7. Simple statements

A simple statement is comprised within a single logical line. Several simple statements may occur on a single line separated by semicolons. The syntax for simple statements is:

```
simple_stmt ::=   expression_stmt
                | assert_stmt
                | assignment_stmt
                | augmented_assignment_stmt
                | annotated_assignment_stmt
                | pass_stmt
                | del_stmt
                | return_stmt
                | yield_stmt
                | raise_stmt
                | break_stmt
                | continue_stmt
                | import_stmt
                | future_stmt
                | global_stmt
                | nonlocal_stmt
                | type_stmt
```

7.1. Expression statements

Expression statements are used (mostly interactively) to compute and write a value, or (usually) to call a procedure (a function that returns no meaningful result; in Python, procedures return the value None). Other uses of expression statements are allowed and occasionally useful. The syntax for an expression statement is:

```
expression_stmt ::=   starred_expression
```

An expression statement evaluates the expression list (which may be a single expression).

In interactive mode, if the value is not None, it is converted to a string using the built-in repr() function and the resulting string is written to standard output on a line by itself (except if the result is None, so that procedure calls do not cause any output.)

7.2. Assignment statements

Assignment statements are used to (re)bind names to values and to modify attributes or items of mutable objects:

```
assignment_stmt ::=   (target_list "=")+ (starred_expression | yield_expression)
target_list     ::=   target ("," target)* [","]
target          ::=   identifier
                    | "(" [target_list] ")"
                    | "[" [target_list] "]"
                    | attributeref
                    | subscription
                    | slicing
                    | "*" target
```

(See section Primaries for the syntax definitions for *attributeref*, *subscription*, and *slicing*.)

Assignment is defined recursively depending on the form of the target (list). When a target is part of a mutable object (an attribute reference, subscription or slicing), the mutable object must ultimately perform the assignment and decide about its validity, and may raise an exception if the assignment is unacceptable. The rules observed by various types and the exceptions raised are given with the definition of the object types (see section The standard type hierarchy).

Assignment of an object to a target list, optionally enclosed in parentheses or square brackets, is recursively defined as follows.

- If the target list is a single target with no trailing comma, optionally in parentheses, the object is assigned to that target.
- Else:
  - If the target list contains one target prefixed with an asterisk, called a "starred" target: The object must be an iterable with at least as many items as there are targets in the target list, minus one. The first items of the iterable are assigned, from left to right, to the targets before the starred target. The final items of the iterable are assigned to the targets after the starred target. A list of the remaining items in the iterable is then assigned to the starred target (the list can be empty).
  - Else: The object must be an iterable with the same number of items as there are targets in the target list, and the items are assigned, from left to right, to the corresponding targets.

Assignment of an object to a single target is recursively defined as follows.

- If the target is an identifier (name):

  - If the name does not occur in a `global` or `nonlocal` statement in the current code block: the name is bound to the object in the current local namespace.
  - Otherwise: the name is bound to the object in the global namespace or the outer namespace determined by `nonlocal`, respectively.

  The name is rebound if it was already bound. This may cause the reference count for the object previously bound to the name to reach zero, causing the object to be deallocated and its destructor (if it has one) to be called.

- If the target is an attribute reference: The primary expression in the reference is evaluated. It should yield an object with assignable attributes; if this is not the case, `TypeError` is raised. That object is then asked to assign the assigned object to the given attribute; if it cannot perform the assignment, it raises an exception (usually but not necessarily `AttributeError`).

  Note: If the object is a class instance and the attribute reference occurs on both sides of the assignment operator, the right-hand side expression, `a.x` can access either an instance attribute or (if no instance attribute exists) a class attribute. The left-hand side target `a.x` is always set as an instance attribute, creating it if necessary. Thus, the two occurrences of `a.x` do not necessarily refer to the same attribute: if the right-hand side expression refers to a class attribute, the left-hand side creates a new instance attribute as the target of the assignment:

  ```
  class Cls:
      x = 3               # class variable
  inst = Cls()
  inst.x = inst.x + 1    # writes inst.x as 4 leaving Cls.x as 3
  ```

  This description does not necessarily apply to descriptor attributes, such as properties created with `property()`.

