

Python Fundamentals Training

Contents

1	Getting Started	1
1.1	The Interactive Interpreter	1
1.2	Lab	1
1.3	Lab	2
2	Types	3
2.1	Strings	3
2.2	Integers	3
2.2.1	Integer Division	3
2.3	Floats	4
2.4	Complex	4
3	Variables	5
3.1	Defining	5
3.2	Dynamic Typing	5
3.3	Strong Typing	5
3.4	Internals	6
4	Simple Expressions	8
4.1	Boolean Evaluation	8
4.2	Truthiness	9
4.3	Branching (if / elif / else)	9
4.4	Block Structure and Whitespace	9
4.5	Lab	9
4.6	Multiple Cases	10
4.7	Lab	10

5	Advanced Types: Containers	11
5.1	Lists	11
5.2	Lab	13
5.3	Strings Revisited	14
5.4	Tuples	14
5.5	Lab	16
5.6	Dictionaries	16
5.7	Lab	19
5.8	Sets	19
5.9	Collection Transitions	20
6	A Bit More Iteration	21
6.1	Loop-Else	21
7	Functions	23
7.1	Defining	23
7.2	Arguments	23
7.3	Mutable Arguments and Binding of Default Values	24
7.4	Accepting Variable Arguments	25
7.5	Unpacking Argument Lists	25
7.6	Scope	26
7.7	Lab	27
8	Exceptions	28
8.1	Basic Error Handling	28
9	Code Organization	29
9.1	Namespaces	29
9.2	Importing modules	30
9.3	Creating Modules	32
10	Working with Files	33
10.1	File I/O	33
11	Interacting with the Outside World	36
11.1	Options	36
12	Regular Expressions (re)	38
12.1	Lab	39

13 Functional Programming	40
13.1 Functions as Objects	40
13.2 Higher-Order Functions	40
13.3 Sorting: An Example of Higher-Order Functions	41
13.4 Anonymous Functions	41
13.5 Nested Functions	41
13.6 Closures	42
13.7 Lexical Scoping	42
13.8 Useful Function Objects: <code>operator</code>	43
13.9 Lab	44
13.10Decorators	44
13.11Lab	48
14 Advanced Iteration	49
14.1 List Comprehensions	49
14.2 Generator Expressions	50
14.3 Generator Functions	50
14.4 Iteration Helpers: <code>itertools</code>	51
14.4.1 <code>chain()</code>	51
14.4.2 <code>izip()</code>	52
14.5 Lab	52
15 Debugging Tools	53
15.1 logging	53
15.2 pprint	54
15.3 Lab	55
16 Object-Oriented Programming	56
16.1 Classes	56
16.2 Emulation	58
16.3 <code>classmethod</code> and <code>staticmethod</code>	59
16.4 Lab	60
16.5 Inheritance	61
16.6 Lab	65
16.7 Encapsulation	65
16.7.1 Intercepting Attribute Access	66
16.7.2 Properties	67
16.7.3 Descriptors	68
16.8 Lab	70

17 Easter Eggs

71

Chapter 1

Getting Started

1.1 The Interactive Interpreter

The Python installation includes an interactive interpreter that you can use to execute code as you type it. This is a great tool to use to try small samples and see the result immediately without having to manage output or print statements.

1.2 Lab

If you have not done it yet, download the lab files at the following URL:

https://marakana.com/static/student-files/python_fundamentals_labs.zip

(Linux / Mac) Open a terminal and type `python`. (Windows) Open the Python IDLE IDE from the Start menu.

1. What is the version number?
2. Type `help(str)` This is a simple way to get documentation for a builtin or standard library function. You can also use the online HTML documentation.

```
>>> help(str)
Help on class str in module __builtin__:

class str(basestring)
| str(object) -> string
|
| Return a nice string representation of the object.
| If the argument is a string, the return value is the same object.
|
| Method resolution order:
|   str
|   basestring
|   object
|
| Methods defined here:
|
| __add__(...)
|     x.__add__(y) <==> x+y
```

```
|  
...
```

1. Note the use of methods with names with two underscores at the beginning and end of the name. These are methods that you will generally never call directly. How do you think the `__add__()` method gets executed?
2. Now try typing the following commands to see the output. Note that you don't assign a result, you get that result in the interpreter.

```
>>> 'hello world'  
'hello world'  
>>> _ + '!'  
'hello world!'  
>>> hw = _  
>>> hw  
'hello world!'
```

Tip

In the interactive interpreter, you can use the special variable `"_"` to refer to the result of the last statement. Handy in this mode, but meaningless in scripts.

Note

Throughout the rest of this courseware, the `">>>"` in a listing indicates that the code is being executed in the interactive interpreter.

1.3 Lab

Enter the following into a new file "hello.py" in your text editor of choice.

```
print 'hello world!'
```

Save and exit, then execute the script as shown below.

```
$ python hello.py  
hello world
```

Tip

On unix, you can also use shebang (`#!/`) notation on the first line.

Chapter 2

Types

2.1 Strings

String literals can be defined with any of single quotes ('), double quotes (") or triple quotes (''' or """). All give the same result with two important differences.

1. If you quote with single quotes, you do not have to escape double quotes and vice-versa.
2. If you quote with triple quotes, your string can span multiple lines.

```
>>> 'hello' + " " + '''world'''  
'hello world'
```

2.2 Integers

Integer literals are created by any number without a decimal or complex component.

```
>>> 1 + 2  
3
```

2.2.1 Integer Division

Some programming tasks make extensive use of integer division and Python behaves in the expected manner.

```
>>> 10 / 3  
3  
>>> 10 % 3  
1  
>>> divmod(10, 3)  
(3, 1)
```

2.3 Floats

Float literals can be created by adding a decimal component to a number.

```
>>> 1.0 / .99
1.0101010101010102
```

2.4 Complex

Complex literals can be created by using the notation $x + yj$ where x is the real component and y is the imaginary component.

```
>>> 1j * 1j
(-1+0j)
```

Chapter 3

Variables

3.1 Defining

A variable in Python is defined through assignment. There is no concept of declaring a variable outside of that assignment.

```
>>> ten = 10
>>> ten
10
```

3.2 Dynamic Typing

In Python, while the value that a variable points to has a type, the variable itself has no strict type in its definition. You can re-use the same variable to point to an object of a different type. It may be helpful to think of variables as "labels" associated with objects.

```
>>> ten = 10
>>> ten
10
>>> ten = 'ten'
>>> ten
'ten'
```

3.3 Strong Typing

**Caution**

While Python allows you to be very flexible with your types, you must still be aware of what those types are. Certain operations will require certain types as arguments.

```
>>> 'Day ' + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

This behavior is different from some other loosely-typed languages. If you were to do the same thing in JavaScript, you would get a different result.

```
d8> 'Day ' + 1
Day 1
```

In Python, however, it is possible to change the type of an object through builtin functions.

```
>>> 'Day ' + str(1)
'Day 1'
```

This type conversion can be a necessity to get the outcome that you want. For example, to force float division instead of integer division. Without an explicit conversion, the division of two integers will result in a third integer.

```
>>> 10 / 3
3
```

By converting one of the operands to a float, Python will perform float division and give you the result you were looking for (at least within floating point precision).

```
>>> float(10) / 3
3.3333333333333335
>>> 10 / float(3)
3.3333333333333335
```

You can also force the initial value to be a float by including a decimal point.

```
>>> 10.0 / 3
3.3333333333333335
```

Make sure to account for order of operations, though. If you convert the result of integer division, it will be too late to get the precision back.

```
>>> float(10 / 3)
3.0
```

3.4 Internals

Each object in Python has three key attributes: a type, a value, and an id. The type and the id can be examined using the `type()` and `id()` functions respectively. The id is implementation-dependent, but in most standard Python interpreters represents the location in memory of the object.

```
>>> a = 1
>>> type(a)
<type 'int'>
>>> id(a)
4298185352
>>> b = 2
```

```
>>> type(b)
<type 'int'>
>>> id(b)
4298185328
```

Multiple instances of the same immutable may be optimized by the interpreter to, in fact, be the same instance with multiple labels. The next example is a continuation of the previous. Notice that by subtracting 1 from `b`, its value becomes the same as `a` and so does its `id`. So rather than changing the value at the location to which `b` points, `b` itself is changed to point to a location that holds the right value. This location may be the location of an already existing object or it may be a new location. That choice is an optimization made by the interpreter.

```
>>> b = b - 1
>>> b
1
>>> id(b)
4298185352
```

Python uses reference counting to track how many of these labels are currently pointing to a particular object. When that count reaches 0, the object is marked for garbage collection after which it may be removed from memory.

Chapter 4

Simple Expressions

4.1 Boolean Evaluation

Boolean expressions are created with the keywords `and`, `or`, `not` and `is`. For example:

```
>>> True and False
False
>>> True or False
True
>>> not True
False
>>> not False
True
>>> True is True
True
>>> True is False
False
>>> 'a' is 'a'
True
```

Compound boolean evaluations shortcut and return the last expression evaluated. In other words:

```
>>> False and 'a' or 'b'
'b'
>>> True and 'a' or 'b'
'a'
```

Up until Python 2.6, this mechanism was the simplest way to implement a "ternary if" (`a = expression ? if_true : if_false`) statement. As of Python 2.6, it is also possible to use:

```
>>> 'a' if True else 'b'
'a'
>>> 'a' if False else 'b'
'b'
```

4.2 Truthiness

Many of the types in Python have truth values that can be used implicitly in boolean checks. However, it is important to note that this behavior is different from C where almost everything ends up actually being a zero. True, False and None in Python are all singleton objects and comparisons are best done with the `is` keyword.

Table 4.1: Truthiness values

Value	truthy	None	True	False
None	N	Y	N	N
0	N	N	N	N
1	Y	N	N	N
'hi'	Y	N	N	N
True	Y	N	Y	N
False	N	N	N	Y
[]	N	N	N	N
[0]	Y	N	N	N

4.3 Branching (if / elif / else)

Python provides the `if` statement to allow branching based on conditions. Multiple `elif` checks can also be performed followed by an optional `else` clause. The `if` statement can be used with any evaluation of truthiness.

```
>>> i = 3
>>> if i < 3:
...     print 'less than 3'
... elif i < 5:
...     print 'less than 5'
... else:
...     print '5 or more'
...
less than 5
```

4.4 Block Structure and Whitespace

The code that is executed when a specific condition is met is defined in a "block." In Python, the block structure is signalled by changes in indentation. Each line of code in a certain block level must be indented equally and indented more than the surrounding scope. The standard (defined in PEP-8) is to use 4 spaces for each level of block indentation. Statements preceding blocks generally end with a colon (:).

Because there are no semi-colons or other end-of-line indicators in Python, breaking lines of code requires either a continuation character (`\` as the last char) or for the break to occur inside an unfinished structure (such as open parentheses).

4.5 Lab

Edit `hello.py` as follows:

```
from datetime import datetime

hour = datetime.now().hour
if hour < 12:
    time_of_day = 'morning'
else:
    time_of_day = 'afternoon'

print 'Good %s, world!' % time_of_day
```

4.6 Multiple Cases

Python does not have a switch or case statement. Generally, multiple cases are handled with an *if-elif-else* structure and you can use as many *elif*'s as you need.

4.7 Lab

Fix *hello.py* to handle evening and the midnight case.

Chapter 5

Advanced Types: Containers

One of the great advantages of Python as a programming language is the ease with which it allows you to manipulate containers. Containers (or collections) are an integral part of the language and, as you'll see, built in to the core of the language's syntax. As a result, thinking in a Pythonic manner means thinking about containers.

5.1 Lists

The first container type that we will look at is the list. A list represents an ordered, mutable collection of objects. You can mix and match any type of object in a list, add to it and remove from it at will.

Creating Empty Lists To create an empty list, you can use empty square brackets or use the `list()` function with no arguments.

