3.1 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
```

In [2]:
```
tup
```

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

In [3]:
```
nested_tup = (4, 5, 6), (7, 8)
```

In [4]:
```
nested_tup
```

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

You can convert any sequence or iterator to a tuple by invoking tuple:

In [9]:
```
tuple([4, 0, 2])
```

Out[9]:
```
(4, 0, 2)
```

In [10]:
```
tup=tuple('string')
```

In [11]:
```
tup
```

Out[11]:
```
('s', 't', 'r', 'i', 'n', 'g')
```

Elements can be accessed with square brackets [] as with most other sequence types.

In [12]:
```
tup[0]
```

Out[12]:
```
's'
```

While the objects stored in a tuple may be mutable themselves, once the tuple is created it's not possible to modify which object is stored in each slot:

In [13]:
```
tup = tuple(['foo', [1, 2], True])
```

In [14]:
```
tup[2] = False
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-14-b89d0c4ae599> in <module>
----> 1 tup[2] = False

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

If an object inside a tuple is mutable, such as a list, you can modify it in-place:

In [15]:
```
tup[1].append(3)
```

In [16]:
```
tup
```

Out[16]:
```
('foo', [1, 2, 3], True)
```

You can concatenate tuples using the + operator to produce longer tuples:

In [21]:
```
(4, None, 'foo') + (6, 0) + ('bar',)
```

Out[21]:
```
(4, None, 'foo', 6, 0, 'bar')
```

Multiplying a tuple by an integer, as with lists, has the effect of concatenating together that many copies of the tuple:

In [22]:
```
('foo', 'bar') * 4
```

Out[22]:
```
('foo', 'bar', 'foo', 'bar', 'foo', 'bar', 'foo', 'bar')
```

Note that the objects themselves are not copied, only the references to them.

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 [23]:
```
tup = (4, 5, 6)
a, b, c = tup
b
```

Out[23]:
```
5
```

Even sequences with nested tuples can be unpacked:

In [24]:
```
tup = 4, 5, (6, 7)
a, b, (c, d) = tup
d
```

Out[24]:
```
7
```

Using this functionality you can easily swap variable names, a task which in many languages might look like

In [25]:
```
tmp = a
a = b
b = tmp
```

But, in Python, the swap can be done like this:

In [26]:
```
a, b = 1, 2
```

In [27]:
```
b, a = a, b
print(b,a)
```

```
1 2
```

A common use of variable unpacking is iterating over sequences of tuples or lists:

In [28]:
```
 seq = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
```

In [29]:
```
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
```

The Python language recently acquired some more advanced tuple unpacking to help with situations where you may want to “pluck” a few elements from the beginning of a tuple. This uses the special syntax *rest, which is also used in function signatures to capture an arbitrarily long list of positional arguments:

In [30]:
```
values = 1, 2, 3, 4, 5
a, b, *rest = values
```

In [31]:
```
a,b
```

Out[31]:
```
(1, 2)
```

In [32]:
```
rest
```

Out[32]:
```
[3, 4, 5]
```

many Python programmers will use the underscore (_) for unwanted variables:

In [33]:
```
a, b, *_ = values
```

In [34]:
```
a,b
```

Out[34]:
```
(1, 2)
```

In [35]:
```
_
```

Out[35]:
```
[3, 4, 5]
```

Tuple methods

Since the size and contents of a tuple cannot be modified, it is very light on instance methods. A particularly useful one (also available on lists) is count, which counts the number of occurrences of a value:

In [36]:
```
a = (1, 2, 2, 2, 3, 4, 2)
a.count(2)
```

Out[36]:
```
4
```

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 function:

In [37]:
```
a_list = [2, 3, 7, None]
a_list
```

Out[37]:
```
[2, 3, 7, None]
```

In [2]:
```
tup = ('foo', 'bar', 'baz')
b_list = list(tup)
```

In [3]:
```
b_list
```

Out[3]:
```
['foo', 'bar', 'baz']
```

In [4]:
```
b_list[1] = 'peekaboo'
```

In [5]:
```
b_list
```

Out[5]:
```
['foo', 'peekaboo', 'baz']
```

Lists and tuples are semantically similar (though tuples cannot be modified) and can be used interchangeably in many functions.

The list function is frequently used in data processing as a way to materialize an iterator or generator expression:

In [7]:
```
gen = range(10)
gen
```

Out[7]:
```
range(0, 10)
```

In [8]:
```
list(gen)
```

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

Adding and removing elements

Elements can be appended to the end of the list with the append method

In [9]:
```
b_list.append('dwarf')
b_list
```

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

Using insert you can insert an element at a specific location in the list:

In [10]:
```
 b_list.insert(1, 'red')
b_list
```

Out[10]:
```
['foo', 'red', 'peekaboo', 'baz', 'dwarf']
```

The insertion index must be between 0 and the length of the list, inclusive.

insert is computationally expensive compared with append, because references to subsequent elements have to be shifted internally to make room for the new element. If you need to insert elements at both the beginning and end of a sequence, you may wish to explore collections.deque, a double-ended queue, for this purpose

The inverse operation to insert is pop, which removes and returns an element at a particular index:

