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
                   | match_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" assignment_expression ":" suite
             ("elif" assignment_expression ":" suite)*
             ["else" ":" suite]
```

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" assignment_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" starred_list ":" suite
              ["else" ":" suite]
```

The `starred_list` expression is evaluated once; it should yield an iterable object. An iterator is created for that iterable. The first item provided by the iterator is then assigned to the target list using the standard rules for assignments (see Assignment statements), and the suite is executed. This repeats for each item provided by the iterator. When the iterator is exhausted, 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 type `range()` represents immutable arithmetic sequences of integers. For instance, iterating `range(3)` successively yields 0, 1, and then 2.

> *Changed in version 3.11:* Starred elements are now allowed in the expression list.

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 | try3_stmt
try1_stmt ::=  "try" ":" suite
               ("except" [expression ["as" identifier]] ":" suite)+
               ["else" ":" suite]
```

```
                ("except" [expression ["as" identifier]] ":" suite)+
                ["else" ":" suite]
                ["finally" ":" suite]
try3_stmt ::=   "try" ":" suite
                "finally" ":" suite
```

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.

8.4.1. `except` clause

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 the object is the class or a non-virtual base class of the exception object, or a tuple containing an item that is the class or a non-virtual base class of the exception object.

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, the exception is stored in the `sys` module, where it can be accessed from within the body of the `except` clause by calling `sys.exception()`. When leaving an exception handler, the exception stored in the `sys` module is reset to its previous value:

```
>>> print(sys.exception())
None
```

```
...     print(repr(sys.exception()))
...     try:
...         raise ValueError
...     except:
...         print(repr(sys.exception()))
...     print(repr(sys.exception()))
...
TypeError()
ValueError()
TypeError()
>>> print(sys.exception())
None
```

8.4.2. except* clause

The except* clause(s) are used for handling ExceptionGroups. The exception type for matching is interpreted as in the case of except, but in the case of exception groups we can have partial matches when the type matches some of the exceptions in the group. This means that multiple except* clauses can execute, each handling part of the exception group. Each clause executes at most once and handles an exception group of all matching exceptions. Each exception in the group is handled by at most one except* clause, the first that matches it.

```
>>> try:
...     raise ExceptionGroup("eg",
...         [ValueError(1), TypeError(2), OSError(3), OSError(4)])
... except* TypeError as e:
...     print(f'caught {type(e)} with nested {e.exceptions}')
... except* OSError as e:
...     print(f'caught {type(e)} with nested {e.exceptions}')
...
caught <class 'ExceptionGroup'> with nested (TypeError(2),)
caught <class 'ExceptionGroup'> with nested (OSError(3), OSError(4))
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 2, in <module>
  | ExceptionGroup: eg
  +-+---------------- 1 ----------------
    | ValueError: 1
    +------------------------------------
```

Any remaining exceptions that were not handled by any except* clause are re-raised at the end, along with all exceptions that were raised from within the except* clauses. If this list contains more than one exception to reraise, they are combined into an exception group.

If the raised exception is not an exception group and its type matches one of the except* clauses, it is caught and wrapped by an exception group with an empty message string.

```
>>> try:
...     raise BlockingIOError
... except* BlockingIOError as e:
...     print(repr(e))
...
ExceptionGroup('', (BlockingIOError()))
```

An except* clause must have a matching type, and this type cannot be a subclass of BaseExceptionGroup. It is not possible to mix except and except* in the same try. break, continue and return cannot appear in an except* clause.

8.4.3. else clause

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

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'
```

> *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_stmt_contents ","? ")" | with_stmt_contents ) ":" suite
with_stmt_contents ::=  with_item ("," with_item)*
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 `__enter__()` is loaded for later use.
3. The context manager's `__exit__()` is loaded for later use.
4. The context manager's `__enter__()` method is invoked.
5. If a target was included in the `with` statement, the return value from `__enter__()` is assigned to it.

__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 7 below.

6. The suite is executed.

7. 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.

