



Capítulo 3. Estructuras de datos integradas, funciones y archivos

This chapter discusses capabilities built into the Python language that will be used ubiquitously throughout the book. While add-on libraries like pandas and NumPy add advanced computational functionality for larger datasets, they are designed to be used together with Python's built-in data manipulation tools.

We'll start with Python's workhorse data structures: tuples, lists, dicts, and sets. Then, we'll discuss creating your own reusable Python functions. Finally, we'll look at the mechanics of Python file objects and interacting with your local hard drive.

3.1 Data Structures and Sequences

Python's data structures are simple but powerful. Mastering their use is a critical part of becoming a proficient Python programmer.

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)
```



When you're defining tuples in more complicated expressions, it's often necessary to enclose the values in parentheses, as in this example of creating a tuple of tuples:

```
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 [5]: tuple([4, 0, 2])
Out[5]: (4, 0, 2)

In [6]: tup = tuple('string')

In [7]: tup
Out[7]: ('s', 't', 'r', 'i', 'n', 'g')
```

Elements can be accessed with square brackets `[]` as with most other sequence types. As in C, C++, Java, and many other languages, sequences are 0-indexed in Python:

```
In [8]: tup[0]
Out[8]: '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 [9]: tup = tuple(['foo', [1, 2], True])

In [10]: tup[2] = False
-----
TypeError                                 Traceback (most recent call)
<ipython-input-10-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 [11]: tup[1].append(3)

In [12]: tup
Out[12]: ('foo', [1, 2, 3], True)
```

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

```
In [13]: (4, None, 'foo') + (6, 0) + ('bar',)
Out[13]: (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 [14]: ('foo', 'bar') * 4
Out[14]: ('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 [15]: tup = (4, 5, 6)

In [16]: a, b, c = tup

In [17]: b
Out[17]: 5
```

Even sequences with nested tuples can be unpacked:

```
In [18]: tup = 4, 5, (6, 7)

In [19]: a, b, (c, d) = tup

In [20]: d
Out[20]: 7
```

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

```
tmp = a
a = b
b = tmp
```

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

```
In [21]: a, b = 1, 2

In [22]: a
Out[22]: 1

In [23]: b
Out[23]: 2

In [24]: b, a = a, b

In [25]: a
Out[25]: 2

In [26]: b
Out[26]: 1
```

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

```
In [27]: seq = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]

In [28]: 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
```

Another common use is returning multiple values from a function. I'll cover this in more detail later.

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 [29]: values = 1, 2, 3, 4, 5

In [30]: a, b, *rest = values

In [31]: a, b
Out[31]: (1, 2)

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

This `rest` bit is sometimes something you want to discard; there is nothing special about the `rest` name. As a matter of convention, many Python programmers will use the underscore (`_`) for unwanted variables:

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

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 [34]: a = (1, 2, 2, 2, 3, 4, 2)

In [35]: a.count(2)
Out[35]: 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 [36]: a_list = [2, 3, 7, None]

In [37]: tup = ('foo', 'bar', 'baz')

In [38]: b_list = list(tup)

In [39]: b_list
Out[39]: ['foo', 'bar', 'baz']

In [40]: b_list[1] = 'peekaboo'
```



```
In [41]: b_list
Out[41]: ['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 [42]: gen = range(10)

In [43]: gen
Out[43]: range(0, 10)

In [44]: list(gen)
Out[44]: [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 [45]: b_list.append('dwarf')

In [46]: b_list
Out[46]: ['foo', 'peekaboo', 'baz', 'dwarf']
```

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

```
In [47]: b_list.insert(1, 'red')

In [48]: b_list
Out[48]: ['foo', 'red', 'peekaboo', 'baz', 'dwarf']
```

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

WARNING

`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 [49]: b_list.pop(2)
Out[49]: 'peekaboo'
```



```
In [50]: b_list
Out[50]: ['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 [51]: b_list.append('foo')

In [52]: b_list
Out[52]: ['foo', 'red', 'baz', 'dwarf', 'foo']

In [53]: b_list.remove('foo')

In [54]: b_list
Out[54]: ['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 [55]: 'dwarf' in b_list
Out[55]: True
```

The keyword `not` can be used to negate `in`:

```
In [56]: 'dwarf' not in b_list
Out[56]: 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 [57]: [4, None, 'foo'] + [7, 8, (2, 3)]
Out[57]: [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 [58]: x = [4, None, 'foo']

In [59]: x.extend([7, 8, (2, 3)])

In [60]: x
Out[60]: [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. Thus,

```
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 [61]: a = [7, 2, 5, 1, 3]

In [62]: a.sort()

In [63]: a
Out[63]: [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 [64]: b = ['saw', 'small', 'He', 'foxes', 'six']

In [65]: b.sort(key=len)

In [66]: b
Out[66]: ['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 [67]: import bisect

In [68]: c = [1, 2, 2, 2, 3, 4, 7]

In [69]: bisect.bisect(c, 2)
Out[69]: 4

In [70]: bisect.bisect(c, 5)
Out[70]: 6

In [71]: bisect.insort(c, 6)
```



```
In [72]: c
Out[72]: [1, 2, 2, 2, 3, 4, 6, 7]
```

CAUTION

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 [73]: seq = [7, 2, 3, 7, 5, 6, 0, 1]

In [74]: seq[1:5]
Out[74]: [2, 3, 7, 5]
```

Slices can also be assigned to with a sequence:

```
In [75]: seq[3:4] = [6, 3]

In [76]: seq
Out[76]: [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 [77]: seq[:5]
Out[77]: [7, 2, 3, 6, 3]

In [78]: seq[3:]
Out[78]: [6, 3, 5, 6, 0, 1]
```

Negative indices slice the sequence relative to the end:

```
In [79]: seq[-4:]
Out[79]: [5, 6, 0, 1]

In [80]: seq[-6:-2]
Out[80]: [6, 3, 5, 6]
```

Slicing semantics takes a bit of getting used to, especially if you're coming from R or MATLAB. See [Figure 3-1](#) for a helpful illustration of slicing with positive and negative integers. In the figure, the indices are shown at the “bin edges” to



help show where the slice selections start and stop using positive or negative indices.

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

```
In [81]: seq[::2]
Out[81]: [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 [82]: seq[::-1]
Out[82]: [1, 0, 6, 5, 3, 6, 3, 2, 7]
```

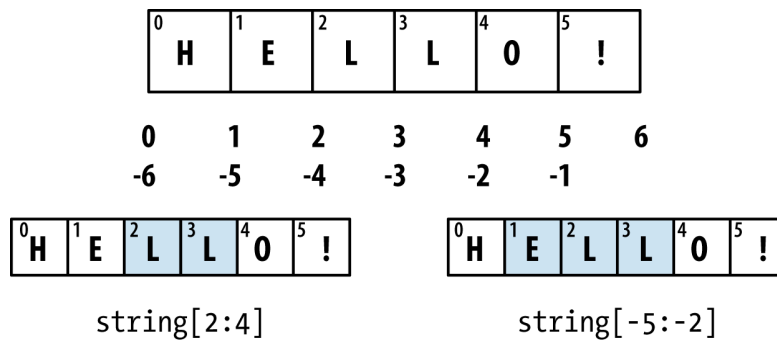


Figure 3-1. Illustration of Python slicing conventions

Built-in Sequence Functions

Python has a handful of useful sequence functions that you should familiarize yourself 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
```

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 [83]: some_list = ['foo', 'bar', 'baz']

In [84]: mapping = {}

In [85]: for i, v in enumerate(some_list):
....:     mapping[v] = i

In [86]: mapping
Out[86]: {'bar': 1, 'baz': 2, 'foo': 0}
```

SORTED

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

```
In [87]: sorted([7, 1, 2, 6, 0, 3, 2])
Out[87]: [0, 1, 2, 2, 3, 6, 7]

In [88]: sorted('horse race')
Out[88]: [' ', 'a', 'c', 'e', 'e', 'h', 'o', 'r', 'r', 's']
```

The `sorted` function accepts the same arguments as the `sort` method on lists.

ZIP

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

```
In [89]: seq1 = ['foo', 'bar', 'baz']

In [90]: seq2 = ['one', 'two', 'three']

In [91]: zipped = zip(seq1, seq2)

In [92]: list(zipped)
Out[92]: [('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 [93]: seq3 = [False, True]

In [94]: list(zip(seq1, seq2, seq3))
Out[94]: [('foo', 'one', False), ('bar', 'two', True)]
```

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

```
In [95]: 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 [96]: pitchers = [('Nolan', 'Ryan'), ('Roger', 'Clemens'),
.....:               ('Schilling', 'Curt')]

In [97]: first_names, last_names = zip(*pitchers)

In [98]: first_names
Out[98]: ('Nolan', 'Roger', 'Schilling')

In [99]: last_names
Out[99]: ('Ryan', 'Clemens', 'Curt')
```

REVERSED

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

```
In [100]: list(reversed(range(10)))
Out[100]: [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 [101]: empty_dict = {}

In [102]: d1 = {'a': 'some value', 'b': [1, 2, 3, 4]}

In [103]: d1
Out[103]: {'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 [104]: d1[7] = 'an integer'

In [105]: d1
Out[105]: {7: 'an integer', 'a': 'some value', 'b': [1, 2, 3, 4]}

In [106]: d1['b']
Out[106]: [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 [107]: 'b' in d1
Out[107]: True
```



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

```
In [108]: d1[5] = 'some value'

In [109]: d1
Out[109]:
{5: 'some value',
 7: 'an integer',
 'a': 'some value',
 'b': [1, 2, 3, 4]}

In [110]: d1['dummy'] = 'another value'

In [111]: d1
Out[111]:
{5: 'some value',
 7: 'an integer',
 'a': 'some value',
 'b': [1, 2, 3, 4],
 'dummy': 'another value'}

In [112]: del d1[5]

In [113]: d1
Out[113]:
{7: 'an integer',
 'a': 'some value',
 'b': [1, 2, 3, 4],
 'dummy': 'another value'}

In [114]: ret = d1.pop('dummy')

In [115]: ret
Out[115]: 'another value'

In [116]: d1
Out[116]: {7: 'an integer', 'a': 'some value', 'b': [1, 2, 3, 4]}
```

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 [117]: list(d1.keys())
Out[117]: ['a', 'b', 7]

In [118]: list(d1.values())
Out[118]: ['some value', [1, 2, 3, 4], 'an integer']
```

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

```
In [119]: d1.update({'b': 'foo', 'c': 12})

In [120]: d1
Out[120]: {7: 'an integer', 'a': 'some value', 'b': 'foo', '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:

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

In [122]: mapping
Out[122]: {0: 4, 1: 3, 2: 2, 3: 1, 4: 0}
```

Later we'll talk about *dict comprehensions*, another elegant way to construct dicts.

