Built-in Data Structures, Functions, and Files

Data Structures and Sequences

Tuple

A tuple is a fixed-length, immutable sequence of Python objects. The easiest way to create one is with a comma-separated sequence of values:

In [1]:
```
tup = 4, 5, 6
tup
```

Out[1]:
```
(4, 5, 6)
```

In [2]:
```
nested_tup = (4, 5, 6), (7, 8)
nested_tup
```

Out[2]:
```
((4, 5, 6), (7, 8))
```

In [3]:
```
tuple([4, 0, 2])
tup = tuple('string')
tup
```

Out[3]:
```
('s', 't', 'r', 'i', 'n', 'g')
```

In [4]:
```
tup[0]
```

Out[4]:
```
's'
```

In [31]:
```
tup = tuple(['foo', [1, 2], True])
tup[2] = False
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-31-11b694945ab9> in <module>
      1 tup = tuple(['foo', [1, 2], True])
----> 2 tup[2] = False

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

In [6]:
```
tup[1].append(3)
tup
```

Out[6]:
```
('foo', [1, 2, 3], True)
```

Unpacking tuples

If you try to assign to a tuple-like expression of variables, Python will attempt to unpack the value on the righthand side of the equals sign:

In [7]:
```
tup = (4, 5, 6)
a, b, c = tup
b
```

Out[7]:
```
5
```

In [8]:
```
seq = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
for a, b, c in seq:
    print('a={0}, b={1}, c={2}'.format(a, b, c))
```

```
a=1, b=2, c=3
a=4, b=5, c=6
a=7, b=8, c=9
```

List

In contrast with tuples, lists are variable-length and their contents can be modified in-place. You can define them using square brackets [] or using the list type func- tion:

In [9]:
```
a_list = [2, 3, 7, None]
tup = ('foo', 'bar', 'baz')
b_list = list(tup)
b_list
b_list[1] = 'peekaboo'
b_list
```

Out[9]:
```
['foo', 'peekaboo', 'baz']
```

In [10]:
```
gen = range(10)
gen
list(gen)
```

Out[10]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Adding and removing elements

In [11]:
```
print(b_list)
b_list.append('dwarf')
b_list
```

```
['foo', 'peekaboo', 'baz']
```

Out[11]:
```
['foo', 'peekaboo', 'baz', 'dwarf']
```

In [12]:
```
b_list.insert(1, 'red')
b_list
```

Out[12]:
```
['foo', 'red', 'peekaboo', 'baz', 'dwarf']
```

In [13]:
```
b_list.pop(2)
b_list
```

Out[13]:
```
['foo', 'red', 'baz', 'dwarf']
```

In [14]:
```
b_list.append('foo')
print(b_list)
b_list.remove('foo')
b_list
```

```
['foo', 'red', 'baz', 'dwarf', 'foo']
```

Out[14]:
```
['red', 'baz', 'dwarf', 'foo']
```

In [15]:
```
'dwarf' in b_list
```

Out[15]:
```
True
```

In [16]:
```
'dwarf' not in b_list
```

Out[16]:
```
False
```

Concatenating and combining lists

In [17]:
```
[4, None, 'foo'] + [7, 8, (2, 3)]
```

Out[17]:
```
[4, None, 'foo', 7, 8, (2, 3)]
```

In [18]:
```
x = [4, None, 'foo']
x.extend([7, 8, (2, 3)])
x
```

Out[18]:
```
[4, None, 'foo', 7, 8, (2, 3)]
```

Sorting

You can sort a list in-place (without creating a new object) by calling its sort function:

In [19]:
```
a = [7, 2, 5, 1, 3]
a.sort()
a
```

Out[19]:
```
[1, 2, 3, 5, 7]
```

In [20]:
```
b = ['saw', 'small', 'He', 'foxes', 'six']
b.sort(key=len)
b
```

Out[20]:
```
['He', 'saw', 'six', 'small', 'foxes']
```

Slicing

In [21]:
```
seq = [7, 2, 3, 7, 5, 6, 0, 1]
seq[1:5]
```

Out[21]:
```
[2, 3, 7, 5]
```

In [22]:
```
print(seq)
seq[3:4] = [6, 3]
seq
```