- If the target is a subscription: The primary expression in the reference is evaluated. It should yield either a mutable sequence object (such as a list) or a mapping object (such as a dictionary). Next, the subscript expression is evaluated.

sequence is asked to assign the assigned object to its item with that index. If the index is out of range, `IndexError` is raised (assignment to a subscripted sequence cannot add new items to a list).

If the primary is a mapping object (such as a dictionary), the subscript must have a type compatible with the mapping's key type, and the mapping is then asked to create a key/value pair which maps the subscript to the assigned object. This can either replace an existing key/value pair with the same key value, or insert a new key/value pair (if no key with the same value existed).

For user-defined objects, the `__setitem__()` method is called with appropriate arguments.

- If the target is a slicing: The primary expression in the reference is evaluated. It should yield a mutable sequence object (such as a list). The assigned object should be a sequence object of the same type. Next, the lower and upper bound expressions are evaluated, insofar they are present; defaults are zero and the sequence's length. The bounds should evaluate to integers. If either bound is negative, the sequence's length is added to it. The resulting bounds are clipped to lie between zero and the sequence's length, inclusive. Finally, the sequence object is asked to replace the slice with the items of the assigned sequence. The length of the slice may be different from the length of the assigned sequence, thus changing the length of the target sequence, if the target sequence allows it.

CPython implementation detail: In the current implementation, the syntax for targets is taken to be the same as for expressions, and invalid syntax is rejected during the code generation phase, causing less detailed error messages.

Although the definition of assignment implies that overlaps between the left-hand side and the right-hand side are 'simultaneous' (for example `a, b = b, a` swaps two variables), overlaps *within* the collection of assigned-to variables occur left-to-right, sometimes resulting in confusion. For instance, the following program prints `[0, 2]`:

```
x = [0, 1]
i = 0
i, x[i] = 1, 2          # i is updated, then x[i] is updated
print(x)
```

7.2.1. Augmented assignment statements

Augmented assignment is the combination, in a single statement, of a binary operation and an assignment statement:

```
augmented_assignment_stmt ::=  augtarget augop (expression_list | yield_expression)
augtarget                 ::=  identifier | attributeref | subscription | slicing
augop                     ::=  "+=" | "-=" | "*=" | "@=" | "/=" | "//=" | "%=" | "**="
                               | ">>=" | "<<=" | "&=" | "^=" | "|="
```

(See section Primaries for the syntax definitions of the last three symbols.)

An augmented assignment evaluates the target (which, unlike normal assignment statements, cannot be an unpacking) and the expression list, performs the binary operation specific to the type of assignment on the two operands, and assigns the result to the original target. The target is only evaluated once.

meaning that rather than creating a new object and assigning that to the target, the old object is modified instead.

Unlike normal assignments, augmented assignments evaluate the left-hand side *before* evaluating the right-hand side. For ex-ample, `a[i] += f(x)` first looks-up `a[i]`, then it evaluates `f(x)` and performs the addition, and lastly, it writes the result back to `a[i]`.

With the exception of assigning to tuples and multiple targets in a single statement, the assignment done by augmented as-signment statements is handled the same way as normal assignments. Similarly, with the exception of the possible *in-place* be-havior, the binary operation performed by augmented assignment is the same as the normal binary operations.

For targets which are attribute references, the same [caveat about class and instance attributes](#) applies as for regular assignments.

7.2.2. Annotated assignment statements

[Annotation](#) assignment is the combination, in a single statement, of a variable or attribute annotation and an optional assign-ment statement:

```
annotated_assignment_stmt ::=   augtarget ":" expression
                                ["=" (starred_expression | yield_expression)]
```

The difference from normal [Assignment statements](#) is that only a single target is allowed.

For simple names as assignment targets, if in class or module scope, the annotations are evaluated and stored in a special class or module attribute `__annotations__` that is a dictionary mapping from variable names (mangled if private) to evaluated an-notations. This attribute is writable and is automatically created at the start of class or module body execution, if annotations are found statically.

For expressions as assignment targets, the annotations are evaluated if in class or module scope, but not stored.