DEFAULT VALUES

It's very common to have logic like:

```
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:

```
value = some_dict.get(key, default_value)
```

`get` by default will return `None` if the key is not present, while `pop` will raise an exception. With *setting* values, a common case is for the values in a dict to be other collections, like lists. For example, you could imagine categorizing a list of words by their first letters as a dict of lists:

```
In [123]: words = ['apple', 'bat', 'bar', 'atom', 'book']

In [124]: by_letter = {}

In [125]: for word in words:
.....:     letter = word[0]
.....:     if letter not in by_letter:
.....:         by_letter[letter] = [word]
.....:     else:
.....:         by_letter[letter].append(word)
.....:

In [126]: by_letter
Out[126]: {'a': ['apple', 'atom'], 'b': ['bat', 'bar', 'book']}
```

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



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

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:

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

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 [127]: hash('string')
Out[127]: -305472831944956388

In [128]: hash((1, 2, (2, 3)))
Out[128]: 1097636502276347782

In [129]: hash((1, 2, [2, 3])) # fails because lists are mutable
-----
TypeError                                 Traceback (most recent call)
<ipython-input-129-473c35a62c0b> 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 [130]: d = {}

In [131]: d[tuple([1, 2, 3])] = 5

In [132]: d
Out[132]: {(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 [133]: set([2, 2, 2, 1, 3, 3])
Out[133]: {1, 2, 3}

In [134]: {2, 2, 2, 1, 3, 3}
Out[134]: {1, 2, 3}
```



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

```
In [135]: a = {1, 2, 3, 4, 5}
```

```
In [136]: 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 [137]: a.union(b)
```

```
Out[137]: {1, 2, 3, 4, 5, 6, 7, 8}
```

```
In [138]: a | b
```

```
Out[138]: {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 [139]: a.intersection(b)
```

```
Out[139]: {3, 4, 5}
```

```
In [140]: a & b
```

```
Out[140]: {3, 4, 5}
```

See [Table 3-1](#) for a list of commonly used set methods.



Table 3-1. Python set operations

Function	Alternative syntax	Description
<code>a.add(x)</code>	N/A	Add element <code>x</code> to the set <code>a</code>
<code>a.clear()</code>	N/A	Reset the set <code>a</code> to an empty state, discarding all of its elements
<code>a.remove(x)</code>	N/A	Remove element <code>x</code> from the set <code>a</code>
<code>a.pop()</code>	N/A	Remove an arbitrary element from the set <code>a</code> , raising <code>KeyError</code> if the set is empty
<code>a.union(b)</code>	<code>a b</code>	All of the unique elements in <code>a</code> and <code>b</code>
<code>a.update(b)</code>	<code>a = b</code>	Set the contents of <code>a</code> to be the union of the elements in <code>a</code> and <code>b</code>
<code>a.intersection(b)</code>	<code>a & b</code>	All of the elements in <i>both</i> <code>a</code> and <code>b</code>
<code>a.intersection_update(b)</code>	<code>a &= b</code>	Set the contents of <code>a</code> to be the intersection of the elements in <code>a</code> and <code>b</code>
<code>a.difference(b)</code>	<code>a - b</code>	The elements in <code>a</code> that are not in <code>b</code>
<code>a.difference_update(b)</code>	<code>a -= b</code>	Set <code>a</code> to the elements in <code>a</code> that are not in <code>b</code>



Function	Alternative syntax	Description
<code>a.symmetric_difference(b)</code>	<code>a ^ b</code>	All of the elements in either <code>a</code> or <code>b</code> but <i>not both</i>
<code>a.symmetric_difference_update(b)</code>	<code>a ^= b</code>	Set <code>a</code> to contain the elements in either <code>a</code> or <code>b</code> but <i>not both</i>
<code>a.issubset(b)</code>	N/A	True if the elements of <code>a</code> are all contained in <code>b</code>
<code>a.issuperset(b)</code>	N/A	True if the elements of <code>b</code> are all contained in <code>a</code>
<code>a.isdisjoint(b)</code>	N/A	True if <code>a</code> and <code>b</code> 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 [141]: c = a.copy()

In [142]: c |= b

In [143]: c
Out[143]: {1, 2, 3, 4, 5, 6, 7, 8}

In [144]: d = a.copy()

In [145]: d &= b

In [146]: d
Out[146]: {3, 4, 5}
```

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

```
In [147]: my_data = [1, 2, 3, 4]

In [148]: my_set = {tuple(my_data)}

In [149]: my_set
Out[149]: {(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 [150]: a_set = {1, 2, 3, 4, 5}

In [151]: {1, 2, 3}.issubset(a_set)
Out[151]: True

In [152]: a_set.issuperset({1, 2, 3})
Out[152]: True
```

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

```
In [153]: {1, 2, 3} == {3, 2, 1}
Out[153]: 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)
```

The filter condition can be omitted, leaving only the expression. For example, given a list of strings, we could filter out strings with length 2 or less and also convert them to uppercase like this:

```
In [154]: strings = ['a', 'as', 'bat', 'car', 'dove', 'python']

In [155]: [x.upper() for x in strings if len(x) > 2]
Out[155]: ['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:

```
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:

```
set_comp = {expr for value in collection if condition}
```



Like list comprehensions, set and dict comprehensions are mostly conveniences, but they similarly can make code both easier to write and read. Consider the list of strings from before. Suppose we wanted a set containing just the lengths of the strings contained in the collection; we could easily compute this using a set comprehension:

```
In [156]: unique_lengths = {len(x) for x in strings}

In [157]: unique_lengths
Out[157]: {1, 2, 3, 4, 6}
```

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

```
In [158]: set(map(len, strings))
Out[158]: {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 [159]: loc_mapping = {val : index for index, val in enumerate(strings)}

In [160]: loc_mapping
Out[160]: {'a': 0, 'as': 1, 'bat': 2, 'car': 3, 'dove': 4, 'python':
```



NESTED LIST COMPREHENSIONS

Suppose we have a list of lists containing some English and Spanish names:

```
In [161]: all_data = [['John', 'Emily', 'Michael', 'Mary', 'Steven'],
.....:               ['Maria', 'Juan', 'Javier', 'Natalia', 'Pilar']]
```



You might have gotten these names from a couple of files and decided to organize them by language. Now, suppose we wanted to get a single list containing all names with two or more `e`'s in them. We could certainly do this with a simple `for` loop:

```
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)
```

You can actually wrap this whole operation up in a single *nested list comprehension*, which will look like:

```
In [162]: result = [name for names in all_data for name in names
.....:               if name.count('e') >= 2]

In [163]: result
Out[163]: ['Steven']
```



At first, nested list comprehensions are a bit hard to wrap your head around. The **for** parts of the list comprehension are arranged according to the order of nesting, and any filter condition is put at the end as before. Here is another example where we “flatten” a list of tuples of integers into a simple list of integers:

```
In [164]: some_tuples = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]

In [165]: flattened = [x for tup in some_tuples for x in tup]

In [166]: flattened
Out[166]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Keep in mind that the order of the **for** expressions would be the same if you wrote a nested **for** loop instead of a list comprehension:

```
flattened = []

for tup in some_tuples:
    for x in tup:
        flattened.append(x)
```

You can have arbitrarily many levels of nesting, though if you have more than two or three levels of nesting you should probably start to question whether this makes sense from a code readability standpoint. It’s important to distinguish the syntax just shown from a list comprehension inside a list comprehension, which is also perfectly valid:

```
In [167]: [[x for x in tup] for tup in some_tuples]
Out[167]: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

This produces a list of lists, rather than a flattened list of all of the inner elements.

3.2 Functions

Functions are the primary and most important method of code organization and reuse in Python. As a rule of thumb, if you anticipate needing to repeat the same or very similar code more than once, it may be worth writing a reusable function. Functions can also help make your code more readable by giving a name to a group of Python statements.

Functions are declared with the **def** keyword and returned from with the **return** keyword:

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

There is no issue with having multiple **return** statements. If Python reaches the end of a function without encountering a **return** statement, **None** is returned automatically.



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:

```
my_function(5, 6, z=0.7)
my_function(3.14, 7, 3.5)
my_function(10, 20)
```

La principal restricción en los argumentos de función es que los argumentos de palabras clave *deben* seguir los argumentos posicionales (si los hay). Puede especificar argumentos de palabras clave en cualquier orden; esto le libera de tener que recordar en qué orden se especificaron los argumentos de la función y solo cuáles son sus nombres.

NOTA

También es posible usar palabras clave para pasar argumentos posicionales. En el ejemplo anterior, también podríamos haber escrito:

```
my_function(x=5, y=6, z=7)
my_function(y=6, x=5, z=7)
```

En algunos casos, esto puede ayudar con la legibilidad.

Espacios de nombres, alcance y funciones locales

Las funciones pueden acceder a variables en dos ámbitos diferentes: *global* y *local*. Un nombre alternativo y más descriptivo que describe un alcance variable en Python es un *espacio de nombres*. Cualquier variable asignada dentro de una función por defecto se asigna al espacio de nombres local. El espacio de nombres local se crea cuando se llama a la función y se llena inmediatamente con los argumentos de la función. Una vez finalizada la función, se destruye el espacio de nombres local (con algunas excepciones que están fuera del alcance de este capítulo). Considere la siguiente función:

```
def func():
    a = []
    for i in range(5):
        a.append(i)
```

Cuando `func()` se llama, se crea la lista vacía, se añaden cinco elementos y luego se destruye cuando la función sale. Supongamos, en cambio, que hubiéramos declarado lo siguiente:

```
a = []
def func():
```



```
for i in range(5):  
    a.append(i)
```

Es posible asignar variables fuera del alcance de la función, pero esas variables deben declararse como globales mediante la `global` palabra clave:

```
In [168]: a = None  
  
In [169]: def bind_a_variable():  
.....:     global a  
.....:     a = []  
.....:     bind_a_variable()  
.....:  
  
In [170]: print(a)  
[]
```

PRECAUCIÓN

Generalmente desaconsejo el uso de la `global` palabra clave. Por lo general, las variables globales se utilizan para almacenar algún tipo de estado en un sistema. Si se encuentra usando muchos de ellos, puede indicar la necesidad de una programación orientada a objetos (usando clases).

Devolviendo valores múltiples

Cuando programé por primera vez Python después de haber programado en Java y C++, una de mis características favoritas fue la capacidad de devolver múltiples valores de una función con sintaxis simple. Aquí hay un ejemplo:

```
def f():  
    a = 5  
    b = 6  
    c = 7  
    return a, b, c  
  
a, b, c = f()
```

En el análisis de datos y otras aplicaciones científicas, puede encontrarse haciendo esto a menudo. Lo que sucede aquí es que la función en realidad solo devuelve *un* objeto, es decir, una tupla, que luego se descomprime en las variables de resultado. En el ejemplo anterior, podríamos haber hecho esto en su lugar:

```
return_value = f()
```

En este caso, `return_value` sería una tupla de 3 con las tres variables devueltas. Una alternativa potencialmente atractiva para devolver múltiples valores como antes podría ser devolver un dict en su lugar:



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

Esta técnica alternativa puede ser útil según lo que intente hacer.

Las funciones son objetos

Desde Python las funciones son objetos, muchas construcciones se pueden expresar fácilmente y son difíciles de hacer en otros idiomas. Supongamos que estábamos limpiando algunos datos y necesitábamos aplicar un montón de transformaciones a la siguiente lista de cadenas:

```
In [171]: states = [' Alabama ', 'Georgia!', 'Georgia', 'georgia',
.....:             'south carolina##', 'West virginia?']
```

Cualquiera que haya trabajado con datos de encuestas enviadas por usuarios ha visto resultados desordenados como estos. Es necesario que sucedan muchas cosas para que esta lista de cadenas sea uniforme y esté lista para el análisis: eliminar espacios en blanco, eliminar símbolos de puntuación y estandarizar la capitalización adecuada. Una forma de hacerlo es usar métodos de cadena integrados junto con el `reestándar` módulo de biblioteca para expresiones regulares:

```
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
```

El resultado se ve así:

```
In [173]: clean_strings(states)
Out[173]:
['Alabama',
 'Georgia',
 'Georgia',
 'Georgia',
 'Florida',
 'South Carolina',
 'West Virginia']
```

Un enfoque alternativo que puede resultarle útil es hacer una lista de las operaciones que desea aplicar a un conjunto particular de cadenas:

```
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
```

Luego tenemos lo siguiente:

```
In [175]: clean_strings(states, clean_ops)
Out[175]:
['Alabama',
 'Georgia',
 'Georgia',
 'Georgia',
 'Florida',
 'South Carolina',
 'West Virginia']
```

Un patrón más *funcional* como este le permite modificar fácilmente cómo se transforman las cadenas en un nivel muy alto. La `clean_strings` función ahora también es más reutilizable y genérica.

Puede usar funciones como argumentos para otras funciones como el incorporado en `map` función, que aplica una función a una secuencia de algun tipo:

```
In [176]: for x in map(remove_punctuation, states):
.....:     print(x)
Alabama
Georgia
Georgia
georgia
FlOrIda
south carolina
West virginia
```

Funciones anónimas (lambda)

Python tiene soporte para las llamadas funciones *anónimas* o *lambda*, que son una forma de escribir funciones que consisten en una sola declaración, cuyo resultado es el valor de retorno. Se definen con la `lambda` palabra clave, que no tiene otro significado que "estamos declarando una función anónima":

```
def short_function(x):
    return x * 2

equiv_anon = lambda x: x * 2
```

Por lo general, me refiero a estas funciones `lambda` en el resto del libro. Son especialmente convenientes en el análisis de datos porque, como verá, hay muchos casos en los que las funciones de transformación de datos tomarán funciones como argumentos. A menudo es menos tipeado (y más claro) pasar una función `lambda` en lugar de escribir una declaración de función completa o



incluso asignar la función lambda a una variable local. Por ejemplo, considere este ejemplo tonto:

```
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)
```

También podría haber escrito `[x * 2 for x in ints]`, pero aquí pudimos pasar sucintamente un operador personalizado a la `apply_to_list` función.

Como otro ejemplo, suponga que desea ordenar una colección de cadenas por el número de letras distintas en cada cadena:

```
In [177]: strings = ['foo', 'card', 'bar', 'aaaa', 'abab']
```

Aquí podríamos pasar una función lambda a `sort` método de la lista :

```
In [178]: strings.sort(key=lambda x: len(set(list(x))))  
  
In [179]: strings  
Out[179]: ['aaaa', 'foo', 'abab', 'bar', 'card']
```

NOTA

Una razón por la cual las funciones lambda se llaman funciones anónimas es que, a diferencia de las funciones declaradas con la `def` palabra clave, el objeto de función en sí mismo nunca recibe un `__name__` atributo explícito .

Curry: aplicación de argumento parcial

Curry es computadorajerga científica (llamada así por el matemático Haskell Curry) que significa derivar nuevas funciones de las existentes mediante *la aplicación de argumentos parciales* . Por ejemplo, supongamos que tenemos una función trivial que suma dos números:

```
def add_numbers(x, y):  
    return x + y
```

Usando esta función, podríamos derivar una nueva función de una variable `add_five`, que agrega 5 a su argumento:

```
add_five = lambda y: add_numbers(5, y)
```

Se `add_numbers` dice que el segundo argumento es *curry* . Aquí no hay nada muy elegante, ya que todo lo que hemos hecho realmente es definir una nueva



función que llame a una función existente. El `functools` módulo incorporado puede simplificar este proceso usando la `partial` función:

```
from functools import partial
add_five = partial(add_numbers, 5)
```

Generadores

Tener un consistente La forma de iterar sobre secuencias, como objetos en una lista o líneas en un archivo, es una característica importante de Python. Esto se logra por medio del *protocolo iterador*, una forma genérica de hacer que los objetos sean iterables. Por ejemplo, iterar sobre un dict produce las claves dict:

```
In [180]: some_dict = {'a': 1, 'b': 2, 'c': 3}

In [181]: for key in some_dict:
.....:     print(key)
a
b
c
```

Cuando escribe `for key in some_dict`, el intérprete de Python primero intenta crear un iterador a partir de `some_dict`:

```
In [182]: dict_iterator = iter(some_dict)

In [183]: dict_iterator
Out[183]: <dict_keyiterator at 0x7ff84e90ee58>
```

Un iterador es cualquier objeto que cederá objetos al intérprete de Python cuando se use en un contexto como un `for` bucle. La mayoría de los métodos que esperan una lista o un objeto similar a una lista también aceptarán cualquier objeto iterable. Esto incluye una función de métodos tales como `min`, `max`, `sum`, y constructores de tipos como `list` y `tuple`:

```
In [184]: list(dict_iterator)
Out[184]: ['a', 'b', 'c']
```

Un *generador* es una forma concisa de construir un nuevo objeto iterable. Mientras que las funciones normales ejecutan y devuelven un solo resultado a la vez, los generadores devuelven una secuencia de resultados múltiples perezosamente, haciendo una pausa después de cada uno hasta que se solicite el siguiente. Para crear un generador, usa la `yield` palabra clave en lugar de `return` en una función:

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

Cuando realmente llama al generador, no se ejecuta ningún código de inmediato:



```
In [186]: gen = squares()

In [187]: gen
Out[187]: <generator object squares at 0x7ff84e92bbf8>
```

No es hasta que solicita elementos del generador que comienza a ejecutar su código:

```
In [188]: for x in gen:
.....:     print(x, end=' ')
Generating squares from 1 to 100
1 4 9 16 25 36 49 64 81 100
```

GENERADOR DE EXPRESIONES

Otro aun mas La forma concisa de hacer un generador es mediante el uso de una *expresión de generador* . Este es un generador análogo para enumerar, dictar y establecer comprensiones; para crear uno, encierre lo que de otro modo sería una comprensión de lista entre paréntesis en lugar de paréntesis:

```
In [189]: gen = (x ** 2 for x in range(100))

In [190]: gen
Out[190]: <generator object <genexpr> at 0x7ff84e92b150>
```

Esto es completamente equivalente al siguiente generador más detallado:

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

Las expresiones generadoras se pueden usar en lugar de las comprensiones de listas como argumentos de función en muchos casos:

```
In [191]: sum(x ** 2 for x in range(100))
Out[191]: 328350

In [192]: dict((i, i **2) for i in range(5))
Out[192]: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

MÓDULO ITERTOOLS

El `itertools` módulo de biblioteca estándar tiene una colección de generadores para muchos algoritmos de datos comunes. Por ejemplo, `groupby` toma cualquier secuencia y una función, agrupando elementos consecutivos en la secuencia por valor de retorno de la función. Aquí hay un ejemplo:

```
In [193]: import itertools

In [194]: first_letter = lambda x: x[0]

In [195]: names = ['Alan', 'Adam', 'Wes', 'Will', 'Albert', 'Steven']

In [196]: 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']
```

Consulte la [Tabla 3-2](#) para obtener una lista de algunas otras `itertools` funciones que he encontrado útiles con frecuencia. Es posible que desee consultar [la documentación oficial de Python](#) para obtener más información sobre este útil módulo de utilidad incorporado .

Tabla 3-2. Algunas funciones útiles de `itertools`

Función	Descripción
<code>combinations(iterable, k)</code>	Genera un secuencia de todas las ktuplas posibles de elementos en el orden iterable, ignorando y sin reemplazo (ver también la función complementaria <code>combinations_with_replacement</code>)
<code>permutations(iterable, k)</code>	Genera un secuencia de todas las ktuplas de elementos posibles en el orden iterable y respetuoso
<code>groupby(iterable[, keyfunc])</code>	Genera (key, sub-iterator) paracada clave única
<code>product(*iterables, repeat=1)</code>	Genera el Producto cartesiano de los iterables de entrada como tuplas, similar a un <code>for</code> bucle anidado

Manejo de errores y excepciones

Manejo de Python errores o *excepciones* con gracia es una parte importante de la construcción de programas robustos. En las aplicaciones de análisis de datos, muchas funciones solo funcionan en ciertos tipos de entrada. Como ejemplo, la `float` función de Python es capaz de lanzar una cadena a un número de coma flotante, pero falla con `ValueError` entradas incorrectas:

```
In [197]: float('1.2345')
Out[197]: 1.2345

In [198]: float('something')
-----
ValueError                                Traceback (most recent call)
<ipython-input-198-2649e4ade0e6> in <module>()
----> 1 float('something')
ValueError: could not convert string to float: 'something'
```



Supongamos que deseamos que una versión de `float` falle con gracia, devolviendo el argumento de entrada. Podemos hacer esto escribiendo una función que encierra la llamada a `float` en un `try/except` bloque:

```
def attempt_float(x):  
    try:  
        return float(x)  
    except:  
        return x
```

El código en la `except` parte del bloque solo se ejecutará si se `float(x)` produce una excepción:

```
In [200]: attempt_float('1.2345')  
Out[200]: 1.2345  
  
In [201]: attempt_float('something')  
Out[201]: 'something'
```

Puede notar que `float` puede generar excepciones que no sean `ValueError`:

```
In [202]: float((1, 2))  
-----  
TypeError                                 Traceback (most recent call  
<ipython-input-202-82f777b0e564> in <module>()  
----> 1 float((1, 2))  
TypeError: float() argument must be a string or a number, not 'tuple'
```

Es posible que solo desee suprimir `ValueError`, ya que a `TypeError` (ella entrada no era una cadena o un valor numérico) podría indicar un error legítimo en su programa. Para hacer eso, escriba el tipo de excepción después de `except`:

```
def attempt_float(x):  
    try:  
        return float(x)  
    except ValueError:  
        return x
```

Tenemos entonces:

```
In [204]: attempt_float((1, 2))  
-----  
TypeError                                 Traceback (most recent call  
<ipython-input-204-8b0026e9e6b7> in <module>()  
----> 1 attempt_float((1, 2))  
<ipython-input-203-d99a2a135508> in attempt_float(x)  
      1 def attempt_float(x):  
      2     try:  
----> 3         return float(x)  
      4     except ValueError:  
      5         return x  
TypeError: float() argument must be a string or a number, not 'tuple'
```



Puede atrapar varios tipos de excepción escribiendo una tupla de tipos de excepción (los paréntesis son obligatorios):

```
def attempt_float(x):
    try:
        return float(x)
    except (TypeError, ValueError):
        return x
```

En algunos casos, es posible que no desee suprimir una excepción, pero desea que se ejecute algún código independientemente de si el código en el `try` bloque tiene éxito o no. Para hacer esto, use `finally`:

```
f = open(path, 'w')

try:
    write_to_file(f)
finally:
    f.close()
```

En este caso, el identificador de archivo `f` será *siempre* se cierran. Del mismo modo, puede tener código que se ejecute solo si el `try` bloque tiene éxito utilizando `else`:

```
f = open(path, 'w')

try:
    write_to_file(f)
except:
    print('Failed')
else:
    print('Succeeded')
finally:
    f.close()
```

