4. Execution model

4.1. Structure of a program

A Python program is constructed from code blocks. A *block* is a piece of Python program text that is executed as a unit. The following are blocks: a module, a function body, and a class definition. Each command typed interactively is a block. A script file (a file given as standard input to the interpreter or specified as a command line argument to the interpreter) is a code block. A script command (a command specified on the interpreter command line with the `-c` option) is a code block. A module run as a top level script (as module `__main__`) from the command line using a `-m` argument is also a code block. The string argument passed to the built-in functions `eval()` and `exec()` is a code block.

A code block is executed in an *execution frame*. A frame contains some administrative information (used for debugging) and determines where and how execution continues after the code block's execution has completed.

4.2. Naming and binding

4.2.1. Binding of names

Names refer to objects. Names are introduced by name binding operations.

The following constructs bind names:

- formal parameters to functions,
- class definitions,
- function definitions,
- assignment expressions,
- targets that are identifiers if occurring in an assignment:
  - `for` loop header,
  - after `as` in a `with` statement, `except` clause, `except*` clause, or in the as-pattern in structural pattern matching,
  - in a capture pattern in structural pattern matching
- `import` statements.
- `type` statements.
- type parameter lists.

The `import` statement of the form `from ... import *` binds all names defined in the imported module, except those beginning with an underscore. This form may only be used at the module level.

A target occurring in a `del` statement is also considered bound for this purpose (though the actual semantics are to unbind the name).

Each assignment or import statement occurs within a block defined by a class or function definition or at the module level (the top-level code block).

If a name is bound in a block, it is a local variable of that block, unless declared as `nonlocal` or `global`. If a name is bound at the module level, it is a global variable. (The variables of the module code block are local and global.) If a variable is used in a code block but not defined there, it is a *free variable*.

Each occurrence of a name in the program text refers to the *binding* of that name established by the following name resolution rules.

4.2.2. Resolution of names

tained block introduces a different binding for the name.

When a name is used in a code block, it is resolved using the nearest enclosing scope. The set of all such scopes visible to a code block is called the block's *environment*.

When a name is not found at all, a `NameError` exception is raised. If the current scope is a function scope, and the name refers to a local variable that has not yet been bound to a value at the point where the name is used, an `UnboundLocalError` exception is raised. `UnboundLocalError` is a subclass of `NameError`.

If a name binding operation occurs anywhere within a code block, all uses of the name within the block are treated as references to the current block. This can lead to errors when a name is used within a block before it is bound. This rule is subtle. Python lacks declarations and allows name binding operations to occur anywhere within a code block. The local variables of a code block can be determined by scanning the entire text of the block for name binding operations. See the FAQ entry on UnboundLocalError for examples.

If the `global` statement occurs within a block, all uses of the names specified in the statement refer to the bindings of those names in the top-level namespace. Names are resolved in the top-level namespace by searching the global namespace, i.e. the namespace of the module containing the code block, and the builtins namespace, the namespace of the module `builtins`. The global namespace is searched first. If the names are not found there, the builtins namespace is searched. The `global` statement must precede all uses of the listed names.

The `global` statement has the same scope as a name binding operation in the same block. If the nearest enclosing scope for a free variable contains a global statement, the free variable is treated as a global.

The `nonlocal` statement causes corresponding names to refer to previously bound variables in the nearest enclosing function scope. `SyntaxError` is raised at compile time if the given name does not exist in any enclosing function scope. Type parameters cannot be rebound with the `nonlocal` statement.

The namespace for a module is automatically created the first time a module is imported. The main module for a script is always called `__main__`.

Class definition blocks and arguments to `exec()` and `eval()` are special in the context of name resolution. A class definition is an executable statement that may use and define names. These references follow the normal rules for name resolution with an exception that unbound local variables are looked up in the global namespace. The namespace of the class definition becomes the attribute dictionary of the class. The scope of names defined in a class block is limited to the class block; it does not extend to the code blocks of methods. This includes comprehensions and generator expressions, but it does not include annotation scopes, which have access to their enclosing class scopes. This means that the following will fail:

```python
class A:
    a = 42
    b = list(a + i for i in range(10))
```

However, the following will succeed:

```python
class A:
    type Alias = Nested
    class Nested: pass

print(A.Alias.__value__)  # <type 'A.Nested'>
```

4.2.3. Annotation scopes

tation scopes in Python 3.13 when **PEP 649** is implemented.

Annotation scopes are used in the following contexts:

- Type parameter lists for generic type aliases.
- Type parameter lists for generic functions. A generic function's annotations are executed within the annotation scope, but its defaults and decorators are not.
- Type parameter lists for generic classes. A generic class's base classes and keyword arguments are executed within the annotation scope, but its decorators are not.
- The bounds and constraints for type variables (lazily evaluated).
- The value of type aliases (lazily evaluated).

