Readers of the Fourth Edition of *Learning Python* might recognize some aspects of the running example used in this chapter—the characters here are similar in spirit to those in the OOP tutorial chapter in that book, and the later class-based examples here are essentially a variation on a theme. Despite some redundancy, I'm revisiting the example here for three reasons: it serves its purpose as a review of language fundamentals; some readers of this book haven't read *Learning Python*; and the example receives expanded treatment here, with the addition of GUI and Web interfaces. That is, this chapter picks up where *Learning Python* left off, pushing this core language example into the realm of realistic applications—which, in a nutshell, reflects the purpose of this book.

The Task

Imagine, if you will, that you need to keep track of information about people for some reason. Maybe you want to store an address book on your computer, or perhaps you need to keep track of employees in a small business. For whatever reason, you want to write a program that keeps track of details about these people. In other words, you want to keep records in a database—to permanently store lists of people's attributes on your computer.

Naturally, there are off-the-shelf programs for managing databases like these. By writing a program for this task yourself, however, you'll have complete control over its operation. You can add code for special cases and behaviors that precoded software may not have anticipated. You won't have to install and learn to use yet another database product. And you won't be at the mercy of a software vendor to fix bugs or add new features. You decide to write a Python program to manage your people.

Step 1: Representing Records

If we're going to store records in a database, the first step is probably deciding what those records will look like. There are a variety of ways to represent information about people in the Python language. Built-in object types such as lists and dictionaries are often sufficient, especially if we don't initially care about processing the data we store.

Using Lists

Lists, for example, can collect attributes about people in a positionally ordered way. Start up your Python interactive interpreter and type the following two statements:

```
>>> bob = ['Bob Smith', 42, 30000, 'software']
>>> sue = ['Sue Jones', 45, 40000, 'hardware']
```

We've just made two records, albeit simple ones, to represent two people, Bob and Sue (my apologies if you really are Bob or Sue, generically or otherwise*). Each record is a list of four properties: name, age, pay, and job fields. To access these fields, we simply index by position; the result is in parentheses here because it is a tuple of two results:

```
>>> bob[0], sue[2]             # fetch name, pay
('Bob Smith', 40000)
```

Processing records is easy with this representation; we just use list operations. For example, we can extract a last name by splitting the name field on blanks and grabbing the last part, and we can give someone a raise by changing their list in-place:

```
>>> bob[0].split()[-1]         # what's bob's last name?
'Smith'
>>> sue[2] *= 1.25             # give sue a 25% raise
>>> sue
['Sue Jones', 45, 50000.0, 'hardware']
```

The last-name expression here proceeds from left to right: we fetch Bob's name, split it into a list of substrings around spaces, and index his last name (run it one step at a time to see how).

Start-up pointers

Since this is the first code in this book, here are some quick pragmatic pointers for reference:

- This code may be typed in the IDLE GUI; after typing **python** at a shell prompt (or the full directory path to it if it's not on your system path); and so on.
- The >>> characters are Python's prompt (not code you type yourself).
- The informational lines that Python prints when this prompt starts up are usually omitted in this book to save space.
- I'm running all of this book's code under Python 3.1; results in any 3.X release should be similar (barring unforeseeable Python changes, of course).
- Apart from some system and C integration code, most of this book's examples are run under Windows 7, though thanks to Python portability, it generally doesn't matter unless stated otherwise.

If you've never run Python code this way before, see an introductory resource such as O'Reilly's *Learning Python* for help with getting started. I'll also have a few words to say about running code saved in script files later in this chapter.

* No, I'm serious. In the Python classes I teach, I had for many years regularly used the name "Bob Smith," age 40.5, and jobs "developer" and "manager" as a supposedly fictitious database record—until a class in Chicago, where I met a student named Bob Smith, who was 40.5 and was a developer and manager. The world is stranger than it seems.

A database list

Of course, what we've really coded so far is just two variables, not a database; to collect Bob and Sue into a unit, we might simply stuff them into another list:

```
>>> people = [bob, sue]                         # reference in list of lists
>>> for person in people:
        print(person)

['Bob Smith', 42, 30000, 'software']
['Sue Jones', 45, 50000.0, 'hardware']
```

Now the people list represents our database. We can fetch specific records by their relative positions and process them one at a time, in loops:

```
>>> people[1][0]
'Sue Jones'

>>> for person in people:
        print(person[0].split()[-1])            # print last names
        person[2] *= 1.20                       # give each a 20% raise

Smith
Jones

>>> for person in people: print(person[2])     # check new pay

36000.0
60000.0
```

Now that we have a list, we can also collect values from records using some of Python's more powerful iteration tools, such as list comprehensions, maps, and generator expressions:

```
>>> pays = [person[2] for person in people]    # collect all pay
>>> pays
[36000.0, 60000.0]

>>> pays = map((lambda x: x[2]), people)       # ditto (map is a generator in 3.X)
>>> list(pays)
[36000.0, 60000.0]

>>> sum(person[2] for person in people)       # generator expression, sum built-in
96000.0
```

To add a record to the database, the usual list operations, such as `append` and `extend`, will suffice:

```
>>> people.append(['Tom', 50, 0, None])
>>> len(people)
3
>>> people[-1][0]
'Tom'
```

Lists work for our people database, and they might be sufficient for some programs, but they suffer from a few major flaws. For one thing, Bob and Sue, at this point, are just fleeting objects in memory that will disappear once we exit Python. For another, every time we want to extract a last name or give a raise, we'll have to repeat the kinds of code we just typed; that could become a problem if we ever change the way those operations work—we may have to update many places in our code. We'll address these issues in a few moments.

Field labels

Perhaps more fundamentally, accessing fields by position in a list requires us to memorize what each position means: if you see a bit of code indexing a record on magic position 2, how can you tell it is extracting a pay? In terms of understanding the code, it might be better to associate a field name with a field value.

We might try to associate names with relative positions by using the Python `range` built-in function, which generates successive integers when used in iteration contexts (such as the sequence assignment used initially here):

```
>>> NAME, AGE, PAY = range(3)                  # 0, 1, and 2
>>> bob = ['Bob Smith', 42, 10000]
>>> bob[NAME]
'Bob Smith'
>>> PAY, bob[PAY]
(2, 10000)
```

This addresses readability: the three uppercase variables essentially become field names. This makes our code dependent on the field position assignments, though—we have to remember to update the range assignments whenever we change record structure. Because they are not directly associated, the names and records may become out of sync over time and require a maintenance step.

Moreover, because the field names are independent variables, there is no direct mapping from a record list back to its field's names. A raw record list, for instance, provides no way to label its values with field names in a formatted display. In the preceding record, without additional code, there is no path from value 42 to label AGE: `bob.index(42)` gives 1, the value of AGE, but not the name AGE itself.

We might also try this by using lists of tuples, where the tuples record both a field name and a value; better yet, a list of lists would allow for updates (tuples are immutable). Here's what that idea translates to, with slightly simpler records:

```
>>> bob = [['name', 'Bob Smith'], ['age', 42], ['pay', 10000]]
>>> sue = [['name', 'Sue Jones'], ['age', 45], ['pay', 20000]]
>>> people = [bob, sue]
```

This really doesn't fix the problem, though, because we still have to index by position in order to fetch fields:

```
>>> for person in people:
        print(person[0][1], person[2][1])     # name, pay
```

```
Bob Smith 10000
Sue Jones 20000

>>> [person[0][1] for person in people]       # collect names
['Bob Smith', 'Sue Jones']

>>> for person in people:
        print(person[0][1].split()[-1])       # get last names
        person[2][1] *= 1.10                  # give a 10% raise

Smith
Jones
>>> for person in people: print(person[2])

['pay', 11000.0]
['pay', 22000.0]
```

All we've really done here is add an extra level of positional indexing. To do better, we might inspect field names in loops to find the one we want (the loop uses tuple assignment here to unpack the name/value pairs):

```
>>> for person in people:
        for (name, value) in person:
            if name == 'name': print(value)   # find a specific field

Bob Smith
Sue Jones
```

Better yet, we can code a fetcher function to do the job for us:

```
>>> def field(record, label):
        for (fname, fvalue) in record:
            if fname == label:                # find any field by name
                return fvalue

>>> field(bob, 'name')
'Bob Smith'
>>> field(sue, 'pay')
22000.0

>>> for rec in people:
        print(field(rec, 'age'))              # print all ages

42
45
```

If we proceed down this path, we'll eventually wind up with a set of record interface functions that generically map field names to field data. If you've done any Python coding in the past, though, you probably already know that there is an easier way to code this sort of association, and you can probably guess where we're headed in the next section.

Using Dictionaries

The list-based record representations in the prior section work, though not without some cost in terms of performance required to search for field names (assuming you need to care about milliseconds and such). But if you already know some Python, you also know that there are more efficient and convenient ways to associate property names and values. The built-in dictionary object is a natural:

```
>>> bob = {'name': 'Bob Smith', 'age': 42, 'pay': 30000, 'job': 'dev'}
>>> sue = {'name': 'Sue Jones', 'age': 45, 'pay': 40000, 'job': 'hdw'}
```

Now, Bob and Sue are objects that map field names to values automatically, and they make our code more understandable and meaningful. We don't have to remember what a numeric offset means, and we let Python search for the value associated with a field's name with its efficient dictionary indexing:

```
>>> bob['name'], sue['pay']            # not bob[0], sue[2]
('Bob Smith', 40000)

>>> bob['name'].split()[-1]
'Smith'

>>> sue['pay'] *= 1.10
>>> sue['pay']
44000.0
```

Because fields are accessed mnemonically now, they are more meaningful to those who read your code (including you).

Other ways to make dictionaries

Dictionaries turn out to be so useful in Python programming that there are even more convenient ways to code them than the traditional literal syntax shown earlier—e.g., with keyword arguments and the type constructor, as long as the keys are all strings:

```
>>> bob = dict(name='Bob Smith', age=42, pay=30000, job='dev')
>>> sue = dict(name='Sue Jones', age=45, pay=40000, job='hdw')
>>> bob
{'pay': 30000, 'job': 'dev', 'age': 42, 'name': 'Bob Smith'}
>>> sue
{'pay': 40000, 'job': 'hdw', 'age': 45, 'name': 'Sue Jones'}
```

by filling out a dictionary one field at a time (recall that dictionary keys are pseudo-randomly ordered):

```
>>> sue = {}
>>> sue['name'] = 'Sue Jones'
>>> sue['age']  = 45
>>> sue['pay']  = 40000
>>> sue['job']  = 'hdw'
>>> sue
{'job': 'hdw', 'pay': 40000, 'age': 45, 'name': 'Sue Jones'}
```

and by zipping together name/value lists:

```
>>> names  = ['name', 'age', 'pay', 'job']
>>> values = ['Sue Jones', 45, 40000, 'hdw']
>>> list(zip(names, values))
[('name', 'Sue Jones'), ('age', 45), ('pay', 40000), ('job', 'hdw')]
>>> sue = dict(zip(names, values))
>>> sue
{'job': 'hdw', 'pay': 40000, 'age': 45, 'name': 'Sue Jones'}
```

We can even make dictionaries from a sequence of key values and an optional starting value for all the keys (handy to initialize an empty dictionary):

```
>>> fields = ('name', 'age', 'job', 'pay')
>>> record = dict.fromkeys(fields, '?')
>>> record
{'job': '?', 'pay': '?', 'age': '?', 'name': '?'}
```

Lists of dictionaries

Regardless of how we code them, we still need to collect our dictionary-based records into a database; a list does the trick again, as long as we don't require access by key at the top level:

```
>>> bob
{'pay': 30000, 'job': 'dev', 'age': 42, 'name': 'Bob Smith'}
>>> sue
{'job': 'hdw', 'pay': 40000, 'age': 45, 'name': 'Sue Jones'}

>>> people = [bob, sue]                                    # reference in a list
>>> for person in people:
        print(person['name'], person['pay'], sep=', ')    # all name, pay

Bob Smith, 30000
Sue Jones, 40000

>>> for person in people:
        if person['name'] == 'Sue Jones':                 # fetch sue's pay
            print(person['pay'])

40000
```

Iteration tools work just as well here, but we use keys rather than obscure positions (in database terms, the list comprehension and map in the following code project the database on the "name" field column):

```
>>> names = [person['name'] for person in people]         # collect names
>>> names
['Bob Smith', 'Sue Jones']

>>> list(map((lambda x: x['name']), people))              # ditto, generate
['Bob Smith', 'Sue Jones']

>>> sum(person['pay'] for person in people)               # sum all pay
70000
```

Interestingly, tools such as list comprehensions and on-demand generator expressions can even approach the utility of SQL queries here, albeit operating on in-memory objects:

```
>>> [rec['name'] for rec in people if rec['age'] >= 45]   # SQL-ish query
['Sue Jones']

>>> [(rec['age'] ** 2 if rec['age'] >= 45 else rec['age']) for rec in people]
[42, 2025]

>>> G = (rec['name'] for rec in people if rec['age'] >= 45)
>>> next(G)
'Sue Jones'

>>> G = ((rec['age'] ** 2 if rec['age'] >= 45 else rec['age']) for rec in people)
>>> G.__next__()
42
```

And because dictionaries are normal Python objects, these records can also be accessed and updated with normal Python syntax:

```
>>> for person in people:
        print(person['name'].split()[-1])             # last name
        person['pay'] *= 1.10                         # a 10% raise

Smith
Jones

>>> for person in people: print(person['pay'])

33000.0
44000.0
```

Nested structures

Incidentally, we could avoid the last-name extraction code in the prior examples by further structuring our records. Because all of Python's compound datatypes can be nested inside each other and as deeply as we like, we can build up fairly complex information structures easily—simply type the object's syntax, and Python does all the work of building the components, linking memory structures, and later reclaiming their space. This is one of the great advantages of a scripting language such as Python.

The following, for instance, represents a more structured record by nesting a dictionary, list, and tuple inside another dictionary:

```
>>> bob2 = {'name': {'first': 'Bob', 'last': 'Smith'},
            'age':  42,
            'job':  ['software', 'writing'],
            'pay':  (40000, 50000)}
```

Because this record contains nested structures, we simply index twice to go two levels deep:

```
>>> bob2['name']                        # bob's full name
{'last': 'Smith', 'first': 'Bob'}
>>> bob2['name']['last']                # bob's last name
'Smith'
>>> bob2['pay'][1]                      # bob's upper pay
50000
```

The name field is another dictionary here, so instead of splitting up a string, we simply index to fetch the last name. Moreover, people can have many jobs, as well as minimum and maximum pay limits. In fact, Python becomes a sort of query language in such cases—we can fetch or change nested data with the usual object operations:

```
>>> for job in bob2['job']: print(job)      # all of bob's jobs
software
writing

>> bob2['job'][-1]                      # bob's last job
'writing'
>>> bob2['job'].append('janitor')       # bob gets a new job
>>> bob2
{'job': ['software', 'writing', 'janitor'], 'pay': (40000, 50000), 'age': 42, 'name':
{'last': 'Smith', 'first': 'Bob'}}
```

It's OK to grow the nested list with `append`, because it is really an independent object. Such nesting can come in handy for more sophisticated applications; to keep ours simple, we'll stick to the original flat record structure.

Dictionaries of dictionaries

One last twist on our people database: we can get a little more mileage out of dictionaries here by using one to represent the database itself. That is, we can use a dictionary of dictionaries—the outer dictionary is the database, and the nested dictionaries are the records within it. Rather than a simple list of records, a dictionary-based database allows us to store and retrieve records by symbolic key:

```
>>> bob = dict(name='Bob Smith', age=42, pay=30000, job='dev')
>>> sue = dict(name='Sue Jones', age=45, pay=40000, job='hdw')
>>> bob
{'pay': 30000, 'job': 'dev', 'age': 42, 'name': 'Bob Smith'}

>>> db = {}
>>> db['bob'] = bob                   # reference in a dict of dicts
>>> db['sue'] = sue
>>>
>>> db['bob']['name']                 # fetch bob's name
'Bob Smith'
>>> db['sue']['pay'] = 50000          # change sue's pay
>>> db['sue']['pay']                  # fetch sue's pay
50000
```

Notice how this structure allows us to access a record directly instead of searching for it in a loop—we get to Bob's name immediately by indexing on key `bob`. This really is a dictionary of dictionaries, though you won't see all the gory details unless you display the database all at once (the Python `pprint` pretty-printer module can help with legibility here):

```
>>> db
{'bob': {'pay': 30000, 'job': 'dev', 'age': 42, 'name': 'Bob Smith'}, 'sue':
{'pay': 50000, 'job': 'hdw', 'age': 45, 'name': 'Sue Jones'}}

>>> import pprint
>>> pprint.pprint(db)
{'bob': {'age': 42, 'job': 'dev', 'name': 'Bob Smith', 'pay': 30000},
 'sue': {'age': 45, 'job': 'hdw', 'name': 'Sue Jones', 'pay': 50000}}
```

If we still need to step through the database one record at a time, we can now rely on dictionary iterators. In recent Python releases, a dictionary iterator produces one key in a `for` loop each time through (for compatibility with earlier releases, we can also call the `db.keys` method explicitly in the `for` loop rather than saying just `db`, but since Python 3's `keys` result is a generator, the effect is roughly the same):

```
>>> for key in db:
        print(key, '=>', db[key]['name'])

bob => Bob Smith
sue => Sue Jones

>>> for key in db:
        print(key, '=>', db[key]['pay'])

bob => 30000
sue => 50000
```

To visit all records, either index by key as you go:

```
>>> for key in db:
        print(db[key]['name'].split()[-1])
        db[key]['pay'] *= 1.10

Smith
Jones
```

or step through the dictionary's values to access records directly:

```
>>> for record in db.values(): print(record['pay'])

33000.0
55000.0

>>> x = [db[key]['name'] for key in db]
>>> x
['Bob Smith', 'Sue Jones']

>>> x = [rec['name'] for rec in db.values()]
```

```
>>> x
['Bob Smith', 'Sue Jones']
```

And to add a new record, simply assign it to a new key; this is just a dictionary, after all:

```
>>> db['tom'] = dict(name='Tom', age=50, job=None, pay=0)
>>>
>>> db['tom']
{'pay': 0, 'job': None, 'age': 50, 'name': 'Tom'}
>>> db['tom']['name']
'Tom'
>>> list(db.keys())
['bob', 'sue', 'tom']
>>> len(db)
3
>>> [rec['age'] for rec in db.values()]
[42, 45, 50]
>>> [rec['name'] for rec in db.values() if rec['age'] >= 45]     # SQL-ish query
['Sue Jones', 'Tom']
```

Although our database is still a transient object in memory, it turns out that this dictionary-of-dictionaries format corresponds exactly to a system that saves objects permanently—the shelve (yes, this should probably be *shelf*, grammatically speaking, but the Python module name and term is *shelve*). To learn how, let's move on to the next section.

Step 2: Storing Records Persistently

So far, we've settled on a dictionary-based representation for our database of records, and we've reviewed some Python data structure concepts along the way. As mentioned, though, the objects we've seen so far are temporary—they live in memory and they go away as soon as we exit Python or the Python program that created them. To make our people persistent, they need to be stored in a file of some sort.

Using Formatted Files

One way to keep our data around between program runs is to write all the data out to a simple text file, in a formatted way. Provided the saving and loading tools agree on the format selected, we're free to use any custom scheme we like.

Test data script

So that we don't have to keep working interactively, let's first write a script that initializes the data we are going to store (if you've done any Python work in the past, you know that the interactive prompt tends to become tedious once you leave the realm of simple one-liners). Example 1-1 creates the sort of records and database dictionary we've been working with so far, but because it is a module, we can import it repeatedly without having to retype the code each time. In a sense, this module is a database itself, but its program code format doesn't support automatic or end-user updates as is.

Example 1-1. PP4E\Preview\initdata.py

```
# initialize data to be stored in files, pickles, shelves

# records
bob = {'name': 'Bob Smith', 'age': 42, 'pay': 30000, 'job': 'dev'}
sue = {'name': 'Sue Jones', 'age': 45, 'pay': 40000, 'job': 'hdw'}
tom = {'name': 'Tom',       'age': 50, 'pay': 0,     'job': None}

# database
db = {}
db['bob'] = bob
db['sue'] = sue
db['tom'] = tom

if __name__ == '__main__':       # when run as a script
    for key in db:
        print(key, '=>\n  ', db[key])
```

As usual, the `__name__` test at the bottom of Example 1-1 is true only when this file is run, not when it is imported. When run as a top-level script (e.g., from a command line, via an icon click, or within the IDLE GUI), the file's self-test code under this test dumps the database's contents to the standard output stream (remember, that's what `print` function-call statements do by default).

Here is the script in action being run from a system command line on Windows. Type the following command in a Command Prompt window after a `cd` to the directory where the file is stored, and use a similar console window on other types of computers:

```
...\PP4E\Preview> python initdata.py
bob =>
   {'job': 'dev', 'pay': 30000, 'age': 42, 'name': 'Bob Smith'}
sue =>
   {'job': 'hdw', 'pay': 40000, 'age': 45, 'name': 'Sue Jones'}
tom =>
   {'job': None, 'pay': 0, 'age': 50, 'name': 'Tom'}
```

File name conventions

Since this is our first source file (a.k.a. "script"), here are three usage notes for this book's examples:

- The text `...\PP4E\Preview>` in the first line of the preceding example listing stands for your operating system's prompt, which can vary per platform; you type just the text that follows this prompt (`python initdata.py`).
- Like all examples in this book, the system prompt also gives the directory in the downloadable book examples package where this command should be run. When running this script using a command-line in a system shell, make sure the shell's current working directory is *PP4E\Preview*. This can matter for examples that use files in the working directory.

- Similarly, the label that precedes every example file's code listing tells you where the source file resides in the examples package. Per the Example 1-1 listing label shown earlier, this script's full filename is *PP4E\Preview\initdata.py* in the examples tree.

We'll use these conventions throughout the book; see the Preface for more on getting the examples if you wish to work along. I occasionally give more of the directory path in system prompts when it's useful to provide the extra execution context, especially in the system part of the book (e.g., a "C:\" prefix from Windows or more directory names).

Script start-up pointers

I gave pointers for using the interactive prompt earlier. Now that we've started running script files, here are also a few quick startup pointers for using Python scripts in general:

- On some platforms, you may need to type the full directory path to the Python program on your machine; if Python isn't on your system path setting on Windows, for example, replace `python` in the command with `C:\Python31\python` (this assumes you're using Python 3.1).
- On most Windows systems you also don't need to type `python` on the command line at all; just type the file's name to run it, since Python is registered to open ".py" script files.
- You can also run this file inside Python's standard IDLE GUI (open the file and use the Run menu in the text edit window), and in similar ways from any of the available third-party Python IDEs (e.g., Komodo, Eclipse, NetBeans, and the Wing IDE).
- If you click the program's file icon to launch it on Windows, be sure to add an `input()` call to the bottom of the script to keep the output window up. On other systems, icon clicks may require a `#!` line at the top and executable permission via a `chmod` command.

I'll assume here that you're able to run Python code one way or another. Again, if you're stuck, see other books such as *Learning Python* for the full story on launching Python programs.

Data format script

Now, all we have to do is store all of this in-memory data in a file. There are a variety of ways to accomplish this; one of the most basic is to write one piece of data at a time, with separators between each that we can use when reloading to break the data apart. Example 1-2 shows one way to code this idea.

Example 1-2. PP4E\Preview\make_db_file.py

```
"""
Save in-memory database object to a file with custom formatting;
assume 'endrec.', 'enddb.', and '=>' are not used in the data;
assume db is dict of dict;  warning: eval can be dangerous - it
runs strings as code;  could also eval() record dict all at once;
could also dbfile.write(key + '\n') vs print(key, file=dbfile);
"""

dbfilename = 'people-file'
ENDDB  = 'enddb.'
ENDREC = 'endrec.'
RECSEP = '=>'

def storeDbase(db, dbfilename=dbfilename):
    "formatted dump of database to flat file"
    dbfile = open(dbfilename, 'w')
    for key in db:
        print(key, file=dbfile)
        for (name, value) in db[key].items():
            print(name + RECSEP + repr(value), file=dbfile)
        print(ENDREC, file=dbfile)
    print(ENDDB, file=dbfile)
    dbfile.close()

def loadDbase(dbfilename=dbfilename):
    "parse data to reconstruct database"
    dbfile = open(dbfilename)
    import sys
    sys.stdin = dbfile
    db = {}
    key = input()
    while key != ENDDB:
        rec = {}
        field = input()
        while field != ENDREC:
            name, value = field.split(RECSEP)
            rec[name] = eval(value)
            field = input()
        db[key] = rec
        key = input()
    return db

if __name__ == '__main__':
    from initdata import db
    storeDbase(db)
```

This is a somewhat complex program, partly because it has both saving and loading logic and partly because it does its job the hard way; as we'll see in a moment, there are better ways to get objects into files than by manually formatting and parsing them. For simple tasks, though, this does work; running Example 1-2 as a script writes the database out to a flat file. It has no printed output, but we can inspect the database file interactively after this script is run, either within IDLE or from a console window where you're running these examples (as is, the database file shows up in the current working directory):

```
...\PP4E\Preview> python make_db_file.py
...\PP4E\Preview> python
>>> for line in open('people-file'):
...     print(line, end='')
...
bob
job=>'dev'
pay=>30000
age=>42
name=>'Bob Smith'
endrec.
sue
job=>'hdw'
pay=>40000
age=>45
name=>'Sue Jones'
endrec.
tom
job=>None
pay=>0
age=>50
name=>'Tom'
endrec.
enddb.
```

This file is simply our database's content with added formatting. Its data originates from the test data initialization module we wrote in Example 1-1 because that is the module from which Example 1-2's self-test code imports its data. In practice, Example 1-2 itself could be imported and used to store a variety of databases and files.

Notice how data to be written is formatted with the as-code `repr` call and is re-created with the `eval` call, which treats strings as Python code. That allows us to store and re-create things like the `None` object, but it is potentially unsafe; you shouldn't use `eval` if you can't be sure that the database won't contain malicious code. For our purposes, however, there's probably no cause for alarm.

Utility scripts

To test further, Example 1-3 reloads the database from a file each time it is run.

Example 1-3. PP4E\Preview\dump_db_file.py

```
from make_db_file import loadDbase
db = loadDbase()
for key in db:
    print(key, '=>\n  ', db[key])
print(db['sue']['name'])
```

And Example 1-4 makes changes by loading, updating, and storing again.

Example 1-4. PP4E\Preview\update_db_file.py

```
from make_db_file import loadDbase, storeDbase
db = loadDbase()
db['sue']['pay'] *= 1.10
db['tom']['name'] = 'Tom Tom'
storeDbase(db)
```

Here are the dump script and the update script in action at a system command line; both Sue's pay and Tom's name change between script runs. The main point to notice is that the data stays around after each script exits—our objects have become persistent simply because they are mapped to and from text files:

```
...\PP4E\Preview> python dump_db_file.py
bob =>
   {'pay': 30000, 'job': 'dev', 'age': 42, 'name': 'Bob Smith'}
sue =>
   {'pay': 40000, 'job': 'hdw', 'age': 45, 'name': 'Sue Jones'}
tom =>
   {'pay': 0, 'job': None, 'age': 50, 'name': 'Tom'}
Sue Jones

...\PP4E\Preview> python update_db_file.py
...\PP4E\Preview> python dump_db_file.py
bob =>
   {'pay': 30000, 'job': 'dev', 'age': 42, 'name': 'Bob Smith'}
sue =>
   {'pay': 44000.0, 'job': 'hdw', 'age': 45, 'name': 'Sue Jones'}
tom =>
   {'pay': 0, 'job': None, 'age': 50, 'name': 'Tom Tom'}
Sue Jones
```

As is, we'll have to write Python code in scripts or at the interactive command line for each specific database update we need to perform (later in this chapter, we'll do better by providing generalized console, GUI, and web-based interfaces instead). But at a basic level, our text file is a database of records. As we'll learn in the next section, though, it turns out that we've just done a lot of pointless work.

Using Pickle Files

The formatted text file scheme of the prior section works, but it has some major limitations. For one thing, it has to read the entire database from the file just to fetch one record, and it must write the entire database back to the file after each set of updates. Although storing one record's text per file would work around this limitation, it would also complicate the program further.

For another thing, the text file approach assumes that the data separators it writes out to the file will not appear in the data to be stored: if the characters `=>` happen to appear in the data, for example, the scheme will fail. We might work around this by generating XML text to represent records in the text file, using Python's XML parsing tools, which we'll meet later in this text, to reload; XML tags would avoid collisions with actual data's text, but creating and parsing XML would complicate the program substantially too.

Perhaps worst of all, the formatted text file scheme is already complex without being general: it is tied to the dictionary-of-dictionaries structure, and it can't handle anything else without being greatly expanded. It would be nice if a general tool existed that could translate any sort of Python data to a format that could be saved in a file in a single step.

That is exactly what the Python `pickle` module is designed to do. The `pickle` module translates an in-memory Python object into a *serialized* byte stream—a string of bytes that can be written to any file-like object. The `pickle` module also knows how to reconstruct the original object in memory, given the serialized byte stream: we get back the exact same object. In a sense, the `pickle` module replaces proprietary data formats—its serialized format is general and efficient enough for any program. With `pickle`, there is no need to manually translate objects to data when storing them persistently, and no need to manually parse a complex format to get them back. Pickling is similar in spirit to XML representations, but it's both more Python-specific, and much simpler to code.

The net effect is that pickling allows us to store and fetch native Python objects as they are and in a single step—we use normal Python syntax to process pickled records. Despite what it does, the `pickle` module is remarkably easy to use. Example 1-5 shows how to store our records in a flat file, using `pickle`.

Example 1-5. PP4E\Preview\make_db_pickle.py

```
from initdata import db
import pickle
dbfile = open('people-pickle', 'wb')                # use binary mode files in 3.X
pickle.dump(db, dbfile)                             # data is bytes, not str
dbfile.close()
```

When run, this script stores the entire database (the dictionary of dictionaries defined in Example 1-1) to a flat file named *people-pickle* in the current working directory. The `pickle` module handles the work of converting the object to a string. Example 1-6 shows how to access the pickled database after it has been created; we simply open the file and pass its content back to `pickle` to remake the object from its serialized string.

Example 1-6. PP4E\Preview\dump_db_pickle.py

```
import pickle
dbfile = open('people-pickle', 'rb')                # use binary mode files in 3.X
db = pickle.load(dbfile)
for key in db:
    print(key, '=>\n  ', db[key])
print(db['sue']['name'])
```

Here are these two scripts at work, at the system command line again; naturally, they can also be run in IDLE, and you can open and inspect the pickle file by running the same sort of code interactively as well:

```
...\PP4E\Preview> python make_db_pickle.py
...\PP4E\Preview> python dump_db_pickle.py
bob =>
   {'pay': 30000, 'job': 'dev', 'age': 42, 'name': 'Bob Smith'}
sue =>
   {'pay': 40000, 'job': 'hdw', 'age': 45, 'name': 'Sue Jones'}
tom =>
   {'pay': 0, 'job': None, 'age': 50, 'name': 'Tom'}
Sue Jones
```

Updating with a pickle file is similar to a manually formatted file, except that Python is doing all of the formatting work for us. Example 1-7 shows how.

Example 1-7. PP4E\Preview\update-db-pickle.py

```
import pickle
dbfile = open('people-pickle', 'rb')
db = pickle.load(dbfile)
dbfile.close()

db['sue']['pay'] *= 1.10
db['tom']['name'] = 'Tom Tom'

dbfile = open('people-pickle', 'wb')
pickle.dump(db, dbfile)
dbfile.close()
```

Notice how the entire database is written back to the file after the records are changed in memory, just as for the manually formatted approach; this might become slow for very large databases, but we'll ignore this for the moment. Here are our update and dump scripts in action—as in the prior section, Sue's pay and Tom's name change between scripts because they are written back to a file (this time, a pickle file):

```
...\PP4E\Preview> python update_db_pickle.py
...\PP4E\Preview> python dump_db_pickle.py
bob =>
   {'pay': 30000, 'job': 'dev', 'age': 42, 'name': 'Bob Smith'}
sue =>
   {'pay': 44000.0, 'job': 'hdw', 'age': 45, 'name': 'Sue Jones'}
tom =>
   {'pay': 0, 'job': None, 'age': 50, 'name': 'Tom Tom'}
Sue Jones
```

As we'll learn in Chapter 17, the Python pickling system supports nearly arbitrary object types—lists, dictionaries, class instances, nested structures, and more. There, we'll also learn about the pickler's text and binary storage protocols; as of Python 3, all protocols use `bytes` objects to represent pickled data, which in turn requires pickle files to be opened in binary mode for all protocols. As we'll see later in this chapter, the pickler and its data format also underlie shelves and ZODB databases, and pickled class instances provide both data and behavior for objects stored.

In fact, pickling is more general than these examples may imply. Because they accept any object that provides an interface compatible with files, pickling and unpickling may be used to transfer native Python objects to a variety of media. Using a network socket, for instance, allows us to ship pickled Python objects across a network and provides an alternative to larger protocols such as SOAP and XML-RPC.

Using Per-Record Pickle Files

As mentioned earlier, one potential disadvantage of this section's examples so far is that they may become slow for very large databases: because the entire database must be loaded and rewritten to update a single record, this approach can waste time. We could improve on this by storing each record in the database in a separate flat file. The next three examples show one way to do so; Example 1-8 stores each record in its own flat file, using each record's original key as its filename with a *.pkl* appended (it creates the files *bob.pkl*, *sue.pkl*, and *tom.pkl* in the current working directory).

Example 1-8. PP4E\Preview\make_db_pickle_recs.py

```
from initdata import bob, sue, tom
import pickle
for (key, record) in [('bob', bob), ('tom', tom), ('sue', sue)]:
    recfile = open(key + '.pkl', 'wb')
    pickle.dump(record, recfile)
    recfile.close()
```

Next, Example 1-9 dumps the entire database by using the standard library's `glob` module to do filename expansion and thus collect all the files in this directory with a *.pkl* extension. To load a single record, we open its file and deserialize with `pickle`; we must load only one record file, though, not the entire database, to fetch one record.

Example 1-9. PP4E\Preview\dump_db_pickle_recs.py

```
import pickle, glob
for filename in glob.glob('*.pkl'):         # for 'bob','sue','tom'
    recfile = open(filename, 'rb')
    record  = pickle.load(recfile)
    print(filename, '=>\n  ', record)

suefile = open('sue.pkl', 'rb')
print(pickle.load(suefile)['name'])         # fetch sue's name
```

Finally, Example 1-10 updates the database by fetching a record from its file, changing it in memory, and then writing it back to its pickle file. This time, we have to fetch and rewrite only a single record file, not the full database, to update.

Example 1-10. PP4E\Preview\update_db_pickle_recs.py

```
import pickle
suefile = open('sue.pkl', 'rb')
sue = pickle.load(suefile)
suefile.close()
```

```
sue['pay'] *= 1.10
suefile = open('sue.pkl', 'wb')
pickle.dump(sue, suefile)
suefile.close()
```

Here are our file-per-record scripts in action; the results are about the same as in the prior section, but database keys become real filenames now. In a sense, the filesystem becomes our top-level dictionary—filenames provide direct access to each record.

```
...\PP4E\Preview> python make_db_pickle_recs.py
...\PP4E\Preview> python dump_db_pickle_recs.py
bob.pkl =>
   {'pay': 30000, 'job': 'dev', 'age': 42, 'name': 'Bob Smith'}
sue.pkl =>
   {'pay': 40000, 'job': 'hdw', 'age': 45, 'name': 'Sue Jones'}
tom.pkl =>
   {'pay': 0, 'job': None, 'age': 50, 'name': 'Tom'}
Sue Jones

...\PP4E\Preview> python update_db_pickle_recs.py
...\PP4E\Preview> python dump_db_pickle_recs.py
bob.pkl =>
   {'pay': 30000, 'job': 'dev', 'age': 42, 'name': 'Bob Smith'}
sue.pkl =>
   {'pay': 44000.0, 'job': 'hdw', 'age': 45, 'name': 'Sue Jones'}
tom.pkl =>
   {'pay': 0, 'job': None, 'age': 50, 'name': 'Tom'}
Sue Jones
```

Using Shelves

Pickling objects to files, as shown in the preceding section, is an optimal scheme in many applications. In fact, some applications use pickling of Python objects across network sockets as a simpler alternative to network protocols such as the SOAP and XML-RPC web services architectures (also supported by Python, but much heavier than `pickle`).

Moreover, assuming your filesystem can handle as many files as you'll need, pickling one record per file also obviates the need to load and store the entire database for each update. If we really want keyed access to records, though, the Python standard library offers an even higher-level tool: shelves.

Shelves automatically pickle objects to and from a keyed-access filesystem. They behave much like dictionaries that must be opened, and they persist after each program exits. Because they give us key-based access to stored records, there is no need to manually manage one flat file per record—the shelve system automatically splits up stored records and fetches and updates only those records that are accessed and changed. In this way, shelves provide utility similar to per-record pickle files, but they are usually easier to code.

The shelve interface is just as simple as pickle: it is identical to dictionaries, with extra open and close calls. In fact, to your code, a shelve really does appear to be a persistent dictionary of persistent objects; Python does all the work of mapping its content to and from a file. For instance, Example 1-11 shows how to store our in-memory dictionary objects in a shelve for permanent keeping.

Example 1-11. PP4E\Preview\make_db_shelve.py

```
from initdata import bob, sue
import shelve
db = shelve.open('people-shelve')
db['bob'] = bob
db['sue'] = sue
db.close()
```

This script creates one or more files in the current directory with the name *people-shelve* as a prefix (in Python 3.1 on Windows, *people-shelve.bak*, *people-shelve.dat*, and *people-shelve.dir*). You shouldn't delete these files (they are your database!), and you should be sure to use the same base name in other scripts that access the shelve. Example 1-12, for instance, reopens the shelve and indexes it by key to fetch its stored records.

Example 1-12. PP4E\Preview\dump_db_shelve.py

```
import shelve
db = shelve.open('people-shelve')
for key in db:
    print(key, '=>\n  ', db[key])
print(db['sue']['name'])
db.close()
```

We still have a dictionary of dictionaries here, but the top-level dictionary is really a shelve mapped onto a file. Much happens when you access a shelve's keys—it uses pickle internally to serialize and deserialize objects stored, and it interfaces with a keyed-access filesystem. From your perspective, though, it's just a persistent dictionary. Example 1-13 shows how to code shelve updates.

Example 1-13. PP4E\Preview\update_db_shelve.py

```
from initdata import tom
import shelve
db = shelve.open('people-shelve')
sue = db['sue']                       # fetch sue
sue['pay'] *= 1.50
db['sue'] = sue                       # update sue
db['tom'] = tom                       # add a new record
db.close()
```

Notice how this code fetches sue by key, updates in memory, and then reassigns to the key to update the shelve; this is a requirement of shelves by default, but not always of more advanced shelve-like systems such as ZODB, covered in Chapter 17. As we'll see later, `shelve.open` also has a newer `writeback` keyword argument, which, if passed `True`, causes all records loaded from the shelve to be cached in memory, and automatically written back to the shelve when it is closed; this avoids manual write backs on changes, but can consume memory and make closing slow.

Also note how shelve files are explicitly closed. Although we don't need to pass mode flags to `shelve.open` (by default it creates the shelve if needed, and opens it for reads and writes otherwise), some underlying keyed-access filesystems may require a `close` call in order to flush output buffers after changes.

Finally, here are the shelve-based scripts on the job, creating, changing, and fetching records. The records are still dictionaries, but the database is now a dictionary-like shelve which automatically retains its state in a file between program runs:

```
...\PP4E\Preview> python make_db_shelve.py
...\PP4E\Preview> python dump_db_shelve.py
bob =>
   {'pay': 30000, 'job': 'dev', 'age': 42, 'name': 'Bob Smith'}
sue =>
   {'pay': 40000, 'job': 'hdw', 'age': 45, 'name': 'Sue Jones'}
Sue Jones

...\PP4E\Preview> python update_db_shelve.py
...\PP4E\Preview> python dump_db_shelve.py
bob =>
   {'pay': 30000, 'job': 'dev', 'age': 42, 'name': 'Bob Smith'}
sue =>
   {'pay': 60000.0, 'job': 'hdw', 'age': 45, 'name': 'Sue Jones'}
tom =>
   {'pay': 0, 'job': None, 'age': 50, 'name': 'Tom'}
Sue Jones
```

When we ran the update and dump scripts here, we added a new record for key `tom` and increased Sue's pay field by 50 percent. These changes are permanent because the record dictionaries are mapped to an external file by shelve. (In fact, this is a particularly good script for Sue—something she might consider scheduling to run often, using a cron job on Unix, or a Startup folder or msconfig entry on Windows...)

What's in a Name?

Though it's a surprisingly well-kept secret, Python gets its name from the 1970s British TV comedy series *Monty Python's Flying Circus*. According to Python folklore, Guido van Rossum, Python's creator, was watching reruns of the show at about the same time he needed a name for a new language he was developing. And as they say in show business, "the rest is history."

Because of this heritage, references to the comedy group's work often show up in examples and discussion. For instance, the name *Brian* appears often in scripts; the words *spam, lumberjack*, and *shrubbery* have a special connotation to Python users; and presentations are sometimes referred to as *The Spanish Inquisition*. As a rule, if a Python user starts using phrases that have no relation to reality, they're probably borrowed from the Monty Python series or movies. Some of these phrases might even pop up in this book. You don't have to run out and rent *The Meaning of Life* or *The Holy Grail* to do useful work in Python, of course, but it can't hurt.

While "Python" turned out to be a distinctive name, it has also had some interesting side effects. For instance, when the Python newsgroup, comp.lang.python, came online in 1994, its first few weeks of activity were almost entirely taken up by people wanting to discuss topics from the TV show. More recently, a special Python supplement in the *Linux Journal* magazine featured photos of Guido garbed in an obligatory "nice red uniform."

Python's news list still receives an occasional post from fans of the show. For instance, one early poster innocently offered to swap Monty Python scripts with other fans. Had he known the nature of the forum, he might have at least mentioned whether they were portable or not.

Step 3: Stepping Up to OOP

Let's step back for a moment and consider how far we've come. At this point, we've created a database of records: the shelve, as well as per-record pickle file approaches of the prior section suffice for basic data storage tasks. As is, our records are represented as simple dictionaries, which provide easier-to-understand access to fields than do lists (by key, rather than by position). Dictionaries, however, still have some limitations that may become more critical as our program grows over time.

For one thing, there is no central place for us to collect record processing logic. Extracting last names and giving raises, for instance, can be accomplished with code like the following:

```
>>> import shelve
>>> db = shelve.open('people-shelve')
>>> bob = db['bob']
>>> bob['name'].split()[-1]             # get bob's last name
'Smith'
>>> sue = db['sue']
>>> sue['pay'] *= 1.25                  # give sue a raise
>>> sue['pay']
75000.0
>>> db['sue'] = sue
>>> db.close()
```

This works, and it might suffice for some short programs. But if we ever need to change the way last names and raises are implemented, we might have to update this kind of code in many places in our program. In fact, even finding all such magical code snippets could be a challenge; hardcoding or cutting and pasting bits of logic redundantly like this in more than one place will almost always come back to haunt you eventually.

It would be better to somehow hide—that is, *encapsulate*—such bits of code. Functions in a module would allow us to implement such operations in a single place and thus avoid code redundancy, but still wouldn't naturally associate them with the records themselves. What we'd like is a way to bind processing logic with the data stored in the database in order to make it easier to understand, debug, and reuse.

Another downside to using dictionaries for records is that they are difficult to expand over time. For example, suppose that the set of data fields or the procedure for giving raises is different for different kinds of people (perhaps some people get a bonus each year and some do not). If we ever need to extend our program, there is no natural way to customize simple dictionaries. For future growth, we'd also like our software to support extension and customization in a natural way.

If you've already studied Python in any sort of depth, you probably already know that this is where its OOP support begins to become attractive:

Structure
: With OOP, we can naturally associate processing logic with record data—classes provide both a program unit that combines logic and data in a single package and a hierarchy that allows code to be easily factored to avoid redundancy.

Encapsulation
: With OOP, we can also wrap up details such as name processing and pay increases behind method functions—i.e., we are free to change method implementations without breaking their users.

Customization
: And with OOP, we have a natural growth path. Classes can be extended and customized by coding new subclasses, without changing or breaking already working code.

That is, under OOP, we program by customizing and reusing, not by rewriting. OOP is an option in Python and, frankly, is sometimes better suited for strategic than for tactical tasks. It tends to work best when you have time for upfront planning—something that might be a luxury if your users have already begun storming the gates.

But especially for larger systems that change over time, its code reuse and structuring advantages far outweigh its learning curve, and it can substantially cut development time. Even in our simple case, the customizability and reduced redundancy we gain from classes can be a decided advantage.

Using Classes

OOP is easy to use in Python, thanks largely to Python's dynamic typing model. In fact, it's so easy that we'll jump right into an example: Example 1-14 implements our database records as class instances rather than as dictionaries.

Example 1-14. PP4E\Preview\person_start.py

```
class Person:
    def __init__(self, name, age, pay=0, job=None):
```

```
        self.name = name
        self.age  = age
        self.pay  = pay
        self.job  = job

if __name__ == '__main__':
    bob = Person('Bob Smith', 42, 30000, 'software')
    sue = Person('Sue Jones', 45, 40000, 'hardware')
    print(bob.name, sue.pay)

    print(bob.name.split()[-1])
    sue.pay *= 1.10
    print(sue.pay)
```

There is not much to this class—just a constructor method that fills out the instance with data passed in as arguments to the class name. It's sufficient to represent a database record, though, and it can already provide tools such as defaults for pay and job fields that dictionaries cannot. The self-test code at the bottom of this file creates two instances (records) and accesses their attributes (fields); here is this file's output when run under IDLE (a system command-line works just as well):

```
Bob Smith 40000
Smith
44000.0
```

This isn't a database yet, but we could stuff these objects into a list or dictionary as before in order to collect them as a unit:

```
>>> from person_start import Person
>>> bob = Person('Bob Smith', 42)
>>> sue = Person('Sue Jones', 45, 40000)

>>> people = [bob, sue]                          # a "database" list
>>> for person in people:
        print(person.name, person.pay)

Bob Smith 0
Sue Jones 40000

>>> x = [(person.name, person.pay) for person in people]
>>> x
[('Bob Smith', 0), ('Sue Jones', 40000)]

>>> [rec.name for rec in people if rec.age >= 45]     # SQL-ish query
['Sue Jones']

>>> [(rec.age ** 2 if rec.age >= 45 else rec.age) for rec in people]
[42, 2025]
```

Notice that Bob's pay defaulted to zero this time because we didn't pass in a value for that argument (maybe Sue is supporting him now?). We might also implement a class that represents the database, perhaps as a subclass of the built-in list or dictionary types, with insert and delete methods that encapsulate the way the database is implemented. We'll abandon this path for now, though, because it will be more useful to store these records persistently in a shelve, which already encapsulates stores and fetches behind an interface for us. Before we do, though, let's add some logic.

Adding Behavior

So far, our class is just data: it replaces dictionary keys with object attributes, but it doesn't add much to what we had before. To really leverage the power of classes, we need to add some behavior. By wrapping up bits of behavior in class method functions, we can insulate clients from changes. And by packaging methods in classes along with data, we provide a natural place for readers to look for code. In a sense, classes combine records and the programs that process those records; methods provide logic that interprets and updates the data (we say they are *object-oriented*, because they always process an object's data).

For instance, Example 1-15 adds the last-name and raise logic as class methods; methods use the `self` argument to access or update the instance (record) being processed.

Example 1-15. PP4E\Preview\person.py

```
class Person:
    def __init__(self, name, age, pay=0, job=None):
        self.name = name
        self.age  = age
        self.pay  = pay
        self.job  = job
    def lastName(self):
        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay *= (1.0 + percent)

if __name__ == '__main__':
    bob = Person('Bob Smith', 42, 30000, 'software')
    sue = Person('Sue Jones', 45, 40000, 'hardware')
    print(bob.name, sue.pay)

    print(bob.lastName())
    sue.giveRaise(.10)
    print(sue.pay)
```

The output of this script is the same as the last, but the results are being computed by methods now, not by hardcoded logic that appears redundantly wherever it is required:

```
Bob Smith 40000
Smith
44000.0
```

Adding Inheritance

One last enhancement to our records before they become permanent: because they are implemented as classes now, they naturally support customization through the inheritance search mechanism in Python. Example 1-16, for instance, customizes the last section's Person class in order to give a 10 percent bonus by default to managers whenever they receive a raise (any relation to practice in the real world is purely coincidental).

Example 1-16. PP4E\Preview\manager.py

```
from person import Person

class Manager(Person):
    def giveRaise(self, percent, bonus=0.1):
        self.pay *= (1.0 + percent + bonus)

if __name__ == '__main__':
    tom = Manager(name='Tom Doe', age=50, pay=50000)
    print(tom.lastName())
    tom.giveRaise(.20)
    print(tom.pay)
```

When run, this script's self-test prints the following:

```
Doe
65000.0
```

Here, the Manager class appears in a module of its own, but it could have been added to the person module instead (Python doesn't require just one class per file). It inherits the constructor and last-name methods from its superclass, but it customizes just the giveRaise method (there are a variety of ways to code this extension, as we'll see later). Because this change is being added as a new subclass, the original Person class, and any objects generated from it, will continue working unchanged. Bob and Sue, for example, inherit the original raise logic, but Tom gets the custom version because of the class from which he is created. In OOP, we program by *customizing*, not by changing.

In fact, code that uses our objects doesn't need to be at all aware of what the raise method does—it's up to the object to do the right thing based on the class from which it is created. As long as the object supports the expected interface (here, a method called giveRaise), it will be compatible with the calling code, regardless of its specific type, and even if its method works differently than others.

If you've already studied Python, you may know this behavior as *polymorphism*; it's a core property of the language, and it accounts for much of your code's flexibility. When the following code calls the giveRaise method, for example, what happens depends on the obj object being processed; Tom gets a 20 percent raise instead of 10 percent because of the Manager class's customization:

```
>>> from person import Person
>>> from manager import Manager

>>> bob = Person(name='Bob Smith', age=42, pay=10000)
>>> sue = Person(name='Sue Jones', age=45, pay=20000)
>>> tom = Manager(name='Tom Doe',  age=55, pay=30000)
>>> db = [bob, sue, tom]
```

```
>>> for obj in db:
        obj.giveRaise(.10)         # default or custom

>>> for obj in db:
        print(obj.lastName(), '=>', obj.pay)

Smith => 11000.0
Jones => 22000.0
Doe => 36000.0
```

Refactoring Code

Before we move on, there are a few coding alternatives worth noting here. Most of these underscore the Python OOP model, and they serve as a quick review.

Augmenting methods

As a first alternative, notice that we have introduced some redundancy in Example 1-16: the raise calculation is now repeated in two places (in the two classes). We could also have implemented the customized `Manager` class by *augmenting* the inherited raise method instead of replacing it completely:

```
class Manager(Person):
    def giveRaise(self, percent, bonus=0.1):
        Person.giveRaise(self, percent + bonus)
```

The trick here is to call back the superclass's version of the method directly, passing in the `self` argument explicitly. We still redefine the method, but we simply run the general version after adding 10 percent (by default) to the passed-in percentage. This coding pattern can help reduce code redundancy (the original raise method's logic appears in only one place and so is easier to change) and is especially handy for kicking off superclass constructor methods in practice.

If you've already studied Python OOP, you know that this coding scheme works because we can always call methods through either an instance or the class name. In general, the following are equivalent, and both forms may be used explicitly:

```
instance.method(arg1, arg2)
class.method(instance, arg1, arg2)
```

In fact, the first form is mapped to the second—when calling through the instance, Python determines the class by searching the inheritance tree for the method name and passes in the instance automatically. Either way, within `giveRaise`, `self` refers to the instance that is the subject of the call.

Display format

For more object-oriented fun, we could also add a few operator overloading methods to our people classes. For example, a `__str__` method, shown here, could return a string to give the display format for our objects when they are printed as a whole—much better than the default display we get for an instance:

```
class Person:
    def __str__(self):
        return '<%s => %s>' % (self.__class__.__name__, self.name)

tom = Manager('Tom Jones', 50)
print(tom)                              # prints: <Manager => Tom Jones>
```

Here `__class__` gives the lowest class from which `self` was made, even though `__str__` may be inherited. The net effect is that `__str__` allows us to print instances directly instead of having to print specific attributes. We could extend this `__str__` to loop through the instance's `__dict__` attribute dictionary to display all attributes generically; for this preview we'll leave this as a suggested exercise.

We might even code an `__add__` method to make `+` expressions automatically call the `giveRaise` method. Whether we should is another question; the fact that a `+` expression gives a person a raise might seem more magical to the next person reading our code than it should.

Constructor customization

Finally, notice that we didn't pass the `job` argument when making a manager in Example 1-16; if we had, it would look like this with keyword arguments:

```
tom = Manager(name='Tom Doe', age=50, pay=50000, job='manager')
```

The reason we didn't include a job in the example is that it's redundant with the class of the object: if someone is a manager, their class should imply their job title. Instead of leaving this field blank, though, it may make more sense to provide an explicit constructor for managers, which fills in this field automatically:

```
class Manager(Person):
    def __init__(self, name, age, pay):
        Person.__init__(self, name, age, pay, 'manager')
```

Now when a manager is created, its job is filled in automatically. The trick here is to call to the superclass's version of the method explicitly, just as we did for the `giveRaise` method earlier in this section; the only difference here is the unusual name for the constructor method.

Alternative classes

We won't use any of this section's three extensions in later examples, but to demonstrate how they work, Example 1-17 collects these ideas in an alternative implementation of our `Person` classes.

Example 1-17. PP4E\Preview\person_alternative.py

```
"""
Alternative implementation of person classes, with data, behavior,
and operator overloading (not used for objects stored persistently)
"""

class Person:
    """
    a general person: data+logic
    """
    def __init__(self, name, age, pay=0, job=None):
        self.name = name
        self.age  = age
        self.pay  = pay
        self.job  = job
    def lastName(self):
        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay *= (1.0 + percent)
    def __str__(self):
        return ('<%s => %s: %s, %s>' %
               (self.__class__.__name__, self.name, self.job, self.pay))

class Manager(Person):
    """
    a person with custom raise
    inherits general lastname, str
    """
    def __init__(self, name, age, pay):
        Person.__init__(self, name, age, pay, 'manager')
    def giveRaise(self, percent, bonus=0.1):
        Person.giveRaise(self, percent + bonus)

if __name__ == '__main__':
    bob = Person('Bob Smith', 44)
    sue = Person('Sue Jones', 47, 40000, 'hardware')
    tom = Manager(name='Tom Doe', age=50, pay=50000)
    print(sue, sue.pay, sue.lastName())
    for obj in (bob, sue, tom):
        obj.giveRaise(.10)                 # run this obj's giveRaise
        print(obj)                         # run common __str__ method
```

Notice the polymorphism in this module's self-test loop: all three objects share the constructor, last-name, and printing methods, but the raise method called is dependent upon the class from which an instance is created. When run, Example 1-17 prints the following to standard output—the manager's job is filled in at construction, we get the new custom display format for our objects, and the new version of the manager's raise method works as before:

```
<Person => Sue Jones: hardware, 40000> 40000 Jones
<Person => Bob Smith: None, 0.0>
<Person => Sue Jones: hardware, 44000.0>
<Manager => Tom Doe: manager, 60000.0>
```

Such *refactoring* (restructuring) of code is common as class hierarchies grow and evolve. In fact, as is, we still can't give someone a raise if his pay is zero (Bob is out of luck); we probably need a way to set pay, too, but we'll leave such extensions for the next release. The good news is that Python's flexibility and readability make refactoring easy—it's simple and quick to restructure your code. If you haven't used the language yet, you'll find that Python development is largely an exercise in rapid, incremental, and interactive programming, which is well suited to the shifting needs of real-world projects.

Adding Persistence

It's time for a status update. We now have encapsulated in the form of classes customizable implementations of our records and their processing logic. Making our class-based records persistent is a minor last step. We could store them in per-record pickle files again; a shelve-based storage medium will do just as well for our goals and is often easier to code. Example 1-18 shows how.

Example 1-18. PP4E\Preview\make_db_classes.py

```
import shelve
from person import Person
from manager import Manager

bob = Person('Bob Smith', 42, 30000, 'software')
sue = Person('Sue Jones', 45, 40000, 'hardware')
tom = Manager('Tom Doe',  50, 50000)

db = shelve.open('class-shelve')
db['bob'] = bob
db['sue'] = sue
db['tom'] = tom
db.close()
```

This file creates three class instances (two from the original class and one from its customization) and assigns them to keys in a newly created shelve file to store them permanently. In other words, it creates a shelve of class instances; to our code, the database looks just like a dictionary of class instances, but the top-level dictionary is mapped to a shelve file again. To check our work, Example 1-19 reads the shelve and prints fields of its records.

Example 1-19. PP4E\Preview\dump_db_classes.py

```
import shelve
db = shelve.open('class-shelve')
for key in db:
    print(key, '=>\n  ', db[key].name, db[key].pay)

bob = db['bob']
print(bob.lastName())
print(db['tom'].lastName())
```

Note that we don't need to reimport the Person class here in order to fetch its instances from the shelve or run their methods. When instances are shelved or pickled, the underlying pickling system records both instance attributes and enough information to locate their classes automatically when they are later fetched (the class's module simply has to be on the module search path when an instance is loaded). This is on purpose; because the class and its instances in the shelve are stored separately, you can change the class to modify the way stored instances are interpreted when loaded (more on this later in the book). Here is the shelve dump script's output just after creating the shelve with the maker script:

```
bob =>
   Bob Smith 30000
sue =>
   Sue Jones 40000
tom =>
   Tom Doe 50000
Smith
Doe
```

As shown in Example 1-20, database updates are as simple as before (compare this to Example 1-13), but dictionary keys become attributes of instance objects, and updates are implemented by class method calls instead of hardcoded logic. Notice how we still fetch, update, and reassign to keys to update the shelve.

Example 1-20. PP4E\Preview\update_db_classes.py

```
import shelve
db = shelve.open('class-shelve')

sue = db['sue']
sue.giveRaise(.25)
db['sue'] = sue

tom = db['tom']
tom.giveRaise(.20)
db['tom'] = tom
db.close()
```

And last but not least, here is the dump script again after running the update script; Tom and Sue have new pay values, because these objects are now persistent in the shelve. We could also open and inspect the shelve by typing code at Python's interactive command line; despite its longevity, the shelve is just a Python object containing Python objects.

```
bob =>
   Bob Smith 30000
sue =>
   Sue Jones 50000.0
tom =>
   Tom Doe 65000.0
Smith
Doe
```

Tom and Sue both get a raise this time around, because they are persistent objects in the shelve database. Although shelves can also store simpler object types such as lists and dictionaries, class instances allow us to combine both data and behavior for our stored items. In a sense, instance attributes and class methods take the place of records and processing programs in more traditional schemes.

Other Database Options

At this point, we have a full-fledged database system: our classes simultaneously implement record data and record processing, and they encapsulate the implementation of the behavior. And the Python `pickle` and `shelve` modules provide simple ways to store our database persistently between program executions. This is not a relational database (we store objects, not tables, and queries take the form of Python object processing code), but it is sufficient for many kinds of programs.

If we need more functionality, we could migrate this application to even more powerful tools. For example, should we ever need full-blown SQL query support, there are interfaces that allow Python scripts to communicate with relational databases such as MySQL, PostgreSQL, and Oracle in portable ways.

ORMs (object relational mappers) such as SQLObject and SqlAlchemy offer another approach which retains the Python class view, but translates it to and from relational database tables—in a sense providing the best of both worlds, with Python class syntax on top, and enterprise-level databases underneath.

Moreover, the open source ZODB system provides a more comprehensive object database for Python, with support for features missing in shelves, including concurrent updates, transaction commits and rollbacks, automatic updates on in-memory component changes, and more. We'll explore these more advanced third-party tools in Chapter 17. For now, let's move on to putting a good face on our system.

"Buses Considered Harmful"

Over the years, Python has been remarkably well supported by the volunteer efforts of both countless individuals and formal organizations. Today, the nonprofit Python Software Foundation (PSF) oversees Python conferences and other noncommercial activities. The PSF was preceded by the PSA, a group that was originally formed in response to an early thread on the Python newsgroup that posed the semiserious question: "What would happen if Guido was hit by a bus?"

These days, Python creator Guido van Rossum is still the ultimate arbiter of proposed Python changes. He was officially anointed the BDFL—Benevolent Dictator for Life—of Python at the first Python conference and still makes final yes and no decisions on language changes (and apart from 3.0's deliberate incompatibilities, has usually said no: a good thing in the programming languages domain, because Python tends to change slowly and in backward-compatible ways).

But Python's user base helps support the language, work on extensions, fix bugs, and so on. It is a true community project. In fact, Python development is now a completely open process—anyone can inspect the latest source code files or submit patches by visiting a website (see *http://www.python.org* for details).

As an open source package, Python development is really in the hands of a very large cast of developers working in concert around the world—so much so that if the BDFL ever does pass the torch, Python will almost certainly continue to enjoy the kind of support its users have come to expect. Though not without pitfalls of their own, open source projects by nature tend to reflect the needs of their user communities more than either individuals or shareholders.

Given Python's popularity, bus attacks seem less threatening now than they once did. Of course, I can't speak for Guido.

Step 4: Adding Console Interaction

So far, our database program consists of class instances stored in a shelve file, as coded in the preceding section. It's sufficient as a storage medium, but it requires us to run scripts from the command line or type code interactively in order to view or process its content. Improving on this is straightforward: simply code more general programs that interact with users, either from a console window or from a full-blown graphical interface.

A Console Shelve Interface

Let's start with something simple. The most basic kind of interface we can code would allow users to type keys and values in a console window in order to process the database (instead of writing Python program code). Example 1-21, for instance, implements a simple interactive loop that allows a user to query multiple record objects in the shelve by key.

Example 1-21. PP4E\Preview\peopleinteract_query.py

```
# interactive queries
import shelve
fieldnames = ('name', 'age', 'job', 'pay')
maxfield   = max(len(f) for f in fieldnames)
db = shelve.open('class-shelve')

while True:
    key = input('\nKey? => ')          # key or empty line, exc at eof
    if not key: break
    try:
        record = db[key]               # fetch by key, show in console
```

```
    except:
        print('No such key "%s"!' % key)
    else:
        for field in fieldnames:
            print(field.ljust(maxfield), '=>', getattr(record, field))
```

This script uses the `getattr` built-in function to fetch an object's attribute when given its name string, and the `ljust` left-justify method of strings to align outputs (`maxfield`, derived from a generator expression, is the length of the longest field name). When run, this script goes into a loop, inputting keys from the interactive user (technically, from the standard input stream, which is usually a console window) and displaying the fetched records field by field. An empty line ends the session. If our shelve of class instances is still in the state we left it near the end of the last section:

```
...\PP4E\Preview> dump_db_classes.py
bob =>
   Bob Smith 30000
sue =>
   Sue Jones 50000.0
tom =>
   Tom Doe 65000.0
Smith
Doe
```

We can then use our new script to query the object database interactively, by key:

```
...\PP4E\Preview> peopleinteract_query.py

Key? => sue
name => Sue Jones
age  => 45
job  => hardware
pay  => 50000.0

Key? => nobody
No such key "nobody"!

Key? =>
```

Example 1-22 goes further and allows interactive updates. For an input key, it inputs values for each field and either updates an existing record or creates a new object and stores it under the key.

Example 1-22. PP4E\Preview\peopleinteract_update.py

```
# interactive updates
import shelve
from person import Person
fieldnames = ('name', 'age', 'job', 'pay')

db = shelve.open('class-shelve')
while True:
    key = input('\nKey? => ')
    if not key: break
```

```
    if key in db:
        record = db[key]                      # update existing record
    else:                                     # or make/store new rec
        record = Person(name='?', age='?')    # eval: quote strings
    for field in fieldnames:
        currval = getattr(record, field)
        newtext = input('\t[%s]=%s\n\t\tnew?=>' % (field, currval))
        if newtext:
            setattr(record, field, eval(newtext))
    db[key] = record
db.close()
```

Notice the use of `eval` in this script to convert inputs (as usual, that allows any Python object type, but it means you must quote string inputs explicitly) and the use of `setattr` call to assign an attribute given its name string. When run, this script allows any number of records to be added and changed; to keep the current value of a record's field, press the Enter key when prompted for a new value:

```
Key? => tom
        [name]=Tom Doe
                new?=>
        [age]=50
                new?=>56
        [job]=None
                new?=>'mgr'
        [pay]=65000.0
                new?=>90000

Key? => nobody
        [name]=?
                new?=>'John Doh'
        [age]=?
                new?=>55
        [job]=None
                new?=>
        [pay]=0
                new?=>None

Key? =>
```

This script is still fairly simplistic (e.g., errors aren't handled), but using it is much easier than manually opening and modifying the shelve at the Python interactive prompt, especially for nonprogrammers. Run the query script to check your work after an update (we could combine query and update into a single script if this becomes too cumbersome, albeit at some cost in code and user-experience complexity):

```
Key? => tom
name => Tom Doe
age  => 56
job  => mgr
pay  => 90000

Key? => nobody
name => John Doh
```

```
age  => 55
job  => None
pay  => None

Key? =>
```

Step 5: Adding a GUI

The console-based interface approach of the preceding section works, and it may be sufficient for some users assuming that they are comfortable with typing commands in a console window. With just a little extra work, though, we can add a GUI that is more modern, easier to use, less error prone, and arguably sexier.

GUI Basics

As we'll see later in this book, a variety of GUI toolkits and builders are available for Python programmers: tkinter, wxPython, PyQt, PythonCard, Dabo, and more. Of these, tkinter ships with Python, and it is something of a de facto standard.

tkinter is a lightweight toolkit and so meshes well with a scripting language such as Python; it's easy to do basic things with tkinter, and it's straightforward to do more advanced things with extensions and OOP-based code. As an added bonus, tkinter GUIs are portable across Windows, Linux/Unix, and Macintosh; simply copy the source code to the machine on which you wish to use your GUI. tkinter doesn't come with all the bells and whistles of larger toolkits such as wxPython or PyQt, but that's a major factor behind its relative simplicity, and it makes it ideal for getting started in the GUI domain.

Because tkinter is designed for scripting, coding GUIs with it is straightforward. We'll study all of its concepts and tools later in this book. But as a first example, the first program in tkinter is just a few lines of code, as shown in Example 1-23.

Example 1-23. PP4E\Preview\tkinter001.py

```
from tkinter import *
Label(text='Spam').pack()
mainloop()
```

From the tkinter module (really, a module package in Python 3), we get screen device (a.k.a. "widget") construction calls such as `Label`; geometry manager methods such as `pack`; widget configuration presets such as the `TOP` and `RIGHT` attachment side hints we'll use later for `pack`; and the `mainloop` call, which starts event processing.

This isn't the most useful GUI ever coded, but it demonstrates tkinter basics and it builds the fully functional window shown in Figure 1-1 in just three simple lines of code. Its window is shown here, like all GUIs in this book, running on Windows 7; it works the same on other platforms (e.g., Mac OS X, Linux, and older versions of Windows), but renders in with native look and feel on each.