The following code:

```
with EXPRESSION as TARGET:
    SUITE
```

is semantically equivalent to:

```
manager = (EXPRESSION)
enter = type(manager).__enter__
exit = type(manager).__exit__
value = enter(manager)
hit_except = False

try:
    TARGET = value
    SUITE
except:
    hit_except = True
    if not exit(manager, *sys.exc_info()):
        raise
finally:
    if not hit_except:
        exit(manager, None, None, None)
```

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 semantically equivalent to:

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

You can also write multi-item context managers in multiple lines if the items are surrounded by parentheses. For example:

```
with (
    A() as a,
    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. The `match` statement

New in version 3.10.

The match statement is used for pattern matching. Syntax:

```
match_stmt   ::=  'match' subject_expr ":" NEWLINE INDENT case_block+ DEDENT
subject_expr ::=  star_named_expression "," star_named_expressions?
                  | named_expression
case_block   ::=  'case' patterns [guard] ":" block
```

Note: This section uses single quotes to denote soft keywords.

Pattern matching takes a pattern as input (following `case`) and a subject value (following `match`). The pattern (which may contain subpatterns) is matched against the subject value. The outcomes are:

- A match success or failure (also termed a pattern success or failure).
- Possible binding of matched values to a name. The prerequisites for this are further discussed below.

The `match` and `case` keywords are soft keywords.

See also:

- **PEP 634** – Structural Pattern Matching: Specification
- **PEP 636** – Structural Pattern Matching: Tutorial

8.6.1. Overview

Here's an overview of the logical flow of a match statement:

1. The subject expression `subject_expr` is evaluated and a resulting subject value obtained. If the subject expression contains a comma, a tuple is constructed using the standard rules.

2. Each pattern in a `case_block` is attempted to match with the subject value. The specific rules for success or failure are described below. The match attempt can also bind some or all of the standalone names within the pattern. The precise pattern binding rules vary per pattern type and are specified below. **Name bindings made during a successful pattern match outlive the executed block and can be used after the match statement**.

   Note: During failed pattern matches, some subpatterns may succeed. Do not rely on bindings being made for a failed match. Conversely, do not rely on variables remaining unchanged after a failed match. The exact behavior is dependent on implementation and may vary. This is an intentional decision made to allow different implementations to add optimizations.

3. If the pattern succeeds, the corresponding guard (if present) is evaluated. In this case all name bindings are guaranteed to have happened.

   - If the guard evaluates as true or is missing, the `block` inside `case_block` is executed.

Note: Users should generally never rely on a pattern being evaluated. Depending on implementation, the interpreter may cache values or use other optimizations which skip repeated evaluations.

A sample match statement:

```
>>> flag = False
>>> match (100, 200):
...     case (100, 300):  # Mismatch: 200 != 300
...         print('Case 1')
...     case (100, 200) if flag:  # Successful match, but guard fails
...         print('Case 2')
...     case (100, y):  # Matches and binds y to 200
...         print(f'Case 3, y: {y}')
...     case _:  # Pattern not attempted
...         print('Case 4, I match anything!')
...
Case 3, y: 200
```

In this case, `if flag` is a guard. Read more about that in the next section.

8.6.2. Guards