```
[7, 2, 3, 7, 5, 6, 0, 1]
```

Out[22]:
```
[7, 2, 3, 6, 3, 5, 6, 0, 1]
```

While the element at the start index is included, the stop index is not included.

In [23]:
```
print(seq)
print(seq[:5])
seq[3:]
```

```
[7, 2, 3, 6, 3, 5, 6, 0, 1]
[7, 2, 3, 6, 3]
```

Out[23]:
```
[6, 3, 5, 6, 0, 1]
```

In [24]:
```
print(seq)
seq[-4:]
```

```
[7, 2, 3, 6, 3, 5, 6, 0, 1]
```

Out[24]:
```
[5, 6, 0, 1]
```

A step can also be used after a second colon to, say, take every other element:

In [25]:
```
print(seq)
seq[::3]
```

```
[7, 2, 3, 6, 3, 5, 6, 0, 1]
```

Out[25]:
```
[7, 6, 6]
```

A clever use of this is to pass -1, which has the useful effect of reversing a list or tuple:

In [26]:
```
print(seq)
seq[::-1]
```

```
[7, 2, 3, 6, 3, 5, 6, 0, 1]
```

Out[26]:
```
[1, 0, 6, 5, 3, 6, 3, 2, 7]
```

dict

dict is likely the most important built-in Python data structure. A more common name for it is hash map or associative array. It is a flexibly sized collection of key-value pairs, where key and value are Python objects. One approach for creating one is to use curly braces {} and colons to separate keys and values:

In [27]:
```
empty_dict = {}
d1 = {'a' : 'some value', 'b' : [1, 2, 3, 4]}
d1
```

Out[27]:
```
{'a': 'some value', 'b': [1, 2, 3, 4]}
```

In [28]:
```
d1[7] = 'an integer'
print(d1)
d1[7] = 'UNO'
print(d1)

# d1['b']
# d1[5] = 'some value'
# d1
d1['dummy'] = 'another value'
d1
print(d1)
```

```
{'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an integer'}
{'a': 'some value', 'b': [1, 2, 3, 4], 7: 'UNO'}
{'a': 'some value', 'b': [1, 2, 3, 4], 7: 'UNO', 'dummy': 'another value'}
```

In [29]:
```
'b' in d1
```

Out[29]: True

In [30]:
```
del d1[5]
d1
ret = d1.pop('dummy')
ret
d1
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-30-2ba71f4e5422> in <module>
----> 1 del d1[5]
      2 d1
      3 ret = d1.pop('dummy')
      4 ret
      5 d1

