Python lists have a built-in `sort()`method that modifies the list in-place and a `sorted()` built-in function that builds a new sorted list from an iterable.

There are many ways to use them to sort data and there doesn't appear to be a single, central place in the various manuals describing them, so I'll do so here.

Sorting Basics

A simple ascending sort is very easy -- just call the `sorted()` function. It returns a new sorted list:

```
>>> sorted([5, 2, 3, 1, 4])
[1, 2, 3, 4, 5]
```

You can also use the `list.sort()`method of a list. It modifies the list in-place (and returns None to avoid confusion). Usually it's less convenient than `sorted()` - but if you don't need the original list, it's slightly more efficient.

```
>>> a = [5, 2, 3, 1, 4]
>>> a.sort()
>>> a
[1, 2, 3, 4, 5]
```

Another difference is that the`list.sort()` method is only defined for lists. In contrast, the `sorted()` function accepts any iterable.

```
>>> sorted({1: 'D', 2: 'B', 3: 'B', 4: 'E', 5: 'A'})
[1, 2, 3, 4, 5]
```

Key Functions

Starting with Python 2.4, both`list.sort()` and `sorted()` added a `key`parameter to specify a function to be called on each list element prior to making comparisons.

For example, here's a case-insensitive string comparison:

```
>>> sorted("This is a test string from Andrew".split(), key=str.lower)
['a', 'Andrew', 'from', 'is', 'string', 'test', 'This']
```

The value of the `key` parameter should be a function that takes a single argument and returns a key to use for sorting purposes. This technique is fast because the key function is called exactly once for each input record.

A common pattern is to sort complex objects using some of the object's indices as a key. For example:

```
>>> student_tuples = [
        ('john', 'A', 15),
        ('jane', 'B', 12),
```

```
        ('dave', 'B', 10),
]
>>> sorted(student_tuples, key=lambda student: student[2])   # sort by age
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

The same technique works for objects with named attributes. For example:

```
>>> class Student:
        def __init__(self, name, grade, age):
                self.name = name
                self.grade = grade
                self.age = age
        def __repr__(self):
                return repr((self.name, self.grade, self.age))
        def weighted_grade(self):
                return 'CBA'.index(self.grade) / float(self.age)

>>> student_objects = [
        Student('john', 'A', 15),
        Student('jane', 'B', 12),
        Student('dave', 'B', 10),
]
>>> sorted(student_objects, key=lambda student: student.age)   # sort by age
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

Operator Module Functions

The key-function patterns shown above are very common, so Python provides convenience functions to make accessor functions easier and faster. The operator module has `itemgetter`, `attrgetter`, and starting in Python 2.6 a `methodcaller` function.

Using those functions, the above examples become simpler and faster.

```
>>> from operator import itemgetter, attrgetter, methodcaller

>>> sorted(student_tuples, key=itemgetter(2))
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]

>>> sorted(student_objects, key=attrgetter('age'))
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

The operator module functions allow multiple levels of sorting. For example, to sort by grade then by age:

```
>>> sorted(student_tuples, key=itemgetter(1,2))
[('john', 'A', 15), ('dave', 'B', 10), ('jane', 'B', 12)]
```

```
>>> sorted(student_objects, key=attrgetter('grade', 'age'))
[('john', 'A', 15), ('dave', 'B', 10), ('jane', 'B', 12)]
```

The third function from the operator module, `methodcaller` is used in the following example in which the weighted grade of each student is shown before sorting on it:

```
>>> [(student.name, student.weighted_grade()) for student in student_objects]
[('john', 0.13333333333333333), ('jane', 0.08333333333333333), ('dave', 0.1)]
>>> sorted(student_objects, key=methodcaller('weighted_grade'))
[('jane', 'B', 12), ('dave', 'B', 10), ('john', 'A', 15)]
```

Ascending and Descending

Both `list.sort()` and `sorted()` accept a`reverse` parameter with a boolean value. This is using to flag descending sorts. For example, to get the student data in reverse age order:

```
>>> sorted(student_tuples, key=itemgetter(2), reverse=True)
[('john', 'A', 15), ('jane', 'B', 12), ('dave', 'B', 10)]

>>> sorted(student_objects, key=attrgetter('age'), reverse=True)
[('john', 'A', 15), ('jane', 'B', 12), ('dave', 'B', 10)]
```

Sort Stability and Complex Sorts

Starting with Python 2.2, sorts are guaranteed to be stable. That means that when multiple records have the same key, their original order is preserved.

```
>>> data = [('red', 1), ('blue', 1), ('red', 2), ('blue', 2)]
>>> sorted(data, key=itemgetter(0))
[('blue', 1), ('blue', 2), ('red', 1), ('red', 2)]
```

Notice how the two records for `'blue'`retain their original order so that`('blue', 1)` is guaranteed to precede`('blue', 2)`.

This wonderful property lets you build complex sorts in a series of sorting steps. For example, to sort the student data by descending grade and then ascending age, do the age sort first and then sort again using grade:

```
>>> s = sorted(student_objects, key=attrgetter('age'))     # sort on secondary
key
>>> sorted(s, key=attrgetter('grade'), reverse=True)       # now sort on primary
key, descending
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```