# 8. 复合语句

复合语句是包含其它语句（语句组）的语句；它们会以某种方式影响或控制所包含其它语句的执行。 通常，复合语句会跨越多行，虽然在某些简单形式下整个复合语句也可能包含于一行之内。

`if`, `while` 和 `for` 语句用来实现传统的控制流程构造。 `try` 语句为一组语句指定异常处理和/和清理代码，而 `with` 语句允许在一个代码块周围执行初始化和终结化代码。 函数和类定义在语法上也属于复合语句。

一条复合语句由一个或多个'子句'组成。 一个子句则包含一个句头和一个'句体'。 特定复合语句的子句头都处于相同的缩进层级。 每个子句头以一个作为唯一标识的关键字开始并以一个冒号结束。 子句体是由一个子句控制的一组语句。 子句体可以是在子句头的冒号之后与其同处一行的一条或由分号分隔的多条简单语句，或者也可以是在其之后缩进的一行或多行语句。 只有后一种形式的子句体才能包含嵌套的复合语句；以下形式是不合法的，这主要是因为无法分清某个后续的 `else` 子句应该属于哪个 `if` 子句:

```
if test1: if test2: print(x)
```

还要注意的是在这种情形下分号的绑定比冒号更紧密，因此在以下示例中，所有 `print()` 调用或者都不执行，或者都执行:

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

总结:

```
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)* [";"]
```

请注意语句总是以 `NEWLINE` 结束，之后可能跟随一个 `DEDENT`。 还要注意可选的后续子句总是以一个不能作为语句开头的关键字作为开头，因此不会产生歧义（'悬空的 `else`'问题在 Python 中是通过要求嵌套的 `if` 语句必须缩进来解决的）。

为了保证清晰，以下各节中语法规则采用将每个子句都放在单独行中的格式。

## 8.1. if 语句

if 语句用于有条件的执行:

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

它通过对表达式逐个求值直至找到一个真值（请参阅 布尔运算 了解真值与假值的定义）在子句体中选择唯一匹配的一个；然后执行该子句体（而且 if 语句的其他部分不会被执行或求值）。 如果所有表达式均为假值，则如果 else 子句体如果存在就会被执行。

## 8.2. while 语句

while 语句用于在表达式保持为真的情况下重复地执行:

```
while_stmt: "while" assignment_expression ":" suite
            ["else" ":" suite]
```

这将重复地检验表达式，并且如果其值为真就执行第一个子句体；如果表达式值为假（这可能在第一次检验时就发生）则如果 else 子句体存在就会被执行并终止循环。

第一个子句体中的 break 语句在执行时将终止循环且不执行 else 子句体。 第一个子句体中的 continue 语句在执行时将跳过子句体中的剩余部分并返回检验表达式。

## 8.3. for 语句

for 语句用于对序列（例如字符串、元组或列表）或其他可迭代对象中的元素进行迭代:

```
for_stmt: "for" target_list "in" starred_expression_list ":" suite
          ["else" ":" suite]
```

starred_expression_list 表达式会被求值一次；它应当产生一个 iterable 对象。 将针对该可迭代对象创建一个 iterator。 随后该迭代器所提供的第一个条目将使用标准的赋值规则被赋值给目标列表 (参见 赋值语句)，而代码块将被执行。 此过程将针对该迭代器所提供每个条目重复进行。 当迭代器被耗尽时，如果存在 else 子句中的代码块，则它将被执行，并终结循环。

第一个子句体中的 break 语句在执行时将终止循环且不执行 else 子句体。 第一个子句体中的 continue 语句在执行时将跳过子句体中的剩余部分并转往下一项继续执行，或者在没有下一项时转往 else 子句执行。

for 循环会对目标列表中的变量进行赋值。 这将覆盖之前对这些变量的所有赋值，包括在 for 循环体中的赋值:

```
for i in range(10):
    print(i)
    i = 5             # 这不会影响 for 循环
```

```
                # 因为它将被 range 对象中的下一个索引
                # 所覆盖
```

目标列表中的名称在循环结束时不会被删除，但是如果序列为空，则它们将根本不会被循环所赋值。 提示：内置类型 range() 代表由整数组成的不可变算数序列。 例如，迭代 range(3) 将依次产生 0, 1 和 2。

> *在 3.11 版本发生变更:* 现在允许在表达式列表中使用带星号的元素。

## 8.4. try 语句

try 语句可为一组语句指定异常处理器和/或清理代码:

```
try_stmt:  try1_stmt | try2_stmt | try3_stmt
try1_stmt: "try" ":" suite
           ("except" [expression ["as" identifier]] ":" suite)+
           ["else" ":" suite]
           ["finally" ":" suite]
try2_stmt: "try" ":" suite
           ("except" "*" expression ["as" identifier] ":" suite)+
           ["else" ":" suite]
           ["finally" ":" suite]
try3_stmt: "try" ":" suite
           "finally" ":" suite
```

有关异常的更多信息可以在 异常 一节找到，有关使用 raise 语句生成异常的信息可以在 raise 语句 一节找到。

> *在 3.14 版本发生变更:* 支持在使用多个异常类型时可以选择略去分组圆括号。 参见 **PEP 758**。

### 8.4.1. except 子句

except 子句指定一个或多个异常处理器。 当在 try 子句中未发生异常时，将不会执行任何异常处理器。 当在 try 语句块中发生异常时，将启动对异常处理器的搜索。 此搜索会依次检查 except 子句直至找到与异常相匹配的处理器。 不带表达式的 except 子句如果存在，则它必须是最后一个；它将匹配任何异常。

对于带有表达式的 except 子句，该表达式必须被求值为一个异常类型或是由异常类型组成的元组。 如果提供了多个异常类型并且没有使用``as``子句，则可以去掉括号。引发的异常匹配一个 except 子句，其表达式的计算结果为异常对象的类或 非虚基类，或包含此类的元组。