If a name is annotated in a function scope, then this name is local for that scope. Annotations are never evaluated and stored in function scopes.

If the right hand side is present, an annotated assignment performs the actual assignment before evaluating annotations (where applicable). If the right hand side is not present for an expression target, then the interpreter evaluates the target ex-cept for the last `__setitem__()` or `__setattr__()` call.

> **See also:**
>
> [PEP 526](#) - **Syntax for Variable Annotations**
>   The proposal that added syntax for annotating the types of variables (including class variables and instance variables), instead of expressing them through comments.
>
> [PEP 484](#) - **Type hints**
>   The proposal that added the `typing` module to provide a standard syntax for type annotations that can be used in static analysis tools and IDEs.

Changed in version 3.8: Now annotated assignments allow the same expressions in the right hand side as regular assign-ments. Previously, some expressions (like un-parenthesized tuple expressions) caused a syntax error.

Assert statements are a convenient way to insert debugging assertions into a program:

```
assert_stmt ::=  "assert" expression ["," expression]
```

The simple form, `assert expression`, is equivalent to

```
if __debug__:
    if not expression: raise AssertionError
```

The extended form, `assert expression1, expression2`, is equivalent to

```
if __debug__:
    if not expression1: raise AssertionError(expression2)
```

These equivalences assume that `__debug__` and `AssertionError` refer to the built-in variables with those names. In the current implementation, the built-in variable `__debug__` is `True` under normal circumstances, `False` when optimization is requested (command line option `-O`). The current code generator emits no code for an assert statement when optimization is requested at compile time. Note that it is unnecessary to include the source code for the expression that failed in the error message; it will be displayed as part of the stack trace.

Assignments to `__debug__` are illegal. The value for the built-in variable is determined when the interpreter starts.

7.4. The pass statement

```
pass_stmt ::=  "pass"
```

`pass` is a null operation — when it is executed, nothing happens. It is useful as a placeholder when a statement is required syntactically, but no code needs to be executed, for example:

```
def f(arg): pass    # a function that does nothing (yet)

class C: pass       # a class with no methods (yet)
```

7.5. The del statement

```
del_stmt ::=  "del" target_list
```

Deletion is recursively defined very similar to the way assignment is defined. Rather than spelling it out in full details, here are some hints.

Deletion of a target list recursively deletes each target, from left to right.

Deletion of a name removes the binding of that name from the local or global namespace, depending on whether the name occurs in a `global` statement in the same code block. If the name is unbound, a `NameError` exception will be raised.

Deletion of attribute references, subscriptions and slicings is passed to the primary object involved; deletion of a slicing is in general equivalent to assignment of an empty slice of the right type (but even this is determined by the sliced object).

> *Changed in version 3.2:* Previously it was illegal to delete a name from the local namespace if it occurs as a free variable in a nested block.

```
return_stmt ::=  "return" [expression_list]
```

return may only occur syntactically nested in a function definition, not within a nested class definition.

If an expression list is present, it is evaluated, else None is substituted.

return leaves the current function call with the expression list (or None) as return value.

When return passes control out of a try statement with a finally clause, that finally clause is executed before really leaving the function.

In a generator function, the return statement indicates that the generator is done and will cause StopIteration to be raised. The returned value (if any) is used as an argument to construct StopIteration and becomes the StopIteration.value attribute.

In an asynchronous generator function, an empty return statement indicates that the asynchronous generator is done and will cause StopAsyncIteration to be raised. A non-empty return statement is a syntax error in an asynchronous generator function.

7.7. The yield statement

```
yield_stmt ::=  yield_expression
```

A yield statement is semantically equivalent to a yield expression. The yield statement can be used to omit the parentheses that would otherwise be required in the equivalent yield expression statement. For example, the yield statements

```
yield <expr>
yield from <expr>
```

are equivalent to the yield expression statements

```
(yield <expr>)
(yield from <expr>)
```

Yield expressions and statements are only used when defining a generator function, and are only used in the body of the generator function. Using yield in a function definition is sufficient to cause that definition to create a generator function instead of a normal function.

For full details of yield semantics, refer to the Yield expressions section.