In [11]:
```
b_list.pop(2)
```

Out[11]:
```
 'peekaboo'
```

In [12]:
```
b_list
```

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

Elements can be removed by value with remove, which locates the first such value and removes it from the last

In [13]:
```
b_list.append('foo')
```

In [14]:
```
b_list
```

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

In [15]:
```
b_list.remove('foo')
b_list
```

Out[15]:
```
['red', 'baz', 'dwarf', 'foo']
```

If performance is not a concern, by using append and remove, you can use a Python list as a perfectly suitable "multiset" data structure

Check if a list contains a value using the in keyword:

In [16]:
```
'dwarf' in b_list
```

Out[16]:
```
True
```

In [17]:
```
'dwarf' not in b_list
```

Out[17]:
```
False
```

Checking whether a list contains a value is a lot slower than doing so with dicts and sets (to be introduced shortly), as Python makes a linear scan across the values of the list, whereas it can check the others (based on hash tables) in constant time.

Concatenating and combining lists

Similar to tuples, adding two lists together with + concatenates them:

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

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

If you have a list already defined, you can append multiple elements to it using the extend method:

In [20]:
```
x = [4, None, 'foo']
```

In [21]:
```
x.extend([7, 8, (2, 3)])
x
```

Out[21]:
```
[4, None, 'foo', 7, 8, (2, 3)]
```

Note that list concatenation by addition is a comparatively expensive operation since a new list must be created and the objects copied over. Using extend to append elements to an existing list, especially if you are building up a large list, is usually preferable.
everything = []
for chunk in list_of_lists:
everything.extend(chunk)
is faster than the concatenative alternative:
everything = []

for chunk in list_of_lists:
everything = everything + chunk

Sorting

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

In [22]:
```
a = [7, 2, 5, 1, 3]
a.sort()
a
```

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

sort has a few options that will occasionally come in handy. One is the ability to pass a secondary sort key—that is, a function that produces a value to use to sort the objects. For example, we could sort a collection of strings by their lengths:

In [23]:
```
b = ['saw', 'small', 'He', 'foxes', 'six']
b.sort(key=len)
b
```

Out[23]:
```
['He', 'saw', 'six', 'small', 'foxes']
```

Soon, we'll look at the sorted function, which can produce a sorted copy of a general sequence.

Binary search and maintaining a sorted list

The built-in bisect module implements binary search and insertion into a sorted list. bisect.bisect finds the location where an element should be inserted to keep it sorted, while bisect.insort actually inserts the element into that location:

In [24]:
```
import bisect
c = [1, 2, 2, 2, 3, 4, 7]
bisect.bisect(c, 2)
```

Out[24]:
```
4
```

In [25]:
```
bisect.bisect(c, 5)
```

Out[25]:
```
6
```

In [26]:
```
bisect.insort(c, 6)
```

In [27]:
```
c
```

Out[27]:
```
[1, 2, 2, 2, 3, 4, 6, 7]
```

The bisect module functions do not check whether the list is sorted, as doing so would be computationally expensive. Thus, using them with an unsorted list will succeed without error but may lead to incorrect results.

Slicing

You can select sections of most sequence types by using slice notation, which in its basic form consists of start:stop passed to the indexing operator []:

In [30]:
```
seq = [7, 2, 3, 7, 5, 6, 0, 1]
seq[1:5]
```

Out[30]:
```
[2, 3, 7, 5]
```

Slices can also be assigned to with a sequence:

In [31]:
```
seq[3:4] = [6, 3]
seq
```

Out[31]:
```
[7, 2, 3, 6, 3, 5, 6, 0, 1]
```

While the element at the start index is included, the stop index is not included, so that the number of elements in the result is stop - start.

Either the start or stop can be omitted, in which case they default to the start of the sequence and the end of the sequence, respectively:

In [32]:
```
seq[:5]
```

Out[32]:
```
[7, 2, 3, 6, 3]
```

In [33]:
```
seq[3:]
```

Out[33]:
```
[6, 3, 5, 6, 0, 1]
```

Negative indices slice the sequence relative to the end:

In [34]:
```
 seq[-4:]
```

Out[34]:
```
[5, 6, 0, 1]
```

In [35]:
```
seq[-6:-2]
```

Out[35]:
```
[6, 3, 5, 6]
```

Slicing semantics takes a bit of getting used to, especially if you're coming from R or MATLAB.

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

In [36]:
```
seq[::2]
```

Out[36]:
```
[7, 3, 3, 6, 1]
```

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

In [37]:
```
seq[::-1]
```

Out[37]:
```
[1, 0, 6, 5, 3, 6, 3, 2, 7]
```

Built-in Sequence Functions

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:

In [39]:
```
# 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:

In [40]:
```
# for i, value in enumerate(collection):
#  # do something with value
```

When you are indexing data, a helpful pattern that uses enumerate is computing a dict mapping the values of a sequence (which are assumed to be unique) to their locations in the sequence:

In [41]:
```
some_list = ['foo', 'bar', 'baz']
mapping = {}
for i, v in enumerate(some_list):
    mapping[v] = i
mapping
```

Out[41]:
```
{'foo': 0, 'bar': 1, 'baz': 2}
```

sorted

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

In [42]:
```
sorted([7, 1, 2, 6, 0, 3, 2])
```

Out[42]:
```
[0, 1, 2, 2, 3, 6, 7]
```

In [43]:
```
sorted('horse race')
```

Out[43]:
```
[' ', 'a', 'c', 'e', 'e', 'h', 'o', 'r', 'r', 's']
```

zip

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

In [44]:
```
seq1 = ['foo', 'bar', 'baz']
seq2 = ['one', 'two', 'three']
zipped = zip(seq1, seq2)
zipped
```

Out[44]:
```
<zip at 0x22ab9269408>
```

In [45]:
```
list(zipped)
```

Out[45]:
```
[('foo', 'one'), ('bar', 'two'), ('baz', 'three')]
```

zip can take an arbitrary number of sequences, and the number of elements it produces is determined by the shortest sequence:

In [46]:
```
seq3 = [False, True]
list(zip(seq1, seq2, seq3))
```

Out[46]:
```
[('foo', 'one', False), ('bar', 'two', True)]
```

A very common use of zip is simultaneously iterating over multiple sequences, possibly also combined with enumerate:

In [47]:
```
for i, (a, b) in enumerate(zip(seq1, seq2)):
    print('{0}: {1}, {2}'.format(i, a, b))
```

```
0: foo, one
1: bar, two
2: baz, three
```

Given a "zipped" sequence, zip can be applied in a clever way to "unzip" the sequence. Another way to think about this is converting a list of rows into a list of columns. The syntax, which looks a bit magical, is:

In [48]:
```
pitchers = [('Nolan', 'Ryan'), ('Roger', 'Clemens'),('Schilling', 'Curt')]
first_names, last_names = zip(*pitchers)
```

In [49]:
```
first_names
```

Out[49]:
```
('Nolan', 'Roger', 'Schilling')
```

In [50]:
```
last_names
```

```
Out[50]: ('Ryan', 'Clemens', 'Curt')
```

reversed

reversed iterates over the elements of a sequence in reverse order:

```
In [51]: list(reversed(range(10)))
```

```
Out[51]: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Keep in mind that reversed is a generator (to be discussed in some more detail later), so it does not create the reversed sequence until materialized (e.g., with list or a for loop).

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 [52]: empty_dict = {}
```

```
In [53]: empty_dict
```

```
Out[53]: {}
```

```
In [54]: d1 = {'a' : 'some value', 'b' : [1, 2, 3, 4]}
```

```
In [55]: d1
```

```
Out[55]: {'a': 'some value', 'b': [1, 2, 3, 4]}
```

You can access, insert, or set elements using the same syntax as for accessing elements of a list or tuple:

```
In [56]: d1[7] = 'an integer'
```

```
In [57]: d1
```

```
Out[57]: {'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an integer'}
```

```
In [58]: d1['b']
```

```
Out[58]: [1, 2, 3, 4]
```

You can check if a dict contains a key using the same syntax used for checking whether a list or tuple contains a value:

In [59]:
```
'b' in d1
```

Out[59]:
```
True
```

You can delete values either using the del keyword or the pop method (which simultaneously returns the value and deletes the key):

In [60]:
```
d1[5] = 'some value'
```

In [61]:
```
d1
```

Out[61]:
```
{'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an integer', 5: 'some value'}
```

In [62]:
```
d1['dummy'] = 'another value'
```

In [63]:
```
d1
```

Out[63]:
```
{'a': 'some value',
 'b': [1, 2, 3, 4],
 7: 'an integer',
 5: 'some value',
 'dummy': 'another value'}
```

In [64]:
```
del d1[5]
```

In [65]:
```
d1
```

Out[65]:
```
{'a': 'some value',
 'b': [1, 2, 3, 4],
 7: 'an integer',
 'dummy': 'another value'}
```

In [66]:
```
ret = d1.pop('dummy')
ret
```

Out[66]:
```
'another value'
```

In [67]:
```
d1
```

Out[67]:
```
{'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an integer'}
```

The keys and values method give you iterators of the dict's keys and values, respectively. While the key-value pairs are not in any particular order, these functions output the keys and values in the same order:

In [68]:
```
list(d1.keys())
```

Out[68]:
```
['a', 'b', 7]
```

In [69]:
```
list(d1.values())
```

Out[69]:
```
['some value', [1, 2, 3, 4], 'an integer']
```

You can merge one dict into another using the update method:

In [70]:
```
d1.update({'b' : 'foo', 'c' : 12})
d1
```

Out[70]:
```
{'a': 'some value', 'b': 'foo', 7: 'an integer', 'c': 12}
```

The update method changes dicts in-place, so any existing keys in the data passed to update will have their old values discarded

Creating dicts from sequences

It's common to occasionally end up with two sequences that you want to pair up element-wise in a dict. As a first cut, you might write code like this

In [72]:
```
# mapping = {}
# for key, value in zip(key_list, value_list):
#     mapping[key] = value
```

Since a dict is essentially a collection of 2-tuples, the dict function accepts a list of 2-tuples:

In [73]:
```
mapping = dict(zip(range(5), reversed(range(5))))
mapping
```

Out[73]:
```
{0: 4, 1: 3, 2: 2, 3: 1, 4: 0}
```

Default values

In [74]:
```
# if key in some_dict:
#  value = some_dict[key]
# else:
#  value = default_value
```

Thus, the dict methods get and pop can take a default value to be returned, so that the above if-else block can be written simply as:

In [75]:
```
# value = some_dict.get(key, default_value)
```

get by default will return None if the key is not present, while pop will raise an excep- tion

In [77]:
```
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
```

Out[77]:
```
{'a': ['apple', 'atom'], 'b': ['bat', 'bar', 'book']}
```

The setdefault dict method is for precisely this purpose. The preceding for loop can be rewritten as:

In [79]:
```
by_letter = {}
for word in words:
    letter = word[0]
    by_letter.setdefault(letter, []).append(word)
by_letter
```

Out[79]:
```
{'a': ['apple', 'atom'], 'b': ['bat', 'bar', 'book']}
```

The built-in collections module has a useful class, defaultdict, which makes this even easier. To create one, you pass a type or function for generating the default value for each slot in the dict:

In [80]:
```
from collections import defaultdict
by_letter = defaultdict(list)
for word in words:
    by_letter[word[0]].append(word)
by_letter
```

Out[80]:
```
defaultdict(list, {'a': ['apple', 'atom'], 'b': ['bat', 'bar', 'book']})
```

Valid dict key types

While the values of a dict can be any Python object, the keys generally have to be immutable objects like scalar types (int, float, string) or tuples (all the objects in the tuple need to be immutable, too). The technical term here is hashability. You can check whether an object is hashable (can be used as a key in a dict) with the hash function

In [85]:
```
hash('string')
```

Out[85]:
```
6526927556070756424
```

In [83]:
```
hash((1, 2, (2, 3)))
```

Out[83]:
```
1097636502276347782
```

In [86]:
```
hash((1, 2, [2, 3])) # fails because lists are mutable
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-86-473c35a62c0b> in <module>
----> 1 hash((1, 2, [2, 3])) # fails because lists are mutable