如果没有 except 子句与异常相匹配，则会在周边代码和唤起栈上继续搜索异常处理器。 [1]

如果在对 except 子句头部的表达式求值时引发了异常，则对处理器的原始搜索会被取消并在周边代码和调用栈上启动对新异常的搜索（它会被视作是整个 try 语句所引发的异常）。

当代到一个匹配的 except 子句时，异常将被赋值给该 except 子句在 as 关键字之后指定的目标，如果存在此关键字的话，并且该 except 子句的代码块将被执行。 所有 except 子句都必须有可执行的代码块。 当到达此类代码块的末尾时，通常会转到整个 try 语句之后继续执行。 （这意味着如

果对同一异常存在两个嵌套的处理器，并且异常发生在内层处理器的 try 子句中，则外层处理器将不会处理该异常。）

当使用 as target 来为异常赋值时，它将在 except 子句结束时被清除。 这就相当于

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

被转写为

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

这意味着异常必须被赋值给一个不同的名称才能在 except 子句之后引用它。 异常会被清除是因为在附加了回溯信息的情况下它们会形成栈帧的循环引用，使得帧中的所有局部变量保持存活直到发生下一次垃圾回收。

在 except 子句的代码块被执行之前，异常将保存在 sys 模块中，在那里它可以从 except 子句的语句体内部通过 sys.exception() 被访问。 当离开一个异常处理器时，保存在 sys 模块中的异常将被重置为在此之前的值:

```
>>> print(sys.exception())
None
>>> try:
...     raise TypeError
... except:
...     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* 子句

except* 子句针对异常组 (BaseExceptionGroup 实例) 指定一个或多个处理器。 一条 try 语句可以具有 except 或 except* 子句，但不可同时存在。 用于匹配的异常类型对于 except* 来说是强制性的，因此 except*: 将导致语法错误。 异常类型将如在 except 中一样被解读，但匹配则是在包含于被处理分组中的异常上执行的。 如果匹配类型为 BaseExceptionGroup 的子类则会引发 TypeError，因为其中存在模糊的语义。

当一个异常分组在 try 代码块中被引发时，每个 except* 子句会将其拆分 (参见 split()) 为匹配和不匹配异常的子分组。 如果匹配子分组不为空，它将成为被处理的异常 (从 sys.exception() 返回

的值) 并分配给 except* 子句 (如果存在) 的目标。 随后，except* 子句体将被执行。 如果不匹配分组不为空，它将由下一个 except* 以同样的方式进行处理。 此过程将持续至分组中的所有异常都已被匹配，或者最后一个 except* 子句运行完成。

After all except* clauses execute, the group of unhandled exceptions is merged with any exceptions that were raised or re-raised from within except* clauses. This merged exception group propagates on.:

```
>>> 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 "<doctest default[0]>", line 2, in <module>
  |     raise ExceptionGroup("eg",
  |         [ValueError(1), TypeError(2), OSError(3), OSError(4)])
  | ExceptionGroup: eg (1 sub-exception)
  +-+---------------- 1 ----------------
    | ValueError: 1
    +------------------------------------
```

如果从 try 代码块引发的异常不是一个异常组并且其类型与某个 except* 子句相匹配，它将被捕获并由附带空消息字符串的异常组来包装。 这将确保目标 e 的类型与一定为 BaseExceptionGroup:

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

break, continue 和 return 不可在 except* 子句中出现。

### 8.4.3. else 子句

如果控制流离开 try 子句体时没有引发异常，并且没有执行 return, continue 或 break 语句，可选的 else 子句将被执行。 else 语句中的异常不会由之前的 except 子句处理。

### 8.4.4. finally 子句

如果存在 finally，它将指定一个“清理”处理器。 try 子句会被执行，包括任何 except 和 else 子句。 如果在这些子句中发生任何未处理的异常，该异常会被临时保存。 finally 子句将被执行。 如果存在被保存的异常，它会在 finally 子句的末尾被重新引发。 如果 finally 子句引发了另一个异常，被保存的异常会被设为新异常的上下文。 如果 finally 子句执行了 return, break 或 continue 语句，则被保存的异常会被丢弃。例如，这个函数返回42。

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

在 finally 子句执行期间程序将不能获取到异常信息。

当 return, break 或 continue 语句在一个 try…finally 语句的 try 子句的代码块中被执行时，finally 子句也会在'离开时'被执行。

函数的返回值是由最后被执行的 return 语句来决定的。 由于 finally 子句总是会被执行，因此在 finally 子句中被执行的 return 语句将总是最后被执行的。下面的函数返回"finally"。

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

> *在 3.8 版本发生变更:* 在 Python 3.8 之前， continue 语句不允许在 finally 子句中使用，这是因为具体实现中存在一个问题。

> *在 3.14 版本发生变更:* 当 return, break 或 continue 在 finally 代码块中出现时编译器将发出 SyntaxWarning (参见 **PEP 765**)。

## 8.5. with 语句

with 语句用于包装带有使用上下文管理器 (参见 with 语句上下文管理器 一节) 定义的方法的代码块的执行。 这允许对普通的 try…except…finally 使用模式进行封装以方便地重用。