KeyError: 5
```

In [ ]:
```
print(list(d1.keys()))
list(d1.values())
```

In [ ]:
```
d1.update({'b' : 'foo', 'c' : 12})
d1
```

In [ ]:
```
words = ['apple', 'bat', 'bar', 'atom', 'book']
by_letter = {}
for word in words:
    letter = word[0]
    if letter not in by_letter:
        by_letter[letter] = [word]
    else:
        by_letter[letter].append(word)
by_letter
```

set

A set is an unordered collection of unique elements. You can think of them like dicts, but keys only, no values. A set can be created in two ways: via the set function or via a set literal with curly braces:

In [ ]:
```
set([2, 2, 2, 1, 3, 3])
{2, 2, 2, 1, 3, 3}
```

Sets support mathematical set operations like union, intersection, difference, and symmetric difference. Consider these two example sets:

In [ ]:
```
a = {1, 2, 3, 4, 5}
b = {3, 4, 5, 6, 7, 8}
```

In [ ]:
```
a.union(b)
a | b
```

In [ ]:
```
a.intersection(b)
a & b
```

Table 3-1. Python set operations

Function	Alternative syntax	Description
`a.add(x)`	N/A	Add element `x` to the set `a`
`a.clear()`	N/A	Reset the set `a` to an empty state, discarding all of its elements
`a.remove(x)`	N/A	Remove element `x` from the set `a`
`a.pop()`	N/A	Remove an arbitrary element from the set `a`, raising `KeyError` if the set is empty
`a.union(b)`	`a | b`	All of the unique elements in `a` and `b`
`a.update(b)`	`a |= b`	Set the contents of `a` to be the union of the elements in `a` and `b`
`a.intersection(b)`	`a & b`	All of the elements in *both* `a` and `b`
`a.intersection_update(b)`	`a &= b`	Set the contents of `a` to be the intersection of the elements in `a` and `b`
`a.difference(b)`	`a - b`	The elements in `a` that are not in `b`
`a.difference_update(b)`	`a -= b`	Set `a` to the elements in `a` that are not in `b`
`a.symmetric_difference(b)`	`a ^ b`	All of the elements in either `a` or `b` but *not both*
`a.symmetric_difference_update(b)`	`a ^= b`	Set `a` to contain the elements in either `a` or `b` but *not both*
`a.issubset(b)`	`<=`	`True` if the elements of `a` are all contained in `b`
`a.issuperset(b)`	`>=`	`True` if the elements of `b` are all contained in `a`
`a.isdisjoint(b)`	N/A	`True` if `a` and `b` have no elements in common

In [ ]:
```
c = a.copy()
c |= b
c
d = a.copy()
print(d)
d &= b
d
```

Like dicts, set elements generally must be immutable. To have list-like elements, you must convert it to a tuple:

In [ ]:
```
my_data = [1, 2, 3, 4]
my_set = {tuple(my_data)}
my_set
```

In [ ]:
```
a_set = {1, 2, 3, 4, 5}
{1, 2, 3}.issubset(a_set)
a_set.issuperset({1, 2, 3})
```

In [ ]:
```
{1, 2, 3} == {3, 2, 1}
```

List, Set, and Dict Comprehensions

List comprehensions are one of the most-loved Python language features. They allow you to concisely form a new list by filtering the elements of a collection, transforming the elements passing

the filter in one concise expression. They take the basic form: Consider a for loop from a sorting algorithm:

```
[expr for val in collection if condition]

This is equivalent to the following for loop:

result = []
for val in collection:
    if condition:
        result.append(expr)
```

In [ ]:
```
strings = ['a', 'as', 'bat', 'car', 'dove', 'python']
[x.upper() for x in strings if len(x) > 2]
```

In [ ]:
```
unique_lengths = {len(x) for x in strings}
unique_lengths
```

Built-in Sequence Functions

Python has a handful of useful sequence functions that you should familiarize your- self with and use at any opportunity.

enumerate

It's common when iterating over a sequence to want to keep track of the index of the current item. A do-it-yourself approach would look like:

```
i = 0
for value in collection:
   # do something with value
   i += 1
```

Since this is so common, Python has a built-in function, enumerate, which returns a sequence of (i, value) tuples:

```
for i, value in enumerate(collection):
   # do something with value
   i += 1
```

In [ ]:
```
some_list = ['foo', 'bar', 'baz']
mapping = {}
for i, v in enumerate(some_list):
    mapping[v] = i
mapping
```

sorted

The sorted function returns a new sorted list from the elements of any sequence:

In [ ]:
```
sorted([7, 1, 2, 6, 0, 3, 2])
sorted('horse race')
```

zip

zip "pairs" up the elements of a number of lists, tuples, or other sequences to create a list of tuples:

In [ ]:
```
seq1 = ['foo', 'bar', 'baz']
seq2 = ['one', 'two', 'three', 'three']
zipped = zip(seq1, seq2)
list(zipped)
```

In [ ]:
```
seq3 = [False, True]
list(zip(seq1, seq2, seq3))
```

In [ ]:
```
for i, (a, b) in enumerate(zip(seq1, seq2)):
    print('{0}: {1}, {2}'.format(i, a, b))
```

In [ ]:
```

```

In [ ]:
```

```