```
guard ::=  "if" named_expression
```

A `guard` (which is part of the `case`) must succeed for code inside the `case` block to execute. It takes the form: `if` followed by an expression.

The logical flow of a `case` block with a `guard` follows:

1. Check that the pattern in the `case` block succeeded. If the pattern failed, the `guard` is not evaluated and the next `case` block is checked.
2. If the pattern succeeded, evaluate the `guard`.
   - If the `guard` condition evaluates as true, the case block is selected.
   - If the `guard` condition evaluates as false, the case block is not selected.
   - If the `guard` raises an exception during evaluation, the exception bubbles up.

Guards are allowed to have side effects as they are expressions. Guard evaluation must proceed from the first to the last case block, one at a time, skipping case blocks whose pattern(s) don't all succeed. (I.e., guard evaluation must happen in order.) Guard evaluation must stop once a case block is selected.

8.6.3. Irrefutable Case Blocks

An irrefutable case block is a match-all case block. A match statement may have at most one irrefutable case block, and it must be last.

A case block is considered irrefutable if it has no guard and its pattern is irrefutable. A pattern is considered irrefutable if we can prove from its syntax alone that it will always succeed. Only the following patterns are irrefutable:

- AS Patterns whose left-hand side is irrefutable
- OR Patterns containing at least one irrefutable pattern
- Capture Patterns
- Wildcard Patterns
- parenthesized irrefutable patterns

Note: This section uses grammar notations beyond standard EBNF:

- the notation `SEP.RULE+` is shorthand for `RULE (SEP RULE)*`
- the notation `!RULE` is shorthand for a negative lookahead assertion

The top-level syntax for `patterns` is:

```
patterns        ::=  open_sequence_pattern | pattern
pattern         ::=  as_pattern | or_pattern
closed_pattern  ::=  | literal_pattern
                     | capture_pattern
                     | wildcard_pattern
                     | value_pattern
                     | group_pattern
                     | sequence_pattern
                     | mapping_pattern
                     | class_pattern
```

The descriptions below will include a description “in simple terms” of what a pattern does for illustration purposes (credits to Raymond Hettinger for a document that inspired most of the descriptions). Note that these descriptions are purely for illustration purposes and **may not** reflect the underlying implementation. Furthermore, they do not cover all valid forms.

8.6.4.1. OR Patterns

An OR pattern is two or more patterns separated by vertical bars `|`. Syntax:

```
or_pattern ::=  "|".closed_pattern+
```

Only the final subpattern may be irrefutable, and each subpattern must bind the same set of names to avoid ambiguity.

An OR pattern matches each of its subpatterns in turn to the subject value, until one succeeds. The OR pattern is then considered successful. Otherwise, if none of the subpatterns succeed, the OR pattern fails.

In simple terms, `P1 | P2 | ...` will try to match `P1`, if it fails it will try to match `P2`, succeeding immediately if any succeeds, failing otherwise.

8.6.4.2. AS Patterns

An AS pattern matches an OR pattern on the left of the `as` keyword against a subject. Syntax:

```
as_pattern ::=  or_pattern "as" capture_pattern
```

If the OR pattern fails, the AS pattern fails. Otherwise, the AS pattern binds the subject to the name on the right of the as keyword and succeeds. `capture_pattern` cannot be a `_`.

In simple terms `P as NAME` will match with `P`, and on success it will set `NAME = <subject>`.

8.6.4.3. Literal Patterns

A literal pattern corresponds to most literals in Python. Syntax:

```
literal_pattern ::=  signed_number
                     | signed_number "+" NUMBER
                     | signed_number "-" NUMBER
                     | strings
```

```
| "false"
| signed_number: NUMBER | "-" NUMBER
```

The rule `strings` and the token `NUMBER` are defined in the standard Python grammar. Triple-quoted strings are supported. Raw strings and byte strings are supported. f-strings are not supported.

The forms `signed_number '+' NUMBER` and `signed_number '-' NUMBER` are for expressing complex numbers; they require a real number on the left and an imaginary number on the right. E.g. `3 + 4j`.

In simple terms, `LITERAL` will succeed only if `<subject> == LITERAL`. For the singletons `None`, `True` and `False`, the `is` operator is used.

8.6.4.4. Capture Patterns

A capture pattern binds the subject value to a name. Syntax:

```
capture_pattern ::=  !'_' NAME
```

A single underscore `_` is not a capture pattern (this is what `!'_'` expresses). It is instead treated as a `wildcard_pattern`.

In a given pattern, a given name can only be bound once. E.g. `case x, x: ...` is invalid while `case [x] | x: ...` is allowed.

Capture patterns always succeed. The binding follows scoping rules established by the assignment expression operator in **PEP 572**; the name becomes a local variable in the closest containing function scope unless there's an applicable `global` or `nonlocal` statement.

In simple terms `NAME` will always succeed and it will set `NAME = <subject>`.

8.6.4.5. Wildcard Patterns

A wildcard pattern always succeeds (matches anything) and binds no name. Syntax:

```
wildcard_pattern ::=  '_'
```

`_` is a soft keyword within any pattern, but only within patterns. It is an identifier, as usual, even within `match` subject expressions, `guards`, and `case` blocks.

In simple terms, `_` will always succeed.

8.6.4.6. Value Patterns

A value pattern represents a named value in Python. Syntax:

```
value_pattern ::=  attr
attr          ::=  name_or_attr "." NAME
name_or_attr  ::=  attr | NAME
```

The dotted name in the pattern is looked up using standard Python name resolution rules. The pattern succeeds if the value found compares equal to the subject value (using the `==` equality operator).

In simple terms `NAME1.NAME2` will succeed only if `<subject> == NAME1.NAME2`

Note: If the same value occurs multiple times in the same match statement, the interpreter may cache the first value found and reuse it rather than repeat the same lookup. This cache is strictly tied to a given execution of a given match

8.6.4.7. Group Patterns

A group pattern allows users to add parentheses around patterns to emphasize the intended grouping. Otherwise, it has no additional syntax. Syntax:

```
group_pattern ::=  "(" pattern ")"
```

In simple terms `(P)` has the same effect as `P`.

8.6.4.8. Sequence Patterns

A sequence pattern contains several subpatterns to be matched against sequence elements. The syntax is similar to the unpacking of a list or tuple.

```
sequence_pattern       ::=  "[" [maybe_sequence_pattern] "]"
                          | "(" [open_sequence_pattern] ")"
open_sequence_pattern  ::=  maybe_star_pattern "," [maybe_sequence_pattern]
maybe_sequence_pattern ::=  ",".maybe_star_pattern+ ","?
maybe_star_pattern     ::=  star_pattern | pattern
star_pattern           ::=  "*" (capture_pattern | wildcard_pattern)
```

There is no difference if parentheses or square brackets are used for sequence patterns (i.e. `(...)` vs `[...]`).

Note: A single pattern enclosed in parentheses without a trailing comma (e.g. `(3 | 4)`) is a group pattern. While a single pattern enclosed in square brackets (e.g. `[3 | 4]`) is still a sequence pattern.

At most one star subpattern may be in a sequence pattern. The star subpattern may occur in any position. If no star subpattern is present, the sequence pattern is a fixed-length sequence pattern; otherwise it is a variable-length sequence pattern.

The following is the logical flow for matching a sequence pattern against a subject value:

1. If the subject value is not a sequence [2], the sequence pattern fails.
2. If the subject value is an instance of `str`, `bytes` or `bytearray` the sequence pattern fails.
3. The subsequent steps depend on whether the sequence pattern is fixed or variable-length.

   If the sequence pattern is fixed-length:

   1. If the length of the subject sequence is not equal to the number of subpatterns, the sequence pattern fails
   2. Subpatterns in the sequence pattern are matched to their corresponding items in the subject sequence from left to right. Matching stops as soon as a subpattern fails. If all subpatterns succeed in matching their corresponding item, the sequence pattern succeeds.

   Otherwise, if the sequence pattern is variable-length:

   1. If the length of the subject sequence is less than the number of non-star subpatterns, the sequence pattern fails.
   2. The leading non-star subpatterns are matched to their corresponding items as for fixed-length sequences.
   3. If the previous step succeeds, the star subpattern matches a list formed of the remaining subject items, excluding the remaining items corresponding to non-star subpatterns following the star subpattern.
   4. Remaining non-star subpatterns are matched to their corresponding subject items, as for a fixed-length sequence.

In simple terms `[P1, P2, P3, ... , P<N>]` matches only if all the following happens:

- check `<subject>` is a sequence
- `len(subject) == <N>`
- `P1` matches `<subject>[0]` (note that this match can also bind names)
- `P2` matches `<subject>[1]` (note that this match can also bind names)
- ... and so on for the corresponding pattern/element.

8.6.4.9. Mapping Patterns

A mapping pattern contains one or more key-value patterns. The syntax is similar to the construction of a dictionary. Syntax:

```
mapping_pattern     ::=  "{" [items_pattern] "}"
items_pattern       ::=  ",".key_value_pattern+ ","?
key_value_pattern   ::=  (literal_pattern | value_pattern) ":" pattern
                         | double_star_pattern
double_star_pattern ::=  "**" capture_pattern
```

At most one double star pattern may be in a mapping pattern. The double star pattern must be the last subpattern in the mapping pattern.

Duplicate keys in mapping patterns are disallowed. Duplicate literal keys will raise a `SyntaxError`. Two keys that otherwise have the same value will raise a `ValueError` at runtime.

The following is the logical flow for matching a mapping pattern against a subject value:

1. If the subject value is not a mapping [3],the mapping pattern fails.
2. If every key given in the mapping pattern is present in the subject mapping, and the pattern for each key matches the corresponding item of the subject mapping, the mapping pattern succeeds.
3. If duplicate keys are detected in the mapping pattern, the pattern is considered invalid. A `SyntaxError` is raised for duplicate literal values; or a `ValueError` for named keys of the same value.

Note: Key-value pairs are matched using the two-argument form of the mapping subject's `get()` method. Matched key-value pairs must already be present in the mapping, and not created on-the-fly via `__missing__()` or `__getitem__()`.

In simple terms `{KEY1: P1, KEY2: P2, ... }` matches only if all the following happens:

- check `<subject>` is a mapping
- `KEY1 in <subject>`
- `P1` matches `<subject>[KEY1]`
- ... and so on for the corresponding KEY/pattern pair.

8.6.4.10. Class Patterns

A class pattern represents a class and its positional and keyword arguments (if any). Syntax:

```
class_pattern       ::=  name_or_attr "(" [pattern_arguments ","?] ")"
pattern_arguments   ::=  positional_patterns ["," keyword_patterns]
                         | keyword_patterns
positional_patterns ::=  ",".pattern+
```

The same keyword should not be repeated in class patterns.

The following is the logical flow for matching a class pattern against a subject value:

1. If `name_or_attr` is not an instance of the builtin `type` , raise `TypeError`.

2. If the subject value is not an instance of `name_or_attr` (tested via `isinstance()`), the class pattern fails.

3. If no pattern arguments are present, the pattern succeeds. Otherwise, the subsequent steps depend on whether keyword or positional argument patterns are present.

   For a number of built-in types (specified below), a single positional subpattern is accepted which will match the entire subject; for these types keyword patterns also work as for other types.

   If only keyword patterns are present, they are processed as follows, one by one:

   I. The keyword is looked up as an attribute on the subject.

   - If this raises an exception other than `AttributeError`, the exception bubbles up.
   - If this raises `AttributeError`, the class pattern has failed.
   - Else, the subpattern associated with the keyword pattern is matched against the subject's attribute value. If this fails, the class pattern fails; if this succeeds, the match proceeds to the next keyword.

   II. If all keyword patterns succeed, the class pattern succeeds.

   If any positional patterns are present, they are converted to keyword patterns using the `__match_args__` attribute on the class `name_or_attr` before matching:

   I. The equivalent of `getattr(cls, "__match_args__", ())` is called.

   - If this raises an exception, the exception bubbles up.
   - If the returned value is not a tuple, the conversion fails and `TypeError` is raised.
   - If there are more positional patterns than `len(cls.__match_args__)`, `TypeError` is raised.
   - Otherwise, positional pattern `i` is converted to a keyword pattern using `__match_args__[i]` as the keyword. `__match_args__[i]` must be a string; if not `TypeError` is raised.
   - If there are duplicate keywords, `TypeError` is raised.

   > **See also:** Customizing positional arguments in class pattern matching

   II. Once all positional patterns have been converted to keyword patterns,
   the match proceeds as if there were only keyword patterns.

   For the following built-in types the handling of positional subpatterns is different:

- `bool`
- `bytearray`
- `bytes`
- `dict`
- `float`
- `frozenset`
- `int`
- `list`

- tuple

These classes accept a single positional argument, and the pattern there is matched against the whole object rather than an attribute. For example `int(0|1)` matches the value `0`, but not the value `0.0`.

In simple terms `CLS(P1, attr=P2)` matches only if the following happens:

- `isinstance(<subject>, CLS)`
- convert `P1` to a keyword pattern using `CLS.__match_args__`
- For each keyword argument `attr=P2`:
  - `hasattr(<subject>, "attr")`
  - `P2` matches `<subject>.attr`
- ... and so on for the corresponding keyword argument/pattern pair.

See also:

- **PEP 634** – Structural Pattern Matching: Specification
- **PEP 636** – Structural Pattern Matching: Tutorial

8.7. Function definitions

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

