8. 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, while the `with` statement allows the execution of initialization and finalization code around a block of code. Function and class definitions are also syntactically compound statements.

A compound statement consists 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 a 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()` calls 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
                   | async_with_stmt
                   | async_for_stmt
                   | async_funcdef
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.

8.1. 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.

8.2. 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.

8.3. 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 returned by the iterator. Each item in turn is assigned to the target list using the standard rules for assignments (see Assignment statements), and then the suite is executed. When the items are exhausted (which is immediately when the sequence is empty or an iterator raises a `StopIteration` exception), 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 is no next item.

The for-loop makes assignments to the variables in the target list. This overwrites all previous assignments to those variables including those made in the suite of the for-loop:

```
for i in range(10):
    print(i)
    i = 5             # this will not affect the for-loop
                      # because i will be overwritten with the next
                      # index in the range
```

Names in the target list are not deleted when the loop is finished, but if the sequence is empty, they will not have been assigned to at all by the loop. Hint: the built-in function `range()` returns an iterator of integers suitable to emulate the effect of Pascal's `for i := a to b do`; e.g., `list(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, e.g. 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)
```

8.4. 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
```

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 after the `as` keyword 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.)

When an exception has been assigned using `as target`, it is cleared at the end of the except clause. This is as if

```
except E as N:
    foo
```

was translated to

```
except E as N:
    try:
        foo
    finally:
        del N
```

This means the exception must be assigned to a different name to be able to refer to it after the except clause. Exceptions are cleared because with the traceback attached to them, they form a reference cycle with the stack frame, keeping all locals in that frame alive until the next garbage collection occurs.

Before an except clause's suite is executed, details about the exception are stored in the `sys` module and can be accessed via `sys.exc_info()`. `sys.exc_info()` returns a 3-tuple consisting of the exception class, the exception instance and a traceback object (see section The standard type hierarchy) identifying the point in the program where the exception occurred. `sys.exc_info()` values 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 the control flow leaves the `try` suite, no exception was raised, and no `return`, `continue`, or `break` statement was executed. 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, the saved exception is set as the context of the new exception. If the `finally` clause executes a `return`, `break` or `continue` 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.'

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.

Changed in version 3.8: Prior to Python 3.8, a `continue` statement was illegal in the `finally` clause due to a problem with the implementation.

8.5. The `with` statement

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
```

Changed in version 3.1: Support for multiple context expressions.

See also:

PEP 343 - The "with" statement
The specification, background, and examples for the Python `with` statement.

8.6. Function definitions

A function definition defines a user-defined function object (see section The standard type hierarchy):

```
funcdef                 ::=  [decorators] "def" funcname "(" [parameter_list] "
                             ["->" expression] ":" suite
decorators              ::=  decorator+
decorator               ::=  "@" dotted_name ["(" [argument_list [","]] ")"] NE
dotted_name             ::=  identifier ("." identifier)*
parameter_list          ::=  defparameter ("," defparameter)* "," "/" ["," [par
                               | parameter_list_no_posonly
parameter_list_no_posonly ::=  defparameter ("," defparameter)* ["," [parameter_l
                             | parameter_list_starargs
parameter_list_starargs ::=  "*" [parameter] ("," defparameter)* ["," ["**" par
                             | "**" parameter [","]
parameter               ::=  identifier [":" expression]
defparameter            ::=  parameter ["=" expression]
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. [2]

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 roughly equivalent to

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

except that the original function is not temporarily bound to the name `func`.

When one or more 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 up until the “`*`” must also have a default value — this is a syntactic restriction that is not expressed by the grammar.

Default parameter values are evaluated from left to right 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 ordered mapping receiving any excess keyword arguments, defaulting to a new empty mapping of the same type. Parameters after “`*`” or “`*identifier`” are keyword-only parameters and may only be passed used keyword arguments.

Parameters may have an annotation of the form “`: expression`” following the parameter name. Any parameter may have an annotation, even those of the form `*identifier` or `**identifier`. Functions may have “return” annotation of the form “`-> expression`” after the parameter list. These annotations can be any valid Python expression. The presence of annotations does not change the semantics of a function. The annotation values are available as values of a dictionary keyed by the parameters’ names in the `__annotations__` attribute of the function object. If the

`annotations` import from `__future__` is used, annotations are preserved as strings at runtime which enables postponed evaluation. Otherwise, they are evaluated when the function definition is executed. In this case annotations may be evaluated in a different order than they appear in the source code.

