The notes are being complied. Final draft will be posted Shortly.

Programming For Artists Notes

Why Do People Use Python?

Because there are many programming languages available today, this is the usual first question of newcomers. Given that there are roughly 1 million Python users out there at the moment, there really is no way to answer this question with complete accuracy; the choice of development tools is sometimes based on unique constraints or personal preference.

But after teaching Python to roughly 260 groups and over 4,000 students during the last 16 years, I have seen some common themes emerge. The primary factors cited by Python users seem to be these:

Software quality

For many, Python's focus on readability, coherence, and software quality in general sets it apart from other tools in the scripting world. Python code is designed to be readable, and hence reusable and maintainable—much more so than traditional scripting languages. The uniformity of Python code makes it easy to understand, even if you did not write it. In addition, Python has deep support for more advanced software reuse mechanisms, such as object-oriented (OO) and function programming.

Developer productivity

Python boosts developer productivity many times beyond compiled or statically typed languages such as C, C++, and Java. Python code is typically one-third toone-fifth the size of equivalent C++ or Java code. That means there is less to type, less to debug, and less to maintain after the fact. Python programs also run immediately, without the lengthy compile and link steps required by some other tools, further boosting programmer speed.

Program portability

Most Python programs run unchanged on all major computer platforms. Porting

Python code between Linux and Windows, for example, is usually just a matter of

copying a script's code between machines. Moreover, Python offers multiple options for coding portable graphical user interfaces, database access programs, web based systems, and more. Even operating system interfaces, including program

launches and directory processing, are as portable in Python as they can possibly

be.

Support libraries

Python comes with a large collection of prebuilt and portable functionality, known

as the standard library. This library supports an array of application-level programming tasks, from text pattern matching to network scripting. In addition,

Python can be extended with both homegrown libraries and a vast collection of

third-party application support software. Python's third-party domain offers tools

for website construction, numeric programming, serial port access, game development, and much more (see ahead for a sampling). The NumPy extension, for

instance, has been described as a free and more powerful equivalent to the Matlab

numeric programming system.

Component integration

Python scripts can easily communicate with other parts of an application, using a

variety of integration mechanisms. Such integrations allow Python to be used as a

product customization and extension tool. Today, Python code can invoke C and

C++ libraries, can be called from C and C++ programs, can integrate with Java

and .NET components, can communicate over frameworks such as COM and Silverlight, can interface with devices over serial ports, and can interact over networks

with interfaces like SOAP, XML-RPC, and CORBA. It is not a standalone tool.

Enjoyment

Because of Python's ease of use and built-in toolset, it can make the act of programming more pleasure than chore. Although this may be an intangible benefit,

its effect on productivity is an important asset.

Of these factors, the first two (quality and productivity) are probably the most compelling benefits to most Python users, and merit a fuller description.

Who Uses Python Today?

At this writing, the best estimate anyone can seem to make of the size of the Python user base is that there are roughly 1 million Python users around the world today (plus or minus a few). This estimate is based on various statistics, like download rates, web statistics, and developer surveys. Because Python is open source, a more exact count is difficult—there are no license registrations to tally. Moreover, Python is automatically included with Linux distributions, Macintosh computers, and a wide range of products and hardware, further clouding the user-base picture.

In general, though, Python enjoys a large user base and a very active developer community. It is generally considered to be in the top 5 or top 10 most widely used programming languages in the world today (its exact ranking varies per source and date).

Because Python has been around for over two decades and has been widely used, it is also very stable and robust.

Besides being leveraged by individual users, Python is also being applied in real revenuegenerating products by real companies. For instance, among the generally known Python user base:

- Google makes extensive use of Python in its web search systems.
- The popular YouTube video sharing service is largely written in Python.
- The Dropbox storage service codes both its server and desktop client software primarily in Python.
- The Raspberry Pi single-board computer promotes Python as its educational language.
- EVE Online, a massively multiplayer online game (MMOG) by CCP Games, uses Python broadly.
- The widespread BitTorrent peer-to-peer file sharing system began its life as a Python program.
- Industrial Light & Magic, Pixar, and others use Python in the production of animated movies.

• ESRI uses Python as an end-user customization tool for its popular GIS mapping products.

• Google's App Engine web development framework uses Python as an application language.

• The IronPort email server product uses more than 1 million lines of Python code to do its job.

• Maya, a powerful integrated 3D modeling and animation system, provides a Python scripting API.

