Python

To start Python from the shell, simply type `python`

A second way of starting the interpreter is `python -c command [arg] …`

Python modules can be invoked with `python -m module [arg] ...`

To run a script file and then go into the interactive mode, use the `-i` option

Quitting

To quit Python, use `quit()` or `Ctl-d`.

Interactive Mode

When you type `python` into the terminal, you go into interactive mode, where you can type in code and have it execute in front of you.

`>>>` is the primary prompt, and means you need to type something

`...` is the secondary prompt, and means you are in a multi-line command.

A secondary prompt on a line by itself in an example means you must type a blank line; this is used to end a multi-line command.

Executable Scripts

On BSD'ish Unix systems, Python scripts can be made directly executable, like shell scripts, by putting the line

```
#! /usr/bin/env python
```

at the beginning of the script and giving the file an executable mode. The `#!`

must be the first two characters of the file. On some platforms, this first line must end with a Unix-style line ending `('\n')`

The script can be given an executable mode, or permission, using the chmod command:

```
chmod +x myscript.py
```

Comments

The hash, or pound, character, `#`, is used to start a comment in Python.

Special Encoding

Place `# -*- coding: encoding -*-` after the `#!` line (above) to instruct Python to use special encoding.

With that declaration, all characters in the source file will be treated as having the encoding.

E.g.

```
# -*- coding: iso-8859-15 -*-

currency = u"€"
print ord(currency)
```

By using `UTF-8` (either through the signature or an encoding declaration), characters of most languages in the world can be used simultaneously in string literals and comments.

Customizing Python

By setting an environment variable named `PYTHONSTARTUP` to the name of a file containing your start-up commands, you can customize the interactive mode of Python. This is similar to the `.profile` feature of the Unix shells.

To customize every invocation of Python, there are two hooks that you can use to perform global customizations: `sitecustomize` and `usercustomize`.

Read up about it in `2.2.5` for more information.

Using Python as a Calculator

The interpreter acts as a simple calculator: you can type an expression at it and it will write the value.

```
python
>>>2 + 2
4
```

Initializing Variables

The equal sign `=` is used to assign a value to a variable.

```
>>> ben = 33
>>> mari = 24
>>> ben - mari
9
```

A value can be assigned to several variables simultaneously:

```
>>> x = y = z = 0  # Zero x, y and z
```

Multiple variables can be initialized on the same line with different values:

```
a, b = 0, 1
```

To print out the value of a variable, simply type its name

```
>>>ben
33
```

Converting a number to a float, int, or long

```
>>> a = 1
>>> b = 2
>>> c = 3
>>> float(a)
1.0
>>> int(b)
2
>>> long(c)
3L
```

The `_` Variable

In interactive mode, the last printed expression is assigned to the variable `_`. Especially useful when using Python as a calculator.

```
>>> ben + _
36L
```

This variable should be treated as read-only by the user. Don't explicitly assign a value to it.

Strings

Strings can be enclosed in single or double quotes. You must escape any special characters.

```
>>> "hello there"
'hello there'
>>> 'hello there'
'hello there'
>>> "'hello', he said"
"'hello', he said"
>>> 'don\'t do that!'
"don't do that!"
```

String literals can span multiple lines in several ways. Continuation lines can be used, with a backslash as the last character on the line indicating that the next line is a logical continuation of the line:

```
>>> question = "What are you doing today?\n\
... What time is it?\n\
... How do you feel?"
>>> print question
What are you doing today?
What time is it?
How do you feel?
```

Or, strings can be surrounded in a pair of matching triple-quotes: `"""` or `'''`. End of lines do not need to be escaped when using triple-quotes, but they will be included in the string.

```
>>> print """
... Hello there
...     Mr X.
...     Your hat is on crooked
... """

Hello there
    Mr X.
    Your hat is on crooked
```

To make a string where special characters are printed as is (i.e. a `raw` string), use:

```
>>> greeting = r"Hello there you\n\
... interesting person."
>>> print greeting
Hello there you\n\
interesting person.
```

Strings can be concatenated (glued together) with the `+` operator, and repeated with `*`:

```
>>> bad = 'Hep' + 'A'
>>> bad
'HepA'
>>> 'Don\'t get' + bad*5 + ' ever.'
"Don't getHepAHepAHepAHepAHepA ever."
```

Two string literals next to each other are automatically concatenated; the first line above could also have been written `bad = 'Hep' 'A';`

Strings can be subscripted (indexed); like in C, the first character of a string has subscript (index) `0`.

Substrings can be specified with the slice notation: two indices separated by a colon.

```
>>> bad[3]
'A'
>>> bad[0:2]
'He'
>>> bad[2:4]
'pA'
```

With slices, an omitted first index defaults to zero, an omitted second index defaults to the size of the string being sliced.

Python strings cannot be changed. Assigning to an indexed position in the string results in an error:

```
bad[2] = 'M'        #Error!
```

But you can create a new string by concatenating input with an old string:

```
>>> 'M' + bad
'MHepC'
>>> bad[3] + 'd'
'Cd'
>>> bad[2:] + bad[2:]
'pCpC'
```

Indicies can be negative numbers

```
>>> word[-1]     # The last character
'A'
>>> word[-2]     # The last-but-one character
'p'
>>> word[-2:]    # The last two characters
'pA'
>>> word[:-2]    # Everything except the last two characters
'Hel'
```

The built-in function `len()` returns the length of a string:

```
>>> meep = 'meeeeeeeeeeeeeeeeeeeeeeeeeeeeep'
>>> len(meep)
31
```

String methods - `http://docs.python.org/2/library/stdtypes.html#string-methods`

The Unicode Object

Starting with Python 2.0 a new data type for storing text data is available to the programmer: the Unicode object. It can be used to store and manipulate Unicode data.

Creating Unicode strings in Python is just as simple as creating normal strings:

```
>>> u'Hello World !'
u'Hello World !'
```

The small `u` in front of the quote indicates that a Unicode string is supposed to be created.

If you want to include special characters in the string, you can do so by using the Python Unicode-Escape encoding. The following example shows how:

```
>>> u'Hello\u0020World !'
u'Hello World !'
```

The built-in function `unicode()` provides access to all registered Unicode codecs (COders and DECoders). Some of the more well known encodings which these codecs can convert are Latin-1, ASCII, UTF-8, and UTF-16.