```
with_stmt:          "with" ( "(" with_stmt_contents ","? ")" | with_stmt_contents )
with_stmt_contents: with_item (","  with_item)*
with_item:          expression ["as" target]
```

带有一个"项目"的 with 语句的执行过程如下:

1. 对上下文表达式（在 with_item 中给出的表达式）进行求值来获得上下文管理器。
2. 载入上下文管理器的 __enter__() 以便后续使用。
3. 载入上下文管理器的 __exit__() 以便后续使用。
4. 唤起上下文管理器的 __enter__() 方法。
5. 如果一个目标被包括在 with 语句中，则把它赋值为 __enter__() 的返回值。

> **备注:** with 语句会保证如果 __enter__() 方法未发生错误地返回，则 __exit__() 将一定被调用。 因此，如果在对目标列表赋值期间发生错误，它将被当作在语句体内部发生的错误来处理。 参见下面的第 7 步。

6. 执行语句体。

7. 唤起上下文管理器的 `__exit__()` 方法。 如果语句体的退出是由异常导致的，则其类型、值和回溯信息将被作为参数传递给 `__exit__()`。 否则的话，将提供三个 `None` 参数。

   如果语句体的退出是由异常导致的，并且来自 `__exit__()` 方法的返回值为假，则该异常会被重新引发。 如果返回值为真，则该异常会被抑制，并会继续执行 `with` 语句之后的语句。

   如果语句体由于异常以外的任何原因退出，则来自 `__exit__()` 的返回值会被忽略，并会在该类退出正常的发生位置继续执行。

以下代码:

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

在语义上等价于:

```
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)
```

如果有多个项目，则会视作存在多个 `with` 语句嵌套来处理多个上下文管理器:

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

在语义上等价于:

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

也可以用圆括号包围的多行形式的多项目上下文管理器。例如:

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

> *在 3.1 版本发生变更:* 支持多个上下文表达式。

> *在 3.10 版本发生变更:* Support for using grouping parentheses to break the statement in multiple lines.

**参见:**

**PEP 343 - "with" 语句**
Python `with` 语句的规范描述、背景和示例。

## 8.6. match 语句

> *Added in version 3.10.*

匹配语句用于进行模式匹配。语法如下:

```
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`
```

**备注:** 本节使用单引号来表示 软关键字。

模式匹配接受一个模式作为输入（跟在 `case` 后），一个目标值（跟在 `match` 后）。该模式（可能包含子模式）将与目标值进行匹配。输出是:

- 匹配成功或失败（也被称为模式成功或失败）。
- 可能将匹配的值绑定到一个名字上。 这方面的先决条件将在下面进一步讨论。

关键字 `match` 和 `case` 是 soft keywords 。

**参见:**

- **PEP 634** —— 结构化模式匹配：规范
- **PEP 636** —— 结构化模式匹配：教程

### 8.6.1. 概述

匹配语句逻辑流程的概述如下:

1. 对目标表达式 `subject_expr` 求值后将结果作为匹配用的目标值。 如果目标表达式包含逗号，则使用 the standard rules 构建一个元组。

2. 目标值将依次与 `case_block` 中的每个模式进行匹配。匹配成功或失败的具体规则在下面描述。匹配尝试也可以与模式中的一些或所有的独立名称绑定。准确的模式绑定规则因模式类型而异，具体规定见下文。**成功的模式匹配过程中产生的名称绑定将超越所执行的块的范围，可以在匹配语句之后使用。**

> **备注:** 在模式匹配失败时，一些子模式可能会成功。 不要依赖于失败匹配进行的绑定。 反过来说，不要认为变量在匹配失败后保持不变。 确切的行为取决于实现，可能会有所不同。 这是一个有意的决定，允许不同的实现添加优化。

3. 如果该模式匹配成功，并且完成了对相应的约束项（如果存在）的求值。在这种情况下，保证完成所有的名称绑定。

    - 如果约束项求值为真或缺失，执行 `case_block` 中的 `block` 。
    - 否则，将按照上述方法尝试下一个 `case_block` 。
    - 如果没有进一步的 case 块，匹配语句终止。

> **备注:** 用户一般不应依赖正在求值的模式。 根据不同的实现方式，解释器可能会缓存数值或使用其他优化方法来避免重复求值。

匹配语句示例:

```
>>> flag = False
>>> match (100, 200):
...     case (100, 300):  # 不匹配: 200 != 300
...         print('Case 1')
...     case (100, 200) if flag:  # 成功匹配，但防护检查失败
...         print('Case 2')
...     case (100, y):  # 匹配并将 y 绑定到 200
...         print(f'Case 3, y: {y}')
...     case _:  # 未尝试的模式
...         print('Case 4, I match anything!')
...
Case 3, y: 200
```

在这个示例中，`if flag` 是约束项。请阅读下一节以了解更多相关内容。

### 8.6.2. 约束项

```
guard: "if" `!named_expression`
```

`guard` (它是 `case` 的一部分) 必须成立才能让 `case` 语句块中的代码被执行。 它所采用的形式为: `if` 之后跟一个表达式。

拥有 `guard` 的 `case` 块的逻辑流程如下:

1. 检查 `case` 块中的模式是否匹配成功。如果该模式匹配失败，则不对 `guard` 进行求值，检查下一个 `case` 块。
2. 如果该模式匹配成功，对 `guard` 求值。
    - 如果 `guard` 求值为真，则选用该 case 块。
    - 如果 `guard` 求值为假，则不选用该 case 块。
    - 如果在对 `guard` 求值过程中引发了异常，则异常将被抛出。

允许约束项产生副作用，因为他们是表达式。约束项求值必须从第一个 case 块到最后一个 case 块依次逐个进行，模式匹配失败的 case 块将被跳过。（也就是说，约束项求值必须按顺序进行。）一

旦选用了一个 case 块，约束项求值必须由此终止。

### 8.6.3. 必定匹配的 case 块

必定匹配的 case 块是能匹配所有情况的 case 块。一个匹配语句最多可以有一个必定匹配的 case 块，而且必须是最后一个。

如果一个 case 块没有约束项，并且其模式是必定匹配的，那么它就被认为是必定匹配的。 如果我们可以仅从语法上证明一个模式总是能匹配成功，那么这个模式就被认为是必定匹配的。 只有以下模式是必定匹配的：

- 左侧模式是必定匹配的 AS 模式
- 包含至少一个必定匹配模式的 或模式
- 捕获模式
- 通配符模式
- 括号内的必定匹配模式

### 8.6.4. 模式

> **备注：** 本节使用了超出标准 EBNF 的语法符号。
>
> - 符号 `SEP.RULE+` 是 `RULE (SEP RULE)*` 的简写
> - 符号 `!RULE` 是前向否定断言的简写

`patterns` 的顶层语法是：