TypeError: unhashable type: 'list'
```

To use a list as a key, one option is to convert it to a tuple, which can be hashed as long as its elements also can:

In [87]:
```
d = {}
d[tuple([1, 2, 3])] = 5
d
```

Out[87]:
```
{(1, 2, 3): 5}
```

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 [88]:
```
set([2, 2, 2, 1, 3, 3])
```

Out[88]:
```
{1, 2, 3}
```

In [89]:
```
{2, 2, 2, 1, 3, 3}
```

Out[89]:
```
{1, 2, 3}
```

Sets support mathematical set operations like union, intersection, difference, and symmetric difference

In [90]:
```
a = {1, 2, 3, 4, 5}
b = {3, 4, 5, 6, 7, 8}
```

The union of these two sets is the set of distinct elements occurring in either set. This can be computed with either the union method or the | binary operator:

In [91]:
```
a.union(b)
```

Out[91]:
```
{1, 2, 3, 4, 5, 6, 7, 8}
```

In [92]:
```
a | b
```

Out[92]:
```
{1, 2, 3, 4, 5, 6, 7, 8}
```

The intersection contains the elements occurring in both sets. The & operator or the intersection method can be used:

In [93]:
```
a.intersection(b)
```

Out[93]:
```
{3, 4, 5}
```

In [94]:
```
a & b
```

Out[94]:
```
{3, 4, 5}
```

In [ ]:
```
# 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 el
# 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
# 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
# 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 intersect
# 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
# a.symmetric_difference(b) --> a ^ b --> All of the elements in either a or b but
# a.symmetric_difference_update(b) --> a ^= b --> Set a to contain the elements in
# a.issubset(b) --> N/A --> True if the elements of a are all contained in b
# a.issuperset(b) --> N/A --> 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
```

All of the logical set operations have in-place counterparts, which enable you to replace the contents of the set on the left side of the operation with the result. For very large sets, this may be more efficient:

In [95]:
```
c = a.copy()
c |= b
c
```

Out[95]:
```
{1, 2, 3, 4, 5, 6, 7, 8}
```

In [96]:
```
d = a.copy()
d &= b
d
```

Out[96]:
```
{3, 4, 5}
```

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

In [97]:
```
my_data = [1, 2, 3, 4]
my_set = {tuple(my_data)}
my_set
```

Out[97]:
```
{(1, 2, 3, 4)}
```

You can also check if a set is a subset of (is contained in) or a superset of (contains all elements of) another set:

In [98]:
```
a_set = {1, 2, 3, 4, 5}
{1, 2, 3}.issubset(a_set)
```

Out[98]: True

In [99]:
```
a_set.issuperset({1, 2, 3})
```

Out[99]: True

Sets are equal if and only if their contents are equal:

In [100…
```
{1, 2, 3} == {3, 2, 1}
```

Out[100… True

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:

[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 [101…
```
strings = ['a', 'as', 'bat', 'car', 'dove', 'python']
[x.upper() for x in strings if len(x) > 2]
```

Out[101… ['BAT', 'CAR', 'DOVE', 'PYTHON']

Set and dict comprehensions are a natural extension, producing sets and dicts in an idiomatically similar way instead of lists. A dict comprehension looks like this:

In [102…
```
# dict_comp = {key-expr : value-expr for value in collection if condition}
```

A set comprehension looks like the equivalent list comprehension except with curly braces instead of square brackets:

In [103…
```
# set_comp = {expr for value in collection if condition}
```

In [104…
```
unique_lengths = {len(x) for x in strings}
unique_lengths
```

Out[104…
```
{1, 2, 3, 4, 6}
```

We could also express this more functionally using the map function, introduced shortly:

In [105…
```
set(map(len, strings))
```

Out[105…
```
{1, 2, 3, 4, 6}
```

As a simple dict comprehension example, we could create a lookup map of these strings to their locations in the list:

In [106…
```
loc_mapping = {val : index for index, val in enumerate(strings)}
loc_mapping
```

Out[106…
```
{'a': 0, 'as': 1, 'bat': 2, 'car': 3, 'dove': 4, 'python': 5}
```

Nested list comprehensions

In [107…
```
all_data = [['John', 'Emily', 'Michael', 'Mary', 'Steven'],['Maria', 'Juan', 'Javie
```

In [108…
```
names_of_interest = []
for names in all_data:
 enough_es = [name for name in names if name.count('e') >= 2]
 names_of_interest.extend(enough_es)
```

In [109…
```
names_of_interest
```

Out[109…
```
['Steven']
```

In [110…
```
result = [name for names in all_data for name in names if name.count('e') >= 2]
result
```

Out[110…
```
['Steven']
```

In [112…
```
some_tuples = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
flattened = [x for tup in some_tuples for x in tup]
flattened
```

Out[112…
```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [113…
```
[[x for x in tup] for tup in some_tuples]
```