```
>>> l = []
>>> l
[]
>>> l = list()
>>> l
[]
```

Initializing Lists You can initialize a list with content of any sort using the same square bracket notation. The `list()` function also takes an iterable as a single argument and returns a shallow copy of that iterable as a new list. A list is one such iterable as we'll see soon, and we'll see others later.

```
>>> l = ['a', 'b', 'c']
>>> l
['a', 'b', 'c']
>>> l2 = list(l)
>>> l2
['a', 'b', 'c']
```

A Python string is also a sequence of characters and can be treated as an iterable over those characters. Combined with the `list()` function, a new list of the characters can easily be generated.

```
>>> list('abcdef')
['a', 'b', 'c', 'd', 'e', 'f']
```

Adding You can append to a list very easily (add to the end) or insert at an arbitrary index.

```
>>> l = []
>>> l.append('b')
>>> l.append('c')
>>> l.insert(0, 'a')
>>> l
['a', 'b', 'c']
```

Note

While inserting at position 0 will work, the underlying structure of a list is not optimized for this behavior. If you need to do it a lot, use `collections.deque`, which is optimized for this behavior (at the expense of some pointer overhead) and has an `appendleft()` function.

Iterating Iterating over a list is very simple. All iterables in Python allow access to elements using the `for ... in` statement. In this structure, each element in the iterable is sequentially assigned to the "loop variable" for a single pass of the loop, during which the enclosed block is executed.

```
>>> for letter in l:
...     print letter,
...
a b c
```

Note

The `print` statement adds a newline character when called. Using a trailing `,` in the `print` statement prevents a newline character from being automatically appended.

Iterating with while It is also possible to use a `while` loop for this iteration. A `while` loop is most commonly used to perform an iteration of unknown length, either checking a condition on each entry or using a `break` statement to exit when a condition is met.

For the simplicity of the example, here we will use the `list.pop()` method to consume the list entries from the right.

```
>>> l = ['a', 'b', 'c']
>>> while len(l):
...     print l.pop(),
...
c b a
```

Iterating with an Index In some instances, you will actually want to know the index of the item that you are accessing inside the `for` loop. You can handle this in a traditional form using the builtin `len()` and `range()` functions.

```
>>> len(l)
3
>>> range(3)
[0, 1, 2]

>>> for i in range(len(l)):
...     print i, l[i]
...
0 a
1 b
2 c
```

However, with a little more foundation, we will see a better way.

Access and Slicing Accessing individual items in a list is very similar to accessing the elements of an array in many languages, often referred to as subscripting, or more accurately, using the subscript operator. One less common, but very useful addition, is the ability to use negative indexing, where `alist[-1]` returns the last element of `alist`. Note that 0 represents the first item in a list while -1 represents the last.

Slices are another extension of this subscripting syntax providing access to subsets of the list. The slice is marked with one or two colons (:) within the square bracket subscript.

In the single colon form, the first argument represents the starting index (inclusive) and the second argument represents the end index (exclusive). If the first is omitted (e.g. `l[:2]`), the start index is the beginning of the list. If the second argument is omitted (e.g. `l[2:]`) the end index is the last item in the list.

In the double colon form, the first two arguments are unchanged and the third represents *stride*. For example, `l[::2]` would take every second item from a list.

```
>>> l = list('abcdefgh')
>>> l
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']

>>> l[3]
'd'
>>> l[-3]
'f'

>>> l[1:4]
['b', 'c', 'd']

>>> l[1:-1]
['b', 'c', 'd', 'e', 'f', 'g']

>>> l[1:-1:2]
['b', 'd', 'f']
```

Presence and Finding Checking for the presence of an object in a list is performed with the `in` keyword. The `index()` method of a list returns the actual location of the object.

```
>>> chars = list('abcdef')
>>> chars
['a', 'b', 'c', 'd', 'e', 'f']
>>> 'g' in chars
False
>>> 'c' in chars
True
>>> chars.index('c')
2
```

5.2 Lab

1. Create a new file `class.py`
2. Use a list to store the first names of everyone in the class.
3. Use a for loop to print `Hello <name>` to stdout for everyone.

5.3 Strings Revisited

Python's string object not only acts as a sequence but has many useful methods for moving back and forth to other types of sequences. A very common use case in data processing is splitting strings into substrings based on a delimiter. This is done with the `split()` method, which returns a list of the components.

```
>>> s = 'abc.def'
>>> parts = s.split('.')
>>> parts
['abc', 'def']
```

The converse method to `split()` is `join()` which joins a list together separating each element by the contents of the string on which the method was called.

This method looks backwards to many people when they first see it (thinking that `.join()` should be a method of the list object). It is also important to realize that a string literal in Python is just another instance of a string object.

Using `'/'` from the following example, that string is a string object and the `.join()` method can be called on it. There is no point assigning it to a variable before using it, because it is of no value after that single call.

```
>>> new_string = '/'.join(parts)
>>> new_string
'abc/def'
```

Other standard sequence operations also apply.

```
>>> s = 'hello world'
>>> len(s)
11
>>> s[4]
'o'
>>> s[2:10:2]
'lowr'
```

A less sequence-oriented, but still quite common method for dealing with strings is trimming whitespace with the `strip()` method and its relatives: `lstrip()` and `rstrip()`.

```
>>> s = '  abc  '
>>> s.strip()
'abc'
```

5.4 Tuples

A tuple is like an immutable list. It is slightly faster and smaller than a list, so it is useful. Tuples are also commonly used in core program features, so recognize them.

Creating Similar to lists, empty tuples can be created with an empty pair of parentheses, or with the `tuple()` builtin function.

```
>>> t = ()
>>> t
()
>>> tuple()
()
```

Tuples can also be initialized with values (and generally are, since they are immutable). There is one important distinction to make due to the use of parentheses, which is that a 1-tuple (tuple with one item) requires a trailing comma to indicate that the desired result is a tuple. Otherwise, the interpreter will see the parentheses as nothing more than a grouping operation.

```
>>> t = ('Othello')
>>> t
'Othello'
>>> t = ('Othello',)
>>> t
('Othello',)
```

The behavior for a 2-tuple and beyond is nothing new. Note that the parentheses become optional at this point. The implication that any comma-separated list without parentheses becomes a tuple is both useful and important in Python programming.

```
>>> t = ('Othello', 'Iago')
>>> t
('Othello', 'Iago')
>>> t = 'Othello', 'Iago'
>>> t
('Othello', 'Iago')
```

Tuples can also be created by passing an iterable to the `tuple()` function.

```
>>> l = ['Othello', 'Iago']
>>> tuple(l)
('Othello', 'Iago')
```

Unpacking A very common paradigm for accessing tuple content in Python is called unpacking. It can be used on lists as well, but since it requires knowledge of the size of the container, it is far more common with tuples.

By assigning a tuple to a list of variables that matches the count of items in the tuple, the variables are individually assigned ordered values from the tuple.

```
>>> t = ('Othello', 'Iago')
>>> hero, villain = t
>>> hero
'Othello'
>>> villain
'Iago'
```

An interesting and valuable side-effect of the natural use of tuples is the ability to elegantly swap variables.

```
>>> t = ('Othello', 'Iago')
>>> t
('Othello', 'Iago')

>>> hero, villain = t

>>> hero
'Othello'
>>> villain
'Iago'

>>> hero, villain = villain, hero
```

```
>>> hero
'Iago'
>>> villain
'Othello'
```

Accessing and Slicing Tuples can be accessed and sliced in the same manner as lists. Note that tuple slices are tuples themselves.

```
>>> t[0]
'Othello'
>>> t = ('Othello', 'Iago', 'Desdemona')
>>> t[0::2]
('Othello', 'Desdemona')
```

Iterating Tuples are iterable, in exactly the same manner as lists.

```
>>> t = ('Othello', 'Iago')
>>> for character in t:
...     print character
...
Othello
Iago
```

Since a tuple is iterable, a mutable copy is easily created using the `list()` builtin.

```
>>> t = ('Othello', 'Iago')
>>> list(t)
['Othello', 'Iago']
```

Indexed List Iteration Revisited Now that you know how to unpack tuples, you can see a better way to iterate lists with an index. The builtin `enumerate()` function takes a single argument (an iterable) and returns an iterator of 2-tuples. Each 2-tuple contains an index and an item from the original iterable. These 2-tuples can be unpacked into separate loop variables as part of the `for` statement.

```
>>> l = ['a', 'b', 'c']
>>> for i, letter in enumerate(l):
...     print i, letter
...
0 a
1 b
2 c
```

5.5 Lab

1. Update `classmates.py` to use a tuple of 3-tuples of first, last, role.
2. Print to screen using a `for` loop and unpacking.

5.6 Dictionaries

A dictionary is an implementation of a key-value mapping that might go by the name "hashtable" or "associative array" in another language. Dictionaries are the building blocks of the Python language itself, so they are quite prevalent and also quite efficient.

**Warning**

Dictionary order is undefined and implementation-specific. It can be different across interpreters, versions, architectures, and more. Even multiple executions in the same environment.

Creating Following the analogy of the other container types, dictionaries are created using braces, i.e. `{}`. There is also a `dict()` builtin function that accepts an arbitrary set of keyword arguments.

Note

Unlike some similar languages, string keys in Python must always be quoted.

```
>>> characters = {'hero': 'Othello', 'villain': 'Iago', 'friend': 'Cassio'}
>>> characters
{'villain': 'Iago', 'hero': 'Othello', 'friend': 'Cassio'}

>>> characters = dict(hero='Othello', villain='Iago', friend='Cassio')
>>> characters
{'villain': 'Iago', 'hero': 'Othello', 'friend': 'Cassio'}
```

Accessing Dictionary values can be accessed using the subscript operator except you use the key instead of an index as the subscript argument. The presence of keys can also be tested with the `in` keyword.

```
>>> if 'villain' in characters:
...     print characters['villain']
...
Iago
```

Adding A new entry can be created where there is no existing key using the same subscripting notation and assignment.

```
>>> characters['beauty'] = 'Desdemona'
```

Modifying Existing entries are modified in exactly the same manner.

```
>>> characters['villain'] = 'Roderigo'

>>> characters
{'villain': 'Roderigo', 'hero': 'Othello', 'beauty': 'Desdemona', 'friend': 'Cassio'}
```

Failed Lookups If you use the subscript operator and the key is not found, a `KeyError` will be raised. If this behavior is not desired, using the `get()` method of the dictionary will return a supplied default value when the key is not found. If no default is provided, `None` is returned when the key is not found. The `get()` method does not alter the contents of the dictionary itself.

```
>>> characters['horse']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    KeyError: 'horse'
>>> characters.get('horse', 'Ed')
'Ed'
>>> characters
{'villain': 'Roderigo', 'hero': 'Othello', 'friend': 'Cassio', 'beauty': 'Desdemona'}
>>> characters.get('horse')
>>>
```

You can also set the value in the case that it wasn't found using the `setdefault()` method.

```
>>> characters
{'villain': 'Roderigo', 'hero': 'Othello', 'friend': 'Cassio', 'beauty': 'Desdemona'}
>>> characters.setdefault('horse', 'Ed')
'Ed'
>>> characters
{'villain': 'Roderigo', 'horse': 'Ed', 'hero': 'Othello', 'friend': 'Cassio', 'beauty': 'Desdemona'}
```

Iterating Because the dictionary has both keys and values, iterating has a few more options. A simple *for... in* statement will iterate over the keys, which is one method to access both.

```
>>> for role in characters:
...     print role, characters[role]
...
villain Roderigo
hero Othello
beauty Desdemona
friend Cassio
```

However, the `items()` method will return 2-tuples of key, value pairs, which can be unpacked in the for loop.