EXCEPCIONES EN IPYTHON

Si se plantea una excepción mientras que usted está `%run-ing` una script o ejecutando cualquier instrucción, IPython imprimirá por defecto un seguimiento completo de la pila de llamadas (traceback) con algunas líneas de contexto alrededor de la posición en cada punto de la pila:

```
In [10]: %run examples/ipython_bug.py

-----
AssertionError                                Traceback (most recent call
/home/wesm/code/pydata-book/examples/ipython_bug.py in <module>()
     13     throws_an_exception()
     14
--> 15 calling_things()

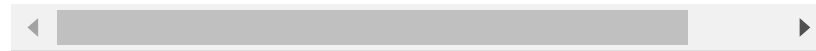
/home/wesm/code/pydata-book/examples/ipython_bug.py in calling_things()
     11 def calling_things():
     12     works_fine()
--> 13     throws_an_exception()
     14
     15 calling_things()

/home/wesm/code/pydata-book/examples/ipython_bug.py in throws_an_except
     7     a = 5
```



```
8     b = 6
----> 9     assert(a + b == 10)
10
11 def calling_things():
```

AssertionError:



Tener un contexto adicional en sí mismo es una gran ventaja sobre el intérprete estándar de Python (que no proporciona ningún contexto adicional). Puedes controlar la cantidad de contexto que se muestra usando el `%xmode` comando mágico, desde `Plain` (igual que el intérprete estándar de Python) a `Verbose` (que incluye valores de argumentos de función y más). Como verá más adelante en el capítulo, puede *ingresar a la pila* (usando las magias `%debugo %pdb`) después de que se haya producido un error para interactivo depuración post mortem.

3.3 Archivos y el sistema operativo

La mayor parte de este libro utiliza herramientas de alto nivel como `pandas.read_csv` para leer archivos de datos del disco en estructuras de datos de Python. Sin embargo, es importante comprender los conceptos básicos sobre cómo trabajar con archivos en Python. Afortunadamente, es muy simple, que es una de las razones por las que Python es tan popular para el texto y la mezcla de archivos.

Para abrir un archivo para leer o escribir, utilizar el incorporado `open` función, ya sea con una ruta de acceso relativa o absoluta:

```
In [207]: path = 'examples/segismundo.txt'

In [208]: f = open(path)
```

Por defecto, el archivo se abre en modo de solo lectura `'r'`. Luego podemos tratar el identificador de archivo `f` como una lista e iterar sobre las líneas de esta manera:

```
for line in f:
    pass
```

Las líneas salen del archivo con el marcadores de fin de línea (EOL) intactos, por lo que a menudo verá código para obtener una lista de líneas sin EOL en un archivo como:

```
In [209]: lines = [x.rstrip() for x in open(path)]

In [210]: lines
Out[210]:
['Sueña el rico en su riqueza,',
 'que más cuidados le ofrece;',
 '',
 'sueña el pobre que padece',
 'su miseria y su pobreza;',
 '',
 'sueña el que a medrar empieza,',
 'sueña el que afana y pretende,',
 'sueña el que agravia y ofende,',
 '',
```



```
'y en el mundo, en conclusión,',  
'todos sueñan lo que son',  
'aunque ninguno lo entiende.',  
'']
```

Cuando usa `open` para crear objetos de archivo, es importante cerrar explícitamente el archivo cuando haya terminado con él. Cerrar el archivo libera sus recursos de vuelta al sistema operativo:

```
In [211]: f.close()
```

Una de las formas de facilitar la limpieza de archivos abiertos es usar la `with` declaración:

```
In [212]: with open(path) as f:  
.....:     lines = [x.rstrip() for x in f]
```

Esto cerrará automáticamente el archivo `f` al salir del `with` bloque.

Si hubiéramos escrito `f = open(path, 'w')`, se habría creado un *nuevo* archivo en `examples / segismundo.txt` (¡cuidado!), Sobrescribiendo a cualquiera en su lugar. También está el `'x'` modo de archivo, que crea un archivo grabable pero falla si la ruta del archivo ya existe. Consulte la [Tabla 3-3](#) para obtener una lista de todos los modos válidos de lectura / escritura de archivos.

Para archivos legibles, algunos de los más utilizados métodos son `read`, `seek`, y `tell`. `read` devuelve una cierta cantidad de caracteres del archivo. Lo que constituye un "carácter" está determinado por la codificación del archivo (por ejemplo, UTF-8) o simplemente bytes sin formato si el archivo se abre en modo binario:

```
In [213]: f = open(path)  
  
In [214]: f.read(10)  
Out[214]: 'Sueña el r'  
  
In [215]: f2 = open(path, 'rb') # Binary mode  
  
In [216]: f2.read(10)  
Out[216]: b'Sue\xc3\xb1a el '
```

El `read` método avanza la posición del identificador de archivo por el número de bytes leídos. `tell` te da la posición actual:

```
In [217]: f.tell()  
Out[217]: 11  
  
In [218]: f2.tell()  
Out[218]: 10
```

Aunque leemos 10 caracteres del archivo, la posición es 11 porque se necesitaron tantos bytes para decodificar 10 caracteres usando la codificación predeterminada. Puede verificar la codificación predeterminada en el `sys` módulo:




```
In [219]: import sys

In [220]: sys.getdefaultencoding()
Out[220]: 'utf-8'
```

`seek` cambia la posición del archivo al byte indicado en el archivo:

```
In [221]: f.seek(3)
Out[221]: 3

In [222]: f.read(1)
Out[222]: 'ñ'
```

Por último, recordamos cerrar los archivos:

```
In [223]: f.close()

In [224]: f2.close()
```