When a Unicode string is printed, written to a file, or converted with `str()`, conversion takes place using this default encoding.

```
>>> u"abc"
u'abc'
>>> str(u"abc")
'abc'
```

To convert a Unicode string into an 8-bit string using a specific encoding, Unicode objects provide an `encode()` method that takes one argument, the name of the encoding. Lowercase names for encodings are preferred.

```
>>> u"äöü".encode('utf-8')
'\xc3\xa4\xc3\xb6\xc3\xbc'
```

Lists

Lists are a data type that is used to group together other values. Lists can be written as a list of comma-separated values (items) between square brackets. List items need not all have the same type.

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
```

Like string indices, list indices start at 0, and lists can be sliced, concatenated and so on:

```
>>> a[0]
'spam'
>>> a[3]
1234
>>> a[-2]
100
>>> a[1:-1]
['eggs', 100]
```

All slice operations return a new list containing the requested elements. This means that the following slice returns a shallow copy of the list a:

```
>>> a[:]
['spam', 'eggs', 100, 1234]
```

Unlike strings, which are immutable, it is possible to change individual elements of a list.

Assignment to slices is also possible, and this can even change the size of the list or clear it entirely:

```
>>> # Replace some items:
... a[0:2] = [1, 12]
>>> a
[1, 12, 123, 1234]
>>> # Remove some:
... a[0:2] = []
[123, 1234]
```

The built-in function len() also applies to lists:

```
len(a)
```

The `WHILE` Loop

```
>>> while b < 10:
...     print b
...     a, b = b, a+b
```

The body of the loop is indented: indentation is Python's way of grouping statements. At the interactive prompt, you have to type a tab or space(s) for each indented line.

Each line within a basic block must be indented by the same amount.

When a compound statement is entered interactively, it must be followed by a blank line to indicate completion.

```
>>> i = 256*256
>>> print 'The value of i is', i
The value of i is 65536
```

A trailing comma avoids the newline after the output:

Note that the interpreter inserts a newline before it prints the next prompt if the last line was not completed.

`IF` Statements

```
>>> x = int(raw_input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...      x = 0
...      print 'Negative changed to zero'
... elif x == 0:
...      print 'Zero'
... else:
...      print 'More'
...
More
```

There can be zero or more `elif` parts, and the `else` part is optional.

`FOR` Statements

The for statement in Python differs a bit from what you may be used to in `C` Rather than always iterating over an arithmetic progression of numbers (like in

Pascal), or giving the user the ability to define both the iteration step and halting condition (as C), Python's for statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. E.g:

```
>>> # Measure some strings:
... words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print w, len(w)
...
cat 3
window 6
defenestrate 12
```

If you need to modify the sequence you are iterating over while inside the loop (for example to duplicate selected items), it is recommended that you first make a copy. Iterating over a sequence does not implicitly make a copy. The slice notation makes this especially convenient:

```
>>> for w in words[:]:  # Loop over a slice copy of the entire list.
...     if len(w) > 6:
...         words.insert(0, w)
...
>>> words
['defenestrate', 'cat', 'window', 'defenestrate']
```

The `RANGE` Statement

If you do need to iterate over a sequence of numbers, the built-in function `range()` comes in handy. It generates lists containing arithmetic progressions:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The given end point is never part of the generated list; range(10) generates a list of 10 values, the legal indices for items of a sequence of length 10. It is possible to let the range start at another number, or to specify a different

increment (even negative; sometimes this is called the ‘step’):

```
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(-10, -100, -30)
[-10, -40, -70]
```

To iterate over the indices of a sequence, you can combine range() and len() as follows:

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print i, a[i]
...
0 Mary
1 had
2 a
3 little
4 lamb
```

4.4 `break` and `continue` Statements, and else

The break statement, like in C, breaks out of the smallest enclosing for or while loop.

Loop statements may have an else clause; it is executed when the loop terminates through exhaustion of the list.

A `try` statement’s else clause runs when no exception occurs, and a loop’s else clause runs when no break occurs. E.g.

```
#! /usr/bin/python
names = ['b', 'e', 'n', 'n', 'y']
for i in range(len(names)):
    print i, names[i]
    if i == 'n':
        break
else:
    print "End of the name"
```

The continue statement, also borrowed from C, continues with the next iteration of the loop:

The `PASS` Statement

The pass statement does nothing. It can be used as a placeholder.

```
>>> while True:
...     pass # Busy-wait
```

or

```
>>> class MyEmptyClass:
...     pass
```

The IN Keyword

The `in` keyword tests whether or not a sequence contains a certain value.

Defining Functions

To define a function:

```
def speakLoud(input):
```

The keyword def introduces a function definition. It must be followed by the function name and the parenthesized list of formal parameters. The statements

that form the body of the function start at the next line, and must be indented.

The first statement of the function body can optionally be a string literal; this string literal is the function's documentation string, or docstring.

There are tools which use docstrings to automatically produce online or printed documentation, or to let the user interactively browse through code; it's good practice to include docstrings in code that you write

It is also possible to define functions with a variable number of arguments. There are three forms, which can be combined.

Default Argument Values

The most useful form is to specify a default value for one or more arguments. This creates a function that can be called with fewer arguments than it is defined to allow. For example:

```
def ask_ok(prompt, retries=4, complaint='Yes or no, please!'):
```

Functions with a Variable Number of Arguments

There are 3 ways to define multiple arguments for a function:

1. Have optional arguments in a function definition, which you can choose to leave out
2. Using argument dictionaries (`*` and `**`)
3. Something weird that I don't understand

1. Have optional arguments in a function definition, which you can choose to leave out

```
def shout(times, message="Chug!", volume=100)
```

With this type of function call, you MUST declare `times` when you call the

function, but since the other 2 are already set, you don't have to declare them. You can declare them as something else, e.g. `message="Stop!"`.

2. Using argument dictionaries (`*` and `**`)

Basically, in a function declaration, `*` stands for any regular argument, and you can make as many of them as you want.