```
funcdef                 ::=  [decorators] "def" funcname [type_params] "(" [parameter_list] ")"
                             ["->" expression] ":" suite
decorators              ::=  decorator+
decorator               ::=  "@" assignment_expression NEWLINE
parameter_list          ::=  defparameter ("," defparameter)* "," "/" ["," [parameter_list_no_posonly
                               | parameter_list_no_posonly
parameter_list_no_posonly ::=  defparameter ("," defparameter)* ["," [parameter_list_starargs]]
                             | parameter_list_starargs
parameter_list_starargs ::=  "*" [parameter] ("," defparameter)* ["," ["**" parameter [","]]]
                             | "**" 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. [4]

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

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

> *Changed in version 3.9:* Functions may be decorated with any valid `assignment_expression`. Previously, the grammar was much more restrictive; see **PEP 614** for details.

A list of type parameters may be given in square brackets between the function's name and the opening parenthesis for its parameter list. This indicates to static type checkers that the function is generic. At runtime, the type parameters can be retrieved from the function's `__type_params__` attribute. See Generic functions for more.

> *Changed in version 3.12:* Type parameter lists are new in Python 3.12.

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 value 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 parameter 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 positional 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 by keyword arguments. Parameters before "`/`" are positional-only parameters and may only be passed by positional arguments.

> *Changed in version 3.8:* The `/` function parameter syntax may be used to indicate positional-only parameters. See **PEP 570** for details.

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 sim-

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.

PEP 318 - Decorators for Functions and Methods
Function and method decorators were introduced. Class decorators were introduced in **PEP 3129**.

8.8. Class definitions

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

```
classdef    ::=  [decorators] "class" classname [type_params] [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. [5] 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.

```
@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.

> *Changed in version 3.9:* Classes may be decorated with any valid `assignment_expression`. Previously, the grammar was much more restrictive; see **PEP 614** for details.

A list of type parameters may be given in square brackets immediately after the class's name. This indicates to static type checkers that the class is generic. At runtime, the type parameters can be retrieved from the class's `__type_params__` attribute. See Generic classes for more.

> *Changed in version 3.12:* Type parameter lists are new in Python 3.12.

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**.

8.9. Coroutines

> *New in version 3.5.*

8.9.1. 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). `await` expressions, `async for` and `async with` can only be used in the body of a coroutine function.

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:

```
    await some_coroutine()
```

> *Changed in version 3.7:* `await` and `async` are now keywords; previously they were only treated as such inside the body of a coroutine function.

8.9.2. The `async for` statement

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

An asynchronous iterable provides an `__aiter__` method that directly returns an asynchronous iterator, which can call asynchronous code in its `__anext__` method.

