7. Compound statements

Compound statements contain (groups of) other statements; they affect or control the execution of those other statements in some way. In general, compound statements span multiple lines, although in simple incarnations a whole compound statement may be contained in one line.

The `if`, `while` and `for` statements implement traditional control flow constructs. `try` specifies exception handlers and/or cleanup code for a group of statements. Function and class definitions are also syntactically compound statements.

Compound statements consist of one or more 'clauses.' A clause consists of a header and a 'suite.' The clause headers of a particular compound statement are all at the same indentation level. Each clause header begins with a uniquely identifying keyword and ends with a colon. A suite is a group of statements controlled by a clause. A suite can be one or more semicolon-separated simple statements on the same line as the header, following the header's colon, or it can be one or more indented statements on subsequent lines. Only the latter form of suite can contain nested compound statements; the following is illegal, mostly because it wouldn't be clear to which `if` clause a following `else` clause would belong:

```
if test1: if test2: print x
```

Also note that the semicolon binds tighter than the colon in this context, so that in the following example, either all or none of the `print` statements are executed:

```
if x < y < z: print x; print y; print z
```

Summarizing:

```
compound_stmt ::=  if_stmt
                   | while_stmt
                   | for_stmt
                   | try_stmt
                   | with_stmt
                   | funcdef
                   | classdef
                   | decorated
suite         ::=  stmt_list NEWLINE | NEWLINE INDENT statement+ DEDENT
statement     ::=  stmt_list NEWLINE | compound_stmt
stmt_list     ::=  simple_stmt (";" simple_stmt)* [";"]
```

Note that statements always end in a `NEWLINE` possibly followed by a `DEDENT`. Also note that optional continuation clauses always begin with a keyword that cannot start a statement, thus there are no ambiguities (the 'dangling `else`' problem is solved in Python by requiring nested `if` statements to be indented).

The formatting of the grammar rules in the following sections places each clause on a separate line for clarity.

7.1. The `if` statement

The `if` statement is used for conditional execution:

```
if_stmt ::=  "if" expression ":" suite
             ( "elif" expression ":" suite )*
             ["else" ":" suite]
```

It selects exactly one of the suites by evaluating the expressions one by one until one is found to be true (see section *Boolean operations* for the definition of true and false); then that suite is executed (and no other part of the `if` statement is executed or evaluated). If all expressions are false, the suite of the `else` clause, if present, is executed.

7.2. The `while` statement

The `while` statement is used for repeated execution as long as an expression is true:

```
while_stmt ::=  "while" expression ":" suite
                ["else" ":" suite]
```

This repeatedly tests the expression and, if it is true, executes the first suite; if the expression is false (which may be the first time it is tested) the suite of the `else` clause, if present, is executed and the loop terminates.

A `break` statement executed in the first suite terminates the loop without executing the `else` clause's suite. A `continue` statement executed in the first suite skips the rest of the suite and goes back to testing the expression.

7.3. The `for` statement

The `for` statement is used to iterate over the elements of a sequence (such as a string, tuple or list) or other iterable object:

```
for_stmt ::=  "for" target_list "in" expression_list ":" suite
              ["else" ":" suite]
```

The expression list is evaluated once; it should yield an iterable object. An iterator is created for the result of the `expression_list`. The suite is then executed once for each item provided by the iterator, in the order of ascending indices. Each item in turn is assigned to the target list using the standard rules for assignments, and then the suite is executed. When the items are exhausted (which is immediately when the sequence is empty), the suite in the `else` clause, if present, is executed, and the loop terminates.

A `break` statement executed in the first suite terminates the loop without executing the `else` clause's suite. A `continue` statement executed in the first suite skips the rest of the suite and continues with the next item, or with the `else` clause if there was no next item.

The suite may assign to the variable(s) in the target list; this does not affect the next item assigned to it.