`**` stands for function dictionaries, which means it has to follow the format `name=value` (as below). In the code example, you can find how to loop through either.

```
#! /usr/bin/python

def shopping_list(item, *questions, **actors):
    print "I need some", item
    print "Sure, we've got a nice big bunch of", item
    for q in questions:
        print q
    print "*" * 50
    print "/" * 50
    acts = sorted(actors.keys())
    for a in acts:
        print a, ":", actors[a]

shopping_list("eggs",
              "Are they fresh?",
              "They don't smell so fresh...",
              customer='Natalie Portman',
              vendor='Musty Eggs',
              attendant='Ron Jeremy')
```

3. (What I don't understand) Arbitrary Argument Lists

Seriosly, I don't get it…

```
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

Unpacking Argument Lists

The reverse situation occurs when the arguments are already in a list or tuple but need to be unpacked for a function call. E.g.

```
>>> range(3, 6)
[3, 4, 5]               #result
>>> args = [3, 6]
>>> range(*args)
[3, 4, 5]               #result
```

In the same fashion, dictionaries can deliver keyword arguments with the `**` operator:

```
def shout(times, message="You're stinky", volume=100):
print message * times
print "at a volume of", volume

args = {"times": 5, "message": "Please take a shower sir \n", "volume
": "50"}
shout(**args)
```

Lambda Forms

With the `lambda` keyword, small anonymous functions can be created. Lambda forms can be used wherever function objects are required. Lambda forms can reference variables from the containing scope. E.g.

```
>>> def inc(n):
...     return lambda x: x + n
...
>>> f = inc(42)
>>> f(3)
```

Lists

```
list.append(x)
```

Add an item to the end of the list; equivalent to `a[len(a):] = [x]`.

```
list.extend(L)
```

Extend the list by appending all the items in the given list; equivalent to `a[len(a):] = L`.

```
list.insert(i, x)
```

Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.

```
list.remove(x)
```

Remove the first item from the list whose value is x. It is an error if there is no such item.

```
list.pop([i])
```

Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. (The square brackets around the `i` in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)

```
list.index(x)
```

Return the index in the list of the first item whose value is x. It is an error if there is no such item.

```
list.count(x)
```

Return the number of times x appears in the list.

```
list.sort()
```

Sort the items of the list, in place.

```
list.reverse()
```

Reverse the elements of the list, in place.

Using Lists as Stacks

The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved (“last-in, first-out”). To add an item to the top of the stack, use `append()`. To retrieve an item from the top of the stack, use `pop()` without an explicit index.

Using Lists as Queues

It is also possible to use a list as a queue, where the first element added is the first element retrieved (“first-in, first- out”); however, lists are not efficient for this purpose. While appends and pops from the end of list are fast, doing inserts or pops from the beginning of a list is slow (because all of the other elements have to be shifted by one).

To implement a queue, use `collections.deque` which was designed to have fast appends and pops from both ends. For example:

```
from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")
>>> queue.append("Graham")
>>> queue.popleft()
```

Built in List Functions

There are three built-in functions that are very useful when used with lists: `filter()`, `map()`, and `reduce()`.

`filter(function, sequence)` returns a sequence consisting of those items from the sequence for which `function(item)` is true. If sequence is a string or tuple, the result will be of the same type; otherwise, it is always a list. E.g.

```
>>> list = range(2, 20)
>>> list
[2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> def func(x):
...     if x % 2 == 0:
...             return x
...
>>> filter(func, list)
[2, 4, 6, 8, 10, 12, 14, 16, 18]
```

`map(function, sequence)` calls `function(item)` for each of the sequence's items and returns a list of the return values. E.g.

```
>>> def add_two(y): return y + 2
...
>>> map(add_two, list)
[4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38]
```

More than one sequence may be passed; the function must then have as many arguments as there are sequences and is called with the corresponding item from each sequence (or None if some sequence is shorter than another).

`reduce(function, sequence)` returns a single value constructed by calling the binary function `function` on the first two items of the sequence, then on the result and the next item, and so on.

```
>>> def add(x,y): return x+y
...
>>> reduce(add, range(1, 11)) 55
```

If there's only one item in the sequence, its value is returned; if the sequence is empty, an exception is raised.

A third argument can be passed to indicate the starting value. In this case the starting value is returned for an empty sequence, and the function is first applied to the starting value and the first sequence item, then to the result and the next item, and so on.

The `**` Operator

Apparently, the `**` operator means "to the power of", so `3**2` means 3 to the power of 2 (or 3 squared), and will equal 9.

List Comprehensions

A list comprehension consists of brackets containing an expression followed by a for clause, then zero or more for or if clauses. The result will be a new list resulting from evaluating the expression in the context of the for and if clauses which follow it.

```
>>> squares = []
>>> for x in range(10):
... squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

We can obtain the same result with:

```
squares = [x**2 for x in range(10)]
```

My explanation: Basically, you first say what you want to do with items in the list (`x-1` in the e.g. below), then you have a `for` loop which tells the compiler how to work through the list. You can also use an `if` to help decide if you want to apply the expression to the list item.

```
>>> nums = [3, 7, 9, 11, 13, 15]
>>> [x-1 for x in nums]
[2, 6, 8, 10, 12, 14]

>>> [x-1 for x in nums if x > 9]
[10, 12, 14]
```

List comprehensions can contain complex expressions and nested functions. E.g.

```
>>> from math import pi
>>> [str(round(pi, i)) for i in range(1, 6)]
```

The initial expression in a list comprehension can be any expression, including another list comprehension. E.g.

```
>>> [[row[i] for row in matrix] for i in range(4)]
```

The `del` statement

The `del` statement removes an item from a list given its index instead of its value. This differs from the pop() method which returns a value. The del statement can also be used to remove slices from a list or clear the entire list.

```
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> del l[3]
>>> l
[0, 1, 2, 4, 5, 6, 7, 8, 9]
>>> del l[2:4]
>>> l
[0, 1, 5, 6, 7, 8, 9]
>>> del l[:]
>>> l
>>> []
```

`del` can also be used to delete entire variables:

```
del l
```

Referencing the `l` after this returns an error unless you declare it with somethign else.

The TUPLE Data Type

Lists and Strings are both of data type "sequence". Tuples are also of type "sequence", although Tuples have different rules to the others.