• The NSA uses Python for cryptography and intelligence analysis.

• iRobot uses Python to develop commercial and military robotic devices.
The Civilization IV game's customizable scripted events are written entirely in Python.

• The One Laptop Per Child (OLPC) project built its user interface and activity model in Python.

• Netflix and Yelp have both documented the role of Python in their software infrastructures.

• Intel, Cisco, Hewlett-Packard, Seagate, Qualcomm, and IBM use Python for hardware testing.

• JPMorgan Chase, UBS, Getco, and Citadel apply Python to financial market forecasting.

• NASA, Los Alamos, Fermilab, JPL, and others use Python for scientific programming tasks.

How Python Runs Programs

When you write a Python program, the Python interpreter reads your program and carries out the instructions it contains. In effect, the interpreter is a layer of software logic between your code and the computer hardware on your machine.

Depending on which flavor of Python you run, the interpreter itself may be implemented as a C program, a set of Java classes, or something else. Whatever form it takes, the Python code you write must always be run by this interpreter. And to enable that, you must install a Python interpreter on your computer.

Import and Reload Basics

In simple terms, every file of Python source code whose name ends in a .py extension is a module. No special code or syntax is required to make a file a module: any such file will do. Other files can access the items a module defines by importing that module —import operations essentially load another file and grant access to that file's contents. The contents of a module are made available to the outside world through its attributes.

This module-based services model turns out to be the core idea behind program architecture in Python. Larger programs usually take the form of multiple module files, which import tools from other module files. One of the modules is designated as the main or top-level file, or "script"—the file launched to start the entire program, which runs line by line as usual.

```
>>> import script1

>>> import sys

>>>import maya.cmds as cmd

>>>from imp import reload

>>>from PyQt5.QtWidgets import QApplication, QDialog, QMessegeBox
```

```
>>> import myfile                    # Run file; load module as a whole
```

```
>>> myfile.title                    # Use its attribute names: '.' to qualify
```

In general, the dot expression syntax object.attribute lets you fetch any attribute attached to any object, and is one of the most common operations in Python code. Here, we've used it to access the string variable title inside the module myfile—in other words, myfile.title. As you'll see in more detail later, from is just like an import, with an extra assignment to names in the importing component. Technically, from copies a module's attributes, such that they become simple variables in the recipient—thus, you can simply refer to the imported string this time as title (a variable) instead of myfile.title (an attribute reference).3 Whether you use import or from to invoke an import operation, the statements in the module file myfile.py are executed, and the importing component (here, the interactive prompt) gains access to names assigned at the top level of the file. There's only one such name in this simple example—the variable title, assigned to a string—but the concept will be more useful when you start defining objects such as functions and classes in your modules: such objects become reusable software components that can be accessed by name from one or more client modules.

A namespace is just a package of variables (i.e., names). It takes the form of an object with attributes in Python. Each module file is automatically a namespace— that is, a package of variables reflecting the assignments made at the top level of the file. Namespaces help avoid name collisions in Python programs: because each module file is a self-contained namespace, files must explicitly import other files in order to use their names.

Python programs can be decomposed into modules, statements, expressions, and objects, as follows:

1. Programs are composed of modules.
2. Modules contain statements.
3. Statements contain expressions.
4. Expressions create and process objects.

Built-in objects preview

Object type	Example literals/creation
Numbers	1234, 3.1415, 3+4j, 0b111, Decimal(), Fraction()
Strings	'spam', "Bob's", b'a\x01c', u'sp\xc4m'
Lists	[1, [2, 'three'], 4.5], list(range(10))
Dictionaries	{'food': 'spam', 'taste': 'yum'}, dict(hours=10)
Tuples	(1, 'spam', 4, 'U'), tuple('spam'), namedtuple
Files	open('eggs.txt'), open(r'C:\ham.bin', 'wb')
Sets	set('abc'), {'a', 'b', 'c'}
Other core types	Booleans, types, None
Program unit types	Functions, modules, classes
Implementation-related types	Compiled code, stack tracebacks

Numbers

Python's core objects set includes the usual suspects: integers that have no fractional part, floating-point numbers that do, and more exotic types—complex numbers with imaginary parts, decimals with fixed precision, rationals with numerator and denominator, and full-featured sets. Built-in numbers are enough to represent most numeric quantities—from your age to your bank balance—but more types are available as third-party add-ons.

```
>>> 123 + 222                          # Integer addition
345
>>> 1.5 * 4                            # Floating-point multiplication
6.0
>>> 2 ** 100                           # 2 to the power 100
1267650600228229401496703205376
>>> len(str(2 ** 1000000))             # How many digits in a really BIG number?
301030
```