```
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
```

下面的描述将包括一个“简而言之”以描述模式的作用，便于说明问题（感谢 Raymond Hettinger 提供的一份文件，大部分的描述受其启发）。请注意，这些描述纯粹是为了说明问题，**可能不** 反映底层的实现。此外，它们并没有涵盖所有有效的形式。

#### 8.6.4.1. 或模式

或模式是由竖杠 `|` 分隔的两个或更多的模式。语法：

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

只有最后的子模式可以是 必定匹配的，且每个子模式必须绑定相同的名字集以避免歧义。

或模式将目标值依次与其每个子模式尝试匹配，直到有一个匹配成功，然后该或模式被视作匹配成功。 否则，如果没有任何子模式匹配成功，则或模式匹配失败。

简而言之，`P1 | P2 | ...` 会首先尝试匹配 `P1`，如果失败将接着尝试匹配 `P2`，如果出现成功的匹配则立即结束且模式匹配成功，否则模式匹配失败。

#### 8.6.4.2. AS 模式

AS 模式将关键字 `as` 左侧的或模式与目标值进行匹配。语法：

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

如果 OR 模式匹配失败，则 AS 模式也会失败。 在其他情况下，AS 模块会将目标与 as 关键字右边的名称绑定并匹配成功。 `capture_pattern` 不可为 `_`。

简而言之， `P as NAME` 将与 `P` 匹配，成功后将设置 `NAME = <subject>`。

#### 8.6.4.3. 字面值模式

字面值模式对应 Python 中的大多数 字面值。 语法为:

```
literal_pattern: signed_number
                 | signed_number "+" NUMBER
                 | signed_number "-" NUMBER
                 | strings
                 | "None"
                 | "True"
                 | "False"
signed_number:   ["-"] NUMBER
```

规则 `strings` 和标记 `NUMBER` 在 标准 Python 语法 中有定义。支持三引号字符串、原生字符串和字节字符串，但不支持 f-字符串 和 t-strings。

`signed_number '+' NUMBER` 和 `signed_number '-' NUMBER` 形式是用于表示 复数；它们要求左边是一个实数而右边是一个虚数。 例如 `3 + 4j`。

简而言之， `LITERAL` 只会在 `<subject> == LITERAL` 时匹配成功。对于单例 `None`、 `True` 和 `False`，会使用 `is` 运算符。

#### 8.6.4.4. 捕获模式

捕获模式将目标值与一个名称绑定。语法：

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

单独的一个下划线 `_` 不是捕获模式（`!'_'` 表达的就是这个含义）。 它会被当作 `wildcard_pattern`。

在给定的模式中，一个名字只能被绑定一次。例如 `case x, x: ...` 时无效的，但 `case [x] | x: ...` 是被允许的。

捕获模式总是能匹配成功。绑定遵循 **PEP 572** 中赋值表达式运算符设立的作用域规则；名字在最接近的包含函数作用域内成为一个局部变量，除非有适用的 `global` 或 `nonlocal` 语句。

简而言之， `NAME` 总是会匹配成功且将设置 `NAME = <subject>` 。

#### 8.6.4.5. 通配符模式

通配符模式总是会匹配成功（匹配任何内容）并且不绑定任何名称。语法：

```
wildcard_pattern: '_'
```

在且仅在任何模式中 `_` 是一个 软关键字。 通常情况下它是一个标识符，即使是在 `match` 的目标表达式、`guard` 和 `case` 代码块中也是如此。

简而言之， `_` 总是会匹配成功。

#### 8.6.4.6. 值模式

值模式代表 Python 中具有名称的值。语法：

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

模式中带点的名称会使用标准的 Python 名称解析规则 来查找。 如果找到的值与目标值比较结果相等则模式匹配成功（使用 `==` 相等运算符）。

简而言之， `NAME1.NAME2` 仅在 `<subject> == NAME1.NAME2` 时匹配成功。

> **备注：** 如果相同的值在同一个匹配语句中出现多次，解释器可能会缓存找到的第一个值并重新使用它，而不是重复查找。 这种缓存与特定匹配语句的执行严格挂钩。

#### 8.6.4.7. 组模式

组模式允许用户在模式周围添加括号，以强调预期的分组。 除此之外，它没有额外的语法。语法：

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

简单来说 `(P)` 具有与 `P` 相同的效果。

#### 8.6.4.8. 序列模式

一个序列模式包含数个将与序列元素进行匹配的子模式。其语法类似于列表或元组的解包。