A tuple consists of a number of values separated by commas, for instance:

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Tuples are immutable:
... t[0] = 88888
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # but they can contain mutable objects:
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

Rules:

1. Tuples can be nested
2. Tuples are immutable
3. Tuples may contain mutable objects

On output tuples are always enclosed in parentheses, so that nested tuples are interpreted correctly; they may be input with or without surrounding

parentheses, although often parentheses are necessary anyway (if the tuple is part of a larger expression).

It is not possible to assign to the individual items of a tuple, however it is possible to create tuples which contain mutable objects, such as lists. You can also simply overwrite a tuple.

Empty tuples are constructed by an empty pair of parentheses; a tuple with one item is constructed by following a value with a comma (it is not sufficient to enclose a single value in parentheses).

```
>>> empty = ()
>>> singleton = 'hello',
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

The statement `t = 12345, 54321,'hello!'` is an example of tuple packing. The values `12345`, `54321` and `hello!` are packed together in a tuple. The reverse operation is also possible:

```
>>> x, y, z = t
```

This is called, appropriately enough, sequence unpacking and works for any sequence on the right-hand side. Sequence unpacking requires the list of variables on the left to have the same number of elements as the length of the sequence.

The SET Datatype

A `set` is an unordered collection with no duplicate elements. `set` objects support mathematical operations like union, intersection, difference, and symmetric difference.

Curly braces or the `set()` function can be used to create sets. Note: to create an empty set you have to use `set()`, not `{}`.

```
>>> order = ['jim', 'max', 'peter', 'jim', 'mark']
>>> roster = set(order)
>>> roster
set(['max', 'jim', 'peter', 'mark'])
>>> 'jim' in roster
True
>>> 'lydon' in roster
False
```

Here is a list of `set` operations:

```
>>> a = set('abcd')
>>> b = set('cdef')
>>> a
set(['a', 'c', 'b', 'd'])
>>> b
set(['c', 'e', 'd', 'f'])
>>> a - b                  #letters that are in a but not b
set(['a', 'b'])
>>> a | b                  #letters that are in either a or b
set(['a', 'c', 'b', 'e', 'd', 'f'])
>>> a & b                  #only letters that are in both a and b
set(['c', 'd'])
>>> a ^ b                  #only letters that are NOT in both a and
b
set(['a', 'b', 'e', 'f'])
```

The DICTIONARY Datatype

Dictionaries are indexed by keys, which can be any immutable type; strings and numbers can always be keys. Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key. You can’t use lists as keys.

Dictionaries are called "associative arrays" in other languages.

Keys must be unique.

A pair of braces creates an empty dictionary: `{}`.

It is possible to delete a key:value pair with `del`.

If you store using a key that is already in use, the old value associated with that key is forgotten.

The `keys()` method of a dictionary object returns a list of all the keys used in the dictionary, in arbitrary order (if you want it sorted, just apply the `sorted()` function to it).

To check whether a single key is in the dictionary, use the `in` keyword.

```
>>> postcodes = {}
>>> postcodes
{}
>>> postcodes['table mesa'] = 80303
>>> postcodes
{'table mesa': 80303}
>>> postcodes['boulder'] = 80302
>>> boulder in postcodes
>>> 'boulder' in postcodes
True
```

The `dict()` constructor builds dictionaries directly from sequences of key-value pairs:

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
>>> dict(sape=4139, guido=4127, jack=4098)
```

In addition, dict comprehensions can be used to create dictionaries from arbitrary key and value expressions:

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}            #result
```

Looping Techniques

When looping through a sequence, the position index and corresponding value can be retrieved at the same time using the `enumerate()` function.

```
>>> for x, y in enumerate(['kafka', 'fritz', 'nichze']):
...     print x, y
...
0 kafka
1 fritz
2 nichze
```

To loop over two or more sequences at the same time, the entries can be paired with the zip() function.

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
... print 'What is your {0}? It is {1}.'.format(q, a)
```

To loop over a sequence in reverse, first specify the sequence in a forward direction and then call the `reversed()` function.

```
>>> for x in reversed(range(3)):
...     print x
...
2
1
0
```

To loop over a sequence in sorted order, use the `sorted()` function which returns a new sorted list while leaving the source unaltered.

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
 >>> for f     in sorted(set(basket)):
... print f
...
```

When looping through dictionaries, the key and corresponding value can be

retrieved at the same time using the `iteritems()` method.

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.iteritems():
... print k, v
```

To change a sequence you are iterating over while inside the loop (for example to duplicate certain items), it is recommended that you first make a copy. Looping over a sequence does not implicitly make a copy. The slice notation makes this especially convenient:

```
>>> words = ['cat', 'window', 'defenestrate']
>>> for w in words[:]:          # Loop over a slice copy of the entir
e list.
...     if len(w) > 6:
...         words.insert(0, w)
...
>>> words
['defenestrate', 'cat', 'window', 'defenestrate']
```

More on Conditions

The conditions used in while and if statements can contain any operators, not just comparisons.

Permitted Statements:

```
in
not in              #for sequences
is
is not              #compare whether 2 objects are the same object (e
.g. lists)
```

Comparisons may be combined using the Boolean operators `and`, `or`, and `not`. As always, parentheses can be used to express the desired composition.

Note that in Python, unlike C, assignment cannot occur inside expressions. C

programmers may grumble about this, but it avoids a common class of problems encountered in C programs: typing `=` in an expression when `==` was intended.

Comparing Sequences and Other Types

Sequence objects may be compared to other objects with the same sequence type. First the first two items are compared, and if they differ this determines the outcome of the comparison; if they are equal, the next two items are compared, and so on, until either sequence is exhausted.

If all items of two sequences compare equal, the sequences are considered equal.

Note that comparing objects of different types is legal. The types are ordered by their name.

Thus, a list is always smaller than a string, a string is always smaller than a tuple, etc.

Modules

Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a module; definitions from a module can be imported into other modules or into the main module.

A module is a file containing Python definitions and statements suffixed with `.py`.

Within a module, the module's name (as a string) is available as the value of the global variable `__name__`.