7.8. The raise statement

```
raise_stmt ::=  "raise" [expression ["from" expression]]
```

If no expressions are present, raise re-raises the exception that is currently being handled, which is also known as the *active exception*. If there isn't currently an active exception, a RuntimeError exception is raised indicating that this is an error.

Otherwise, raise evaluates the first expression as the exception object. It must be either a subclass or an instance of BaseException. If it is a class, the exception instance will be obtained when needed by instantiating the class with no

The *type* of the exception is the exception instance's class, the *value* is the instance itself.

A traceback object is normally created automatically when an exception is raised and attached to it as the `__traceback__` attribute. You can create an exception and set your own traceback in one step using the `with_traceback()` exception method (which returns the same exception instance, with its traceback set to its argument), like so:

```
raise Exception("foo occurred").with_traceback(tracebackobj)
```

The `from` clause is used for exception chaining: if given, the second *expression* must be another exception class or instance. If the second expression is an exception instance, it will be attached to the raised exception as the `__cause__` attribute (which is writable). If the expression is an exception class, the class will be instantiated and the resulting exception instance will be attached to the raised exception as the `__cause__` attribute. If the raised exception is not handled, both exceptions will be printed:

```
>>> try:
...     print(1 / 0)
... except Exception as exc:
...     raise RuntimeError("Something bad happened") from exc
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    print(1 / 0)
          ~~^~~
ZeroDivisionError: division by zero

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
    raise RuntimeError("Something bad happened") from exc
RuntimeError: Something bad happened
```

A similar mechanism works implicitly if a new exception is raised when an exception is already being handled. An exception may be handled when an `except` or `finally` clause, or a `with` statement, is used. The previous exception is then attached as the new exception's `__context__` attribute:

```
>>> try:
...     print(1 / 0)
... except:
...     raise RuntimeError("Something bad happened")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    print(1 / 0)
          ~~^~~
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
    raise RuntimeError("Something bad happened")
RuntimeError: Something bad happened
```

Exception chaining can be explicitly suppressed by specifying None in the from clause:

```
... except:
...     raise RuntimeError("Something bad happened") from None
...
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: Something bad happened
```

Additional information on exceptions can be found in section Exceptions, and information about handling exceptions is in section The try statement.

> *Changed in version 3.3:* None is now permitted as Y in `raise X from Y`.
>
> Added the `__suppress_context__` attribute to suppress automatic display of the exception context.

> *Changed in version 3.11:* If the traceback of the active exception is modified in an except clause, a subsequent `raise` statement re-raises the exception with the modified traceback. Previously, the exception was re-raised with the traceback it had when it was caught.

7.9. The break statement

```
break_stmt ::=  "break"
```

break may only occur syntactically nested in a for or while loop, but not nested in a function or class definition within that loop.

It terminates the nearest enclosing loop, skipping the optional else clause if the loop has one.

If a for loop is terminated by break, the loop control target keeps its current value.

When break passes control out of a try statement with a finally clause, that finally clause is executed before really leaving the loop.

7.10. The continue statement

```
continue_stmt ::=  "continue"
```

continue may only occur syntactically nested in a for or while loop, but not nested in a function or class definition within that loop. It continues with the next cycle of the nearest enclosing loop.

When continue passes control out of a try statement with a finally clause, that finally clause is executed before really starting the next loop cycle.

7.11. The import statement

```
import_stmt     ::=  "import" module ["as" identifier] ("," module ["as" identifier])*
                     | "from" relative_module "import" identifier ["as" identifier]
                     ("," identifier ["as" identifier])*
                     | "from" relative_module "import" "(" identifier ["as" identifier]
                     ("," identifier ["as" identifier])* [","] ")"
                     | "from" relative_module "import" "*"
module          ::=  (identifier ".")* identifier
relative_module ::=  "."* module | "."+
```

1. find a module, loading and initializing it if necessary
2. define a name or names in the local namespace for the scope where the `import` statement occurs.

When the statement contains multiple clauses (separated by commas) the two steps are carried out separately for each clause, just as though the clauses had been separated out into individual import statements.

