from enumerate().

```
In [1]: lst = ['bread', 'milk', 'ham']
```

```
In [2]: for index, value in enumerate(lst):  
        print(index, value)
```

```
0 bread
```

```
1 milk
```

```
2 ham
```

enumerate()

enumerate() is helpful when we want to modify our list.

```
In [1]: lst = ['bread', 'milk', 'ham']
```

```
In [2]: for index, value in enumerate(lst):  
        if value == 'bread':  
            lst[index] = 'butter'  
        print(lst)
```

```
['butter', 'milk', 'ham']
```

enumerate() with 2D lists

enumerate() is useful if we want to access indices of 2D lists. For example, we can use indices to modify the 2D list.

```
In[1]: lst = [[1, 2, 3],  
              [4, 5, 6],  
              [7, 8, 9]]
```

```
In[2]: for row_ind, row in enumerate(lst):  
        for col_ind, value in enumerate(row):  
            lst[row_ind][col_index] = 3
```

```
In[3]: lst
```

```
Out[50]: [[3, 3, 3], [3, 3, 3], [3, 3, 3]]
```


Dictionaries

Python lists are useful but in some applications, it is nice to have a different indexing scheme than the integers. For example, consider a database of students' names and their grades:

Mike: [70, 81, 84]

Sarah: [88, 71, 85]

...

Suppose that this database has hundreds of records. It is hard to access these students' grades using 0-based integer indexing.

Python dictionaries allow "values" to be accessed by meaningful "keys". In the example above, we can access the database of grades by `nam(keys)` instead of integer index.

Dictionaries

Dictionaries are extremely flexible mappings of keys to values, and form the basis of much of Python's internal implementation.

They can be created via a comma-separated list of key:value pairs within curly braces. The "keys" are distinct.

```
In [1]: numbers = {"one":1, "two":2, "three":3}
```

```
In [2]: numbers["two"]
```

```
Out [2]: 2
```

```
In [3]: len(numbers)
```

```
Out [3]: 3
```

Dictionaries

New items can be added to the dictionary using indexing as well.

```
In [1]: numbers = {"one":1, "two":2, "three":3}
```

```
In [2] numbers['ninety'] = 90
```

```
In [3] print(numbers)
{'one': 1, 'two': 2, 'three': 3, 'ninety': 90}
```

```
In [4]: numbers['four'] # KeyError, 'four' not set as key
```

Dictionaries

Modifying dictionary.

```
In [1]: numbers = {"one":1, "two":2, "three":3}
```

```
In [2]: numbers['two'] = 20
```

```
In [3] print(numbers)
{'one': 1, 'two': 20, 'three': 3}
```

```
In [4]: numbers['one'] += 5
```

```
In [5] print(numbers)
{'one': 6, 'two': 20, 'three': 3}
```

Dictionaries

The keys and values of dictionaries can be different types. However, a dictionary key must be immutable(int, float, bool, str, tuple). A dictionary value can be any object.

```
In [1]: misc = {2:5, (3, 5):4.5, True:[1], 3.5:(1, 5)}
```

```
In [2]: misc[2]
```

```
Out [2]: 5
```

```
In [3]: misc[True]
```

```
Out [3]: [1]
```

```
In [4]: misc[(3, 5)]
```

```
Out [4]: 4.5
```

Dictionaries

The keys and values can be extracted from a dictionary via `keys()` and `values()`.

```
In [1]: numbers = {"one":1, "two":2, "three":3}
```

```
In [2]: k = list(numbers.keys())
```

```
    print(k)
```

```
['one', 'two', 'three']
```

```
In [3]: k = list(numbers.values())
```

```
    print(k)
```

```
[1, 2, 3]
```

Membership Operations

By default, membership operations checks keys of a dictionary.

```
In [1]: numbers = {'one':1, 'two':2, 'three':3}
```

```
In [2]: 'one' in numbers
```

```
Out [2]: True
```

```
In [3]: 1 in numbers
```

```
Out [3]: False
```

Iterables

It is easy to iterate over keys of the dictionary. The default loop iterates over the keys.

```
In [1]: numbers = {'one':1, 'two':2, 'three':3}
        for x in numbers:
            print(x, end=" ")
```

one two three

Iterables

To iterate over values of the dictionary use, values().

```
In [1]: for x in numbers.values():  
        print(x, end=' ')
```

| 2 3

OR use indexing.

```
In [2]: numbers = {'one':1, 'two':2, 'three':3}  
        for x in numbers:  
            print(numbers[x], end=" ")
```

| 2 3

Iterables

It is easy to iterate over both keys and values of the dictionary. Use the method `items()` and tuple unpacking.

```
In [1]: numbers = {'one':1, 'two':2, 'three':3}
        for k, v in numbers.items():
            print(k, v)
```

one 1

two 2

three 3

Example of a Use for Dictionaries

Let's manually add test scores to a database.

```
database = {'Mike':[70], 'Sarah':[88]}
names_to_add = [('Sarah',75), ('John',90)]
# unpacking Sarah's info
name, score = names_to_add[0]
# since 'Sarah' is already in database
database[name].append(score) # add Sarah's score
                             # to her record
# since 'John' is not in database, need to initialize
database['John'] = []
database['John'].append(90)
```

Example of a Use for Dictionaries

We can automate the previous example using a for loop.

```
database = {'Mike':[70], 'Sarah':[88]}
names_to_add = [('Sarah',75), ('John',90), ('Mike',81)]
for student in names_to_add:
    name, score = student # tuple unpacking
    if name not in database:
        database[name] = [] # create new key/value
    database[name].append(score) # add score to list
```

References

- I) Vanderplas, Jake, A Whirlwind Tour of Python, O'reilly Media.