If you write and save a module as `file.py`, you import it with the `import` command, minus the suffix.

```
import file
```

This does not enter the names of the functions defined in fibo directly in the current symbol table; it only enters the module name 'file' there. Using the module name you can access the functions. E.g.

```
file.add_nums(10)
```

To get the name of the file:

```
>>> file.__name__
'file'
```

You can assign a function to a local name to speed things up.

```
func = file.add_nums
func(100)
```

More on Modules

A module can contain executable statements as well as function definitions. These statements are intended to initialize the module. They are executed only the first time the module name is encountered in an import statement.

You can touch a module's global variables with the same notation as above:

```
file.somevar
```

There is a variant of the import statement that imports names from a module directly into the importing module’s symbol table. For example:

```
>>> from file import add_nums, sub_nums
```

This does not introduce the module name from which the imports are taken in the local symbol table (so in the example, `file` is not defined).

There is even a variant to import all names that a module defines:

```
>>> from file import *
```

This imports all names except those beginning with an underscore `_`.

Note that in general the practice of importing `*` from a module or package is frowned upon, since it often causes poorly readable code. However, it is okay to use it to save typing in interactive sessions.

Executing Modules as Scripts

You can execute a module with:

```
python fibo.py <arguments>
```

The module will execute normally, but with the `__name__` set to `__main__`.

If you want the module to act both as a main script, as well a module that is imported into other programs, you can mess around with:

```
if __name__ == "__main__":
    import crab
    add_num(crab.claw("schwak"))
```

The `dir()` Function

The built-in function `dir()` is used to find out which names a module defines.

```
>>> dir(fibo)
['__name__', 'fib', 'fib2']
```

Without arguments, `dir()` lists the names you have defined currently:

`dir()` does not list the names of built-in functions and variables. If you want a list of those, they are defined in the standard module `__builtin__`:

```
>>> import __builtin__
>>> dir(__builtin__)
```

Packages

Packages are a way of structuring Python's module namespace by using "dotted module names". For example, the module name A.B designates a submodule named B in a package named A. Just like the use of modules saves the authors of different modules from having to worry about each other's global variable names, the use of dotted module names saves the authors of multi-module packages like NumPy or the Python Imaging Library from having to worry about each other's module names.

Suppose you want to design a collection of modules (a "package") called `sound`.

```
sound/                          #Top-level package
    __init__.py                 #Initialize the sound package
    formats/                    #Subpackage for file format conversions
       __init__.py
        wavread.py
        wavwrite.py
        aiffread.py
        aiffwrite.py
        auread.py
        auwrite.py
        ...

    effects/                    #Subpackage for sound effects
        __init__.py
        echo.py
        surround.py
        reverse.py
        ...
```

The `__init__.py` files are required to make Python treat the directories as containing packages; this is done to prevent directories with a common name, such as string, from unintentionally hiding valid modules that occur later on the module search path. In the simplest case, `__init__.py` can just be an empty file, but it can also execute initialization code for the package or set the `__all__` variable, described later.

Users of the package can import individual modules from the package, for example:

```
import sound.effects.echo
```

This loads the submodule sound.effects.echo. It must be referenced with its full name.

```
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

An alternative way of importing the submodule is:

```
from sound.effects import echo
```

This also loads the submodule echo, and makes it available without its package prefix, so it can be used as follows:

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

Yet another variation is to import the desired function or variable directly:

```
from sound.effects.echo import echofilter
```

Again, this loads the submodule echo, but this makes its function `echofilter()` directly available:

```
echofilter(input, output, delay=0.7, atten=4)
```

Note that when using `from package import item` , the item can be

either a submodule (or subpackage) of the package, or some other name defined in the package, like a function, class or variable.

Importing * From a Package

The import statement uses the following convention: if a package's `__init__.py` code defines a list named `__all__`, it is taken to be the list of module names that should be imported when `from package import *` is encountered.

For example, the file `sounds/effects/__init__.py` could contain the following code:

```
__all__ = ["echo", "surround", "reverse"]
```

This would mean that `from sound.effects import *` would import the three named submodules of the sound package.

If `__all__` is not defined, the statement `from sound.effects import *` does not import all submodules from the package `sound.effects` into the current namespace; it only ensures that the package `sound.effects` has been imported (possibly running any initialization code in `__init__.py`) and then imports whatever names are defined in the package. This includes any names defined (and submodules explicitly loaded) by `__init__.py`.

Output Formatting

To convert any value to a string, use the `repr()` or `str()` functions.

The `str()` function is meant to return representations of values which are fairly human-readable, while `repr()` is meant to generate representations which can be read by the interpreter (or will force a SyntaxError if there is no equivalent syntax).

The `str.rjust()` method of string objects right-justifies a string in a field of a given width by padding it with spaces on the left. There are similar methods `str.ljust()` and `str.center()`.

These methods do not write anything, they just return a new string.

```
>>> for x in range(1, 11):
... print repr(x).rjust(2), repr(x*x).rjust(3),
... # Note trailing comma on previous line
... print repr(x*x*x).rjust(4)
```

There is another method, str.zfill(), which pads a numeric string on the left with zeros. It understands about plus and minus signs. E.g.

```
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
```

Basic usage of the `str.format()` method looks like this:

```
>>> print 'We are the {} who say "{}!"'.format('knights', 'Ni')
We are the knights who say "Ni!"
```

The brackets and characters within them (called format fields) are replaced with the objects passed into the `str.format()` method. A number in the brackets refers to the position of the object passed into the `str.format()` method.

```
>>> print '{0} and {1}'.format('spam', 'eggs')
spam and eggs
>>> print '{1} and {0}'.format('spam', 'eggs')
eggs and spam
```

If keyword arguments are used in the `str.format()` method, their values are referred to by using the name of the argument.

```
>>> print 'This {food} is {adjective}.'.format(
... food='spam', adjective='absolutely horrible')
This spam is absolutely horrible.
```

Positional and keyword arguments can be arbitrarily combined:

```
>>> print 'The story of {0}, {1}, and {other}.'.format('Bill', 'Manfr
ed',
... other='Georg')
The story of Bill, Manfred, and Georg.
```

`'!s'` (apply `str()`) and `'!r'` (apply `repr()`) can be used to convert the value before it is formatted.