The target list is not deleted when the loop is finished, but if the sequence is empty, it will not have been assigned to at all by the loop. Hint: the built-in function `range()` returns a sequence of integers suitable to emulate the effect of Pascal's `for i := a to b do`; e.g., `range(3)` returns the list `[0, 1, 2]`.

Note: There is a subtlety when the sequence is being modified by the loop (this can only occur for mutable sequences, i.e. lists). An internal counter is used to keep track of which item is used next, and this is incremented on each iteration. When this counter has reached the length of the sequence the loop terminates. This means that if the suite deletes the current (or a previous) item from the sequence, the next item will be skipped (since it gets the index of the current item which has already been treated). Likewise, if the suite inserts an item in the sequence before the current item, the current item will be treated again the next time through the loop. This can lead to nasty bugs that can be avoided by making a temporary copy using a slice of the whole sequence, e.g.,

```
for x in a[:]:
    if x < 0: a.remove(x)
```

7.4. The `try` statement

The `try` statement specifies exception handlers and/or cleanup code for a group of statements:

```
try_stmt  ::=  try1_stmt | try2_stmt
try1_stmt ::=  "try" ":" suite
               ("except" [expression [("as" | ",") identifier]] ":" suite)+
               ["else" ":" suite]
               ["finally" ":" suite]
try2_stmt ::=  "try" ":" suite
               "finally" ":" suite
```

Changed in version 2.5: In previous versions of Python, `try`...`except`...`finally` did not work. `try`...`except` had to be nested in `try`...`finally`.

The `except` clause(s) specify one or more exception handlers. When no exception occurs in the `try` clause, no exception handler is executed. When an exception occurs in the `try` suite, a search for an exception handler is started. This search inspects the except clauses in turn until one is found that matches the exception. An expression-less except clause, if present, must be last; it matches any exception. For an except clause with an expression, that expression is

evaluated, and the clause matches the exception if the resulting object is “compatible” with the exception. An object is compatible with an exception if it is the class or a base class of the exception object, or a tuple containing an item compatible with the exception.

If no except clause matches the exception, the search for an exception handler continues in the surrounding code and on the invocation stack. [1]

If the evaluation of an expression in the header of an except clause raises an exception, the original search for a handler is canceled and a search starts for the new exception in the surrounding code and on the call stack (it is treated as if the entire `try` statement raised the exception).

When a matching except clause is found, the exception is assigned to the target specified in that except clause, if present, and the except clause’s suite is executed. All except clauses must have an executable block. When the end of this block is reached, execution continues normally after the entire try statement. (This means that if two nested handlers exist for the same exception, and the exception occurs in the try clause of the inner handler, the outer handler will not handle the exception.)

Before an except clause’s suite is executed, details about the exception are assigned to three variables in the `sys` module: `sys.exc_type` receives the object identifying the exception; `sys.exc_value` receives the exception’s parameter; `sys.exc_traceback` receives a traceback object (see section *The standard type hierarchy*) identifying the point in the program where the exception occurred. These details are also available through the `sys.exc_info()` function, which returns a tuple `(exc_type, exc_value, exc_traceback)`. Use of the corresponding variables is deprecated in favor of this function, since their use is unsafe in a threaded program. As of Python 1.5, the variables are restored to their previous values (before the call) when returning from a function that handled an exception.

The optional `else` clause is executed if and when control flows off the end of the `try` clause. [2] Exceptions in the `else` clause are not handled by the preceding `except` clauses.

If `finally` is present, it specifies a ‘cleanup’ handler. The `try` clause is executed, including any `except` and `else` clauses. If an exception occurs in any of the clauses and is not handled, the exception is temporarily saved. The `finally` clause is executed. If there is a saved exception, it is re-raised at the end of the `finally` clause. If the `finally` clause raises another exception or executes a `return` or `break` statement, the saved exception is discarded:

```
>>> def f():
...     try:
...         1/0
...     finally:
...         return 42
...
>>> f()
42
```

The exception information is not available to the program during execution of the `finally` clause.