```
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)
```

序列模式中使用圆括号或方括号没有区别（例如 `(...)` 和 `[...]` ）。

> **备注：** 用圆括号括起来且没有跟随逗号的单个模式 (例如 `(3 | 4)`) 是一个 分组模式。 而用方括号括起来的单个模式 (例如 `[3 | 4]`) 则仍是一个序列模式。

一个序列模式中最多可以有一个星号子模式。星号子模式可以出现在任何位置。如果没有星号子模式，该序列模式是固定长度的序列模式；否则，其是一个可变长度的序列模式。

下面是将一个序列模式与一个目标值相匹配的逻辑流程：

1. 如果目标值不是一个序列 [2] ，该序列模式匹配失败。
2. 如果目标值是 `str` 、 `bytes` 或 `bytearray` 的实例，则该序列模式匹配失败。
3. 随后的步骤取决于序列模式是固定长度还是可变长度的。

   如果序列模式是固定长度的：

   1. 如果目标序列的长度与子模式的数量不相等，则该序列模式匹配失败
   2. 序列模式中的子模式与目标序列中的相应项目从左到右进行匹配。 一旦一个子模式匹配失败，就停止匹配。 如果所有的子模式都成功地与它们的对应项相匹配，那么该序列模式就匹配成功了。

   否则，如果序列模式是变长的：

   1. 如果目标序列的长度小于非星号子模式的数量，则该序列模式匹配失败。
   2. 与固定长度的序列一样，靠前的非星形子模式与其相应的项目进行匹配。
   3. 如果上一步成功，星号子模式与剩余的目标项形成的列表相匹配，不包括星号子模式之后的非星号子模式所对应的剩余项。
   4. 剩余的非星号子模式将与相应的目标项匹配，就像固定长度的序列一样。

   > **备注：** 目标序列的长度可通过 `len()` (即通过 `__len__()` 协议) 获得。 解释器可能会以类似于 值模式 的方式缓存这个长度信息。

简而言之， `[P1, P2, P3, … , P<N>]` 仅在满足以下情况时匹配成功：

- 检查 `<subject>` 是一个序列
- `len(subject) == <N>`
- 将 `P1` 与 `<subject>[0]` 进行匹配（请注意此匹配可以绑定名称）
- 将 `P2` 与 `<subject>[1]` 进行匹配（请注意此匹配可以绑定名称）
- …… 剩余对应的模式/元素也以此类推。

#### 8.6.4.9. 映射模式

映射模式包含一个或多个键值模式。其语法类似于字典的构造。语法：