```
>>> print 'The value of PI is approximately {!r}.'.format(math.pi)
The value of PI is approximately 3.141592653589793.
```

An optional `:` and format specifier can follow the field name. This allows greater control over how the value is formatted. The following example rounds Pi to three places after the decimal.

```
>>> import math
>>> print 'The value of PI is approximately {0:.3f}.'.format(math.pi)
```

Passing an integer after the `:` will cause that field to be a minimum number of characters wide. This is useful for making tables pretty.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
... print '{0:10} ==> {1:10d}'.format(name, phone)
```

If you have a really long format string that you don't want to split up, it would be nice if you could reference the variables to be formatted by name instead of by position. This can be done by simply passing the dict and using square brackets `[]` to access the keys

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print ('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; '
... 'Dcab: {0[Dcab]:d}'.format(table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

This could also be done by passing the table as keyword arguments with the `**` notation.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print 'Jack: {Jack:d}; Sjoerd: {Sjoerd:d};
... Dcab: {Dcab:d}'.format(**table)
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

The `vars()` Method

`vars()` returns a dictionary containing all local variables.

Reading and Writing Files

`open()` returns a file object, and is most commonly used with two arguments: `open(filename, mode)`.

```
>>> f = open('workfile', 'w')
>>> print f
<open file 'workfile', mode 'w' at 80a0960>
```

The second argument is another string containing a few characters describing the way in which the file will be used. `r` = read. `w` = write (overwrites destructively). `a` = append. Any data written to the file is automatically added to the end.

`r+` opens the file for both reading and writing. The mode argument is optional; `r` will be assumed if it's omitted.

On Windows, you need to use special syntax (see documentation 7.2).

Methods of File Objects

Assume that a file object called `f` has already been created.

To read a file's contents, call `f.read(size)`, which reads some quantity of data and returns it as a string. When size is omitted or negative, the entire contents of the file will be read and returned.

`f.readline()` reads a single line from the file; a newline character `\n` is left at the end of the string, and is only omitted on the last line of the file if the file doesn't end in a newline.

If `f.readline()` returns an empty string, the end of the file has been reached, while a blank line is represented by `\n`.

For reading lines from a file, you can loop over the file object. This is memory efficient, fast, and leads to simple code:

```
>>> for line in f:
        print line,
```

If you want to read all the lines of a file in a list you can also use `list(f)` or `f.readlines()`.

Writing to the file

`f.write(string)` writes the contents of string to the file, returning None.

```
>>> f.write('This is a test\n')
```

To write something other than a string, it needs to be converted to a string first:

Cursor Manipulation

`f.tell()` returns an integer giving the file object's current position in the file, measured in bytes from the beginning of the file. To change the file object's position, use `f.seek(offset, from_what)`.

A from_what value of `0` measures from the beginning of the file, `1` uses the current file position, and `2` uses the end of the file as the reference point. `from_what` can be omitted and defaults to `0`.

Closing the file

When you're done with a file, call `f.close()` to close it and free up any system resources taken up by the open file. After calling `f.close()`, attempts to use the file object will automatically fail.

Using `with` when dealing with file objects

It is good practice to use the `with` keyword when dealing with file objects. This has the advantage that the file is properly closed after its suite finishes, even if an exception is raised on the way.

```
>>> with open('workfile', 'r') as f:
...     read_data = f.read()
>>> f.closed
True
```

The PICKLE Module

Strings can easily be written to and read from a file. Numbers take a bit more effort, since the `read()` method only returns strings, which will have to be passed to a function like `int()`.

The Pickle module helps with this. The Pickle module can take almost any Python object (even some forms of Python code!), and convert it to a string representation; this process is called **pickling**. Reconstructing the object from the string representation is called **unpickling**.

If you have an object x, and a file object f that's been opened for writing:

```
pickle.dump(x, f)
```

To unpickle the object again:

```
x = pickle.load(f)
```

Pickle is the standard way to make Python objects which can be stored and reused by other programs or by a future invocation of the same program.

Errors and Exceptions

There are (at least) two distinguishable kinds of errors: **syntax errors** and **exceptions**.

Errors detected during execution are called **exceptions** and are not unconditionally fatal

The string printed as the exception type is the name of the built-in exception that occurred.

`bltin-exceptions` lists the built-in exceptions and their meanings.

Exception Handling

It is possible to write programs that handle selected exceptions.

Here's how the `try` statement works:

- First, the try clause (the statement(s) between the try and except keywords) is executed.
- If no exception occurs, the except clause is skipped and execution of the try statement is finished.
- If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the except keyword, the except clause is executed, and then execution continues after the try statement.
- If an exception occurs which does not match the exception named in the

except clause, it is passed on to outer try statements; if no handler is found, it is an unhandled exception and execution stops with a message as shown above.

A try statement may have more than one except clause, to specify handlers for different exceptions. At most one handler will be executed.

Handlers only handle exceptions that occur in the corresponding try clause, not in other handlers of the same try statement.

An except clause may name multiple exceptions as a parenthesized tuple, for example:

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

The last except clause may omit the exception name(s), to serve as a wildcard.

```
try:
    f = open('myfile.txt') s = f.readline()
    i = int(s.strip())
except IOError as e:
    print "I/O error({0}): {1}".format(e.errno, e.strerror)
except ValueError:
    print "Could not convert data to an integer."
except:
    print "Unexpected error:", sys.exc_info()[0] raise
```

The `try ... except` statement has an optional `else` clause, which, when present, must follow all except clauses. It is useful for code that must be executed if the try clause does not raise an exception. For example:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print 'cannot open', arg
    else:
        print arg, 'has', len(f.readlines()), 'lines'
        f.close()
```

The use of the else clause is better than adding additional code to the try clause because it avoids accidentally catching an exception that wasn't raised by the code being protected by the `try ... except` statement.

When an exception occurs, it may have an associated value, also known as the exception's argument. The presence and type of the argument depend on the exception type. The except clause may specify a variable after the exception name (or tuple). The variable is bound to an exception instance with the arguments stored in `instance.args`. For convenience, the exception instance defines `__str__()` so the arguments can be printed directly without having to reference `.args`. One may also instantiate an exception first before raising it and add any attributes to it as desired.

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print type(inst)            # the exception instance
...     print inst.args             # arguments stored in .args
...     print inst                  # __str__ allows args to printed
directly
...     x, y = inst.args
...     print 'x =', x
...     print 'y =', y
...

<type 'exceptions.Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

If an exception has an argument, it is printed as the last part (‘detail’) of the message for unhandled exceptions.

Exception handlers also handle exceptions from functions called inside the `try` block.

Raising Exceptions

The raise statement allows the programmer to force a specified exception to occur.

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
    File "<stdin>", line 1, in ?
NameError: HiThere
```