When a `return`, `break` or `continue` statement is executed in the `try` suite of a `try...finally` statement, the `finally` clause is also executed 'on the way out.' A `continue` statement is illegal in the `finally` clause. (The reason is a problem with the current implementation — this restriction may be lifted in the future).

The return value of a function is determined by the last `return` statement executed. Since the `finally` clause always executes, a `return` statement executed in the `finally` clause will always be the last one executed:

```
>>> def foo():
...     try:
...         return 'try'
...     finally:
...         return 'finally'
...
>>> foo()
'finally'
```

Additional information on exceptions can be found in section *Exceptions*, and information on using the `raise` statement to generate exceptions may be found in section *The raise statement*.

7.5. The `with` statement

New in version 2.5.

The `with` statement is used to wrap the execution of a block with methods defined by a context manager (see section *With Statement Context Managers*). This allows common `try...except...finally` usage patterns to be encapsulated for convenient reuse.

```
with_stmt ::=  "with" with_item ("," with_item)* ":" suite
with_item ::=  expression ["as" target]
```

The execution of the `with` statement with one "item" proceeds as follows:

1. The context expression (the expression given in the `with_item`) is evaluated to obtain a context manager.
2. The context manager's `__exit__()` is loaded for later use.
3. The context manager's `__enter__()` method is invoked.
4. If a target was included in the `with` statement, the return value from `__enter__()` is assigned to it.

Note: The `with` statement guarantees that if the `__enter__()` method returns without an error, then `__exit__()` will always be called. Thus, if an error occurs during the assignment to the target list, it will be treated the same as an error occurring within the suite would be. See step 6 below.

5. The suite is executed.

6. The context manager's `__exit__()` method is invoked. If an exception caused the suite to be exited, its type, value, and traceback are passed as arguments to `__exit__()`. Otherwise, three `None` arguments are supplied.

   If the suite was exited due to an exception, and the return value from the `__exit__()` method was false, the exception is reraised. If the return value was true, the exception is suppressed, and execution continues with the statement following the `with` statement.

   If the suite was exited for any reason other than an exception, the return value from `__exit__()` is ignored, and execution proceeds at the normal location for the kind of exit that was taken.

With more than one item, the context managers are processed as if multiple `with` statements were nested:

```
with A() as a, B() as b:
    suite
```

is equivalent to

```
with A() as a:
    with B() as b:
        suite
```

Note: In Python 2.5, the `with` statement is only allowed when the `with_statement` feature has been enabled. It is always enabled in Python 2.6.

Changed in version 2.7: Support for multiple context expressions.

See also:

PEP 0343 - The "with" statement
The specification, background, and examples for the Python `with` statement.

7.6. Function definitions

A function definition defines a user-defined function object (see section *The standard type*

hierarchy):

```
decorated      ::=  decorators (classdef | funcdef)
decorators     ::=  decorator+
decorator      ::=  "@" dotted_name ["(" [argument_list [","]] ")"] NEWLINE
funcdef        ::=  "def" funcname "(" [parameter_list] ")" ":" suite
dotted_name    ::=  identifier ("." identifier)*
parameter_list ::=  (defparameter ",")*
                    (  "*" identifier ["," "**" identifier]
                    | "**" identifier
                    | defparameter [","] )
defparameter   ::=  parameter ["=" expression]
sublist        ::=  parameter ("," parameter)* [","]
parameter      ::=  identifier | "(" sublist ")"
funcname       ::=  identifier
```

A function definition is an executable statement. Its execution binds the function name in the current local namespace to a function object (a wrapper around the executable code for the function). This function object contains a reference to the current global namespace as the global namespace to be used when the function is called.