Out[113…
```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

3.2 Functions

In [114…
```
def my_function(x, y, z=1.5):
 if z > 1:
     return z * (x + y)
 else:
     return z / (x + y)
```

Each function can have positional arguments and keyword arguments. Keyword arguments are most commonly used to specify default values or optional arguments. In the preceding function, x and y are positional arguments while z is a keyword argument. This means that the function can be called in any of these ways:

In [115…
```
my_function(5, 6, z=0.7)
```

Out[115…
```
0.06363636363636363
```

In [116…
```
my_function(3.14, 7, 3.5)
```

Out[116…
```
35.49
```

In [117…
```
my_function(10, 20)
```

Out[117…
```
45.0
```

The main restriction on function arguments is that the keyword arguments must follow the positional arguments (if any). You can specify keyword arguments in any order; this frees you from having to remember which order the function arguments were specified in and only what their names are.

It is possible to use keywords for passing positional arguments as well. In the preceding example, we could also have written:

In [119…
```
my_function(x=5, y=6, z=7)
```

Out[119…
```
77
```

In [120…
```
my_function(y=6, x=5, z=7)
```

Out[120…
```
77
```

Namespaces, Scope, and Local Functions

Functions can access variables in two different scopes: global and local. An alternative and more descriptive name describing a variable scope in Python is a namespace.

Any variables that are assigned within a function by default are assigned to the local namespace. The local namespace is created when the function is called and immediately populated by the function's arguments. After the function is finished, the local namespace is destroyed

In [123…
```
def func():
 a_z = []
 for i in range(5):
     a_z.append(i)
```

In [124…
```
a_z
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-124-7ac8f618dc62> in <module>
----> 1 a_z

NameError: name 'a_z' is not defined
```

In [125…
```
a_z = []
def func():
 for i in range(5):
     a_z.append(i)
```

In [126…
```
a_z
```

Out[126… []

Assigning variables outside of the function's scope is possible, but those variables must be declared as global via the global keyword:

In [127…
```
a = None
def bind_a_variable():
    global a
    a = []
bind_a_variable()
print(a)
```

```
[]
```

In [131…
```
a_z = []
def func():
    global a_z
    for i in range(5):
        a_z.append(i)
func()
print(a_z)
```

```
[0, 1, 2, 3, 4]
```

In [132…
```
global a_z
a_z = []
def func():
    for i in range(5):
        a_z.append(i)
func()
print(a_z)
```

```
[0, 1, 2, 3, 4]
```

I generally discourage use of the global keyword. Typically global variables are used to store some kind of state in a system. If you find yourself using a lot of them, it may indicate a need for objectoriented programming (using classes).

Returning Multiple Values

In [133…
```
def f():
 a = 5
 b = 6
 c = 7
 return a, b, c
a, b, c = f()
```

In [134…
```
print(a,b,c)
```

```
5 6 7
```

What's happening here is that the function is actually just returning one object, namely a tuple, which is then being unpacked into the result variables. In the preceding example, we could have done this instead:

In [135…
```
returned_value = f()
print(returned_value)
```

```
(5, 6, 7)
```

In [136…
```
def f():
 a = 5
 b = 6
 c = 7
 return {'a' : a, 'b' : b, 'c' : c}
```

In [137…
```
returned_value = f()
print(returned_value)
```

```
{'a': 5, 'b': 6, 'c': 7}
```

In [138…
```
a, b, c = f()
```

In [139…
```
print(a,b,c)
```