User Defined Exceptions

Programs may name their own exceptions by creating a new exception class. Exceptions should typically be derived from the Exception class, either directly or indirectly. E.g.

```
class MyError(Exception):
    def __init__(self, value):
        self.value = value
    def __str__(self):
        return repr(self.value)
try:
    raise MyError(2*2)
except MyError as e:
    print 'My exception occurred, value:', e.value
```

In this example, the default `__init__()` of Exception has been overridden. This replaces the default behavior of creating the args attribute.

Exception classes can be defined which do anything any other class can do, but are usually kept simple.

BEST PRACTICE: When creating a module that can raise several distinct errors, a common practice is to create a base class for exceptions defined by that module, and subclass that to create specific exception classes for different error conditions

Most exceptions are defined with names that end in “Error”.

Defining Clean Up Actions

The try statement has another optional clause, `finally`, which is intended to define clean-up actions that must be executed under all circumstances. For example:

```
>>> try:
... raise KeyboardInterrupt
... finally:
... print 'Goodbye, world!'
```

A `finally` clause is always executed before leaving the try statement, whether an exception has occurred or not.

When an exception has occurred in the try clause and has not been handled by an except clause (or it has occurred in a except or else clause), it is re-raised after the finally clause has been executed. The finally clause is also executed “on the way out” when any other clause of the try statement is left via a break, continue or return statement.

Predefined Clean-up Actions

Some objects define standard clean-up actions to be undertaken when the object is no longer needed.

The `with` statement allows objects like files to be used in a way that ensures they are always cleaned up promptly and correctly.

```
with open("myfile.txt") as f:
    for line in f:
        print line,
```

After the statement is executed, the file `f` is always closed, even if a problem was encountered while processing the lines.

Classes

The class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain arbitrary amounts and kinds of data.

Normally class members (including variables) are **public**, and all member functions are virtual.

There are no shorthands for referencing the object's members from its methods.

The method function is declared with an explicit first argument representing the object, which is provided implicitly by the call.

Classes themselves are objects. This provides semantics for importing and renaming.

Built-in types can be used as base classes for extension by the user.

Aliases

Objects have individuality, and multiple names (in multiple scopes) can be bound to the same object. This is known as aliasing in other languages.

Aliases behave like pointers in some respects. For example, passing an object is cheap since only a pointer is passed by the implementation; and if a function modifies an object passed as an argument, the caller will see the change.

Python Scopes and Namespaces

A namespace is a mapping from names to objects. E.g. built-in exception names.

The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces.

According to Python, any name following a dot is called an attribute (e.g. function via module name).

References to names in modules are attribute references. E.g.

In the expression `modname.funcname`, `modname` is a module object and funcname is an attribute of it.

The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of a module called `__main__`, so they have their own global namespace. (The built-in names actually also live in a module; this is called `__builtin__`.)

The local namespace for a function is created when the function is called, and deleted when the function returns or raises an exception that is not handled within the function.

A scope is a textual region of a Python program where a namespace is directly accessible.

At any time during execution, there are at least three nested scopes whose namespaces are directly accessible:

- the innermost scope, which is searched first, contains the local names
- the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope, contains non-local, but also non-global names
- the next-to-last scope contains the current module's global names
- the outermost scope (searched last) is the namespace containing built-in

names

All variables found outside of the innermost scope are read-only (an attempt to write to such a variable will simply create a new local variable in the innermost scope, leaving the identically named outer variable unchanged).

Intro to Classes

The simplest form of class definition looks like this:

```
class ClassName:
    <statement-1>
    .
    <statement-N>
```

Class definitions, like function definitions (def statements) must be executed before they have any effect.

Class Objects

When a class definition is left normally (via the end), a class object is created. This is basically a wrapper around the contents of the namespace created by the class definition

Class objects support two kinds of operations: attribute references and instantiation.

Attribute references use the standard syntax used for all attribute references in Python: `obj.name`.

So with:

```
class MyClass:
"""A simple example class"""
i = 12345
def f(self):
    return 'hello world'
```

`MyClass.i` and `MyClass.f` are valid attribute references, returning an integer and a function object.

Class attributes can also be assigned to, so you can change the value of `MyClass.i` by assignment.

`__doc__` is also a valid attribute, returning the docstring belonging to the class: "A simple example class".

Class instantiation uses function notation. Just pretend that the class object is a parameterless function that returns a new instance of the class. E.g.

```
x = MyClass()
```

Creates a new instance of the class and assigns this object to the local variable `x`.

Many classes like to create objects with instances customized to a specific initial state. Therefore a class may define a special method named `__init__()`, like this:

```
def __init__(self):
    self.data = []
```

When a class defines an `__init__()` method, class instantiation automatically invokes `__init__()` for the newly-created class instance.

If the `__init__()` method has more arguments, then any arguments given to the class instantiation operator are passed on to `__init__()`.

```
class Complex:
    def __init__(self, realpart, imagpart):
        self.r = realpart
        self.i = imagpart

x = Complex(3.0, -4.5)
x.r, x.i
(3.0, -4.5)
```

Instance Objects

The only operations understood by instance objects are attribute references. There are two kinds of valid attribute names, data attributes and methods.

data attributes correspond to **instance variables** in Smalltalk, and to **data members** in C++.

Data attributes need not be declared; like local variables, they spring into existence when they are first assigned to.

The other kind of instance attribute reference is a method. A method is a function that “belongs to” an object.

Method Objects

Usually, a method is called right after it is bound.

```
x.f()
```

But methods can be used later as well.

```
xf = x.f while True:
print xf()
```

The special thing about methods is that the object is passed as the first argument of the function.