The details of the first step, finding and loading modules, are described in greater detail in the section on the import system, which also describes the various types of packages and modules that can be imported, as well as all the hooks that can be used to customize the import system. Note that failures in this step may indicate either that the module could not be located, *or* that an error occurred while initializing the module, which includes execution of the module's code.

If the requested module is retrieved successfully, it will be made available in the local namespace in one of three ways:

- If the module name is followed by `as`, then the name following `as` is bound directly to the imported module.
- If no other name is specified, and the module being imported is a top level module, the module's name is bound in the local namespace as a reference to the imported module
- If the module being imported is *not* a top level module, then the name of the top level package that contains the module is bound in the local namespace as a reference to the top level package. The imported module must be accessed using its full qualified name rather than directly

The `from` form uses a slightly more complex process:

1. find the module specified in the `from` clause, loading and initializing it if necessary;
2. for each of the identifiers specified in the `import` clauses:
   1. check if the imported module has an attribute by that name
   2. if not, attempt to import a submodule with that name and then check the imported module again for that attribute
   3. if the attribute is not found, `ImportError` is raised.
   4. otherwise, a reference to that value is stored in the local namespace, using the name in the `as` clause if it is present, otherwise using the attribute name

Examples:

```
import foo                 # foo imported and bound locally
import foo.bar.baz         # foo, foo.bar, and foo.bar.baz imported, foo bound locally
import foo.bar.baz as fbb  # foo, foo.bar, and foo.bar.baz imported, foo.bar.baz bound as fbb
from foo.bar import baz    # foo, foo.bar, and foo.bar.baz imported, foo.bar.baz bound as baz
from foo import attr       # foo imported and foo.attr bound as attr
```

If the list of identifiers is replaced by a star (`'*'`), all public names defined in the module are bound in the local namespace for the scope where the `import` statement occurs.

The *public names* defined by a module are determined by checking the module's namespace for a variable named `__all__`; if defined, it must be a sequence of strings which are names defined or imported by that module. The names given in `__all__` are all considered public and are required to exist. If `__all__` is not defined, the set of public names includes all names found in the module's namespace which do not begin with an underscore character (`'_'`). `__all__` should contain the entire public API. It is intended to avoid accidentally exporting items that are not part of the API (such as library modules which were imported and used within the module).

When specifying what module to import you do not have to specify the absolute name of the module. When a module or package is contained within another package it is possible to make a relative import within the same top package without having to mention the package name. By using leading dots in the specified module or package after `from` you can specify how high to traverse up the current package hierarchy without specifying exact names. One leading dot means the current package where the module making the import exists. Two dots means up one package level. Three dots is up two levels, etc. So if you execute `from . import mod` from a module in the `pkg` package then you will end up importing `pkg.mod`. If you execute `from ..subpkg2 import mod` from within `pkg.subpkg1` you will import `pkg.subpkg2.mod`. The specification for relative imports is contained in the [Package Relative Imports](#) section.

[`importlib.import_module()`](#) is provided to support applications that determine dynamically the modules to be loaded.

Raises an [auditing event](#) `import` with arguments `module, filename, sys.path, sys.meta_path, sys.path_hooks`.

7.11.1. Future statements

A *future statement* is a directive to the compiler that a particular module should be compiled using syntax or semantics that will be available in a specified future release of Python where the feature becomes standard.

The future statement is intended to ease migration to future versions of Python that introduce incompatible changes to the language. It allows use of the new features on a per-module basis before the release in which the feature becomes standard.

```
future_stmt ::=   "from" "__future__" "import" feature ["as" identifier]
                  ("," feature ["as" identifier])*
                  | "from" "__future__" "import" "(" feature ["as" identifier]
                  ("," feature ["as" identifier])* [","] ")"
feature     ::=   identifier
```

A future statement must appear near the top of the module. The only lines that can appear before a future statement are:

- the module docstring (if any),
- comments,
- blank lines, and
- other future statements.

The only feature that requires using the future statement is `annotations` (see **PEP 563**).

All historical features enabled by the future statement are still recognized by Python 3. The list includes `absolute_import`, `division, generators, generator_stop, unicode_literals, print_function, nested_scopes` and `with_statement`. They are all redundant because they are always enabled, and only kept for backwards compatibility.