Tabla 3-3. Modos de archivo de Python

Modo	Descripción
r	Solo lectura modo
w	Modo de solo escritura; crea un nuevo archivo (borrando los datos de cualquier archivo con el mismo nombre)
x	Modo de solo escritura; crea un nuevo archivo, pero falla si la ruta del archivo ya existe
a	Anexar a archivo existente (cree el archivo si aún no existe)
r+	Lea y escribir
b	Agregar al modo para archivos binarios (es decir, 'rb' o 'wb')
t	Modo de texto para archivos (decodificación automática de bytes a Unicode). Este es el valor predeterminado si no se especifica. Agregar <code>ta</code> otros modos para usar esto (es decir, 'rt' o 'xt')

Para escribir texto en un archivo, puede usar el archivos `w` o `w+` `writelines` métodos. Por ejemplo, podríamos crear una versión de `prof_mod.py` sin líneas en blanco como esta:

```
In [225]: with open('tmp.txt', 'w') as handle:
.....:     handle.writelines(x for x in open(path) if len(x) > 1)
```



```
In [226]: with open('tmp.txt') as f:
.....:     lines = f.readlines()

In [227]: lines
Out[227]:
['Sueña el rico en su riqueza,\n',
'que más cuidados le ofrece;\n',
'sueña el pobre que padece\n',
'su miseria y su pobreza;\n',
'sueña el que a medrar empieza,\n',
'sueña el que afana y pretende,\n',
'sueña el que agravia y ofende,\n',
'y en el mundo, en conclusión,\n',
'todos sueñan lo que son,\n',
'aunque ninguno lo entiende.\n']
```

Consulte la [Tabla 3-4](#) para ver muchos de los más comunes.métodos de archivo utilizados.

Tabla 3-4. Métodos o atributos importantes del archivo Python

Método	Descripción
<code>read([size])</code>	Devolver datos del archivo como una cadena, con un <code>size</code> argumento opcional que indica el número de bytes a leer
<code>readlines([size])</code>	Lista de retorno de líneas en el archivo, con <code>size</code> argumento opcional
<code>write(str)</code>	Escribir pasado cadena para archivar
<code>writelines(strings)</code>	Escribir pasado secuencia de cadenas al archivo
<code>close()</code>	Cierra el encargarse de
<code>flush()</code>	Enjuague el búfer interno de E / S al disco
<code>seek(pos)</code>	Mover a posición de archivo indicada (entero)
<code>tell()</code>	Regreso posición actual del archivo como entero
<code>closed</code>	True Si el el archivo está cerrado

Bytes y Unicode con archivos

El comportamiento predeterminado para Los archivos Python (ya sean legibles o editables) están en *modo texto* , lo que significa que tiene la intención de trabajar



con cadenas Python (es decir, Unicode). Esto contrasta con *modo binario*, que puede obtener agregando el modo de archivo. Veamos el archivo (que contiene caracteres no ASCII con codificación UTF-8) de la sección anterior:

```
In [230]: with open(path) as f:
.....:     chars = f.read(10)

In [231]: chars
Out[231]: 'Sueña el r'
```

UTF-8 es una codificación Unicode de longitud variable, por lo que cuando solicite una cierta cantidad de caracteres del archivo, Python lee suficientes bytes (que podrían ser tan pocos como 10 o hasta 40 bytes) del archivo para decodificar esos caracteres. Si abro el archivo en 'rb' modo, `read` solicita números exactos de bytes:

```
In [232]: with open(path, 'rb') as f:
.....:     data = f.read(10)

In [233]: data
Out[233]: b'Sue\xc3\xb1a el '
```

Dependiendo de la codificación del texto, puede decodificar los bytes a un objeto usted mismo, pero solo si cada uno de los caracteres Unicode codificados está completamente formado:

```
In [234]: data.decode('utf8')
Out[234]: 'Sueña el '

In [235]: data[:4].decode('utf8')
-----
UnicodeDecodeError                                Traceback (most recent call)
<ipython-input-235-0ad9ad6a11bd> in <module>()
----> 1 data[:4].decode('utf8')
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xc3 in position 3:
d end of data
```

El modo de texto, combinado con la `encoding` opción de `open`, proporciona una forma conveniente de convertir de una codificación Unicode a otra:

```
In [236]: sink_path = 'sink.txt'

In [237]: with open(path) as source:
.....:     with open(sink_path, 'xt', encoding='iso-8859-1') as si:
.....:         sink.write(source.read())

In [238]: with open(sink_path, encoding='iso-8859-1') as f:
.....:     print(f.read(10))
Sueña el r
```

Cuidado con usar `seek` cuando abra archivos en cualquier modo que no sea binario. Si la posición del archivo se encuentra en el medio de los bytes que definen un carácter Unicode, las lecturas posteriores generarán un error:



```

In [240]: f = open(path)

In [241]: f.read(5)
Out[241]: 'Sueña'

In [242]: f.seek(4)
Out[242]: 4

In [243]: f.read(1)
-----
UnicodeDecodeError                                Traceback (most recent call:
<ipython-input-243-5a354f952aa4> in <module>()
----> 1 f.read(1)
/miniconda/envs/book-env/lib/python3.6/codecs.py in decode(self, input,
    319         # decode input (taking the buffer into account)
    320         data = self.buffer + input
--> 321         (result, consumed) = self._buffer_decode(data, self.e
    )
    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:
tart byte

In [244]: f.close()

```

Si te encuentras haciendo análisis de datos regularmente en datos de texto que no son ASCII, dominar la funcionalidad Unicode de Python será valioso. Vea [la documentación en línea de Python](#) para mucho más.

3.4 Conclusión

Con algunos de los conceptos básicos y el entorno y el lenguaje de Python ahora en nuestro haber, es hora de seguir adelante y aprender sobre NumPy y la computación orientada a matrices en Python.

[Configuraciones](#) / [Apoyo](#) / [Desconectar](#)

◀ ANTERIOR
Python Language Basics, IPython y Jup...

PRÓXIMO ▶
[Conceptos básicos de NumPy: matrices ...](#)