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 and annotations.

Programmer's note: Functions are first-class objects. A "`def`" statement 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.

See also:

PEP 3107 - Function Annotations
The original specification for function annotations.

PEP 484 - Type Hints
Definition of a standard meaning for annotations: type hints.

PEP 526 - Syntax for Variable Annotations
Ability to type hint variable declarations, including class variables and instance variables

PEP 563 - Postponed Evaluation of Annotations
Support for forward references within annotations by preserving annotations in a string form at runtime instead of eager evaluation.

8.7. Class definitions

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

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

A class definition is an executable statement. The inheritance list usually gives a list of base classes (see Metaclasses for more advanced uses), so each item in the list should evaluate to a class object which allows subclassing. Classes without an inheritance list inherit, by default, from the base class `object`; hence,

```
class Foo:
    pass
```

is equivalent to

```
class Foo(object):
    pass
```

The class's suite is then executed in a new execution frame (see Naming and binding), using a newly created local namespace and the original global namespace. (Usually, the suite contains mostly function definitions.) When the class's suite finishes execution, its execution frame is discarded but its local namespace is saved. [3] 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.

The order in which attributes are defined in the class body is preserved in the new class's `__dict__`. Note that this is reliable only right after the class is created and only for classes that were defined using the definition syntax.

Class creation can be customized heavily using metaclasses.

Classes can also be decorated: just like when decorating functions,

```
@f1(arg)
@f2
class Foo: pass
```

is roughly equivalent to

```
class Foo: pass
Foo = f1(arg)(f2(Foo))
```

The evaluation rules for the decorator expressions are the same as for function decorators. The result is then bound to the class name.

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

See also:

PEP 3115 - Metaclasses in Python 3000

The proposal that changed the declaration of metaclasses to the current syntax, and the semantics for how classes with metaclasses are constructed.

PEP 3129 - Class Decorators
The proposal that added class decorators. Function and method decorators were introduced in **PEP 318**.

8.8. Coroutines

New in version 3.5.

8.8.1. Coroutine function definition

```
async_funcdef ::=  [decorators] "async" "def" funcname "(" [parameter_list] ")"
                   ["->" expression] ":" suite
```

Execution of Python coroutines can be suspended and resumed at many points (see coroutine). Inside the body of a coroutine function, `await` and `async` identifiers become reserved keywords; `await` expressions, `async for` and `async with` can only be used in coroutine function bodies.

Functions defined with `async def` syntax are always coroutine functions, even if they do not contain `await` or `async` keywords.

It is a `SyntaxError` to use a `yield from` expression inside the body of a coroutine function.

An example of a coroutine function:

```
async def func(param1, param2):
    do_stuff()
    await some_coroutine()
```

8.8.2. The `async for` statement

```
async_for_stmt ::=  "async" for_stmt
```

An asynchronous iterable is able to call asynchronous code in its *iter* implementation, and asynchronous iterator can call asynchronous code in its *next* method.

The `async for` statement allows convenient iteration over asynchronous iterators.

The following code:

```
async for TARGET in ITER:
    BLOCK
else:
    BLOCK2
```

Is semantically equivalent to:

```
iter = (ITER)
iter = type(iter).__aiter__(iter)
running = True
while running:
    try:
        TARGET = await type(iter).__anext__(iter)
    except StopAsyncIteration:
        running = False
    else:
        BLOCK
else:
    BLOCK2
```

See also `__aiter__()` and `__anext__()` for details.

It is a `SyntaxError` to use an `async for` statement outside the body of a coroutine function.

8.8.3. The `async with` statement

```
async_with_stmt ::=  "async" with_stmt
```

An asynchronous context manager is a context manager that is able to suspend execution in its *enter* and *exit* methods.

The following code:

```
async with EXPR as VAR:
    BLOCK
```

Is semantically equivalent to:

```
mgr = (EXPR)
aexit = type(mgr).__aexit__
aenter = type(mgr).__aenter__(mgr)

VAR = await aenter
try:
    BLOCK
except:
    if not await aexit(mgr, *sys.exc_info()):
        raise
else:
    await aexit(mgr, None, None, None)
```

See also `__aenter__()` and `__aexit__()` for details.

It is a `SyntaxError` to use an `async with` statement outside the body of a coroutine function.

See also:

PEP 492 - Coroutines with async and await syntax
The proposal that made coroutines a proper standalone concept in Python, and added supporting syntax.

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] 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.

[3] 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.