Annotation scopes differ from function scopes in the following ways:

- Annotation scopes have access to their enclosing class namespace. If an annotation scope is immediately within a class scope, or within another annotation scope that is immediately within a class scope, the code in the annotation scope can use names defined in the class scope as if it were executed directly within the class body. This contrasts with regular functions defined within classes, which cannot access names defined in the class scope.
- Expressions in annotation scopes cannot contain `yield`, `yield from`, `await`, or `:=` expressions. (These expressions are allowed in other scopes contained within the annotation scope.)
- Names defined in annotation scopes cannot be rebound with `nonlocal` statements in inner scopes. This includes only type parameters, as no other syntactic elements that can appear within annotation scopes can introduce new names.
- While annotation scopes have an internal name, that name is not reflected in the __qualname__ of objects defined within the scope. Instead, the `__qualname__` of such objects is as if the object were defined in the enclosing scope.

> *New in version 3.12:* Annotation scopes were introduced in Python 3.12 as part of **PEP 695**.

4.2.4. Lazy evaluation

The values of type aliases created through the `type` statement are *lazily evaluated*. The same applies to the bounds and constraints of type variables created through the type parameter syntax. This means that they are not evaluated when the type alias or type variable is created. Instead, they are only evaluated when doing so is necessary to resolve an attribute access.

Example:

```
>>> type Alias = 1/0
>>> Alias.__value__
Traceback (most recent call last):
  ...
ZeroDivisionError: division by zero
>>> def func[T: 1/0](): pass
>>> T = func.__type_params__[0]
>>> T.__bound__
Traceback (most recent call last):
  ...
ZeroDivisionError: division by zero
```

Here the exception is raised only when the `__value__` attribute of the type alias or the `__bound__` attribute of the type variable is accessed.

This behavior is primarily useful for references to types that have not yet been defined when the type alias or type variable is created. For example, lazy evaluation enables creation of mutually recursive type aliases:

```
type SimpleExpr = int | Parenthesized
type Parenthesized = tuple[Literal["("], Expr, Literal[")"]]
type Expr = SimpleExpr | tuple[SimpleExpr, Literal["+", "-"], Expr]
```

Lazily evaluated values are evaluated in annotation scope, which means that names that appear inside the lazily evaluated value are looked up as if they were used in the immediately enclosing scope.

New in version 3.12.

4.2.5. Builtins and restricted execution

CPython implementation detail: Users should not touch `__builtins__`; it is strictly an implementation detail. Users wanting to override values in the builtins namespace should `import` the `builtins` module and modify its attributes appropriately.

The builtins namespace associated with the execution of a code block is actually found by looking up the name `__builtins__` in its global namespace; this should be a dictionary or a module (in the latter case the module's dictionary is used). By default, when in the `__main__` module, `__builtins__` is the built-in module `builtins`; when in any other module, `__builtins__` is an alias for the dictionary of the `builtins` module itself.

4.2.6. Interaction with dynamic features

Name resolution of free variables occurs at runtime, not at compile time. This means that the following code will print 42:

```
i = 10
def f():
    print(i)
i = 42
f()
```

The `eval()` and `exec()` functions do not have access to the full environment for resolving names. Names may be resolved in the local and global namespaces of the caller. Free variables are not resolved in the nearest enclosing namespace, but in the global namespace. [1] The `exec()` and `eval()` functions have optional arguments to override the global and local namespace. If only one namespace is specified, it is used for both.

4.3. Exceptions

Exceptions are a means of breaking out of the normal flow of control of a code block in order to handle errors or other exceptional conditions. An exception is *raised* at the point where the error is detected; it may be *handled* by the surrounding code block or by any code block that directly or indirectly invoked the code block where the error occurred.

The Python interpreter raises an exception when it detects a run-time error (such as division by zero). A Python program can also explicitly raise an exception with the `raise` statement. Exception handlers are specified with the `try` … `except` statement. The `finally` clause of such a statement can be used to specify cleanup code which does not handle the exception, but is executed whether an exception occurred or not in the preceding code.

Python uses the "termination" model of error handling: an exception handler can find out what happened and continue execution at an outer level, but it cannot repair the cause of the error and retry the failing operation (except by re-entering the offending piece of code from the top).

When an exception is not handled at all, the interpreter terminates execution of the program, or returns to its interactive main loop. In either case, it prints a stack traceback, except when the exception is `SystemExit`.

Exceptions are identified by class instances. The `except` clause is selected depending on the class of the instance: it must reference the class of the instance or a non-virtual base class thereof. The instance can be received by the handler and can

Note: Exception messages are not part of the Python API. Their contents may change from one version of Python to the next without warning and should not be relied on by code which will run under multiple versions of the interpreter.

See also the description of the `try` statement in section The try statement and `raise` statement in section The raise statement.

Footnotes

[1] This limitation occurs because the code that is executed by these operations is not available at the time the module is compiled.