In our example, the call `x.f()` is exactly equivalent to `MyClass.f(x)`.

General Knowledge

Data attributes override method attributes with the same name.

Use verbs for methods and nouns for data attributes to avoid conflicts (or some other convention).

Classes are not usable to implement pure abstract data types. In fact, nothing in Python makes it possible to enforce data hiding — it is all based upon convention.

(On the other hand, the Python implementation, written in C, can completely hide implementation details and control access to an object if necessary; this can be used by extensions to Python written in C.)

Often, the first argument of a method is called `self`. This is nothing more than a convention: the name self has absolutely no special meaning to Python.

It is not necessary that the function definition is textually enclosed in the class definition: assigning a function object to a local variable in the class is also ok. E.g.

```
# Function defined outside the class
def f1(self, x, y):
    return min(x, x+y)
class C:
    f = f1
    def g(self):
        return 'hello world'
    h=g
```

Methods may call other methods by using method attributes of the self argument. E.g.

```
class Bag:
    def __init__(self):
        self.data = []
    def add(self, x):
        self.data.append(x)
    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

Methods may reference global names in the same way as ordinary functions. The global scope associated with a method is the module containing its

definition. (A class is never used as a global scope.)

Each value is an object, and therefore has a class (also called its type). It is stored as `object.__class__`.

Inheritance

The syntax for a derived class definition looks like this:

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    <statement-N>
```

The name BaseClassName must be defined in a scope containing the derived class definition. In place of a base class name, other arbitrary expressions are also allowed. This can be useful, for example, when the base class is defined in another module:

```
class DerivedClassName(modname.BaseClassName):
```

Execution of a derived class definition proceeds the same as for a base class. When the class object is constructed, the base class is remembered.

Derived classes may override methods of their base classes.

There is a simple way to call the base class method directly: just call `BaseClassName.methodname(self, arguments)`.

An overriding method in a derived class may in fact want to extend rather than simply replace the base class method of the same name.

Python has two built-in functions that work with inheritance:

- Use `isinstance()` to check an instance's type: `isinstance(obj, int)` will be True only if `obj.__class__` is int

or some class derived from int.

- Use `issubclass()` to check class inheritance: `issubclass(bool, int)` is True since bool is a subclass of int. However, `issubclass(unicode, str)` is False since unicode is not a subclass of `str` (they only share a common ancestor, basestring).

Mupltiple Inheritance

Python supports a limited form of multiple inheritance as well. A class definition with multiple base classes looks like this:

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

For old-style classes, the only rule is depth-first, left-to-right.

Therefore, if an attribute is not found in `DerivedClassName`, it is searched in `Base1`, then (recursively) in the base classes of `Base1`, and only if it is not found there, it is searched in `Base2`, and so on.

For new-style classes, the method resolution order changes dynamically to support cooperative calls to `super()`.

With new-style classes, dynamic ordering is necessary because all cases of multiple inheritance exhibit one or more diamond relationships (where at least one of the parent classes can be accessed through multiple paths from the bottommost class). For example, all new-style classes inherit from object, so any case of multiple inheritance provides more than one path to reach object.

To keep the base classes from being accessed more than once, the dynamic algorithm linearizes the search order in a way that preserves the left-to-right ordering specified in each class, that calls each parent only once, and that is

monotonic (meaning that a class can be subclassed without affecting the precedence order of its parents).

Private Variables and Class-local References

“Private” instance variables that cannot be accessed except from inside an object don’t exist in Python.

However, there is a convention that is followed by most Python code: a name prefixed with an underscore (e.g. _spam) should be treated as a non-public part of the API (whether it is a function, a method or a data member).

Python believes that the main use case for class-private members is to avoid name clashes with subclasses. So Python has a mechanism called **name mangling**.

Any identifier of the form `__spam` (at least two leading underscores, at most one trailing underscore) is textually replaced with `_classname__spam`, where `classname` is the current class name with leading underscore(s) stripped.

Name mangling is helpful for letting subclasses override methods without breaking intraclass method calls.

Note that the mangling rules are designed mostly to avoid accidents; it still is possible to access or modify a variable that is considered private.

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)

    def update(self, iterable):
    for item in iterable:
        self.items_list.append(item)

    __update = update # private copy of original update() method

class MappingSubclass(Mapping):
    def update(self, keys, values):
        # provides new signature for update()
        # but does not break __init__()
        for item in zip(keys, values):
            self.items_list.append(item)
```

Bundling

Sometimes it is useful to bundle together a few named data items. To do this we use an empty class definition.

```
class Employee:
    pass

john = Employee() # Create an empty employee record

# Fill the fields of the record
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```

A piece of Python code that expects a particular abstract data type can often be passed a class that emulates the methods of that data type instead. For instance, if you have a function that formats some data from a file object, you can define a class with methods `read()` and `readline()` that get the data from a string buffer instead, and pass it as an argument.

Instance method objects have attributes, too: `m.im_self` is the instance object with the method `m()`, and `m.im_func` is the function object corresponding to the method.

Exceptions Are Classes Too

User-defined exceptions are identified by classes as well. Using this mechanism it is possible to create extensible hierarchies of exceptions.

There are two new valid (semantic) forms for the raise statement:

```
raise Class, instance

raise instance
```

In the first form, `instance` must be an instance of `Class` or of a class derived from it. The second form is a shorthand for:

```
raise instance.__class__, instance
```

A class in an except clause is compatible with an exception if it is the same class or a base class thereof (but not the other way around — an except clause listing a derived class is not compatible with a base class).

For example, the following code will print B, C, D in that order:

```
class B:
    pass
class C(B):
    pass
class D(C):
    pass
for c in [B, C, D]:
    try:
        raise c()
    except D:
        print "D"
    except C:
        print "C"
    except B:
        print "B"
```

Iterators

Most container objects can be looped over using a for statement:

```
for element in [1, 2, 3]:
    print element
for element in (1, 2, 3):
    print element
for key in {'one':1, 'two':2}:
    print key
for char in "123":
    print char
for line in open("myfile.txt"):
    print line,
```

This next part (chapter 9.9) explains how to make your own iterator. Feel free to read it if you need to.

Generators

Generators are a simple and powerful tool for creating iterators. Chapter 9.10. As above.