A future statement is recognized and treated specially at compile time: Changes to the semantics of core constructs are often implemented by generating different code. It may even be the case that a new feature introduces new incompatible syntax (such as a new reserved word), in which case the compiler may need to parse the module differently. Such decisions cannot be pushed off until runtime.

For any given release, the compiler knows which feature names have been defined, and raises a compile-time error if a future statement contains a feature not known to it.

The interesting runtime semantics depend on the specific feature enabled by the future statement.

Note that there is nothing special about the statement:

```
import __future__ [as name]
```

That is not a future statement; it's an ordinary import statement with no special semantics or syntax restrictions.

Code compiled by calls to the built-in functions `exec()` and `compile()` that occur in a module M containing a future statement will, by default, use the new syntax or semantics associated with the future statement. This can be controlled by optional arguments to `compile()` — see the documentation of that function for details.

A future statement typed at an interactive interpreter prompt will take effect for the rest of the interpreter session. If an interpreter is started with the `-i` option, is passed a script name to execute, and the script includes a future statement, it will be in effect in the interactive session started after the script is executed.

> **See also:**
>
> **PEP 236** - **Back to the __future__**
>   The original proposal for the __future__ mechanism.

7.12. The `global` statement

```
global_stmt ::=  "global" identifier ("," identifier)*
```

The `global` statement is a declaration which holds for the entire current code block. It means that the listed identifiers are to be interpreted as globals. It would be impossible to assign to a global variable without `global`, although free variables may refer to globals without being declared global.

Names listed in a `global` statement must not be used in the same code block textually preceding that `global` statement.

Names listed in a `global` statement must not be defined as formal parameters, or as targets in `with` statements or `except` clauses, or in a `for` target list, `class` definition, function definition, `import` statement, or variable annotation.

CPython implementation detail: The current implementation does not enforce some of these restrictions, but programs should not abuse this freedom, as future implementations may enforce them or silently change the meaning of the program.

Programmer's note: `global` is a directive to the parser. It applies only to code parsed at the same time as the `global` statement. In particular, a `global` statement contained in a string or code object supplied to the built-in `exec()` function does not affect the code block *containing* the function call, and code contained in such a string is unaffected by `global` statements in the code containing the function call. The same applies to the `eval()` and `compile()` functions.

7.13. The `nonlocal` statement

```
nonlocal_stmt ::=  "nonlocal" identifier ("," identifier)*
```

When the definition of a function or class is nested (enclosed) within the definitions of other functions, its nonlocal scopes are the local scopes of the enclosing functions. The `nonlocal` statement causes the listed identifiers to refer to names previously

`SyntaxError` is raised.

The nonlocal statement applies to the entire scope of a function or class body. A `SyntaxError` is raised if a variable is used or assigned to prior to its nonlocal declaration in the scope.

> **See also:**
>
> **PEP 3104 - Access to Names in Outer Scopes**
> The specification for the `nonlocal` statement.

Programmer's note: `nonlocal` is a directive to the parser and applies only to code parsed along with it. See the note for the `global` statement.

7.14. The type statement

```
type_stmt ::=  'type' identifier [type_params] "=" expression
```

The `type` statement declares a type alias, which is an instance of `typing.TypeAliasType`.

For example, the following statement creates a type alias:

```
type Point = tuple[float, float]
```

This code is roughly equivalent to:

```
annotation-def VALUE_OF_Point():
    return tuple[float, float]
Point = typing.TypeAliasType("Point", VALUE_OF_Point())
```

`annotation-def` indicates an annotation scope, which behaves mostly like a function, but with several small differences.

The value of the type alias is evaluated in the annotation scope. It is not evaluated when the type alias is created, but only when the value is accessed through the type alias's `__value__` attribute (see Lazy evaluation). This allows the type alias to refer to names that are not yet defined.

Type aliases may be made generic by adding a type parameter list after the name. See Generic type aliases for more.

`type` is a soft keyword.

New in version 3.12.

> **See also:**
>
> **PEP 695 - Type Parameter Syntax**
> Introduced the `type` statement and syntax for generic classes and functions.