Besides expressions, there are a handful of useful numeric modules that ship with Python—modules are just packages of additional tools that we import to use:

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.sqrt(85)
9.219544457292887
>>> import random
>>> random.random()
0.7082048489415967
>>> random.choice([1, 2, 3, 4])
1
```

Strings

Strings are used to record both textual information (your name, for instance) as well as arbitrary collections of bytes (such as an image file's contents). They are our first example of what in Python we call a sequence—a positionally ordered collection of other objects. Sequences maintain a left-to-right order among the items they contain: their items are stored and fetched by their relative positions. Strictly speaking, strings are sequences of one-character strings; other, more general sequence types include lists and tuples, covered later.

As sequences, strings support operations that assume a positional ordering among items. For example, if we have a four-character string coded inside quotes (usually of the single variety), we can verify its length with the built-in len function and fetch its components with indexing expressions:

```
>>> S = 'Spam'                  # Make a 4-character string, and assign it to a name
>>> len(S)                      # Length
4
>>> S[0]                        # The first item in S, indexing by zero-based position
'S'
>>> S[1]                        # The second item from the left
'p'
```

In Python, we can also index backward, from the end—positive indexes count from the left, and negative indexes count back from the right:

```
>>> S[-1]                    # The last item from the end in S
'm'
>>> S[-2]                    # The second-to-last item from the end
'a'
```

In addition to simple positional indexing, sequences also support a more general form

of indexing known as slicing, which is a way to extract an entire section (slice) in a single

step. For example:

```
>>> S # A 4-character string
'Spam'
>>> S[1:3] # Slice of S from offsets 1 through 2 (not 3)
'pa'
```

Probably the easiest way to think of slices is that they are a way to extract an entire

column from a string in a single step. Their general form, X[I:J], means “give me everything in X from offset I up to but not including offset J.” The result is returned in a

new object. The second of the preceding operations, for instance, gives us all the characters in string S from offsets 1 through 2 (that is, 1 through 3 – 1) as a new string. The

effect is to slice or “parse out” the two characters in the middle.

In a slice, the left bound defaults to zero, and the right bound defaults to the length of

the sequence being sliced. This leads to some common usage variations:

```
>>> S[1:] # Everything past the first (1:len(S))
'pam'
>>> S # S itself hasn't changed
'Spam'
>>> S[0:3] # Everything but the last
'Spa'
>>> S[:3] # Same as S[0:3]
'Spa'
>>> S[:-1] # Everything but the last again, but simpler (0:-1)
```

```
'Spa'
>>> S[:] # All of S as a top-level copy (0:len(S))
'Spam'
```

Note in the second-to-last command how negative offsets can be used to give bounds for slices, too, and how the last operation effectively copies the entire string. As you'll learn later, there is no reason to copy a string, but this form can be useful for sequences like lists.

Finally, as sequences, strings also support concatenation with the plus sign (joining two strings into a new string) and repetition (making a new string by repeating another):

```
>>> S
'Spam'

>>> S + 'xyz' # Concatenation
'Spamxyz'
>>> S # S is unchanged
'Spam'
>>> S * 8 # Repetition
'SpamSpamSpamSpamSpamSpamSpamSpam'
```

Also notice in the prior examples that we were not changing the original string with any of the operations we ran on it. Every string operation is defined to produce a new string as its result, because strings are immutable in Python—they cannot be changed in place after they are created. In other words, you can never overwrite the values of immutable objects. For example, you can't change a string by assigning to one of its positions, but you can always build a new one and assign it to the same name. Because Python cleans up old objects as you go (as you'll see later), this isn't as inefficient as it may sound:

```
>>> S
'Spam'
>>> S[0] = 'z' # Immutable objects cannot be changed
...error text omitted...
TypeError: 'str' object does not support item assignment
>>> S = 'z' + S[1:] # But we can run expressions to make new objects
>>> S
'zpam'
```

Every object in Python is classified as either immutable (unchangeable) or not. In terms of the core types, numbers, strings, and tuples are immutable; lists, dictionaries, and sets are not—they can be changed in place freely, as can most new objects you'll code with classes. This distinction turns out to be crucial in Python work, in ways that we can't yet fully explore. Among other things, immutability can be used to guarantee that an object remains constant throughout your program; mutable objects' values can be changed at any time and place (and whether you expect it or not).

Strictly speaking, you can change text-based data in place if you either expand it into a list of individual characters and join it back together with nothing between, or use the newer bytearray type available in Pythons 2.6, 3.0, and later:

```
>>> S = 'shrubbery'
>>> L = list(S) # Expand to a list: [...]
>>> L
['s', 'h', 'r', 'u', 'b', 'b', 'e', 'r', 'y']
>>> L[1] = 'c' # Change it in place
>>> ''.join(L) # Join with empty delimiter
'scrubbery'
>>> B = bytearray(b'spam') # A bytes/list hybrid (ahead)
```

```
>>> B.extend(b'eggs') # 'b' needed in 3.X, not 2.X
>>> B # B[i] = ord(c) works here too
bytearray(b'spameggs')
>>> B.decode() # Translate to normal string
'spameggs'
```

The bytearray supports in-place changes for text, but only for text whose characters are all at most 8-bits wide (e.g., ASCII). All other strings are still immutable—bytear ray is a distinct hybrid of immutable bytes strings (whose b'...' syntax is required in 3.X and optional in 2.X) and mutable lists (coded and displayed in []), and we have to learn more about both these and Unicode text to fully grasp this code.

Type-Specific Methods

Every string operation we've studied so far is really a sequence operation—that is, these operations will work on other sequences in Python as well, including lists and tuples. In addition to generic sequence operations, though, strings also have operations all their own, available as methods—functions that are attached to and act upon a specific object, which are triggered with a call expression.

For example, the string find method is the basic substring search operation (it returns the offset of the passed-in substring, or −1 if it is not present), and the string replace method performs global searches and replacements; both act on the subject that they are attached to and called from:

```
>>> S = 'Spam'
>>> S.find('pa') # Find the offset of a substring in S
1
>>> S
'Spam'
>>> S.replace('pa', 'XYZ') # Replace occurrences of a string in S with another
'SXYZm'
>>> S
```

'Spam'

```
>>> line = 'aaa,bbb,ccccc,dd'
>>> line.split(',') # Split on a delimiter into a list of substrings
['aaa', 'bbb', 'ccccc', 'dd']
>>> S = 'spam'
>>> S.upper() # Upper- and lowercase conversions
'SPAM'
>>> S.isalpha() # Content tests: isalpha, isdigit, etc.
True
>>> line = 'aaa,bbb,ccccc,dd\n'
>>> line.rstrip() # Remove whitespace characters on the right side
'aaa,bbb,ccccc,dd'
>>> line.rstrip().split(',') # Combine two operations
['aaa', 'bbb', 'ccccc', 'dd']
```

Pattern Matching

One point worth noting before we move on is that none of the string object's own

methods support pattern-based text processing. Text pattern matching is an advanced

tool outside this book's scope, but readers with backgrounds in other scripting languages may be interested to know that to do pattern matching in Python, we import a

module called re. This module has analogous calls for searching, splitting, and replacement, but because we can use patterns to specify substrings, we can be much more

general:

```
>>> import re
>>> match = re.match('Hello[ \t]*(.*)world', 'Hello Python world')
>>> match.group(1)
'Python '
```

This example searches for a substring that begins with the word "Hello," followed by

zero or more tabs or spaces, followed by arbitrary characters to be saved as a matched

group, terminated by the word “world.” If such a substring is found, portions of the substring matched by parts of the pattern enclosed in parentheses are available as groups. The following pattern, for example, picks out three groups separated by slashes, and is similar to splitting by an alternatives pattern:

```
>>> match = re.match('[/:](.*)[/:](.*)[/:](.*)', '/usr/home:lumberjack')
>>> match.groups()
('usr', 'home', 'lumberjack')
>>> re.split('[/:]', '/usr/home/lumberjack')
['', 'usr', 'home', 'lumberjack']
```

Lists

The Python list object is the most general sequence provided by the language. Lists are positionally ordered collections of arbitrarily typed objects, and they have no fixed size. They are also mutable—unlike strings, lists can be modified in place by assignment to offsets as well as a variety of list method calls. Accordingly, they provide a very flexible tool for representing arbitrary collections—lists of files in a folder, employees in a company, emails in your inbox, and so on.

Sequence Operations

Because they are sequences, lists support all the sequence operations we discussed for strings; the only difference is that the results are usually lists instead of strings. For instance, given a three-item list:

```
>>> L = [123, 'spam', 1.23] # A list of three different-type objects
>>> len(L) # Number of items in the list
3
```

we can index, slice, and so on, just as for strings:

```
>>> L[0] # Indexing by position
123
>>> L[:-1] # Slicing a list returns a new list
```

[123, 'spam']

>>> L + [4, 5, 6] # Concat/repeat make new lists too

[123, 'spam', 1.23, 4, 5, 6]

>>> L * 2

[123, 'spam', 1.23, 123, 'spam', 1.23]

>>> L # We're not changing the original list

[123, 'spam', 1.23]

Type-Specific Operations

Python’s lists may be reminiscent of arrays in other languages, but they tend to be more powerful. For one thing, they have no fixed type constraint—the list we just looked at, for example, contains three objects of completely different types (an integer, a string, and a floating-point number). Further, lists have no fixed size. That is, they can grow and shrink on demand, in response to list-specific operations:

>>> L.append('NI') # Growing: add object at end of list

>>> L

[123, 'spam', 1.23, 'NI']

>>> L.pop(2) # Shrinking: delete an item in the middle

1.23

>>> L # "del L[2]" deletes from a list too

[123, 'spam', 'NI']

Here, the list append method expands the list’s size and inserts an item at the end; the pop method (or an equivalent del statement) then removes an item at a given offset, causing the list to shrink. Other list methods insert an item at an arbitrary position (insert), remove a given item by value (remove), add multiple items at the end (extend), and so on. Because lists are mutable, most list methods also change the list object in place, instead of creating a new one:

>>> M = ['bb', 'aa', 'cc']

>>> M.sort()

>>> M

['aa', 'bb', 'cc']

>>> M.reverse()

>>> M

['cc', 'bb', 'aa']

The list sort method here, for example, orders the list in ascending fashion by default, and reverse reverses it—in both cases, the methods modify the list directly.

Bounds Checking

Although lists have no fixed size, Python still doesn't allow us to reference items that are not present. Indexing off the end of a list is always a mistake, but so is assigning off the end:

>>> L

[123, 'spam', 'NI']

>>> L[99]

...error text omitted...

IndexError: list index out of range

>>> L[99] = 1

...error text omitted...

IndexError: list assignment index out of range

Comprehensions

In addition to sequence operations and list methods, Python includes a more advanced operation known as a list comprehension expression, which turns out to be a powerful way to process structures like our matrix. Suppose, for instance, that we need to extract the second column of our sample matrix. It's easy to grab rows by simple indexing because the matrix is stored by rows, but it's almost as easy to get a column with a list comprehension:

>>> col2 = [row[1] for row in M] # Collect the items in column 2

>>> col2

[2, 5, 8]

```
>>> M # The matrix is unchanged
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

List comprehensions derive from set notation; they are a way to build a new list by running an expression on each item in a sequence, one at a time, from left to right. List comprehensions are coded in square brackets (to tip you off to the fact that they make a list) and are composed of an expression and a looping construct that share a variable name

List comprehensions can be more complex in practice:

```
>>> [row[1] + 1 for row in M] # Add 1 to each item in column 2
[3, 6, 9]
>>> [row[1] for row in M if row[1] % 2 == 0] # Filter out odd items
[2, 8]
```

The first operation here, for instance, adds 1 to each item as it is collected, and the second uses an if clause to filter odd numbers out of the result using the % modulus expression (remainder of division). List comprehensions make new lists of results, but they can be used to iterate over any iterable object—a term we'll flesh out later in this preview. Here, for instance, we use list comprehensions to step over a hardcoded list of coordinates and a string:

```
>>> diag = [M[i][i] for i in [0, 1, 2]] # Collect a diagonal from matrix
>>> diag
[1, 5, 9]
>>> doubles = [c * 2 for c in 'spam'] # Repeat characters in a string
>>> doubles
['ss', 'pp', 'aa', 'mm']
```

These expressions can also be used to collect multiple values, as long as we wrap those values in a nested collection. The following illustrates using range—a built-in that generates successive integers, and requires a surrounding list to display all its values in 3.X only (2.X makes a physical list all at once):

```
>>> list(range(4)) # 0..3 (list() required in 3.X)
[0, 1, 2, 3]
>>> list(range(−6, 7, 2)) # −6 to +6 by 2 (need list() in 3.X)
[−6, −4, −2, 0, 2, 4, 6]
```

```
>>> [[x ** 2, x ** 3] for x in range(4)] # Multiple values, "if" filters
[[0, 0], [1, 1], [4, 8], [9, 27]]
>>> [[x, x / 2, x * 2] for x in range(−6, 7, 2) if x > 0]
[[2, 1, 4], [4, 2, 8], [6, 3, 12]]
```