```
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
```

一个映射模式中最多可以有一个双星号模式。双星号模式必须是映射模式中的最后一个子模式。

映射模式中不允许出现重复的键。重复的字面值键会引发 SyntaxError 。若是两个键有相同的值将会在运行时引发 ValueError 。

以下是映射模式与目标值匹配的逻辑流程：

1. 如果目标值不是一个映射 [3]，则映射模式匹配失败。
2. 若映射模式中给出的每个键都存在于目标映射中，且每个键的模式都与目标映射的相应项匹配成功，则该映射模式匹配成功。
3. 如果在映射模式中检测到重复的键，该模式将被视作无效。对于重复的字面值，会引发 SyntaxError ；对于相同值的命名键，会引发 ValueError 。

**备注：** 键值对使用映射目标的 `get()` 方法的双参数形式进行匹配。 匹配的键值对必须已经存在于映射中，而不是通过 `__missing__()` 或 `__getitem__()` 即时创建。

简而言之，`{KEY1: P1, KEY2: P2, ... }` 仅在满足以下情况时匹配成功：

- 检查 `<subject>` 是映射
- `KEY1 in <subject>`
- `P1` 与 `<subject>[KEY1]` 相匹配
- ..... 剩余对应的键/模式对也以此类推。

#### 8.6.4.10. 类模式

类模式表示一个类以及它的位置参数和关键字参数（如果有的话）。语法：

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

同一个关键词不应该在类模式中重复出现。

以下是类模式与目标值匹配的逻辑流程：

1. 如果 `name_or_attr` 不是内置 type 的实例，引发 TypeError 。
2. 如果目标值不是 `name_or_attr` 的实例（通过 isinstance() 测试），该类模式匹配失败。
3. 如果没有模式参数存在，则该模式匹配成功。 否则，后面的步骤取决于是否有关键字或位置参数模式存在。

   对于一些内置的类型（将在后文详述），接受一个位置子模式，它将与整个目标值相匹配；对于这些类型，关键字模式也像其他类型一样工作。

如果只存在关键词模式，它们将被逐一处理，如下所示：

一. 该关键词被视作主体的一个属性进行查找。

- 如果这引发了除 `AttributeError` 以外的异常，该异常会被抛出。
- 如果这引发了 `AttributeError` ，该类模式匹配失败。
- 否则，与关键词模式相关的子模式将与目标的属性值进行匹配。 如果失败，则类模式匹配失败；如果成功，则继续对下一个关键词进行匹配。

二. 如果所有的关键词模式匹配成功，该类模式匹配成功。

如果存在位置模式，在匹配前会用类 `name_or_attr` 的 `__match_args__` 属性将其转换为关键词模式。

一. 进行与 `getattr(cls, "__match_args__", ())` 等价的调用。

- 如果这引发一个异常，该异常将被抛出。
- 如果返回值不是一个元组，则转换失败且引发 `TypeError` 。
- 若位置模式的数量超出 `len(cls.__match_args__)` ，将引发 `TypeError` 。
- 否则，位置模式 `i` 会使用 `__match_args__[i]` 转换为关键词。 `__match_args__[i]` 必须是一个字符串；如果不是则引发 `TypeError` 。
- 如果有重复的关键词，引发 `TypeError` 。

> **参见:** 定制类模式匹配中的位置参数

二. 若所有的位置模式都被转换为关键词模式，
匹配的过程就像只有关键词模式一样。

对于以下内置类型，位置子模式的处理是不同的：

- `bool`
- `bytearray`
- `bytes`
- `dict`
- `float`
- `frozenset`
- `int`
- `list`
- `set`
- `str`
- `tuple`

这些类接受一个位置参数，其模式是针对整个对象而不是某个属性进行匹配。 例如，`int(0|1)` 匹配值 `0`，但不匹配值 `0.0`。

简而言之， `CLS(P1, attr=P2)` 仅在满足以下情况时匹配成功：

- `isinstance(<subject>, CLS)`
- 用 `CLS.__match_args__` 将 `P1` 转换为关键词模式
- 对于每个关键词参数 `attr=P2`：
  - `hasattr(<subject>, "attr")`
  - 将 `P2` 与 `<subject>.attr` 进行匹配
- ..... 剩余对应的关键字参数/模式对也以此类推。

**参见:**

- **PEP 634** —— 结构化模式匹配：规范
- **PEP 636** —— 结构化模式匹配：教程

## 8.7. 函数定义

函数定义就是对用户自定义函数的定义（参见 标准类型层级结构 一节）：

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

函数定义是一条可执行语句。 它执行时会在当前局部命名空间中将函数名称绑定到一个函数对象（函数可执行代码的包装器）。 这个函数对象包含对当前全局命名空间的引用，作为函数被调用时所使用的全局命名空间。

函数定义并不会执行函数体；只有当函数被调用时才会执行此操作。 [4]

一个函数定义可以被一个或多个 decorator 表达式所包装。 当函数被定义时将在包含该函数定义的作用域中对装饰器表达式求值。 求值结果必须是一个可调用对象，它会以该函数对象作为唯一参数被唤起。 其返回值将被绑定到函数名称而非函数对象。 多个装饰器会以嵌套方式被应用。 例如以下代码

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

大致等价于

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

不同之处在于原始函数并不会被临时绑定到名称 `func`。

> *在 3.9 版本发生变更:* 函数可使用任何有效的 assignment_expression 来装饰。 在之前版本中，此语法则更为受限，详情参见 **PEP 614**。

可以在函数名及其形参列表开头圆括号之间加方括号给出一个 类型形参 的列表。 这将向静态类型检查器指明该函数是泛型尾数。 在运行时，类型形参可以从函数的 `__type_params__` 属性中提取。 请参阅 泛型函数 了解详情。

> *在 3.12 版本发生变更:* 类型形参列表是在 Python 3.12 中新增的。

当一个或多个 形参 具有 *形参* `=` *表达式* 这样的形式时，该函数就被称为具有"默认形参值"。 对于一个具有默认值的形参，其对应的 argument 可以在调用中被省略，在此情况下会用形参的默认值来替代。 如果一个形参具有默认值，后续所有在 "`*`" 之前的形参也必须具有默认值 --- 这个句法限制并未在语法中明确表达。

**默认形参值会在执行函数定义时按从左至右的顺序被求值。** 这意味着当函数被定义时将对表达式求值一次，相同的"预计算"值将在每次调用时被使用。 这一点在默认形参为可变对象，例如列表或字典的时候尤其需要重点理解：如果函数修改了该对象（例如向列表添加了一项），则实际上默认值也会被修改。 这通常不是人们所想要的。 绕过此问题的一个方法是使用 `None` 作为默认值，并在函数体中显式地对其进测试，例如:

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

函数调用的语义在 调用 一节中有更详细的描述。 函数调用总是会给形参列表中列出的所有形参赋值，或是用位置参数，或是用关键字参数，或是用默认值。 如果存在 "`*identifier`" 这样的形式，它会被初始化为一个元组来接收任何额外的位置参数，默认为一个空元组。 如果存在 "`**identifier`" 这样的形式，它会被初始化为一个新的有序映射来接收任何额外的关键字参数，默认为一个相同类型的空映射。 在 "`*`" 或 "`*identifier`" 之后的形参都是仅限关键字形参因而只能通过关键字参数传入。 在 "`/`" 之前的形参都是仅限位置形参因而只能通过位置参数传入。

> *在 3.8 版本发生变更:* 可以使用 `/` 函数形参语法来标示仅限位置形参。 请参阅 **PEP 570** 了解详情。

形参可以带有 注解，其形式为在形参名后加 "`: expression`"。 任何形参都可以带注解，甚至 `*identifier` 或 `**identifier` 这样的形参也可以。 （作为特例，`*identifier` 这样的形参可以有 "`: *expression`" 形式的注解。） 函数可以带有"返回"注解，其形式为在形参列表后加 "`-> expression`"。 这些注解可以是任何有效的 Python 表达式。 注解的存在不会改变函数的语义。 有关注解的更多信息，请参阅 标注。

> *在 3.11 版本发生变更:* 形式为 "`*identifier`" 的形参可以带有 "`: *expression`" 标注。 参见 **PEP 646**。

创建匿名函数（未绑定到一个名称的函数）以便立即在表达式中使用也是可能的。 这需要使用 lambda 表达式，具体描述见 lambda 表达式 一节。 请注意 lambda 只是简单函数定义的一种简化写法；在 "`def`" 语句中定义的函数也可以像用 lambda 表达式定义的函数一样被传递或赋值给其他名称。 "`def`" 形式实际上更为强大，因为它允许执行多条语句和使用标注。

**程序员注意事项:** 函数属于一类对象。 在一个函数内部执行的 "`def`" 语句会定义一个局部函数并可被返回或传递。 在嵌套函数中使用的自由变量可以访问包含该 def 语句的函数的局部变量。 详情参见 命名与绑定 一节。

**参见:**

**PEP 3107** - **函数标注**
最初的函数标注规范说明。

**PEP 484** —— **类型注解**
标注的标准含意定义：类型提示。

**PEP 526** - **变量标注的语法**
变量声明的类型提示功能，包括类变量和实例变量。

**PEP 563** - **延迟的标注求值**
支持在运行时通过以字符串形式保存标注而非不是即求值来实现标注内部的向前引用。

**PEP 318** - **函数和方法的装饰器**
引入了函数和方法的装饰器。 类装饰器是在 **PEP 3129** 中引入的。

## 8.8. 类定义

类定义就是对类对象的定义 (参见 标准类型层级结构 一节):

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

类定义是一条可执行语句。 其中继承列表通常给出基类的列表 (进阶用法请参见 元类)，列表中的每一项都应当被求值为一个允许子类的类对象。 没有继承列表的类默认继承自基类 `object`；因此，：