```
>>> characters.items()
[('villain', 'Roderigo'), ('hero', 'Othello'), ('friend', 'Cassio'), ('beauty', 'Desdemona')]
>>> for role, name in characters.items():
...     print role, name
...
villain Roderigo
hero Othello
friend Cassio
beauty Desdemona
```

Note

The `.items()` method returns a newly allocated list full of newly allocated tuples, that exists only for the duration of the iteration. Whenever possible, it is preferable to use the `.iteritems()` method, which returns a generator. This generator holds a reference to the original dictionary and produces individual 2-tuples on demand. The downside of the iterator is that it expects the state of the dictionary to remain consistent during iteration.

```
>>> characters.iteritems()
<dictionary-itemiterator object at 0x100473b50>
>>> for role, name in characters.iteritems():
...     print role, name
...
villain Roderigo
hero Othello
friend Cassio
beauty Desdemona
```

5.7 Lab

1. One more time on classmates.py
2. Insert the tuples into a dictionary using firstname as the key
3. Ask for a firstname on the command-line and print the data
4. If it's not there, prompt for last name and role
5. Add it
6. Print the new list

5.8 Sets

A set is a mutable, unordered, unique collection of objects. It is designed to reflect the properties and behavior of a true mathematical set. A *frozenset* has the same properties as a set, except that it is immutable.

Creating A new set is created using the `set()` builtin. This function without any parameters will return a new, empty set. It will also accept a single argument of an iterable, in which case it will return a new set containing one element for each unique element in the iterable.

```
>>> s = set()
>>> s
set([])
>>> s = set(['Beta', 'Gamma', 'Alpha', 'Delta', 'Gamma', 'Beta'])
>>> s
set(['Alpha', 'Beta', 'Gamma', 'Delta'])
```

Accessing Sets are not designed for indexed access, so it is not possible to use subscript notation. Like a list, we can use the `.pop()` method to consume elements, but note that the order will be undefined.

```
>>> s
set(['Alpha', 'Beta', 'Gamma', 'Delta'])
>>> while len(s):
...     print s.pop(),
...
Alpha Beta Gamma Delta
```

Set Operations Not surprisingly, the real value of sets shows itself in set operations. Sets use sensibly overloaded operators to calculate unions and intersections of sets. You can also call these methods by name.

```
>>> s1 = set(['Beta', 'Gamma', 'Alpha', 'Delta', 'Gamma', 'Beta'])
>>> s2 = set(['Beta', 'Alpha', 'Epsilon', 'Omega'])
>>> s1
set(['Alpha', 'Beta', 'Gamma', 'Delta'])
>>> s2
set(['Alpha', 'Beta', 'Omega', 'Epsilon'])

>>> s1.union(s2)
set(['Epsilon', 'Beta', 'Delta', 'Alpha', 'Omega', 'Gamma'])
>>> s1 | s2
set(['Epsilon', 'Beta', 'Delta', 'Alpha', 'Omega', 'Gamma'])
```

```
>>> s1.intersection(s2)
set(['Alpha', 'Beta'])
>>> s1 & s2
set(['Alpha', 'Beta'])

>>> s1.difference(s2)
set(['Gamma', 'Delta'])
>>> s1 - s2
set(['Gamma', 'Delta'])

>>> s1.symmetric_difference(s2)
set(['Epsilon', 'Delta', 'Omega', 'Gamma'])
>>> s1 ^ s2
set(['Epsilon', 'Delta', 'Omega', 'Gamma'])
```

5.9 Collection Transitions

As you saw previously, `dict.items()` returns a list of 2-tuples representing key-value pairs. Inversely, a list of 2-tuples can be passed to the `dict()` factory function to create a dictionary using the first item from each tuple as a key and the second item as the value. `zip()` takes `n` lists and returns one list of `n`-tuples.

```
>>> roles = characters.keys()
>>> roles
['villain', 'hero', 'beauty', 'friend']
>>> names = characters.values()
>>> names
['Roderigo', 'Othello', 'Desdemona', 'Cassio']

>>> tuples = zip(roles, names)
>>> tuples
[('villain', 'Roderigo'), ('hero', 'Othello'), ('beauty', 'Desdemona'), ('friend', ' ←
Cassio')]

>>> new_characters = dict(tuples)
>>> new_characters
{'villain': 'Roderigo', 'hero': 'Othello', 'friend': 'Cassio', 'beauty': 'Desdemona'}
```

Chapter 6

A Bit More Iteration

6.1 Loop-Else

An addition to both *for* and *while* loops in Python that is not common to all languages is the availability of an *else* clause. The *else* block after a loop is executed in the case where **no** *break* occurs inside the loop.

The most common paradigm for using this clause occurs when evaluating a dataset for the occurrences of a certain condition and breaking as soon as it is found. Rather than setting a flag when found and checking after to see the result, the *else* clause simplifies the code.

In the following example, if a multiple of 5 is found, the *break* exits the *for* loop without executing the *else* clause.

```
>>> for x in range(1,5):
...     if x % 5 == 0:
...         print '%d is a multiple of 5' % x
...         break
...     else:
...         print 'No multiples of 5'
...
No multiples of 5

>>> for x in range(11,20):
...     if x % 5 == 0:
...         print '%d is a multiple of 5' % x
...         break
...     else:
...         print 'No multiples of 5'
...
15 is a multiple of 5
```

Without this feature, the code would look something like:

```
>>> found = False
>>> for x in range(1,5):
...     if x % 5 == 0:
...         print '{0} is a multiple of 5'.format(x)
...         found = True
...         break
...
...
```

```
>>> if not found:  
...     print 'No multiples of 5'  
...  
No multiples of 5
```

Chapter 7

Functions

7.1 Defining

A function in Python is defined with the `def` keyword. Functions do not have declared return types. A function without an explicit `return` statement returns `None`. In the case of no arguments and no return value, the definition is very simple.

Calling the function is performed by using the call operator `()` after the name of the function.

```
>>> def hello_function():
...     print 'Hello World, it\'s me.  Function.'
...
>>> hello_function()
Hello World, it's me.  Function.
```

7.2 Arguments

The arguments of a function are defined within the `def` statement. Like all other variables in Python, there is no explicit type associated with the function arguments. This fact is important to consider when making assumptions about the types of data that your function will receive.

Function arguments can optionally be defined with a default value. The default value will be assigned in the case that the argument is not present in the call to the function. All arguments without default values must be listed before arguments with default values in the function definition.

Any argument can be passed either implicitly by position or explicitly by name, regardless of whether or not it has a default value defined.

```
>>> def record_score(name, score=0):
...     print '%s scored %s' % (name, score)
...
>>> record_score('Jill', 4)
Jill scored 4
>>> record_score('Jack')
Jack scored 0
```

```
>>> record_score(score=3, name='Pail')
Pail scored 3

>>> record_score(2)
2 scored 0

>>> record_score(score=2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: record_score() takes at least 1 non-keyword argument (0 given)

>>> record_score()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: record_score() takes at least 1 argument (0 given)
```

Note

Look carefully at the example above. There is an asymmetry in the use of the = sign for defining vs. passing arguments that can be confusing to beginners. An argument with a default value can be passed using only position and an argument without a default can be passed using a keyword.

7.3 Mutable Arguments and Binding of Default Values

When defining default arguments, take special care with mutable data types. The instance of the default value is bound at the time the function is defined. Consequently, there is a single instance of the mutable object that will be used across all calls to the function.

```
>>> def add_items(new_items, base_items=[]):
...     for item in new_items:
...         base_items.append(item)
...     return base_items
...

>>> add_items((1, 2, 3))
[1, 2, 3]

>>> add_items((1, 2, 3))
[1, 2, 3, 1, 2, 3]
```

As a result, it is best to use default value of `None` as a flag to signify the absence of the argument and handle the case inside the function body.

```
>>> def add_items(new_items, base_items=None):
...     if base_items is None:
...         base_items = []
...     for item in new_items:
...         base_items.append(item)
...     return base_items
...

>>> add_items((1, 2, 3))
```

```
[1, 2, 3]
>>> add_items((1, 2, 3))
[1, 2, 3]
```

7.4 Accepting Variable Arguments

In addition to named arguments, functions can accept two special collections of arguments.

The first is a variable-length, named tuple of any additional positional arguments received by the function. This special argument is identified by prefixing it with a single asterisk (*).

The second is a variable-length dictionary containing all keyword arguments passed to the function that were not explicitly defined as part of the function arguments. This argument is identified by prefixing it with two asterisks (**).

It is not required, but conventional and therefore highly recommended, to name these two arguments `args` and `kwargs`, respectively.

The use of these two arguments is illustrated in the following set of examples.

```
>>> def variable_function(*args, **kwargs):
...     print 'args:', args
...     print 'kwargs:', kwargs
...

>>> variable_function('simple')
args: ('simple',)
kwargs: {}

>>> variable_function(type='Complex')
args: ()
kwargs: {'type': 'Complex'}

>>> def mixed_function(a, b, c=None, *args, **kwargs):
...     print '(a, b, c):', (a, b, c)
...     print 'args:', args
...     print 'kwargs:', kwargs
...

>>> mixed_function(1, 2, 3, 4, 5, d=10, e=20)
(a, b, c): (1, 2, 3)
args: (4, 5)
kwargs: {'e': 20, 'd': 10}
```

7.5 Unpacking Argument Lists

It is also possible to construct argument lists (positional or keyword) and pass them into a function.

For positional arguments, insert them into a tuple / list and prepend with an asterisk (*) in the function call.

For keyword arguments, use a dictionary and prepend with two asterisks (**).

```
>>> def printer(a, b, c=0, d=None):
...     print 'a: {0}, b: {1}, c: {2}, d: {3}'.format(a, b, c, d)
...
>>> printer(2, 3, 4, 5)
a: 2, b: 3, c: 4, d: 5

>>> ordered_args = (5, 6)
>>> keyword_args = {'c': 7, 'd': 8}

>>> printer(*ordered_args, **keyword_args)
a: 5, b: 6, c: 7, d: 8
```

Note

The example above shows another potentially confusing asymmetry in Python. You can pass arguments using the regular style to a function defined using variable arguments, and you can pass unpacked variable argument lists to a function defined without variable arguments.

7.6 Scope

Each function evaluation creates a local namespace that is manipulated at any level within the function. As a result, variables can be initially defined at a seemingly lower level of scope than they are eventually used.

```
>>> def deep_scope():
...     if True:
...         if True:
...             if True:
...                 x = 5
...     return x
...

>>> deep_scope()
5
```

**Warning**

This model for scope can simplify your code, but pay attention. If you don't anticipate all code paths, you can end up referencing undefined variables.

```
>>> def oops(letter):
...     if letter == 'a':
...         out = 'A'
...     return out
...

>>> oops('a')
'A'

>>> oops('b')
```

```
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
  File "<stdin>", line 4, in oops  
UnboundLocalError: local variable 'out' referenced before assignment
```

7.7 Lab

1. Copy `classmates.py` to `classfilter.py`
 2. Ask the user for a role on the command-line
 3. Using a function and the builtin `filter()`, print a list of your classmates with that role
-

Chapter 8

Exceptions

8.1 Basic Error Handling

Exception handling in Python follows a similar pattern to many other languages with this construct. The statements with exception handling are contained in a `try` block, followed by one or more `except` blocks. The `except` keyword is analogous to `catch` in some other languages.

The body of the `try` block is executed up to and including whatever statement raises (throws) an exception. At that point, the first `except` clause with an associated type that either matches, or is a supertype of, the exception raised will execute.

If no `except` clause matches, execution will exit the current scope and the process will continue back up the stack until the exception is handled or program execution terminates.

Regardless of whether the exception is caught at the current execution level, the optional `finally` clause will execute if present. The `finally` clause will also execute if no exception is raised and is ideally suited for resource cleanup.

If no exception is raised, optional `else` block will be executed. The idea of an `else` block in exceptions is less common and designed to limit the contents of the `try` block, such that the types of exceptions being tested are quite explicit.

```
>>> characters = {'hero': 'Othello', 'villain': 'Iago', 'friend': 'Cassio'}
>>> def get_char(role):
...     try:
...         name = characters[role]
...     except KeyError, e:
...         result = 'This play has no %s' % role
...     else:
...         result = '%s is the %s' % (name, role)
...     finally:
...         result += ' and it\'s a great play'
...     return result
...
>>> get_char('champion')
"This play has no champion and it's a great play"
>>> get_char('friend')
"Cassio is the friend and it's a great play"
```

Chapter 9

Code Organization

In this section, we'll cover some of the tools you need as your scripts get to be bigger than just the contents of a single file. Namely code organization, command-line arguments and file I/O.

9.1 Namespaces

Inside a single python module (file) there are multiple namespaces. The global namespace represents the full contents of the file, while inside each function there is a local namespace. The contents of these namespaces can be accessed as dictionaries using the functions `globals()` and `locals()` respectively.

When objects (variables, functions, etc) are defined in the file, they manipulate the global namespace. Anything defined inside a function manipulates its local namespace. Notice in the example below that the namespace is manipulated as the file is read. In other words, the first print statement occurs before the interpreter is aware of the presence of the function definition.

Note

The following example makes use of `import`, which will be explained in the next section. It also uses `pprint.pformat` which converts a dictionary into a string in a manner that is more easily read when printed.

organization-1-namespaces.py

```
'''Some documentation for this file.'''
import pprint

print 'globals before def: %s\n' % pprint.pformat(globals(), indent=4)

def simple():
    print 'locals before a: %s\n' % locals()
    a = 'simple'
    print 'locals after a: %s\n' % locals()
    return a

print 'globals after def: %s\n' % pprint.pformat(globals(), indent=4)

simple()
```

```
$ python organization-1-namespaces.py
globals before def: {  '__builtins__': <module '__builtin__' (built-in)>,
  '__doc__': 'Some documentation for this file.',
  '__file__': 'samples/organization-1-namespaces.py',
  '__name__': '__main__',
  '__package__': None,
  'pprint': <module 'pprint' from '/opt/local/Library/Frameworks/Python.framework/ ←
    Versions/2.6/lib/python2.6/pprint.pyc'>}

globals after def: {  '__builtins__': <module '__builtin__' (built-in)>,
  '__doc__': 'Some documentation for this file.',
  '__file__': 'samples/organization-1-namespaces.py',
  '__name__': '__main__',
  '__package__': None,
  'pprint': <module 'pprint' from '/opt/local/Library/Frameworks/Python.framework/ ←
    Versions/2.6/lib/python2.6/pprint.pyc'>,
  'simple': <function simple at 0x100425f50>}

locals before a: {}

locals after a: {'a': 'simple'}
```

9.2 Importing modules

Have another look at an example similar to the one above. Notice that the modules that are imported are present in the global namespace.

organization-2-imports.py

```
import collections
import pprint

d = collections.deque()
d.append('a')
d.appendleft('b')

pprint.pprint(globals())

$ python organization-2-imports.py
{'__builtins__': <module '__builtin__' (built-in)>,
  '__doc__': None,
  '__file__': 'samples/organization-2-imports.py',
  '__name__': '__main__',
  '__package__': None,
  'collections': <module 'collections' from '/opt/local/Library/Frameworks/Python. ←
    framework/Versions/2.6/lib/python2.6/collections.pyc'>,
  'd': deque(['b', 'a']),
  'pprint': <module 'pprint' from '/opt/local/Library/Frameworks/Python.framework/ ←
    Versions/2.6/lib/python2.6/pprint.pyc'>}
```

Objects from this module are accessed using dotted notation.

Alternatively, you can import the specific element from the module, using the `from ... import` syntax.

organization-3-import-submodule.py

```
from collections import deque
from pprint import pprint

d = deque()
d.append('a')
d.appendleft('b')

pprint(globals())
```

```
$ python organization-3-import-submodule.py
{'__builtins__': <module '__builtin__' (built-in)>,
 '__doc__': None,
 '__file__': 'samples/organization-3-import-submodule.py',
 '__name__': '__main__',
 '__package__': None,
 'd': deque(['b', 'a']),
 'deque': <type 'collections.deque'>,
 'pprint': <function pprint at 0x100435578>}
```

It is also possible to import an entire namespace. This should be done with caution, as it can lead to unexpected elements in your namespace and conflicts in naming.

organization-4-import-all.py

```
from collections import *
from pprint import pprint

d = deque()
d.append('a')
d.appendleft('b')

pprint(globals())
```

```
$ python organization-4-import-all.py
{'Callable': <class '_abcoll.Callable'>,
 'Container': <class '_abcoll.Container'>,
 'Hashable': <class '_abcoll.Hashable'>,
 'ItemsView': <class '_abcoll.ItemsView'>,
 'Iterable': <class '_abcoll.Iterable'>,
 'Iterator': <class '_abcoll.Iterator'>,
 'KeysView': <class '_abcoll.KeysView'>,
 'Mapping': <class '_abcoll.Mapping'>,
 'MappingView': <class '_abcoll.MappingView'>,
 'MutableMapping': <class '_abcoll.MutableMapping'>,
 'MutableSequence': <class '_abcoll.MutableSequence'>,
 'MutableSet': <class '_abcoll.MutableSet'>,
 'Sequence': <class '_abcoll.Sequence'>,
 'Set': <class '_abcoll.Set'>,
 'Sized': <class '_abcoll.Sized'>,
 'ValuesView': <class '_abcoll.ValuesView'>,
 '__builtins__': <module '__builtin__' (built-in)>,
 '__doc__': None,
 '__file__': 'samples/organization-4-import-all.py',
 '__name__': '__main__',
 '__package__': None,
 'd': deque(['b', 'a']),
```

```
'defaultdict': <type 'collections.defaultdict'>,
'deque': <type 'collections.deque'>,
'namedtuple': <function namedtuple at 0x100425ed8>,
'pprint': <function pprint at 0x1004355f0>}
```

9.3 Creating Modules

Once your code starts to get bigger than a script, you will want to start organizing it into modules. Unlike some other languages (Java for example) each file in Python is a module. Directories can also be used as a further layer of organization with some care.

Using the file-only model, functions can be created in another file in the same directory (or somewhere in the \$PYTHON-PATH) and imported using the filename and the function name.

tools.py

```
def shorten(toolong):
    return toolong[:2]
```

complexity-4-file-module.py

```
from tools import shorten
print shorten('abcdef')
```

```
$ python complexity-4-file-module.py
ab
```

As code starts to require even more organization, perhaps with multiple types of utility functions, this file could be moved to a subdirectory. In order to use a directory as a module, it is required that the special file `__init__.py` be present in the directory, which can be empty.

```
$ ls tools2
tools2/__init__.py
tools2/strings.py
```

tools2/strings.py

```
def shorten(toolong):
    return toolong[:2]
```

complexity-5-directory-module.py

```
from tools2 import strings
print strings.shorten('abcdef')
```

```
$ python complexity-5-directory-module.py
ab
```

Chapter 10

Working with Files

10.1 File I/O

Getting data into and out of files in Python feels a lot like using the low-level methods of C, but it has all the ease of Python layered on top.

For example, to open a file for writing use the builtin (and very C-like) `open()` function. But then write the contents of a list with a single call. The `open()` function returns an open file object and closing the file is done by calling the `close()` method of that object.

Note

the `writelines()` and `readlines()` methods in Python do not handle EOL characters on your behalf, making the naming a bit confusing. In this example, note the inclusion of the `\n` character in the elements of the list.

```
>>> colors = ['red\n', 'yellow\n', 'blue\n']
>>> f = open('colors.txt', 'w')
>>> f.writelines(colors)
>>> f.close()
```

By default, the `open()` function returns a file open for reading. Individual lines can be read with the `readline()` method, which will return an empty string. Since the zero-length string has is not truthy, it makes a simple marker.

```
>>> f = open('colors.txt')
>>> f.readline()
'red\n'
>>> f.readline()
'yellow\n'
>>> f.readline()
'blue\n'
>>> f.readline()
''
>>> f.close()
```

Alternatively, all of the lines of the file can be read into a list with one method call and then iterated over from there.

```
>>> f = open('colors.txt')
>>> f.readlines()
['red\n', 'yellow\n', 'blue\n']
>>> f.close()
```

However, for large files, reading the contents into memory can be impractical. So it is best to use the file object itself as an iterator, which will consume content from the file as needed with no intermediary memory requirement.

```
>>> f = open('colors.txt')
>>> for line in f:
...     print line,
...
red
yellow
blue
>>> f.close()
```

In order to ensure that the files in the above examples were properly closed, they should have been safe-guarded against an abnormal exit (by Exception or other unexpected return) using a `finally` statement. See the following example, where you can see in the final `seek()` that the file is closed in the case of proper execution.

```
>>> f = open('colors.txt')
>>> try:
...     lines = f.readlines()
... finally:
...     f.close()
...
>>> lines
['red\n', 'yellow\n', 'blue\n']

>>> f.seek(0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file
```

In the next example, the code path fails attempting to write to a file opened for reading. The file is still closed in the *finally* clause.

```
>>> f = open('colors.txt')
>>> try:
...     f.writelines('magenta\n')
... finally:
...     f.close()
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
IOError: File not open for writing

>>> f.seek(0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file
```

As of Python 2.6, there is a builtin syntax for handling this paradigm, using the `with` keyword. `with` creates a context and regardless of how that context is exited, calls the `__exit__()` method of the object being managed. In the following example, that object is the file `f`, but this model works for any file-like object (objects with the basic methods of files).

Once again, performing an operation on the file outside of that context shows that it has been closed.

```
>>> with open('colors.txt') as f:
...     for line in f:
...         print line,
...
red
yellow
blue

>>> f.seek(0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file
```

As mentioned above, file-like objects are objects that look and feel like files. In Python, there is no strict inheritance requirement in order to use an object as if it were another. `StringIO` is an example of a file-like object that manages its contents in memory instead of on disk.

It is important to be conscious of the file-like behavior here. Note that a `seek` is required after writing to get back to the beginning and read. Attempting to run the same iterator twice results in no values the second time through.

`StringIO.getvalue()` returns a newly created string object with the full contents of the `StringIO` buffer.

```
>>> colors = ['red\n', 'yellow\n', 'blue\n']
>>> from StringIO import StringIO
>>> buffer = StringIO()
>>> buffer.writelines(colors)
>>> buffer.seek(0)

>>> for line in buffer:
...     print line,
...
red
yellow
blue

>>> for line in buffer:
...     print line,
...

>>> buffer.getvalue()
'red\nyellow\nblue\n'
```

Chapter 11

Interacting with the Outside World

11.1 Options

Each script invoked from the command line in Python has access to any arguments in the `sys.argv` list. While the contents of this list are quite predictable, it is rarely necessary to handle them directly. The preferred option is to use the `optparse` module and the `OptionParser` class from that module.

An instance of `OptionParser` has methods for adding options (long and short flags) and including parameters with those options (if desired). The action to be taken when an option is encountered is also configurable, including setting a boolean flag, storing a value, or executing a callback function.

Defaults and help text are also covered, as well as error handling. Here is a basic example using the standard features.

The call to `parse_args()` returns a two-tuple. The first item in the tuple is an options object containing any parameters or boolean values captured from the command line. The second item is a list containing any additional arguments passed at the end of the command line.

```
from optparse import OptionParser

p = OptionParser()

p.add_option('-d', '--debug', action='store_true',
             dest='debug', help='Turn on debugging')

p.add_option('-s', '--speed', action='store',
             type='int', dest='speed',
             help='Use a bigger number to go faster')

p.add_option('-u', '--username', action='store',
             type='string', dest='user',
             help='Provide your username')

p.set_defaults(debug=False, speed=0)

options, args = p.parse_args()

if options.user is None:
    p.error('Username is required')

print 'debug option is: %s' % options.debug
```

```
print 'speed option is: %s' % options.speed
print 'args are: %s' % args
```

```
$ python options.py
Usage: options.py [options]

options.py: error: Username is required
```

Unless modified in the creation, each instance of `OptionParser` adds a `-h` option to print help for the user.

```
$ python options.py -h
Usage: options.py [options]

Options:
  -h, --help            show this help message and exit
  -d, --debug           Turn on debugging
  -s SPEED, --speed=SPEED
                        Use a bigger number to go faster
  -u USER, --username=USER
                        Provide your username
```

In the output below, note the value *myfile.txt*, which was not associated with any flag.

```
$ python options.py -u me -s 9 myfile.txt
debug option is: False
speed option is: 9
args are: ['myfile.txt']
```

Chapter 12

Regular Expressions (re)

While regular expression handling is very complete in Python, regular expressions are not a first-class language element as they are in Perl or JavaScript. Regular expression handling is found in the `re` module.

At the simplest level, there are module-level functions in `re` that can be used to search for regular expressions. In many cases, calling the `search()` function and checking for the presence of a return value is enough. `search()` returns `None` if the pattern was not found.

Note

It is customary in Python regular expressions to pass the patterns as raw strings (`r'pattern'`) to avoid escaping the special characters that are likely included in the pattern.

```
>>> text = 'All your base are belong to us.'
>>> re.search(r'o\s?u', text)
<_sre.SRE_Match object at 0x10041f718>
```

Take note of the `match()` function, which specifically only matches the beginning of the text being matched and does not search throughout for the pattern.

```
>>> re.match(r'o\s?u', text)
>>> re.match('All', text)
<_sre.SRE_Match object at 0x10041f718>
```

Regular expressions can also be used to split strings in more advanced ways than the `string.split()` method.

```
>>> re.split(r'o\s?u', text)
['All y', 'r base are belong t', 's.']
```

Using the `findall()` or `finditer()` methods, it is possible to process all the matching groups. `findall()` returns a list while `finditer()` returns an iterator.

```
>>> re.findall(r'o\s?u', text)
['ou', 'o u']
>>> re.finditer(r'o\s?u', text)
<callable-iterator object at 0x100516610>
```

If you need to use the same pattern multiple times, you can improve performance by compiling the regex and then using the methods of the regex object, rather than the module-level functions.

If groups are defined in the pattern, they can be accessed using the `group()` method of the returned Match object. Note that they are 1-indexed to conform to most other regex utilities.

```
>>> text = 'All your base are belong to us.'
>>> pattern = re.compile(r'you[r]?\s*(\S*)\s*are belong to us')
>>> match = pattern.search(text)
>>> match.group(1)
'base'
```

12.1 Lab

1. Rename `complexity-1-fileutil.py` to `fileutil.py`
2. Implement `fileutil.py` to pass the doctests
3. Create a second file `grep.py` that accepts command line arguments and calls the function in `grep.py`
4. Accept the following command-line args:
 - a. `-v`, `--invert-match` select non-matching lines
 - b. `-E`, `--extended-regexp PATTERN` is an extended regular expression
 - c. And a list of files

Chapter 13

Functional Programming

When it comes to functional programming, Python is classified as a "hybrid" language. It supports the functional programming paradigm, but equally supports imperative paradigms (both procedural and object-oriented).

13.1 Functions as Objects

Functions are first-class objects in Python, meaning they have attributes and can be referenced and assigned to variables.

```
>>> def i_am_an_object(myarg):
...     '''I am a really nice function.
...     Please be my friend.'''
...     return myarg
...
>>> i_am_an_object(1)
1

>>> an_object_by_any_other_name = i_am_an_object
>>> an_object_by_any_other_name(2)
2

>>> i_am_an_object
<function i_am_an_object at 0x100432aa0>

>>> an_object_by_any_other_name
<function i_am_an_object at 0x100432aa0>

>>> i_am_an_object.__doc__
'I am a really nice function.\n          Please be my friend.'
```

13.2 Higher-Order Functions

Python also supports higher-order functions, meaning that functions can accept other functions as arguments and return functions to the caller.

```
>>> i_am_an_object(i_am_an_object)
<function i_am_an_object at 0x100519848>
```

13.3 Sorting: An Example of Higher-Order Functions

In order to define non-default sorting in Python, both the `sorted()` function and the list's `.sort()` method accept a `key` argument. The value passed to this argument needs to be a function object that returns the sorting key for any item in the list or iterable.

For example, given a list of tuples, Python will sort by default on the first value in each tuple. In order to sort on a different element from each tuple, a function can be passed that returns that element.

```
>>> def second_element(t):
...     return t[1]
...
>>> zepp = [('Guitar', 'Jimmy'), ('Vocals', 'Robert'), ('Bass', 'John Paul'), ('Drums', 'John')]

>>> sorted(zepp)
[('Bass', 'John Paul'), ('Drums', 'John'), ('Guitar', 'Jimmy'), ('Vocals', 'Robert')]

>>> sorted(zepp, key=second_element)
[('Guitar', 'Jimmy'), ('Drums', 'John'), ('Bass', 'John Paul'), ('Vocals', 'Robert')]
```

13.4 Anonymous Functions

Anonymous functions in Python are created using the `lambda` statement. This approach is most commonly used when passing a simple function as an argument to another function. The syntax is shown in the next example and consists of the `lambda` keyword followed by a list of arguments, a colon, and the expression to evaluate and return.

```
>>> def call_func(f, *args):
...     return f(*args)
...
>>> call_func(lambda x, y: x + y, 4, 5)
9
```

Whenever you find yourself writing a simple anonymous function, check for a builtin first. The `operator` module is a good place to start (see the section below). For example, the `lambda` function above could have been replaced with the `operator.add` builtin.

```
>>> import operator
>>> def call_func(f, *args):
...     return f(*args)
...
>>> call_func(operator.add, 4, 5)
9
```

13.5 Nested Functions

Functions can be defined within the scope of another function. If this type of function definition is used, the inner function is only in scope inside the outer function, so it is most often useful when the inner function is being returned (moving it to the outer scope) or when it is being passed into another function.

Notice that in the below example, a new instance of the function `inner()` is created on each call to `outer()`. That is because it is defined during the execution of `outer()`. The creation of the second instance has no impact on the first.

```
>>> def outer():
...     def inner(a):
...         return a
...     return inner
...
>>> f = outer()
>>> f
<function inner at 0x1004340c8>
>>> f(10)
10

>>> f2 = outer()
>>> f2
<function inner at 0x1004341b8>
>>> f2(11)
11

>>> f(12)
12
```

13.6 Closures

A nested function has access to the environment in which it was defined. Remember from above that the definition occurs during the execution of the outer function. Therefore, it is possible to return an inner function that remembers the state of the outer function, even after the outer function has completed execution. This model is referred to as a closure.

```
>>> def outer2(a):
...     def inner2(b):
...         return a + b
...     return inner2
...
>>> add1 = outer2(1)
>>> add1
<function inner2 at 0x100519c80>
>>> add1(4)
5
>>> add1(5)
6
>>> add2 = outer2(2)
>>> add2
<function inner2 at 0x100519cf8>
>>> add2(4)
6
>>> add2(5)
7
```

13.7 Lexical Scoping

A common pattern that occurs while attempting to use closures, and leads to confusion, is attempting to encapsulate an internal variable using an immutable type. When it is re-assigned in the inner scope, it is interpreted as a new variable and fails because it hasn't been defined.

```
>>> def outer():
...     count = 0
...     def inner():
...         count += 1
...         return count
...     return inner
...

>>> counter = outer()

>>> counter()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in inner
UnboundLocalError: local variable 'count' referenced before assignment
```

The standard workaround for this issue is to use a mutable datatype like a list and manage state within that object.

```
>>> def better_outer():
...     count = [0]
...     def inner():
...         count[0] += 1
...         return count[0]
...     return inner
...

>>> counter = better_outer()
>>> counter()
1
>>> counter()
2
>>> counter()
3
```

13.8 Useful Function Objects: `operator`

There are many builtin functions in Python that accept functions as arguments. An example is the `filter()` function that was used previously. However, there are some basic actions that use operators instead of functions (like `+` or the subscript `[]` or dot `.` operators).

The `operator` module provides function versions of these operators.

```
>>> import operator
>>> operator.add(1, 2)
3
```

Using closures, it's possible to create functions dynamically that can, for example, know **which** item to get from a list.

```
>>> get_second = operator.itemgetter(1)
>>> get_second(['a', 'b', 'c', 'd'])
'b'
```

The `itemgetter()` function will return a tuple if it is given more than one index.

```
>>> get_02 = operator.itemgetter(0, 2)
>>> get_02(['a', 'b', 'c', 'd'])
('a', 'c')
```

A typical use for the `itemgetter()` function is as the key argument to a list sort.

```
>>> import operator
>>> zepp = [('Guitar', 'Jimmy'), ('Vocals', 'Robert'), ('Bass', 'John Paul'), ('Drums ←
', 'John')]

>>> sorted(zepp)
[('Bass', 'John Paul'), ('Drums', 'John'), ('Guitar', 'Jimmy'), ('Vocals', 'Robert')]

>>> sorted(zepp, key=operator.itemgetter(1))
[('Guitar', 'Jimmy'), ('Drums', 'John'), ('Bass', 'John Paul'), ('Vocals', 'Robert')]
```

13.9 Lab

functional-1-reduce.py

```
'''
>>> series1 = (0, 1, 2, 3, 4, 5)
>>> series2 = (2, 4, 8, 16, 32)
>>> power_reducer = add_powers(2)
>>> power_reducer(series1)
55
>>> power_reducer(series2)
1364
>>>
>>> power_reducer = add_powers(3)
>>> power_reducer(series1)
225
>>> power_reducer(series2)
37448

Hint: use the builtin reduce() function
'''
if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

13.10 Decorators

A powerful and common, albeit slightly complex, paradigm in Python is the use of function decorators. The usual role of decorators is to reduce repetition in code by consolidating common elements of setup and teardown in functions.

Understanding decorators requires some foundation, so the first building block is a straightforward function that does nothing out of the ordinary.

The function `simple()` accepts a variable list of arguments and prints them to stdout.


```
import itertools

CLAIM = '{0} is the #{1} {2} language'

def best(type, *args):
    langs = []
    for i, arg in enumerate(args):
        langs.append(CLAIM.format(arg, i+1, type))
    return langs

def best_functional(*args):
    return best('functional', *args)

def best_oo(*args):
    return best('OO', *args)

for claim in itertools.chain(best_functional('Haskell', 'Erlang'), [''],
                             best_oo('Objective-C', 'Java')):
    print claim
```

```
$ python functional-1-langs.py
Haskell is the #1 functional language
Erlang is the #2 functional language

Objective-C is the #1 OO language
Java is the #2 OO language
```

Recognizing that Python is universally the best language in both arenas, a decorator can be defined to prevent any caller from making a mistake.

functional-2-langs-decorated.py

```
import itertools

CLAIM = '{0} is the #{1} {2} language'

def python_rules(func):
    def wrapper(*args, **kwargs):
        new_args = ['Python']
        new_args.extend(args)
        return func(*new_args)
    return wrapper

def best(type, *args):
    langs = []
    for i, arg in enumerate(args):
        langs.append(CLAIM.format(arg, i+1, type))
    return langs

def best_functional(*args):
    return best('functional', *args)
best_functional = python_rules(best_functional)

def best_oo(*args):
    return best('OO', *args)
best_oo = python_rules(best_oo)
```

```
for claim in itertools.chain(best_functional('Haskell', 'Erlang'), [''],
                             best_oo('Objective-C', 'Java')):
    print claim
```

```
$ python functional-2-langs-decorated.py
Python is the #1 functional language
Haskell is the #2 functional language
Erlang is the #3 functional language
```

```
Python is the #1 OO language
Objective-C is the #2 OO language
Java is the #3 OO language
```

The decorator paradigm is so common in Python that there is a special syntax using the @ symbol that allows the name of the decorating function to be placed immediately above the function definition line. This syntax, while it stands out, tends to lead to some confusion because it is less explicit than the previous format. So it is important to remember that the @ symbol notation does exactly the same thing under the covers as re-assigning the function variable name after the function definition.

functional-3-langs-atsyntax.py

```
import itertools

CLAIM = '{0} is the #{1} {2} language'

def python_rules(func):
    def wrapper(*args, **kwargs):
        new_args = ['Python']
        new_args.extend(args)
        return func(*new_args)
    return wrapper

def best(type, *args):
    langs = []
    for i, arg in enumerate(args):
        langs.append(CLAIM.format(arg, i+1, type))
    return langs

@python_rules
def best_functional(*args):
    return best('functional', *args)

@python_rules
def best_oo(*args):
    return best('OO', *args)

for claim in itertools.chain(best_functional('Haskell', 'Erlang'), [''],
                             best_oo('Objective-C', 'Java')):
    print claim
```

```
$ python functional-3-langs-atsyntax.py
Python is the #1 functional language
Haskell is the #2 functional language
Erlang is the #3 functional language
```

```
Python is the #1 OO language
Objective-C is the #2 OO language
Java is the #3 OO language
```

13.11 Lab

functional-2-decorate.py

```
'''
>>> data = '{"username": "oscar", "password": "trashcan", "account": 1234, "amount": ←
12.03}'
>>> deposit(data)
'OK'
>>> data = '{"username": "oscar", "password": "trash", "account": 1234, "amount": ←
14.98}'
>>> deposit(data)
'Invalid Password'
>>> data = '{"username": "oscar", "password": "trashcan", "account": 1234, "amount": ←
4.12}'
>>> withdraw(data)
'OK'
>>> data = '{"username": "oscar", "password": "trashcan", "account": 1235, "amount": ←
2.54}'
>>> withdraw(data)
'Invalid Account'
>>> data = '{"username": "oscar", "password": "trashcan", "account": 1234}'
>>> balance(data)
'7.91'
>>> data = '{"username": "oscar", "password": "trashcan}'
>>> balance(data)
'No Account Number Provided'

Hint: that's json data
'''

def deposit(account, amount=0.00):
    pass

def withdraw(account, amount=0.00):
    pass

def balance(account):
    pass

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

Chapter 14

Advanced Iteration

14.1 List Comprehensions

Python provides list comprehension syntax to simplify the task of generating new lists from existing lists (or from any other iterable data type).

In the earlier example of writing to a file, the `\n` character was stored with each color in the list so that it could be passed to the `writelines()` method of the file object. Storing the data in this way would make other processing challenging, so a more common model would be to create a list with line feeds based on a list without them.

Without list comprehensions, this operation would look something like the following:

```
>>> colors = ['red', 'yellow', 'blue']
>>> color_lines = []
>>> for color in colors:
...     color_lines.append('{0}\n'.format(color))
...
>>> color_lines
['red\n', 'yellow\n', 'blue\n']
```

Many functional languages provide either a standalone `map()` function or `map()` list method to perform this task. While rarely used in Python, this function is available.

```
>>> colors = ['red', 'yellow', 'blue']
>>> color_lines = map(lambda c: '{0}\n'.format(c), colors)
>>> color_lines
['red\n', 'yellow\n', 'blue\n']
```

List comprehensions perform this task equally well, but provide additional functionality. To accomplish the same task with a list comprehension, use the following syntax:

```
>>> colors = ['red', 'yellow', 'blue']
>>> color_lines = ['{0}\n'.format(color) for color in colors]
>>> color_lines
['red\n', 'yellow\n', 'blue\n']
```

A conditional filter can also be included in the creation of the new list, like so:

```
>>> colors = ['red', 'yellow', 'blue']
>>> color_lines = ['{0}\n'.format(color) for color in colors if 'l' in color]
>>> color_lines
['yellow\n', 'blue\n']
```

More than one list can be iterated over, as well, which will create a pass for each combination in the lists.

```
>>> colors = ['red', 'yellow', 'blue']
>>> clothes = ['hat', 'shirt', 'pants']

>>> colored_clothes = ['{0} {1}'.format(color, garment) for color in colors for
    garment in clothes]

>>> colored_clothes
['red hat', 'red shirt', 'red pants', 'yellow hat', 'yellow shirt', 'yellow pants', '
blue hat', 'blue shirt', 'blue pants']
```

14.2 Generator Expressions

Storing a new list as the output of a list comprehension is not always optimal behavior. Particularly in a case where that list is intermediary or where the total size of the contents is quite large.

For such cases, a slightly modified syntax (replacing square brackets with parentheses) leads to the creation of a generator instead of a new list. The generator will produce the individual items in the list as each one is requested, which is generally while iterating over that new list.

```
>>> colors = ['red', 'yellow', 'blue']
>>> color_lines = ('{0}\n'.format(color) for color in colors)
>>> color_lines
<generator object <genexpr> at 0x10041ac80>
>>> color_lines.next()
'red\n'
>>> color_lines.next()
'yellow\n'
>>> color_lines.next()
'blue\n'
>>> color_lines.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

14.3 Generator Functions

The type of object created by the generator expression in the previous section is unsurprisingly called a generator. This is a term for a type of iterator that generates values on demand.

While the generator expression notation is very compact, there may be cases where there is more logic to be performed than can be effectively expressed with this notation. For such cases, a generator function can be used.

A generator function uses the `yield` statement in place of `return` and usually does so inside a loop. When the interpreter sees this statement, it will actually return a generator object from the function. Each time the `next()` function

is called on the generator object, the function will be executed up to the next `yield`. When the function completes, the interpreter will raise a `StopIteration` error to the caller.

```
>>> def one_color_per_line():
...     colors = ['red', 'yellow', 'blue']
...     for color in colors:
...         yield '{0}\n'.format(color)
...
>>> gen = one_color_per_line()
>>> gen
<generator object one_color_per_line at 0x10041acd0>
>>> gen.next()
'red\n'
>>> gen.next()
'yellow\n'
>>> gen.next()
'blue\n'
>>> gen.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Note that a second call to the same generator function will return a new generator object (shown at a different address) as each generator should be capable of maintaining its own state.

```
>>> gen = one_color_per_line()
>>> gen
<generator object one_color_per_line at 0x10041ad20>
```

Of course, the more typical use case would be to allow the calls to `next()` to be handled by a `for ... in` loop.

```
>>> for line in one_color_per_line():
...     print line,
...
red
yellow
blue
```

14.4 Iteration Helpers: `itertools`

Iteration is a big part of the flow of Python and aside from the builtin syntax, there are some handy tools in the `itertools` package to make things easier. They also tend to make things run faster.

14.4.1 `chain()`

The `chain()` method accepts an arbitrary number of iterable objects as arguments and returns an iterator that will iterate over each iterable in turn. Once the first is exhausted, it will move onto the next.

Without the `chain()` function, iterating over two lists would require creating a copy with the contents of both or adding the contents of one to the other.

```
>>> l1 = ['a', 'b', 'c']
>>> l2 = ['d', 'e', 'f']
```

```
>>> l1.extend(l2)
>>> l1
['a', 'b', 'c', 'd', 'e', 'f']
```

It's much more efficient to use the `chain()` function which only allocates additional storage for some housekeeping data in the iterator itself.

```
>>> import itertools
>>> l1 = ['a', 'b', 'c']
>>> l2 = ['d', 'e', 'f']

>>> chained = itertools.chain(l1, l2)
>>> chained
<itertools.chain object at 0x100431250>

>>> [l for l in chained]
['a', 'b', 'c', 'd', 'e', 'f']
```

14.4.2 `izip()`

`izip()` is almost identical to the `zip()` builtin, in that it pairs up the contents of two lists into an iterable of 2-tuples. However, where `zip()` allocates a new list, `izip()` only returns an iterator.

```
>>> name = ['Jimmy', 'Robert', 'John Paul', 'John']
>>> instruments = ['Guitar', 'Vocals', 'Bass', 'Drums']

>>> zepp = zip(name, instruments)
>>> zepp
[('Jimmy', 'Guitar'), ('Robert', 'Vocals'), ('John Paul', 'Bass'), ('John', 'Drums')]

>>> zepp = itertools.izip(name, instruments)
>>> zepp
<itertools.izip object at 0x100430998>

>>> [musician for musician in zepp]
[('Jimmy', 'Guitar'), ('Robert', 'Vocals'), ('John Paul', 'Bass'), ('John', 'Drums')]
```

14.5 Lab

1. Convert your `grep` function from the previous lab to use a generator function
2. Each call to `.next()` on the iterator should return the next matching line

Chapter 15

Debugging Tools

15.1 logging

The `logging` module in the Python standard libraries is similar in function to `log4j` and all of its derivatives in other languages. Logging configuration is hierarchical and can range from very simple to very complex to suit application needs.

Here is an example of a very standard logging configuration and its usage.

Note that you should ideally find a place in your application that will only be executed once in order to configure the logging.

logconfig.py

```
import logging
from logging import handlers

def config():
    logger = logging.getLogger('app')
    logger.setLevel(logging.ERROR)
    handler = handlers.RotatingFileHandler(
        'debug.log',
        maxBytes=(1024*1024),
        backupCount=5
    )
    formatter = logging.Formatter('%(asctime)s - %(levelname)s - %(message)s')
    handler.setFormatter(formatter)
    logger.addHandler(handler)

    logger = logging.getLogger('app.othersub')
    logger.setLevel(logging.DEBUG)
```

debug-1-logging.py

```
import logging

from logconfig import config

config()
```

```
logger = logging.getLogger('app.sub')
logger.debug('something interesting just happened')
logger.error('something horrible just happened')

logger2 = logging.getLogger('app.othersub')
logger2.debug('HMM> something interesting just happened')
logger2.setLevel(logging.DEBUG)
logger2.error('UH-OH> something horrible just happened')

class YouCantDoThatException(Exception):
    pass

class ThatIsABadThingException(Exception):
    pass

def dobrokenstuff():
    d = {}
    e = {}
    key = 'missingkey'
    try:
        # val = d[key]
        # raise YouCantDoThatException
        raise KeyError('badkey')
    except KeyError, ke:
        d[key] = 'samplestring'
        logger2.exception('two exceptions')
    else:
        val = val.reverse()
        print val
        val = val.lower()
        finalval = e[val]
        logger2.debug('ELSE: HERE I AM')
    finally:
        logger2.debug('FINALLY: HERE I AM')

def dostuff():
    try:
        dobrokenstuff()
    except YouCantDoThatException, e:
        logger2.exception('Don\'t do that')

try:
    dostuff()
except:
    logger.exception("How did this happen?")
```

```
$ cat debug.log
2010-08-01 21:03:28,074 - DEBUG - something interesting just happened
2010-08-01 21:03:28,074 - ERROR - something horrible just happened
```

15.2 pprint

In conjunction with the logging module, the pprint (pretty print) module can be very helpful. It is easy to include a list or dictionary in a log statement, but it can be hard to read when debugging. In the pprint module, the pprint

function prints to standard out while the `pformat` function returns a string that is more likely to be useful in logging or for other custom output requirements.

```
>>> import pprint
>>> deep = [{'letter': let, 'index': i, 'extended': let*i}
...         for i, let in enumerate(list('abcdefg'))]
>>> deep
[{'index': 0, 'extended': '', 'letter': 'a'}, {'index': 1, 'extended': 'b', 'letter': 'b'}, {'index': 2, 'extended': 'cc', 'letter': 'c'}, {'index': 3, 'extended': 'ddd', 'letter': 'd'}, {'index': 4, 'extended': 'eeee', 'letter': 'e'}, {'index': 5, 'extended': 'fffff', 'letter': 'f'}, {'index': 6, 'extended': 'gggggg', 'letter': 'g'}]

>>> pprint.pprint(deep)
[{'extended': '', 'index': 0, 'letter': 'a'},
 {'extended': 'b', 'index': 1, 'letter': 'b'},
 {'extended': 'cc', 'index': 2, 'letter': 'c'},
 {'extended': 'ddd', 'index': 3, 'letter': 'd'},
 {'extended': 'eeee', 'index': 4, 'letter': 'e'},
 {'extended': 'fffff', 'index': 5, 'letter': 'f'},
 {'extended': 'gggggg', 'index': 6, 'letter': 'g'}]
```

15.3 Lab

1. Create a simple script with three loggers 'app', 'app.sub', 'app.sub.sub'.
2. Log a debug and critical message for each.
3. Set 'app' to CRITICAL and 'app.sub.sub' to DEBUG. What messages do you get?
4. Include a timestamp and the log level in your messages.
5. raise an Exception and log it with `.exception()`

Chapter 16

Object-Oriented Programming

16.1 Classes

A class is defined in Python using the `class` statement. The syntax of this statement is `class <ClassName> (superclass)`. In the absence of anything else, the superclass should always be `object`, the root of all classes in Python.

Note

`object` is technically the root of "new-style" classes in Python, but new-style classes today are as good as being the only style of classes.

Here is a basic class definition for a class with one method. There are a few things to note about this method:

1. The single argument of the method is `self`, which is a reference to the object instance upon which the method is called, is explicitly listed as the first argument of the method. In the example, that instance is `a`. This object is commonly referred to as the "bound instance."
2. However, when the method is called, the `self` argument is inserted implicitly by the interpreter — it does not have to be passed by the caller.
3. The attribute `__class__` of `a` is a reference to the class object `A`.
4. The attribute `__name__` of the class object is a string representing the name, as given in the class definition.

Also notice that "calling" the class object (`A`) produces a newly instantiated object of that type (assigned to `a` in this example). You can think of the class object as a factory that creates objects and gives them the behavior described by the class definition.

```
>>> class A(object):
...     def whoami(self):
...         return self.__class__.__name__
...
>>> a = A()
>>> a
<__main__.A object at 0x100425d90>
```

```
>>> a.whoami()
'A'
```

The most commonly used special method of classes is the `__init__()` method, which is an initializer for the object. The arguments to this method are passed in the call to the class object.

Notice also that the arguments are stored as object attributes, but those attributes are not defined anywhere before the initializer.

Attempting to instantiate an object of this class without those arguments will fail.

```
>>> class Song(object):
...     def __init__(self, title, artist):
...         self.title = title
...         self.artist = artist
...     def get_title(self):
...         return self.title
...     def get_artist(self):
...         return self.artist
...

>>> unknown = Song()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __init__() takes exactly 3 arguments (1 given)
```

Notice again that one argument was actually provided (`self`) and only `title` and `artist` are considered missing.

So, calling `Song` properly gives an instance of `Song` with an `artist` and `title`. Calling the `get_title()` method returns the `title`, but so does just referencing the `title` attribute. It is also possible to directly write the instance attribute. Using boilerplate getter / setter methods is generally considered unnecessary. There are ways to create encapsulation that will be covered later.

```
>>> leave = Song('Leave', 'Glen Hansard')

>>> leave
<__main__.Song object at 0x100431050>

>>> leave.get_title()
'Leave'

>>> leave.title
'Leave'

>>> leave.title = 'Please Leave'

>>> leave.title
'Please Leave'
```

One mechanism that can be utilized to create some data privacy is a preceding double-underscore on attribute names. However, it is possible to find and manipulate these variables if desired, because this approach simply mangles the attribute name with the class name. The goal in this mangling is to prevent clashing between "private" attributes of classes and "private" attributes of their superclasses.

```
>>> class Song(object):
...     def __init__(self, title, artist):
```

```
...     self.__title = title
...     self.__artist = artist
...     def get_title(self):
...         return self.__title
...     def get_artist(self):
...         return self.__artist
...
>>> leave = Song('Leave', 'Glen Hansard')

>>> leave.get_title()
'Leave'

>>> leave.__title
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Song' object has no attribute '__title'

>>> leave._Song__title
'Leave'
```

16.2 Emulation

Python provides many special methods on classes that can be used to emulate other types, such as functions, iterators, containers and more.

Functions In order to emulate a function object, a class must define the method `__call__()`. If the call operator `()` is used on an instance of the class, this method will be called behind the scenes. Here is an example that performs the same task as the adding closure in the functional programming section.

```
>>> class Adder(object):
...     def __init__(self, extra):
...         self.extra = extra
...     def __call__(self, base):
...         return self.extra + base
...
>>> add2 = Adder(2)
>>> add2(3)
5
>>> add5 = Adder(5)
>>> add5(3)
8
>>> add2(1)
3
```

Iterators When an object is used in a `for ... in` statement, the object's `__iter__()` method is called and the returned value should be an iterator. At that point, the interpreter iterates over the result, assigning each object returned from the iterator to the loop variable in the `for ... in` statement.

This example class implements the `__iter__()` method and returns a generator expression based on whatever arguments were passed to the initializer.

```
>>> class Lister(object):
...     def __init__(self, *args):
...         self.items = tuple(args)
```

```
...     def __iter__(self):
...         return (i for i in self.items)
...

>>> l = Lister('a', 'b', 'c')

>>> for letter in l:
...     print letter,
...
a b c
```

Here is the same example using a generator function instead of a generator expression.

```
>>> class Lister(object):
...     def __init__(self, *args):
...         self.items = tuple(args)
...     def __iter__(self):
...         for i in self.items:
...             yield i
...

>>> l = Lister('a', 'b', 'c')

>>> for letter in l:
...     print letter,
...
a b c
```

16.3 classmethod and staticmethod

A class method in Python is defined by creating a method on a class in the standard way, but applying the `classmethod` decorator to the method.

Notice in the following example that instead of `self`, the class method's first argument is named `cls`. This convention is used to clearly denote the fact that in a class method, the first argument received is not a bound instance of the class, but the class object itself.

As a result, class methods are useful when there may not be an existing object of the class type, but the type of the class is important. This example shows a "factory" method, that creates `Song` objects based on a list of tuples.

Also notice the use of the `__str__()` special method in this example. This method returns a string representation of the object when the object is passed to `print` or the `str()` builtin.

```
>>> class Song(object):
...     def __init__(self, title, artist):
...         self.title = title
...         self.artist = artist
...
...     def __str__(self):
...         return ("%s" % self.title) + " by " + ("%s" % self.artist)
...
...     @classmethod
...     def create_songs(cls, songlist):
...         for artist, title in songlist:
```

```
...         yield cls(title, artist)
...
>>> songs = (('Glen Hansard', 'Leave'),
...          ('Stevie Ray Vaughan', 'Lenny'))

>>> for song in Song.create_songs(songs):
...     print song
...
"Leave" by Glen Hansard
"Lenny" by Stevie Ray Vaughan
```

Static methods are very similar to class methods and defined using a similar decorator. The important difference is that static methods receive neither an instance object nor a class object as the first argument. They only receive the passed arguments.

As a result, the only real value in defining static methods is code organization. But in many cases a module-level function would do the same job with fewer dots in each call.

```
>>> class Song(object):
...     def __init__(self, title, artist):
...         self.title = title
...         self.artist = artist
...
...     def __str__(self):
...         return ("%s" % self.title) + " by " + ("%s" % self.artist)
...
...     @staticmethod
...     def create_songs(songlist):
...         for artist, title in songlist:
...             yield Song(title, artist)
...
>>> songs = (('Glen Hansard', 'Leave'),
...          ('Stevie Ray Vaughan', 'Lenny'))

>>> for song in Song.create_songs(songs):
...     print song
...
"Leave" by Glen Hansard
"Lenny" by Stevie Ray Vaughan
```

16.4 Lab

oop-1-parking.py

```
'''
>>> # Create a parking lot with 2 parking spaces
>>> lot = ParkingLot(2)
'''

'''
>>> # Create a car and park it in the lot
>>> car = Car('Audi', 'R8', '2010')
```

```
>>> lot.park(car)
>>> car = Car('VW', 'Vanagon', '1981')
>>> lot.park(car)
>>> car = Car('Buick', 'Regal', '1988')
>>> lot.park(car)
'Lot Full'
>>> lot.spaces = 3
>>> lot.park(car)
>>> car.make
'Buick'
>>> car.model
'Regal'
>>> car.year
'1988'
>>> for c in lot:
...     print c
2010 Audi R8
1981 VW Vanagon
1988 Buick Regal
>>> for c in lot.cars_by_age():
...     print c
1981 VW Vanagon
1988 Buick Regal
2010 Audi R8
>>> for c in lot:
...     print c
2010 Audi R8
1981 VW Vanagon
1988 Buick Regal
'''

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

16.5 Inheritance

As noted in the first class definition example above, a class defines a superclass using the parentheses list in the class definition. The model for overloading methods is very similar to most other languages: define a method in the child class with the same name as that in the parent class and it will be used instead.

oop-1-inheritance.py

```
class Instrument(object):
    def __init__(self, name):
        self.name = name
    def has_strings(self):
        return True

class PercussionInstrument(Instrument):
    def has_strings(self):
        return False
```

```
guitar = Instrument('guitar')
drums = PercussionInstrument('drums')

print 'Guitar has strings: {0}'.format(guitar.has_strings())
print 'Guitar name: {0}'.format(guitar.name)
print 'Drums have strings: {0}'.format(drums.has_strings())
print 'Drums name: {0}'.format(drums.name)
```

```
$ python oop-1-inheritance.py
Guitar has strings: True
Guitar name: guitar
Drums have strings: False
Drums name: drums
```

Calling Superclass Methods Python has a `super()` builtin function instead of a keyword and it makes for slightly clunky syntax. The result, however, is as desired, which is the ability to execute a method on a parent or superclass in the body of the overloading method on the child or subclass.

In this example, an overloaded `__init__()` is used to hard-code the known values for every guitar, saving typing on every instance.

oop-2-super.py

```
class Instrument(object):
    def __init__(self, name):
        self.name = name
    def has_strings(self):
        return True

class StringInstrument(Instrument):
    def __init__(self, name, count):
        super(StringInstrument, self).__init__(name)
        self.count = count

class Guitar(StringInstrument):
    def __init__(self):
        super(Guitar, self).__init__('guitar', 6)

guitar = Guitar()

print 'Guitar name: {0}'.format(guitar.name)
print 'Guitar count: {0}'.format(guitar.count)
```

```
python oop-2-super.py
Guitar name: guitar
Guitar count: 6
```

There is an alternate form for calling methods of the superclass by calling them against the unbound class method and explicitly passing the object as the first parameter. Here is the same example using the direct calling method.

oop-3-super-alt.py

```
class Instrument(object):
    def __init__(self, name):
        self.name = name
    def has_strings(self):
```

```
        return True

class StringInstrument(Instrument):
    def __init__(self, name, count):
        Instrument.__init__(self, name)
        self.count = count

class Guitar(StringInstrument):
    def __init__(self):
        StringInstrument.__init__(self, 'guitar', 6)

guitar = Guitar()

print 'Guitar name: {0}'.format(guitar.name)
print 'Guitar count: {0}'.format(guitar.count)
```

```
python oop-3-super-alt.py
Guitar name: guitar
Guitar count: 6
```

Multiple Inheritance Python supports multiple inheritance using the same definition format as single inheritance. Just provide an ordered list of superclasses to the class definition. The order of superclasses provided can affect method resolution in the case of conflicts, so don't treat it lightly.

The next example shows the use of multiple inheritance to add some functionality to a class that might be useful in many different kinds of classes.

oop-4-multiple.py

```
class Instrument(object):
    def __init__(self, name):
        self.name = name
    def has_strings(self):
        return True

class Analyzable(object):
    def analyze(self):
        print 'I am a {0}'.format(self.__class__.__name__)

class Flute(Instrument, Analyzable):
    def has_strings(self):
        return False

flute = Flute('flute')
flute.analyze()
```

```
$ python oop-4-multiple.py
I am a Flute
```

Abstract Base Classes Python recently added support for abstract base classes. Because it is a more recent addition, its implementation is based on existing capabilities in the language rather than a new set of keywords. To create an abstract base class, override the metaclass in your class definition (metaclasses in general are beyond the scope of this course, but

they define how a class is created). Then, apply the `abstractmethod` decorator to each abstract method. Note that both `ABCMeta` and `abstractmethod` need to be imported.

Here is a simple example. Notice that the base class cannot be instantiated, because it is incomplete.

oop-5-abc.py

```
from abc import ABCMeta, abstractmethod
import sys
import traceback

class Instrument(object):
    __metaclass__ = ABCMeta
    def __init__(self, name):
        self.name = name
    @abstractmethod
    def has_strings(self):
        pass

class StringInstrument(Instrument):
    def has_strings(self):
        return True

guitar = StringInstrument('guitar')
print 'Guitar has strings: {0}'.format(guitar.has_strings())
try:
    guitar = Instrument('guitar')
except:
    traceback.print_exc(file=sys.stdout)
```

```
$ python oop-5-abc.py
Guitar has strings: True
Traceback (most recent call last):
  File "samples/oop-5-abc.py", line 22, in <module>
    guitar = Instrument('guitar')
TypeError: Can't instantiate abstract class Instrument with abstract methods
    has_strings
```

One feature of abstract methods in Python that differs from some other languages is the ability to create a method body for an abstract method. This feature allows common, if incomplete, functionality to be shared between multiple subclasses. The abstract method body is executed using the `super()` method in the subclass.

oop-6-abcbody.py

```
from abc import ABCMeta, abstractmethod

class Instrument(object):
    __metaclass__ = ABCMeta

    def __init__(self, name):
        self.name = name

    @abstractmethod
    def has_strings(self):
        print 'checking for strings in %s' % \
            self.name
```

```
class StringInstrument(Instrument):  
    def has_strings(self):  
        super(StringInstrument,  
              self).has_strings()  
        return True  
  
guitar = StringInstrument('guitar')  
print 'Guitar has strings: {0}'.format(guitar.has_strings())
```

```
$ python oop-6-abcbody.py  
checking for strings in guitar  
Guitar has strings: True
```

16.6 Lab

oop-2-pets.py

```
'''  
>>> cat = Cat('Spike')  
>>> cat.speak()  
'Spike says "Meow"  
>>> dog = Dog('Bowzer')  
>>> cat.can_swim()  
False  
>>> dog.can_swim()  
True  
>>> dog.speak()  
'Bowzer says "Woof"  
>>> fish = Fish('Goldie')  
>>> fish.speak()  
"Goldie can't speak"  
>>> fish.can_swim()  
True  
>>> generic = Pet('Bob')  
Traceback (most recent call last):  
...  
TypeError: Can't instantiate abstract class Pet with abstract methods can_swim  
'''  
  
if __name__ == '__main__':  
    import doctest  
    doctest.testmod()
```

16.7 Encapsulation

As mentioned previously, while Python does not support declarations of attribute visibility (public, private, etc), it does provide mechanisms for encapsulation of object attributes. There are three approaches with different levels of capability that can be used for this purpose.

16.7.1 Intercepting Attribute Access

When an attribute of an object is accessed using dot-notation, there are three special methods of the object that may get called along the way.

For lookups, two separate methods are called: `__getattribute__(self, name)` is called first, passing the name of the attribute that is being requested. Overriding this method allows for the interception of requests for any attribute. By contrast, `__getattr__()` is only called when `__getattribute__()` fails to return a value. So this method is useful if only handling undefined cases.

For setting attributes only one method, `__setattr__(self, name, value)` is called. Note that inside this method body, calling `self.name = value` will lead to infinite recursion. Use the superclass object `object.__setattr__(self, name, value)` instead.

The following example shows a course with two attributes `capacity` and `enrolled`. A third attribute `open` is calculated based on the other two. However, setting it is also allowed and forces the `enrolled` attribute to be modified.

oop-7-intercept.py

```
import traceback
import sys

class Course(object):
    def __init__(self, capacity):
        self.capacity = capacity
        self.enrolled = 0

    def enroll(self):
        self.enrolled += 1

    def __getattr__(self, name):
        if name == 'open':
            return self.capacity - self.enrolled
        else:
            raise AttributeError('%s not found', name)

    def __setattr__(self, name, value):
        if name == 'open':
            self.enrolled = self.capacity - value
        else:
            object.__setattr__(self, name, value)

    def __str__(self):
        return 'Enrolled: \t{0}\nCapacity:\t{1}\nOpen:\t{2}'.format(
            self.enrolled, self.capacity, self.open)

course = Course(12)
course.enroll()
course.enroll()

print course

course.open = 8

print course
```

```
$ python oop-7-properties.py
Enrolled:  2
```

```
Capacity: 12
Open: 10
Enrolled: 4
Capacity: 12
Open: 8
```

16.7.2 Properties

For the simple case of defining a calculated property as shown in the above example, the more appropriate (and simpler) model is to use the `property` decorator to define a method as a property of the class.

Here is the same example again using the `property` decorator. Note that this approach does not handle setting this property. Also notice that while `open()` is initially defined as a method, it cannot be accessed as a method.

oop-8-properties.py

```
import traceback
import sys

class Course(object):
    def __init__(self, capacity):
        self.capacity = capacity
        self.enrolled = 0

    def enroll(self):
        self.enrolled += 1

    @property
    def open(self):
        return self.capacity - self.enrolled

course = Course(12)
course.enroll()
course.enroll()

print 'Enrolled: \t{0}\nCapacity:\t{1}\nOpen:\t{2}'.format(course.enrolled,
    course.capacity, course.open)

print
try:
    course.open()
except:
    traceback.print_exc(file=sys.stdout)

print
try:
    course.open = 9
except:
    traceback.print_exc(file=sys.stdout)
```

```
$ python oop-8-properties.py
Enrolled: 2
Capacity: 12
Open: 10
```

```
Traceback (most recent call last):
  File "samples/oop-8-properties.py", line 25, in <module>
    course.open()
TypeError: 'int' object is not callable

Traceback (most recent call last):
  File "samples/oop-8-properties.py", line 31, in <module>
    course.open = 9
AttributeError: can't set attribute
```

Using the property mechanism, setters can also be defined. A second decorator is dynamically created as `<attribute name>.setter` which must be applied to a method with the **exact same name** but an additional argument (the value to be set).

In this example, we use this additional functionality to encapsulate the speed of a car and enforce a cap based on the type of car being manipulated.

oop-9-propertysetters.py

```
class Car(object):
    def __init__(self, name, maxspeed):
        self.name = name
        self.maxspeed = maxspeed
        self.__speed = 0

    @property
    def speed(self):
        return self.__speed

    @speed.setter
    def speed(self, value):
        s = int(value)
        s = max(0, s)
        self.__speed = min(self.maxspeed, s)

car = Car('Lada', 32)
car.speed = 100
print 'My {name} is going {speed} mph!'.format(name=car.name, speed=car.speed)

car.speed = 24
print 'My {name} is going {speed} mph!'.format(name=car.name, speed=car.speed)
```

```
$ python oop-9-propertysetters.py
My Lada is going 32 mph!
My Lada is going 24 mph!
```

16.7.3 Descriptors

The final mechanism for encapsulating the attributes of a class uses descriptors. A descriptor is itself a class and it defines the type of an attribute. When using a descriptor, the attribute is actually declared at the class level (not in the initializer) because it is adopting some type information that must be preserved.

The descriptor class has a simple protocol with the methods `__get__()`, `__set__()` and `__delete__()` being called on attribute access and manipulation. Notice that the descriptor does not store instance attributes on itself, but

rather on the instance. The descriptor is only instantiated once at class definition time, so any values stored on the descriptor object will be common to all instances.

The following example tackles the speed-limit problem using descriptors.

oop-10-descriptors.py

```
class BoundsCheckingSpeed(object):
    def __init__(self, maxspeed):
        self.maxspeed = maxspeed

    def __get__(self, instance, cls):
        return instance._speed

    def __set__(self, instance, value):
        s = int(value)
        s = max(0, s)
        instance._speed = min(self.maxspeed, s)

class Animal(object):
    speed = BoundsCheckingSpeed(0)

    def __init__(self, name):
        self.name = name

    @property
    def speed_description(self):
        return '{name} the {type} is going {speed} mph!'.format(name=self.name,
                                                                type=self.__class__.__name__.lower(), speed=self.speed)

class Squirrel(Animal):
    speed = BoundsCheckingSpeed(12)

class Cheetah(Animal):
    speed = BoundsCheckingSpeed(70)

squirrel = Squirrel('Jimmy')
squirrel.speed = 20
print squirrel.speed_description

squirrel.speed = 10
print squirrel.speed_description

cheetah = Cheetah('Fred')
cheetah.speed = 100
print cheetah.speed_description
```

```
$ python oop-10-descriptors.py
Jimmy the squirrel is going 12 mph!
Jimmy the squirrel is going 10 mph!
Fred the cheetah is going 70 mph!
```

Tip

Notice that values of the descriptor can be set for all instances of a certain class, while being different in different uses. The descriptor allows for the creation of a generic field "type" that can be shared in a configurable fashion across unrelated classes. It is more complex to use than properties, but can provide more flexibility in a complex object hierarchy.

16.8 Lab

oop-3-portfolio.py

```
'''
>>> p = Portfolio()
>>> stocks = (('APPL', 1000, 251.80, 252.73),
...           ('CSCO', 5000, 23.09, 23.74),
...           ('GOOG', 500, 489.23, 491.34),
...           ('MSFT', 2000, 24.63, 25.44))
...
>>> for stock in stocks:
...     p.add(Investment(*stock))
>>> print p['APPL']
1000 shares of APPL worth 252730.00
>>> p['GOOG'].quantity
500
>>> p['GOOG'].close
491.33999999999997
>>> p['GOOG'].open
489.23000000000002
>>> for stock in p:
...     print stock
1000 shares of APPL worth 252730.00
5000 shares of CSCO worth 118700.00
500 shares of GOOG worth 245670.00
2000 shares of MSFT worth 50880.00
>>> for stock in p.sorted('open'):
...     print stock.name
CSCO
MSFT
APPL
GOOG
>>> p['MSFT'].gain
0.81000000000000227
>>> p['CSCO'].total_gain
3249.9999999999927
>>> 'GOOG' in p
True
>>> 'YHOO' in p
False
'''

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

Chapter 17

Easter Eggs

```
>>> from __future__ import braces
>>> import this
```