Dictionaries

Python dictionaries are something completely different (Monty Python reference intended)—they are not sequences at all, but are instead known as mappings. Mappings are also collections of other objects, but they store objects by key instead of by relative position. In fact, mappings don't maintain any reliable left-to-right order; they simply map keys to associated values. Dictionaries, the only mapping type in Python's core objects set, are also mutable: like lists, they may be changed in place and can grow and shrink on demand. Also like lists, they are a flexible tool for representing collections, but their more mnemonic keys are better suited when a collection's items are named or labeled—fields of a database record, for example.

Mapping Operations

When written as literals, dictionaries are coded in curly braces and consist of a series

of "key: value" pairs. Dictionaries are useful anytime we need to associate a set of values

with keys—to describe the properties of something, for instance. As an example, consider the following three-item dictionary (with keys "food," "quantity," and "color,"

perhaps the details of a hypothetical menu item?):

```
>>> D = {'food': 'Spam', 'quantity': 4, 'color': 'pink'}
```

We can index this dictionary by key to fetch and change the keys' associated values.

The dictionary index operation uses the same syntax as that used for sequences, but

the item in the square brackets is a key, not a relative position:

```
>>> D['food'] # Fetch value of key 'food'
'Spam'
>>> D['quantity'] += 1 # Add 1 to 'quantity' value
>>> D
{'color': 'pink', 'food': 'Spam', 'quantity': 5}
```

Although the curly-braces literal form does see use, it is perhaps more common to see

dictionaries built up in different ways (it's rare to know all your program's data before

your program runs). The following code, for example, starts with an empty dictionary and fills it out one key at a time. Unlike out-of-bounds assignments in lists, which are forbidden, assignments to new dictionary keys create those keys:

```
>>> D = {}
>>> D['name'] = 'Bob' # Create keys by assignment
>>> D['job'] = 'dev'
>>> D['age'] = 40
>>> D
{'age': 40, 'job': 'dev', 'name': 'Bob'}
>>> print(D['name'])
Bob
```

Here, we're effectively using dictionary keys as field names in a record that describes someone. In other applications, dictionaries can also be used to replace searching operations—indexing a dictionary by key is often the fastest way to code a search in Python.

As we'll learn later, we can also make dictionaries by passing to the dict type name either keyword arguments (a special name=value syntax in function calls), or the result of zipping together sequences of keys and values obtained at runtime (e.g., from files). Both the following make the same dictionary as the prior example and its equivalent {} literal form, though the first tends to make for less typing:

```
>>> bob1 = dict(name='Bob', job='dev', age=40) # Keywords
>>> bob1
{'age': 40, 'name': 'Bob', 'job': 'dev'}

>>> bob2 = dict(zip(['name', 'job', 'age'], ['Bob', 'dev', 40])) # Zipping
>>> bob2
{'job': 'dev', 'name': 'Bob', 'age': 40}
```

Notice how the left-to-right order of dictionary keys is scrambled. Mappings are not positionally ordered, so unless you're lucky, they'll come back in a different order than

you typed them. The exact order may vary per Python, but you shouldn't depend on it, and shouldn't expect yours to match that in this book.

Nesting Revisited

In the prior example, we used a dictionary to describe a hypothetical person, with three keys. Suppose, though, that the information is more complex. Perhaps we need to record a first name and a last name, along with multiple job titles. This leads to another application of Python's object nesting in action. The following dictionary, coded all at once as a literal, captures more structured information:

```
>>> rec = {'name': {'first': 'Bob', 'last': 'Smith'},
 'jobs': ['dev', 'mgr'],
 'age': 40.5}
```

Here, we again have a three-key dictionary at the top (keys "name," "jobs," and "age"), but the values have become more complex: a nested dictionary for the name to support multiple parts, and a nested list for the jobs to support multiple roles and future expansion. We can access the components of this structure much as we did for our listbased matrix earlier, but this time most indexes are dictionary keys, not list offsets:

```
>>> rec['name'] # 'name' is a nested dictionary
{'last': 'Smith', 'first': 'Bob'}
>>> rec['name']['last'] # Index the nested dictionary
'Smith'
>>> rec['jobs'] # 'jobs' is a nested list
['dev', 'mgr']
>>> rec['jobs'][-1] # Index the nested list
'mgr'
>>> rec['jobs'].append('janitor') # Expand Bob's job description in place
>>> rec
{'age': 40.5, 'jobs': ['dev', 'mgr', 'janitor'], 'name': {'last': 'Smith',
'first': 'Bob'}}
```

Notice how the last operation here expands the nested jobs list—because the jobs list

is a separate piece of memory from the dictionary that contains it, it can grow and shrink freely (object memory layout will be discussed further later in this book).

Sorting Keys: for Loops

As mentioned earlier, because dictionaries are not sequences, they don't maintain any dependable left-to-right order. If we make a dictionary and print it back, its keys may come back in a different order than that in which we typed them, and may vary per Python version and other variables:

```
>>> D = {'a': 1, 'b': 2, 'c': 3}
>>> D
{'a': 1, 'c': 3, 'b': 2}
```

What do we do, though, if we do need to impose an ordering on a dictionary's items? One common solution is to grab a list of keys with the dictionary keys method, sort that with the list sort method, and then step through the result with a Python for loop (as for if, be sure to press the Enter key twice after coding the following for loop, and omit the outer parenthesis in the print in Python 2.X):

```
>>> Ks = list(D.keys()) # Unordered keys list
>>> Ks # A list in 2.X, "view" in 3.X: use list()
['a', 'c', 'b']
>>> Ks.sort() # Sorted keys list
>>> Ks
['a', 'b', 'c']
>>> for key in Ks: # Iterate though sorted keys
 print(key, '=>', D[key]) # <== press Enter twice here (3.X print)
a => 1
b => 2
c => 3
```

Tuples

The tuple object (pronounced "toople" or "tuhple," depending on whom you ask) is

roughly like a list that cannot be changed—tuples are sequences, like lists, but they are immutable, like strings. Functionally, they're used to represent fixed collections of items: the components of a specific calendar date, for instance. Syntactically, they are normally coded in parentheses instead of square brackets, and they support arbitrary types, arbitrary nesting, and the usual sequence operations:

```
>>> T = (1, 2, 3, 4) # A 4-item tuple
>>> len(T) # Length
4
>> T + (5, 6) # Concatenation
(1, 2, 3, 4, 5, 6)
>>> T[0] # Indexing, slicing, and more
1
```

Tuples also have type-specific callable methods as of Python 2.6 and 3.0, but not nearly as many as lists:

```
>>> T.index(4) # Tuple methods: 4 appears at offset 3
3
>>> T.count(4) # 4 appears once
1
```

The primary distinction for tuples is that they cannot be changed once created. That is, they are immutable sequences (one-item tuples like the one here require a trailing comma):

```
>>> T[0] = 2 # Tuples are immutable
...error text omitted...
TypeError: 'tuple' object does not support item assignment
>>> T = (2,) + T[1:] # Make a new tuple for a new value
>>> T
(2, 2, 3, 4)
```

Files

File objects are Python code's main interface to external files on your computer. They

can be used to read and write text memos, audio clips, Excel documents, saved email messages, and whatever else you happen to have stored on your machine. Files are a core type, but they're something of an oddball—there is no specific literal syntax for creating them. Rather, to create a file object, you call the built-in open function, passing in an external filename and an optional processing mode as strings.

For example, to create a text output file, you would pass in its name and the 'w' processing mode string to write data:

```
>>> f = open('data.txt', 'w') # Make a new file in output mode ('w' is write)
>>> f.write('Hello\n') # Write strings of characters to it
6
>>> f.write('world\n') # Return number of items written in Python 3.X
6
>>> f.close() # Close to flush output buffers to disk
```

This creates a file in the current directory and writes text to it (the filename can be a full directory path if you need to access a file elsewhere on your computer). To read back what you just wrote, reopen the file in 'r' processing mode, for reading text input —this is the default if you omit the mode in the call. Then read the file's content into a string, and display it. A file's contents are always a string in your script, regardless of the type of data the file contains:

```
>>> f = open('data.txt') # 'r' (read) is the default processing mode
>>> text = f.read() # Read entire file into a string
>>> text
'Hello\nworld\n'
>>> print(text) # print interprets control characters
Hello
world
>>> text.split() # File content is always a string
['Hello', 'world']
```