```
class Foo:
    pass
```

等价于

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

随后类体将在一个新的执行帧 (参见 命名与绑定) 中被执行，使用新创建的局部命名空间和原有的全局命名空间。 （通常，类体主要包含函数定义。） 当类体结束执行时，其执行帧将被丢弃而其局部命名空间会被保存。 [5] 一个类对象随后会被创建，其基类使用给定的继承列表，属性字典使用保存的局部命名空间。 类名称将在原有的全局命名空间中绑定到该类对象。

在类体内定义的属性的顺序保存在新类的 `__dict__` 中。 请注意此顺序的可靠性只限于类刚被创建时，并且只适用于定义语法所定义的类。

类的创建可使用 元类 进行重度定制。

类也可以被装饰：就像装饰函数一样，：

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

大致等价于

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

装饰器表达式的求值规则与函数装饰器相同。 结果随后会被绑定到类名称。

> *在 3.9 版本发生变更:* 类可使用任何有效的 `assignment_expression` 来装饰。 在之前版本中，此语法则更为受限，详情参见 **PEP 614**。

可以在类名之后的方括号中列出 类型形参。 这将向静态类型检查器指明该类是泛型类。 在运行时，可以从类的 `__type_params__` 属性中获取类型形参。 请参阅 泛型类 了解详情。

> *在 3.12 版本发生变更:* 类型形参列表是在 Python 3.12 中新增的。

**程序员注意事项:** 在类定义内定义的变量是类属性；它们将被类实例所共享。 实例属性可通过 `self.name = value` 在方法中设定。 类和实例属性均可通过 "`self.name`" 表示法来访问，当通过此方式访问时实例属性会隐藏同名的类属性。 类属性可被用作实例属性的默认值，但在此场景下使用可变值可能导致未预期的结果。 可以使用 描述器 来创建具有不同实现细节的实例变量。

**参见:**

**PEP 3115 - Python 3000 中的元类**
将元类声明修改为当前语法的提议，以及关于如何构建带有元类的类的语义描述。

**PEP 3129 - 类装饰器**
增加类装饰器的提议。 函数和方法装饰器是在 **PEP 318** 中被引入的。

## 8.9. 协程

> *Added in version 3.5.*

### 8.9.1. 协程函数定义

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

Python 协程的执行可以在多个位置上被挂起和恢复 (参见 coroutine)。 `await` 表达式，`async for` 以及 `async with` 只能在协程函数体中使用。

使用 `async def` 语法定义的函数总是为协程函数，即使它们不包含 `await` 或 `async` 关键字。

在协程函数体中使用 `yield from` 表达式将引发 `SyntaxError`。

协程函数的例子:

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

> *在 3.7 版本发生变更:* `await` 和 `async` 现在是保留关键字；在之前版本中它们仅在协程函数内被当作保留关键字。

### 8.9.2. async for 语句

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

asynchronous iterable 提供了 `__aiter__` 方法，该方法会直接返回 asynchronous iterator，它可以在其 `__anext__` 方法中调用异步代码。

`async for` 语句允许方便地对异步可迭代对象进行迭代。

以下代码:

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

在语义上等价于:

```
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
```

另请参阅 `__aiter__()` 和 `__anext__()` 了解详情。

在协程函数体之外使用 `async for` 语句将引发 SyntaxError。

### 8.9.3. `async with` 语句

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

asynchronous context manager 是一种 context manager，能够在其 *enter* 和 *exit* 方法中暂停执行。

以下代码:

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

在语义上等价于:

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

try:
    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)
```

另请参阅 `__aenter__()` 和 `__aexit__()` 了解详情。

在协程函数体之外使用 `async with` 语句将引发 SyntaxError。

**参见:**

**PEP 492 - 使用 async 和 await 语法实现协程**
将协程作为 Python 中的一个正式的单独概念，并增加相应的支持语法。

## 8.10. 类型形参列表

> *Added in version 3.12.*

> *在 3.13 版本发生变更:* 增加了对默认值的支持 (参见 **PEP 696**)。

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

函数 (包括 协程), 类 和 类型别名 可能包含类型形参列表:

```
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]
```

从语义上讲，这表明函数、类或类型别名是类型变量的泛型。 此信息主要供静态类型检查器使用，并且在运行时，泛型对象的行为与其对应的非泛型对象非常相似。

类型参数是紧接在函数、类或类型别名的名称之后的方括号 (`[]`) 中声明的。 类型参数可在泛型对象的作用域内访问，但不能在其他地方访问。 因此，在声明 `def func[T](): pass` 之后，模块作用域中就不能再使用 `T` 这个名称。 在下文中，将更精确地描述泛型对象的语义。 类型形参的作用域是用一个特殊函数 (从技术上说，是一个 标注作用域) 来模拟的，它封装了泛型对象的创建操作。

泛型函数、类和类型别名都有一个 `__type_params__` 属性用来列出它们的类型形参。

类型形参可分为三种:

- `typing.TypeVar`，由一个普通名称 (例如 `T`) 引入。 从语义上讲，这对类型检查器来说代表了一个单独类型。
- `typing.TypeVarTuple`，通过在前面添加一个星号的名称来引入 (例如 `*Ts`)。 从语义上讲，它代表由任意多个类型组成的元组。
- `typing.ParamSpec`，通过在前面添加两个星号的名称来引入 (例如 `**P`)。 从语义上讲，它代表一个可调用对象的形参。

`typing.TypeVar` 声明可以通过在冒号 (`:`) 后跟一个表达式来定义 *范围* 和 *约束*。 冒号后的单独表达式表示一个范围 (例如 `T: int`)。 从语义上讲，这意味着 `typing.TypeVar` 能表示的类型只能是该范围的子类型。 冒号后在圆括号内的表达式元组指定了一组约束 (例如 `T: (str, bytes)`)。 元组中的每个成员都应为一个类型 (同样，在运行时并不强制要求这一点)。 约束的类型变量只能使用约束列表内的类型中选择一种。

对于使用类型形参列表语法声明的 `typing.TypeVar`，范围和约束在创建泛型对象时并不会被求值，只有在通过属性 `__bound__` 和 `__constraints__` 显式地访问它时才会被求值。 要做到这一点，需要在单独的 标注作用域 中对范围和约束进行求值。