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

The following code:

```
async for TARGET in ITER:
    SUITE
else:
    SUITE2
```

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:
        SUITE
else:
    SUITE2
```

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

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

8.9.3. 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 EXPRESSION as TARGET:
    SUITE
```

is semantically equivalent to:

```
manager = (EXPRESSION)
aenter = type(manager).__aenter__
aexit = type(manager).__aexit__
value = await aenter(manager)
hit_except = False
```

```
    TARGET = value
    SUITE
except:
    hit_except = True
    if not await aexit(manager, *sys.exc_info()):
        raise
finally:
    if not hit_except:
        await aexit(manager, 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.

8.10. Type parameter lists

New in version 3.12.

```
type_params  ::=  "[" type_param ("," type_param)* "]"
type_param   ::=  typevar | typevartuple | paramspec
typevar      ::=  identifier (":" expression)?
typevartuple ::=  "*" identifier
paramspec    ::=  "**" identifier
```

Functions (including coroutines), classes and type aliases may contain a type parameter list:

```
def max[T](args: list[T]) -> T:
    ...

async def amax[T](args: list[T]) -> T:
    ...

class Bag[T]:
    def __iter__(self) -> Iterator[T]:
        ...

    def add(self, arg: T) -> None:
        ...

type ListOrSet[T] = list[T] | set[T]
```

Semantically, this indicates that the function, class, or type alias is generic over a type variable. This information is primarily used by static type checkers, and at runtime, generic objects behave much like their non-generic counterparts.

Type parameters are declared in square brackets (`[]`) immediately after the name of the function, class, or type alias. The type parameters are accessible within the scope of the generic object, but not elsewhere. Thus, after a declaration `def func[T](): pass`, the name `T` is not available in the module scope. Below, the semantics of generic objects are described with more precision. The scope of type parameters is modeled with a special function (technically, an annotation scope) that wraps the creation of the generic object.

Generic functions, classes, and type aliases have a `__type_params__` attribute listing their type parameters.

Type parameters come in three kinds:

tuple of any number of types.

- typing.ParamSpec, introduced by a name prefixed with two asterisks (e.g., **P). Semantically, this stands for the parameters of a callable.

typing.TypeVar declarations can define *bounds* and *constraints* with a colon (:) followed by an expression. A single expression after the colon indicates a bound (e.g. T: int). Semantically, this means that the typing.TypeVar can only represent types that are a subtype of this bound. A parenthesized tuple of expressions after the colon indicates a set of constraints (e.g. T: (str, bytes)). Each member of the tuple should be a type (again, this is not enforced at runtime). Constrained type variables can only take on one of the types in the list of constraints.

For typing.TypeVars declared using the type parameter list syntax, the bound and constraints are not evaluated when the generic object is created, but only when the value is explicitly accessed through the attributes __bound__ and __constraints__. To accomplish this, the bounds or constraints are evaluated in a separate annotation scope.

typing.TypeVarTuples and typing.ParamSpecs cannot have bounds or constraints.

The following example indicates the full set of allowed type parameter declarations:

```
def overly_generic[
   SimpleTypeVar,
   TypeVarWithBound: int,
   TypeVarWithConstraints: (str, bytes),
   *SimpleTypeVarTuple,
   **SimpleParamSpec,
](
   a: SimpleTypeVar,
   b: TypeVarWithBound,
   c: Callable[SimpleParamSpec, TypeVarWithConstraints],
   *d: SimpleTypeVarTuple,
): ...
```

8.10.1. Generic functions

Generic functions are declared as follows:

```
def func[T](arg: T): ...
```

This syntax is equivalent to:

```
annotation-def TYPE_PARAMS_OF_func():
    T = typing.TypeVar("T")
    def func(arg: T): ...
    func.__type_params__ = (T,)
    return func
func = TYPE_PARAMS_OF_func()
```

Here annotation-def indicates an annotation scope, which is not actually bound to any name at runtime. (One other liberty is taken in the translation: the syntax does not go through attribute access on the typing module, but creates an instance of typing.TypeVar directly.)

The annotations of generic functions are evaluated within the annotation scope used for declaring the type parameters, but the function's defaults and decorators are not.

The following example illustrates the scoping rules for these cases, as well as for additional flavors of type parameters:

```
    ...
```

Except for the lazy evaluation of the `TypeVar` bound, this is equivalent to:

```
DEFAULT_OF_arg = some_default

annotation-def TYPE_PARAMS_OF_func():

    annotation-def BOUND_OF_T():
        return int
    # In reality, BOUND_OF_T() is evaluated only on demand.
    T = typing.TypeVar("T", bound=BOUND_OF_T())

    Ts = typing.TypeVarTuple("Ts")
    P = typing.ParamSpec("P")

    def func(*args: *Ts, arg: Callable[P, T] = DEFAULT_OF_arg):
        ...

    func.__type_params__ = (T, Ts, P)
    return func
func = decorator(TYPE_PARAMS_OF_func())
```

The capitalized names like `DEFAULT_OF_arg` are not actually bound at runtime.

8.10.2. Generic classes

Generic classes are declared as follows:

```
class Bag[T]: ...
```

This syntax is equivalent to:

```
annotation-def TYPE_PARAMS_OF_Bag():
    T = typing.TypeVar("T")
    class Bag(typing.Generic[T]):
        __type_params__ = (T,)
        ...
    return Bag
Bag = TYPE_PARAMS_OF_Bag()
```

Here again `annotation-def` (not a real keyword) indicates an annotation scope, and the name `TYPE_PARAMS_OF_Bag` is not actually bound at runtime.

Generic classes implicitly inherit from `typing.Generic`. The base classes and keyword arguments of generic classes are evaluated within the type scope for the type parameters, and decorators are evaluated outside that scope. This is illustrated by this example:

```
@decorator
class Bag(Base[T], arg=T): ...
```

This is equivalent to:

```
annotation-def TYPE_PARAMS_OF_Bag():
    T = typing.TypeVar("T")
    class Bag(Base[T], typing.Generic[T], arg=T):
        __type_params__ = (T,)
        ...
    return Bag
Bag = decorator(TYPE_PARAMS_OF_Bag())
```

The type statement can also be used to create a generic type alias:

```
type ListOrSet[T] = list[T] | set[T]
```

Except for the lazy evaluation of the value, this is equivalent to:

```
annotation-def TYPE_PARAMS_OF_ListOrSet():
    T = typing.TypeVar("T")

    annotation-def VALUE_OF_ListOrSet():
        return list[T] | set[T]
    # In reality, the value is lazily evaluated
    return typing.TypeAliasType("ListOrSet", VALUE_OF_ListOrSet(), type_params=(T,))
ListOrSet = TYPE_PARAMS_OF_ListOrSet()
```

Here, `annotation-def` (not a real keyword) indicates an annotation scope. The capitalized names like `TYPE_PARAMS_OF_ListOrSet` are not actually bound at runtime.

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] In pattern matching, a sequence is defined as one of the following:

- a class that inherits from `collections.abc.Sequence`
- a Python class that has been registered as `collections.abc.Sequence`
- a builtin class that has its (CPython) `Py_TPFLAGS_SEQUENCE` bit set
- a class that inherits from any of the above

The following standard library classes are sequences:

- `array.array`
- `collections.deque`
- `list`
- `memoryview`
- `range`
- `tuple`

Note: Subject values of type `str`, `bytes`, and `bytearray` do not match sequence patterns.

[3] In pattern matching, a mapping is defined as one of the following:

- a class that inherits from `collections.abc.Mapping`
- a Python class that has been registered as `collections.abc.Mapping`
- a builtin class that has its (CPython) `Py_TPFLAGS_MAPPING` bit set
- a class that inherits from any of the above

The standard library classes `dict` and `types.MappingProxyType` are mappings.

[4] 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.