The function definition does not execute the function body; this gets executed only when the function is called. [3]

A function definition may be wrapped by one or more *decorator* expressions. Decorator expressions are evaluated when the function is defined, in the scope that contains the function definition. The result must be a callable, which is invoked with the function object as the only argument. The returned value is bound to the function name instead of the function object. Multiple decorators are applied in nested fashion. For example, the following code:

```
@f1(arg)
@f2
def func(): pass
```

is equivalent to:

```
def func(): pass
func = f1(arg)(f2(func))
```

When one or more top-level *parameters* have the form *parameter* = *expression*, the function is said to have “default parameter values.” For a parameter with a default value, the corresponding *argument* may be omitted from a call, in which case the parameter’s default value is substituted. If a parameter has a default value, all following parameters must also have a default value — this is a syntactic restriction that is not expressed by the grammar.

Default parameter values are evaluated when the function definition is executed. This means that the expression is evaluated once, when the function is defined, and that the same “pre-computed” value is used for each call. This is especially important to understand when a default parameter is a mutable object, such as a list or a dictionary: if the function modifies the object (e.g. by appending an item to a list), the default value is in effect modified. This is

generally not what was intended. A way around this is to use `None` as the default, and explicitly test for it in the body of the function, e.g.:

```
def whats_on_the_telly(penguin=None):
    if penguin is None:
        penguin = []
    penguin.append("property of the zoo")
    return penguin
```

Function call semantics are described in more detail in section *Calls*. A function call always assigns values to all parameters mentioned in the parameter list, either from position arguments, from keyword arguments, or from default values. If the form "`*identifier`" is present, it is initialized to a tuple receiving any excess positional parameters, defaulting to the empty tuple. If the form "`**identifier`" is present, it is initialized to a new dictionary receiving any excess keyword arguments, defaulting to a new empty dictionary.

It is also possible to create anonymous functions (functions not bound to a name), for immediate use in expressions. This uses lambda expressions, described in section *Lambdas*. Note that the lambda expression is merely a shorthand for a simplified function definition; a function defined in a "`def`" statement can be passed around or assigned to another name just like a function defined by a lambda expression. The "`def`" form is actually more powerful since it allows the execution of multiple statements.

Programmer's note: Functions are first-class objects. A "`def`" form executed inside a function definition defines a local function that can be returned or passed around. Free variables used in the nested function can access the local variables of the function containing the def. See section *Naming and binding* for details.

7.7. Class definitions

A class definition defines a class object (see section *The standard type hierarchy*):

```
classdef    ::=  "class" classname [inheritance] ":" suite
inheritance ::=  "(" [expression_list] ")"
classname   ::=  identifier
```

A class definition is an executable statement. It first evaluates the inheritance list, if present. Each item in the inheritance list should evaluate to a class object or class type which allows subclassing. The class's suite is then executed in a new execution frame (see section *Naming and binding*), using a newly created local namespace and the original global namespace. (Usually, the suite contains only function definitions.) When the class's suite finishes execution, its execution frame is discarded but its local namespace is saved. [4] A class object is then created using the inheritance list for the base classes and the saved local namespace for the attribute dictionary. The class name is bound to this class object in the original local namespace.

Programmer's note: Variables defined in the class definition are class variables; they are shared by all instances. To create instance variables, they can be set in a method with `self.name = value`. Both class and instance variables are accessible through the notation "`self.name`", and an instance variable hides a class variable with the same name when accessed in this way. Class variables can be used as defaults for instance variables, but using mutable values there can lead to unexpected results. For *new-style classes*, descriptors can be used to create instance variables with different implementation details.

Class definitions, like function definitions, may be wrapped by one or more *decorator* expressions. The evaluation rules for the decorator expressions are the same as for functions. The result must be a class object, which is then bound to the class name.

Footnotes

[1] The exception is propagated to the invocation stack unless there is a **`finally`** clause which happens to raise another exception. That new exception causes the old one to be lost.

[2] Currently, control "flows off the end" except in the case of an exception or the execution of a **`return`**, **`continue`**, or **`break`** statement.

[3] A string literal appearing as the first statement in the function body is transformed into the function's `__doc__` attribute and therefore the function's *docstring*.

[4] A string literal appearing as the first statement in the class body is transformed into the namespace's `__doc__` item and therefore the class's *docstring*.