`typing.TypeVarTuple` 和 `typing.ParamSpec` 不能拥有范围或约束。

所有三种风格的类型形参都还可以具有 *默认值*，它会在未显式提供类型形参值时被使用。 这是通过添加单个等号 (`=`) 跟一个表达式来添加的。 与类型变量的绑定和约束类似，默认值不是在创建对象

时被求值的，而是在类型形参的 `__default__` 属性被访问的时候。 为此，默认值将在单独的 标注作用域 中被求值。 如果没有为类型形参指定默认值，`__default__` 属性将被设为特殊的哨兵对象 `typing.NoDefault`。

下面的例子显示了所有被允许的类型形参声明:

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

### 8.10.1. 泛型函数

泛型函数的声明方式如下:

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

该语法等价于:

```
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()
```

这里 `annotation-def` 指定了一个 标注作用域，它在运行时并不会实际绑定到任何名称。 （另一项自由是在翻译中达成的：该语法没有通过 `typing` 模块的属性访问，而是直接创建了一个 `typing.TypeVar` 的实例）。

泛型函数的标注会在用于声明类型形参的标注作用域内进行求值，但函数的默认值和装饰器则不会。

下面的例子演示了针对这些场景，以及类型形参的变化形式的作用域规则:

```
@decorator
def func[T: int, *Ts, **P](*args: *Ts, arg: Callable[P, T] = some_default):
    ...
```

除了 `TypeVar` 绑定的 惰性求值 以外，这等同于:

```
DEFAULT_OF_arg = some_default

annotation-def TYPE_PARAMS_OF_func():

    annotation-def BOUND_OF_T():
        return int
    # 在现实中，BOUND_OF_T() 仅会在需要时被求值。
    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())
```

大写形式的名称如 `DEFAULT_OF_arg` 在运行时不会被实际绑定。

### 8.10.2. 泛型类

泛型类的声明方式如下:

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

该语法等价于:

```
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()
```

这里还是用 `annotation-def` (不是真正的关键字) 指明 标注作用域，而名称 `TYPE_PARAMS_OF_Bag` 在不会运行时实际被绑定。

泛型类隐式地继承自 `typing.Generic`。 泛型类的基类和关键字参数在类型形参的类型作用域内进行求值，而装饰器则在该作用域之外进行求值。 以下示例对此进行了说明:

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

这相当于:

```
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())
```

### 8.10.3. 泛型类型别名

type 语句也可被用来创建泛型类型别名:

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

除了会对值执行 惰性求值 以外，这等同于:

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

    annotation-def VALUE_OF_ListOrSet():
        return list[T] | set[T]
    # 在现实中，该值将被惰性地求值
    return typing.TypeAliasType("ListOrSet", VALUE_OF_ListOrSet(), type_params=(T,
ListOrSet = TYPE_PARAMS_OF_ListOrSet()
```

这里，`annotation-def` (不是一个真正的关键字) 指明 标注作用域。 像 `TYPE_PARAMS_OF_ListOrSet` 这样的大写名称不会在运行时实际被绑定。

## 8.11. 标注

> *在 3.14 版本发生变更:* 标注现在默认将被惰性求值。

变量和函数形参可能带有 标注，创建方式是在名称后加一个冒号，后面再跟一个表达式:

```
x: annotation = 1
def f(param: annotation): ...
```

函数也可能带有加在一个箭头后的返回值标注:

```
def f() -> annotation: ...
```

注解通常用于 类型提示，但语言不强制这样做，并且通常注解可能包含任意表达式。 注解的存在不会改变代码的运行时语义，除非使用了一些自省和使用注解的机制 (例如 dataclasses 或 functools.singledispatch())。

默认情况下，注解在 注解范围 中惰性求值。 这意味着当包含注解的代码被求值时，它们不会被求值。 相反，解释器会保存信息，以便在以后需要时用于注解求值。 annotationlib 模块提供了注解求值的工具。

如果存在 future 语句 `from __future__ import annotations`，则所有注解将被存储为字符串:

```
>>> from __future__ import annotations
>>> def f(param: annotation): ...
>>> f.__annotations__
{'param': 'annotation'}
```

这个 future语 句将在 Python 的未来版本中被弃用并删除，但不会在 Python 3.13 达到其生命周期结束之前 (参见 **PEP 749**)。 当它被使用时，内省工具，如 `annotationlib.get_annotations()` 和 `typing.get_type_hints()` 不太可能在运行时解析注解。

**备注**

[1] 异常会被传播给唤起栈，除非存在一个 `finally` 子句正好引发了另一个异常。 新引发的异常将导致旧异常的丢失。

[2] 在模式匹配中，序列被定义为以下几种之一:

- 继承自 `collections.abc.Sequence` 的类
- 注册为 `collections.abc.Sequence` 的 Python 类
- 设置了 (CPython) `Py_TPFLAGS_SEQUENCE` 比特位的内置类
- 继承自上述任何一个类的类

  下列标准库中的类都是序列:

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

> **备注:** 类型为 `str`, `bytes` 和 `bytearray` 的目标值不能匹配序列模式。

[3] 在模式匹配中，映射被定义为以下几种之一:

- 继承自 `collections.abc.Mapping` 的类
- 注册为 `collections.abc.Mapping` 的 Python 类
- 设置了 (CPython) `Py_TPFLAGS_MAPPING` 比特位的内置类
- 继承自上述任何一个类的类

  标准库中的 `dict` 和 `types.MappingProxyType` 类都属于映射。

[4] 作为函数体的第一条语句出现的字符串字面值会被转换为函数的 `__doc__` 属性也就是该函数的 docstring。

[5] 作为类体的第一条语句出现的字符串字面值会被转为命名空间的 `__doc__` 条目，也就是该类的 docstring。