```
a b c
```

Functions Are Objects

Since Python functions are objects, many constructs can be easily expressed that are difficult to do in other languages. Suppose we were doing some data cleaning and needed to apply a bunch of transformations to the following list of strings:

In [140…
```
 states = [' Alabama ', 'Georgia!', 'Georgia', 'georgia', 'FlOrIda','south carolina
```

Anyone who has ever worked with user-submitted survey data has seen messy results like these. Lots of things need to happen to make this list of strings uniform and ready for analysis: stripping whitespace, removing punctuation symbols, and standardizing on proper capitalization. One way to do this is to use built-in string methods along with the re standard library module for regular expressions:

In [141…
```
import re
def clean_strings(strings):
 result = []
 for value in strings:
     value = value.strip()
     value = re.sub('[!#?]', '', value)
     value = value.title()
     result.append(value)
 return result
```

In [142…
```
clean_strings(states)
```

Out[142…
```
['Alabama',
 'Georgia',
 'Georgia',
 'Georgia',
 'Florida',
 'South Carolina',
 'West Virginia']
```

In [143…
```
def remove_punctuation(value):
 return re.sub('[!#?]', '', value)
clean_ops = [str.strip, remove_punctuation, str.title]
def clean_strings(strings, ops):
 result = []
 for value in strings:
     for function in ops:
         value = function(value)
     result.append(value)
 return result
```

In [144…
```
clean_strings(states, clean_ops)
```

Out[144…
```
['Alabama',
 'Georgia',
 'Georgia',
 'Georgia',
 'Florida',
 'South Carolina',
 'West Virginia']
```

A more functional pattern like this enables you to easily modify how the strings are transformed at a very high level. The clean_strings function is also now more reusable and generic.

You can use functions as arguments to other functions like the built-in map function, which applies a function to a sequence of some kind:

In [145…
```
for x in map(remove_punctuation, states):
    print(x)
```

```
 Alabama
Georgia
Georgia
georgia
FlOrIda
south carolina
West virginia
```

Anonymous (Lambda) Functions

Python has support for so-called anonymous or lambda functions, which are a way of writing functions consisting of a single statement, the result of which is the return value. They are defined with the lambda keyword, which has no meaning other than "we are declaring an anonymous function":

In [146…
```
def short_function(x):
 return x * 2
```

In [157…
```
equiv_anon = lambda x: x * 2
```

In [160…
```
equiv_anon(3)
```

Out[160…
```
6
```

In [154…
```
def apply_to_list(some_list, f):
 return [f(x) for x in some_list]
ints = [4, 0, 1, 5, 6]
apply_to_list(ints, lambda x: x * 2)
```

Out[154…
```
[8, 0, 2, 10, 12]
```

In [155…
```
strings = ['foo', 'card', 'bar', 'aaaa', 'abab']
strings.sort(key=lambda x: len(set(list(x))))
```

In [156…
```
strings
```

Out[156…
```
['aaaa', 'foo', 'abab', 'bar', 'card']
```

One reason lambda functions are called anonymous functions is that , unlike functions declared with the def keyword, the function object itself is never given an explicit __name__ attribute

Currying: Partial Argument Application

Currying is computer science jargon (named after the mathematician Haskell Curry) that means deriving new functions from existing ones by partial argument application. For example, suppose we had a trivial function that adds two numbers together:

In [161…
```
def add_numbers(x, y):
 return x + y
```

Using this function, we could derive a new function of one variable, add_five, that adds 5 to its argument:

In [162…
```
add_five = lambda y: add_numbers(5, y)
```

In [163…
```
add_five(10)
```

Out[163… 15

The second argument to add_numbers is said to be curried. There's nothing very fancy here, as all we've really done is define a new function that calls an existing function. The built-in functools module can simplify this process using the partial function:

In [164…
```
from functools import partial
add_five = partial(add_numbers, 5)
```

In [165…
```
add_five(101)
```

Out[165… 106

Generators

Having a consistent way to iterate over sequences, like objects in a list or lines in a file, is an important Python feature. This is accomplished by means of the iterator protocol, a generic way to make objects iterable. For example, iterating over a dict yields the dict keys:

In [166…
```
some_dict = {'a': 1, 'b': 2, 'c': 3}
for key in some_dict:
    print(key)
```

```
a
b
c
```

When you write for key in some_dict, the Python interpreter first attempts to create an iterator out of some_dict:

In [167…
```
dict_iterator = iter(some_dict)
dict_iterator
```

Out[167… <dict_keyiterator at 0x22ab9319f48>

An iterator is any object that will yield objects to the Python interpreter when used in a context like a for loop. Most methods expecting a list or list-like object will also accept any iterable object. This includes built-in methods such as min, max, and sum, and type constructors like list and tuple:

In [168…
```
list(dict_iterator)
```

Out[168…
```
['a', 'b', 'c']
```

A generator is a concise way to construct a new iterable object. Whereas normal functions execute and return a single result at a time, generators return a sequence of multiple results lazily, pausing after each one until the next one is requested. To create a generator, use the yield keyword instead of return in a function:

In [169…
```
def squares(n=10):
 print('Generating squares from 1 to {0}'.format(n ** 2))
 for i in range(1, n + 1):
     yield i ** 2
```

When you actually call the generator, no code is immediately executed:

In [170…
```
gen = squares()
```

In [171…
```
gen
```

Out[171…
```
<generator object squares at 0x0000022AB8C020F8>
```

It is not until you request elements from the generator that it begins executing its code:

In [172…
```
for x in gen:
    print(x, end=' ')
```

```
Generating squares from 1 to 100
1 4 9 16 25 36 49 64 81 100
```

Generator expresssions

Another even more concise way to make a generator is by using a generator expression. This is a generator analogue to list, dict, and set comprehensions; to create one, enclose what would otherwise be a list comprehension within parentheses instead of brackets:

In [173…
```
gen = (x ** 2 for x in range(100))
gen
```

Out[173…
```
<generator object <genexpr> at 0x0000022AB8C02150>
```

This is completely equivalent to the following more verbose generator:

In [174…
```
def _make_gen():
 for x in range(100):
     yield x ** 2
gen = _make_gen()
```

In [175…
```
sum(x ** 2 for x in range(100))
```

Out[175… 328350

In [176…
```
dict((i, i **2) for i in range(5))
```

Out[176… {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

The standard library itertools module has a collection of generators for many common data algorithms. For example, groupby takes any sequence and a function, grouping consecutive elements in the sequence by return value of the function. Here's an example:

In [177…
```
import itertools
first_letter = lambda x: x[0]
names = ['Alan', 'Adam', 'Wes', 'Will', 'Albert', 'Steven']
for letter, names in itertools.groupby(names, first_letter):
    print(letter, list(names)) # names is a generator
```

```
A ['Alan', 'Adam']
W ['Wes', 'Will']
A ['Albert']
S ['Steven']
```

In [178…
```
# Function --> Description

# combinations(iterable, k) --> Generates a sequence of all possible k-tuples of el
# permutations(iterable, k) --> Generates a sequence of all possible k-tuples of el
# groupby(iterable[, keyfunc]) --> Generates (key, sub-iterator) for each unique ke
# product(*iterables, repeat=1) --> Generates the Cartesian product of the input it
```

Errors and Exception Handling

Handling Python errors or exceptions gracefully is an important part of building robust programs. In data analysis applications, many functions only work on certain kinds of input. As an example, Python's float function is capable of casting a string to a floating-point number, but fails with ValueError on improper inputs:

In [180…
```
float('1.2345')
```

Out[180… 1.2345

In [181…
```
float('something')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-181-2649e4ade0e6> in <module>
----> 1 float('something')

ValueError: could not convert string to float: 'something'
```

Suppose we wanted a version of float that fails gracefully, returning the input argument. We can do this by writing a function that encloses the call to float in a try/ except block:

In [182…
```
def attempt_float(x):
 try:
     return float(x)
 except:
     return x
```

The code in the except part of the block will only be executed if float(x) raises an exception:

In [183…
```
attempt_float('1.2345')
```

Out[183…
```
1.2345
```

In [184…
```
attempt_float('something')
```

Out[184…
```
'something'
```

You might notice that float can raise exceptions other than ValueError:

In [185…
```
float((1, 2))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-185-82f777b0e564> in <module>
----> 1 float((1, 2))

TypeError: float() argument must be a string or a number, not 'tuple'
```

You might want to only suppress ValueError, since a TypeError (the input was not a string or numeric value) might indicate a legitimate bug in your program. To do that, write the exception type after except:

In [186…
```
def attempt_float(x):
 try:
     return float(x)
 except ValueError:
     return x
```

In [187…
```
attempt_float('something')
```

Out[187…
```
'something'
```

You can catch multiple exception types by writing a tuple of exception types instead (the parentheses are required):

In [188…
```
def attempt_float(x):
 try:
     return float(x)
 except (TypeError, ValueError):
     return x
```

In [190…
```
attempt_float((1, 2))
```

Out[190… (1, 2)

In some cases, you may not want to suppress an exception, but you want some code to be executed regardless of whether the code in the try block succeeds or not. To do this, use finally:

In [191…
```
# f = open(path, 'w')
# try:
#      write_to_file(f)
# finally:
#      f.close()
```

Here, the file handle f will always get closed. Similarly, you can have code that executes only if the try: block succeeds using else:

Similarly, you can have code that executes only if the try: block succeeds using else:

In [192…
```
# f = open(path, 'w')
# try:
#  write_to_file(f)
# except:
#  print('Failed')
# else:
#  print('Succeeded')
# finally:
#  f.close()
```

Exceptions in IPython

If an exception is raised while you are %run-ing a script or executing any statement, IPython will by default print a full call stack trace (traceback) with a few lines of context around the position at each point in the stack:

In [194…
```
# %run examples/ipython_bug.py
```

As you will see later in the chapter, you can step into the stack (using the %debug or %pdb magics) after an error has occurred for interactive post-mortem debugging

3.3 Files and the Operating System

To open a file for reading or writing, use the built-in open function with either a relative or absolute file path:

In [196…
```
# path = 'examples/segismundo.txt'
# f = open(path)
```

By default, the file is opened in read-only mode 'r'. We can then treat the file handle f like a list and iterate over the lines like so:

In [197…
```
# for line in f:
#  pass
```

The lines come out of the file with the end-of-line (EOL) markers intact, so you'll often see code to get an EOL-free list of lines in a file like:

In [198…
```
# lines = [x.rstrip() for x in open(path)]
# lines
```

When you use open to create file objects, it is important to explicitly close the file when you are finished with it. Closing the file releases its resources back to the oper- ating system

In [199…
```
# f.close()
```

One of the ways to make it easier to clean up open files is to use the with statement:

In [200…
```
# with open(path) as f:
#     lines = [x.rstrip() for x in f]
```

This will automatically close the file f when exiting the with block.

If we had typed f = open(path, 'w'), a new file at examples/segismundo.txt would have been created (be careful!), overwriting any one in its place. There is also the 'x' file mode, which creates a writable file but fails if the file path already exists. See Table 3-3 for a list of all valid file read/write modes.

For readable files, some of the most commonly used methods are read, seek, and tell. read returns a certain number of characters from the file. What constitutes a "character" is determined by the file's encoding (e.g., UTF-8) or simply raw bytes if the file is opened in binary mode:

In [201…
```
# f = open(path)
# f.read(10)
# f2 = open(path, 'rb') # Binary mode
# f2.read(10)
```

tell gives you the current position:

In [202…
```
# f.tell()
```

Even though we read 10 characters from the file, the position is 11 because it took that many bytes to decode 10 characters using the default encoding. You can check the default encoding in the sys module:

In [203…
```
import sys
sys.getdefaultencoding()
```

Out[203…
```
'utf-8'
```

seek changes the file position to the indicated byte in the file:

In [205…
```
# f.seek(3)
```

In [206…
```
# f.read(1)
```

In [207…
```
# Table 3-3. Python file modes

# Mode --> Description
# r --> Read-only mode
# w --> Write-only mode; creates a new file (erasing the data for any file with the
# x --> Write-only mode; creates a new file, but fails if the file path already exi
# a --> Append to existing file (create the file if it does not already exist)
# r+ --> Read and write b Add to mode for binary files (i.e., 'rb' or 'wb')
# t -->Text mode for files (automatically decoding bytes to Unicode). This is the d
# modes to use this (i.e., 'rt' or 'xt')
```

To write text to a file, you can use the file's write or writelines methods. For example, we could create a version of prof_mod.py with no blank lines like so:

In [208…
```
# with open('tmp.txt', 'w') as handle:
#     handle.writelines(x for x in open(path) if len(x) > 1)
# with open('tmp.txt') as f:
#     lines = f.readlines()
# lines
```

In [209…
```
# read([size]) --> Return data from file as a string, with optional size argument i
# readlines([size]) --> Return list of lines in the file, with optional size argume
# write(str) --> Write passed string to file
# writelines(strings) --> Write passed sequence of strings to the file
# close() --> Close the handle
# flush() --> Flush the internal I/O buffer to disk
# seek(pos) --> Move to indicated file position (integer)
# tell() --> Return current file position as integer
# closed --> True if the file is closed
```

The default behavior for Python files (whether readable or writable) is text mode, which means that you intend to work with Python strings (i.e., Unicode). This contrasts with binary

mode, which you can obtain by appending b onto the file mode. Let's look at the file (which contains non-ASCII characters with UTF-8 encoding) from the previous section:

In [210…
```
# with open(path) as f:
#     chars = f.read(10)
# chars
```

UTF-8 is a variable-length Unicode encoding, so when I requested some number of characters from the file, Python reads enough bytes (which could be as few as 10 or as many as 40 bytes) from the file to decode that many characters. If I open the file in 'rb' mode instead, read requests exact numbers of bytes:

In [211…
```
# with open(path, 'rb') as f:
#     data = f.read(10)
# data
```

Depending on the text encoding, you may be able to decode the bytes to a str object yourself, but only if each of the encoded Unicode characters is fully formed:

In [212…
```
# data.decode('utf8')
```

In [213…
```
# : data[:4].decode('utf8')
# ---------------------------------------------------------------------------
# UnicodeDecodeError Traceback (most recent call last)
# <ipython-input-235-300e0af10bb7> in <module>()
# ----> 1 data[:4].decode('utf8')
# UnicodeDecodeError: 'utf-8' codec can't decode byte 0xc3 in position 3: unexpecte
# d end of data
```

Text mode, combined with the encoding option of open, provides a convenient way to convert from one Unicode encoding to another:

In [215…
```
# \sink_path = 'sink.txt'
# with open(path) as source:
#     with open(sink_path, 'xt', encoding='iso-8859-1') as sink:
#         sink.write(source.read())
# with open(sink_path, encoding='iso-8859-1') as f:
#     print(f.read(10))
```

Beware using seek when opening files in any mode other than binary. If the file position falls in the middle of the bytes defining a Unicode character, then subsequent reads will result in an error:

In [217…
```
# f = open(path)
# f.read(5)
# Out[241]: 'Sueña'
# In [242]: f.seek(4)
# Out[242]: 4
# In [243]: f.read(1)
# ------------------------
```

```
# UnicodeDecodeError Traceback (most recent call last)
# <ipython-input-243-7841103e33f5> in <module>()
# ----> 1 f.read(1)
# /miniconda/envs/book-env/lib/python3.6/codecs.py in decode(self, input, final)
#  319 # decode input (taking the buffer into account)
#  320 data = self.buffer + input
# --> 321 (result, consumed) = self._buffer_decode(data, self.errors, final
# )
#  322 # keep undecoded input until the next call
#  323 self.buffer = data[consumed:]
# UnicodeDecodeError: 'utf-8' codec can't decode byte 0xb1 in position 0: invalid s
# tart byte
```

In [ ]: