CHAPTER

THIRTYTHREE

PYTHON LANGUAGE SERVICES

Python provides a number of modules to assist in working with the Python language. These modules support tokenizing, parsing, syntax analysis, bytecode disassembly, and various other facilities.

These modules include:

33.1 `parser` — Access Python parse trees

The *parser* module provides an interface to Python's internal parser and byte-code compiler. The primary purpose for this interface is to allow Python code to edit the parse tree of a Python expression and create executable code from this. This is better than trying to parse and modify an arbitrary Python code fragment as a string because parsing is performed in a manner identical to the code forming the application. It is also faster.

Note: From Python 2.5 onward, it's much more convenient to cut in at the Abstract Syntax Tree (AST) generation and compilation stage, using the *ast* module.

There are a few things to note about this module which are important to making use of the data structures created. This is not a tutorial on editing the parse trees for Python code, but some examples of using the *parser* module are presented.

Most importantly, a good understanding of the Python grammar processed by the internal parser is required. For full information on the language syntax, refer to reference-index. The parser itself is created from a grammar specification defined in the file `Grammar/Grammar` in the standard Python distribution. The parse trees stored in the ST objects created by this module are the actual output from the internal parser when created by the *expr()* or *suite()* functions, described below. The ST objects created by *sequence2st()* faithfully simulate those structures. Be aware that the values of the sequences which are considered "correct" will vary from one version of Python to another as the formal grammar for the language is revised. However, transporting code from one Python version to another as source text will always allow correct parse trees to be created in the target version, with the only restriction being that migrating to an older version of the interpreter will not support more recent language constructs. The parse trees are not typically compatible from one version to another, whereas source code has always been forward-compatible.

Each element of the sequences returned by *st2list()* or *st2tuple()* has a simple form. Sequences representing non-terminal elements in the grammar always have a length greater than one. The first element is an integer which identifies a production in the grammar. These integers are given symbolic names in the C header file `Include/graminit.h` and the Python module *symbol*. Each additional element of the sequence represents a component of the production as recognized in the input string: these are always sequences which have the same form as the parent. An important aspect of this structure which should be noted is that keywords used to identify the parent node type, such as the keyword `if` in an `if_stmt`, are included

in the node tree without any special treatment. For example, the `if` keyword is represented by the tuple `(1, 'if')`, where `1` is the numeric value associated with all `NAME` tokens, including variable and function names defined by the user. In an alternate form returned when line number information is requested, the same token might be represented as `(1, 'if', 12)`, where the `12` represents the line number at which the terminal symbol was found.

Terminal elements are represented in much the same way, but without any child elements and the addition of the source text which was identified. The example of the `if` keyword above is representative. The various types of terminal symbols are defined in the C header file `Include/token.h` and the Python module *token*.

The ST objects are not required to support the functionality of this module, but are provided for three purposes: to allow an application to amortize the cost of processing complex parse trees, to provide a parse tree representation which conserves memory space when compared to the Python list or tuple representation, and to ease the creation of additional modules in C which manipulate parse trees. A simple "wrapper" class may be created in Python to hide the use of ST objects.

The *parser* module defines functions for a few distinct purposes. The most important purposes are to create ST objects and to convert ST objects to other representations such as parse trees and compiled code objects, but there are also functions which serve to query the type of parse tree represented by an ST object.

See also:

Module *symbol* Useful constants representing internal nodes of the parse tree.

Module *token* Useful constants representing leaf nodes of the parse tree and functions for testing node values.

33.1.1 Creating ST Objects

ST objects may be created from source code or from a parse tree. When creating an ST object from source, different functions are used to create the `'eval'` and `'exec'` forms.

`parser.expr`(*source*)
: The *expr()* function parses the parameter *source* as if it were an input to `compile(source, 'file.py', 'eval')`. If the parse succeeds, an ST object is created to hold the internal parse tree representation, otherwise an appropriate exception is raised.

`parser.suite`(*source*)
: The *suite()* function parses the parameter *source* as if it were an input to `compile(source, 'file.py', 'exec')`. If the parse succeeds, an ST object is created to hold the internal parse tree representation, otherwise an appropriate exception is raised.

`parser.sequence2st`(*sequence*)
: This function accepts a parse tree represented as a sequence and builds an internal representation if possible. If it can validate that the tree conforms to the Python grammar and all nodes are valid node types in the host version of Python, an ST object is created from the internal representation and returned to the called. If there is a problem creating the internal representation, or if the tree cannot be validated, a *ParserError* exception is raised. An ST object created this way should not be assumed to compile correctly; normal exceptions raised by compilation may still be initiated when the ST object is passed to *compilest()*. This may indicate problems not related to syntax (such as a *MemoryError* exception), but may also be due to constructs such as the result of parsing `del f(0)`, which escapes the Python parser but is checked by the bytecode compiler.

  Sequences representing terminal tokens may be represented as either two-element lists of the form `(1, 'name')` or as three-element lists of the form `(1, 'name', 56)`. If the third element is present, it is assumed to be a valid line number. The line number may be specified for any subset of the terminal symbols in the input tree.

`parser.tuple2st`(*sequence*)
: This is the same function as *sequence2st()*. This entry point is maintained for backward compatibility.

33.1.2 Converting ST Objects

ST objects, regardless of the input used to create them, may be converted to parse trees represented as list- or tuple- trees, or may be compiled into executable code objects. Parse trees may be extracted with or without line numbering information.

`parser.st2list`(*st*, *line_info=False*, *col_info=False*)
This function accepts an ST object from the caller in *st* and returns a Python list representing the equivalent parse tree. The resulting list representation can be used for inspection or the creation of a new parse tree in list form. This function does not fail so long as memory is available to build the list representation. If the parse tree will only be used for inspection, *st2tuple()* should be used instead to reduce memory consumption and fragmentation. When the list representation is required, this function is significantly faster than retrieving a tuple representation and converting that to nested lists.

If *line_info* is true, line number information will be included for all terminal tokens as a third element of the list representing the token. Note that the line number provided specifies the line on which the token *ends*. This information is omitted if the flag is false or omitted.

`parser.st2tuple`(*st*, *line_info=False*, *col_info=False*)
This function accepts an ST object from the caller in *st* and returns a Python tuple representing the equivalent parse tree. Other than returning a tuple instead of a list, this function is identical to *st2list()*.

If *line_info* is true, line number information will be included for all terminal tokens as a third element of the list representing the token. This information is omitted if the flag is false or omitted.

`parser.compilest`(*st*, *filename='<syntax-tree>'*)
The Python byte compiler can be invoked on an ST object to produce code objects which can be used as part of a call to the built-in *exec()* or *eval()* functions. This function provides the interface to the compiler, passing the internal parse tree from *st* to the parser, using the source file name specified by the *filename* parameter. The default value supplied for *filename* indicates that the source was an ST object.

Compiling an ST object may result in exceptions related to compilation; an example would be a *SyntaxError* caused by the parse tree for `del f(0)`: this statement is considered legal within the formal grammar for Python but is not a legal language construct. The *SyntaxError* raised for this condition is actually generated by the Python byte-compiler normally, which is why it can be raised at this point by the *parser* module. Most causes of compilation failure can be diagnosed programmatically by inspection of the parse tree.

33.1.3 Queries on ST Objects

Two functions are provided which allow an application to determine if an ST was created as an expression or a suite. Neither of these functions can be used to determine if an ST was created from source code via *expr()* or *suite()* or from a parse tree via *sequence2st()*.

`parser.isexpr`(*st*)
When *st* represents an `'eval'` form, this function returns true, otherwise it returns false. This is useful, since code objects normally cannot be queried for this information using existing built-in functions. Note that the code objects created by *compilest()* cannot be queried like this either, and are identical to those created by the built-in *compile()* function.

`parser.issuite`(*st*)
This function mirrors *isexpr()* in that it reports whether an ST object represents an `'exec'` form, commonly known as a "suite." It is not safe to assume that this function is equivalent to `not isexpr(st)`, as additional syntactic fragments may be supported in the future.

33.1.4 Exceptions and Error Handling

The parser module defines a single exception, but may also pass other built-in exceptions from other portions of the Python runtime environment. See each function for information about the exceptions it can raise.

exception parser.ParserError
Exception raised when a failure occurs within the parser module. This is generally produced for validation failures rather than the built-in *SyntaxError* raised during normal parsing. The exception argument is either a string describing the reason of the failure or a tuple containing a sequence causing the failure from a parse tree passed to *sequence2st()* and an explanatory string. Calls to *sequence2st()* need to be able to handle either type of exception, while calls to other functions in the module will only need to be aware of the simple string values.

Note that the functions *compilest()*, *expr()*, and *suite()* may raise exceptions which are normally raised by the parsing and compilation process. These include the built in exceptions *MemoryError*, *OverflowError*, *SyntaxError*, and *SystemError*. In these cases, these exceptions carry all the meaning normally associated with them. Refer to the descriptions of each function for detailed information.

33.1.5 ST Objects

Ordered and equality comparisons are supported between ST objects. Pickling of ST objects (using the *pickle* module) is also supported.

parser.STType
The type of the objects returned by *expr()*, *suite()* and *sequence2st()*.

ST objects have the following methods:

ST.compile(*filename='<syntax-tree>'*)
Same as `compilest(st, filename)`.

ST.isexpr()
Same as `isexpr(st)`.

ST.issuite()
Same as `issuite(st)`.

ST.tolist(*line_info=False*, *col_info=False*)
Same as `st2list(st, line_info, col_info)`.

ST.totuple(*line_info=False*, *col_info=False*)
Same as `st2tuple(st, line_info, col_info)`.

33.1.6 Example: Emulation of `compile()`

While many useful operations may take place between parsing and bytecode generation, the simplest operation is to do nothing. For this purpose, using the *parser* module to produce an intermediate data structure is equivalent to the code

```
>>> code = compile('a + 5', 'file.py', 'eval')
>>> a = 5
>>> eval(code)
10
```

The equivalent operation using the *parser* module is somewhat longer, and allows the intermediate internal parse tree to be retained as an ST object:

```
>>> import parser
>>> st = parser.expr('a + 5')
>>> code = st.compile('file.py')
>>> a = 5
>>> eval(code)
10
```

An application which needs both ST and code objects can package this code into readily available functions:

```
import parser

def load_suite(source_string):
    st = parser.suite(source_string)
    return st, st.compile()

def load_expression(source_string):
    st = parser.expr(source_string)
    return st, st.compile()
```

33.2 ast — Abstract Syntax Trees

Source code: Lib/ast.py

The *ast* module helps Python applications to process trees of the Python abstract syntax grammar. The abstract syntax itself might change with each Python release; this module helps to find out programmatically what the current grammar looks like.

An abstract syntax tree can be generated by passing `ast.PyCF_ONLY_AST` as a flag to the *compile()* built-in function, or using the *parse()* helper provided in this module. The result will be a tree of objects whose classes all inherit from *ast.AST*. An abstract syntax tree can be compiled into a Python code object using the built-in *compile()* function.

33.2.1 Node classes

`class ast.AST`

This is the base of all AST node classes. The actual node classes are derived from the `Parser/Python.asdl` file, which is reproduced *below*. They are defined in the `_ast` C module and re-exported in *ast*.

There is one class defined for each left-hand side symbol in the abstract grammar (for example, `ast.stmt` or `ast.expr`). In addition, there is one class defined for each constructor on the right-hand side; these classes inherit from the classes for the left-hand side trees. For example, `ast.BinOp` inherits from `ast.expr`. For production rules with alternatives (aka "sums"), the left-hand side class is abstract: only instances of specific constructor nodes are ever created.

`_fields`

Each concrete class has an attribute *_fields* which gives the names of all child nodes.

Each instance of a concrete class has one attribute for each child node, of the type as defined in the grammar. For example, `ast.BinOp` instances have an attribute `left` of type `ast.expr`.

If these attributes are marked as optional in the grammar (using a question mark), the value might be `None`. If the attributes can have zero-or-more values (marked with an asterisk), the

values are represented as Python lists. All possible attributes must be present and have valid values when compiling an AST with *compile()*.

lineno
col_offset
Instances of `ast.expr` and `ast.stmt` subclasses have *lineno* and *col_offset* attributes. The *lineno* is the line number of source text (1-indexed so the first line is line 1) and the *col_offset* is the UTF-8 byte offset of the first token that generated the node. The UTF-8 offset is recorded because the parser uses UTF-8 internally.

The constructor of a class `ast.T` parses its arguments as follows:

- If there are positional arguments, there must be as many as there are items in `T._fields`; they will be assigned as attributes of these names.
- If there are keyword arguments, they will set the attributes of the same names to the given values.

For example, to create and populate an `ast.UnaryOp` node, you could use

```
node = ast.UnaryOp()
node.op = ast.USub()
node.operand = ast.Num()
node.operand.n = 5
node.operand.lineno = 0
node.operand.col_offset = 0
node.lineno = 0
node.col_offset = 0
```

or the more compact

```
node = ast.UnaryOp(ast.USub(), ast.Num(5, lineno=0, col_offset=0),
                   lineno=0, col_offset=0)
```

33.2.2 Abstract Grammar

The abstract grammar is currently defined as follows:

```
-- ASDL's 7 builtin types are:
-- identifier, int, string, bytes, object, singleton, constant
--
-- singleton: None, True or False
-- constant can be None, whereas None means "no value" for object.

module Python
{
    mod = Module(stmt* body)
        | Interactive(stmt* body)
        | Expression(expr body)

        -- not really an actual node but useful in Jython's typesystem.
        | Suite(stmt* body)

    stmt = FunctionDef(identifier name, arguments args,
                       stmt* body, expr* decorator_list, expr? returns)
          | AsyncFunctionDef(identifier name, arguments args,
```

(continues on next page)

(continued from previous page)

```
                          stmt* body, expr* decorator_list, expr? returns)

        | ClassDef(identifier name,
           expr* bases,
           keyword* keywords,
           stmt* body,
           expr* decorator_list)
        | Return(expr? value)

        | Delete(expr* targets)
        | Assign(expr* targets, expr value)
        | AugAssign(expr target, operator op, expr value)
        -- 'simple' indicates that we annotate simple name without parens
        | AnnAssign(expr target, expr annotation, expr? value, int simple)

        -- use 'orelse' because else is a keyword in target languages
        | For(expr target, expr iter, stmt* body, stmt* orelse)
        | AsyncFor(expr target, expr iter, stmt* body, stmt* orelse)
        | While(expr test, stmt* body, stmt* orelse)
        | If(expr test, stmt* body, stmt* orelse)
        | With(withitem* items, stmt* body)
        | AsyncWith(withitem* items, stmt* body)

        | Raise(expr? exc, expr? cause)
        | Try(stmt* body, excepthandler* handlers, stmt* orelse, stmt* finalbody)
        | Assert(expr test, expr? msg)

        | Import(alias* names)
        | ImportFrom(identifier? module, alias* names, int? level)

        | Global(identifier* names)
        | Nonlocal(identifier* names)
        | Expr(expr value)
        | Pass | Break | Continue

        -- XXX Jython will be different
        -- col_offset is the byte offset in the utf8 string the parser uses
        attributes (int lineno, int col_offset)

        -- BoolOp() can use left & right?
    expr = BoolOp(boolop op, expr* values)
         | BinOp(expr left, operator op, expr right)
         | UnaryOp(unaryop op, expr operand)
         | Lambda(arguments args, expr body)
         | IfExp(expr test, expr body, expr orelse)
         | Dict(expr* keys, expr* values)
         | Set(expr* elts)
         | ListComp(expr elt, comprehension* generators)
         | SetComp(expr elt, comprehension* generators)
         | DictComp(expr key, expr value, comprehension* generators)
         | GeneratorExp(expr elt, comprehension* generators)
        -- the grammar constrains where yield expressions can occur
```

(continues on next page)

(continued from previous page)

```
        | Await(expr value)
        | Yield(expr? value)
        | YieldFrom(expr value)
        -- need sequences for compare to distinguish between
        -- x < 4 < 3 and (x < 4) < 3
        | Compare(expr left, cmpop* ops, expr* comparators)
        | Call(expr func, expr* args, keyword* keywords)
        | Num(object n) -- a number as a PyObject.
        | Str(string s) -- need to specify raw, unicode, etc?
        | FormattedValue(expr value, int? conversion, expr? format_spec)
        | JoinedStr(expr* values)
        | Bytes(bytes s)
        | NameConstant(singleton value)
        | Ellipsis
        | Constant(constant value)

        -- the following expression can appear in assignment context
        | Attribute(expr value, identifier attr, expr_context ctx)
        | Subscript(expr value, slice slice, expr_context ctx)
        | Starred(expr value, expr_context ctx)
        | Name(identifier id, expr_context ctx)
        | List(expr* elts, expr_context ctx)
        | Tuple(expr* elts, expr_context ctx)

        -- col_offset is the byte offset in the utf8 string the parser uses
        attributes (int lineno, int col_offset)

    expr_context = Load | Store | Del | AugLoad | AugStore | Param

    slice = Slice(expr? lower, expr? upper, expr? step)
          | ExtSlice(slice* dims)
          | Index(expr value)

    boolop = And | Or

    operator = Add | Sub | Mult | MatMult | Div | Mod | Pow | LShift
                 | RShift | BitOr | BitXor | BitAnd | FloorDiv

    unaryop = Invert | Not | UAdd | USub

    cmpop = Eq | NotEq | Lt | LtE | Gt | GtE | Is | IsNot | In | NotIn

    comprehension = (expr target, expr iter, expr* ifs, int is_async)

    excepthandler = ExceptHandler(expr? type, identifier? name, stmt* body)
                    attributes (int lineno, int col_offset)

    arguments = (arg* args, arg? vararg, arg* kwonlyargs, expr* kw_defaults,
                 arg? kwarg, expr* defaults)

    arg = (identifier arg, expr? annotation)
           attributes (int lineno, int col_offset)
```

(continues on next page)

(continued from previous page)

```
    -- keyword arguments supplied to call (NULL identifier for **kwargs)
    keyword = (identifier? arg, expr value)

    -- import name with optional 'as' alias.
    alias = (identifier name, identifier? asname)

    withitem = (expr context_expr, expr? optional_vars)
}
```

33.2.3 `ast` Helpers

Apart from the node classes, the *ast* module defines these utility functions and classes for traversing abstract syntax trees:

`ast.parse`(*source*, *filename='<unknown>'*, *mode='exec'*)

Parse the source into an AST node. Equivalent to `compile(source, filename, mode, ast.PyCF_ONLY_AST)`.

> **Warning:** It is possible to crash the Python interpreter with a sufficiently large/complex string due to stack depth limitations in Python's AST compiler.

`ast.literal_eval`(*node_or_string*)

Safely evaluate an expression node or a string containing a Python literal or container display. The string or node provided may only consist of the following Python literal structures: strings, bytes, numbers, tuples, lists, dicts, sets, booleans, and `None`.

This can be used for safely evaluating strings containing Python values from untrusted sources without the need to parse the values oneself. It is not capable of evaluating arbitrarily complex expressions, for example involving operators or indexing.

> **Warning:** It is possible to crash the Python interpreter with a sufficiently large/complex string due to stack depth limitations in Python's AST compiler.

Changed in version 3.2: Now allows bytes and set literals.

`ast.get_docstring`(*node*, *clean=True*)

Return the docstring of the given *node* (which must be a `FunctionDef`, `AsyncFunctionDef`, `ClassDef`, or `Module` node), or `None` if it has no docstring. If *clean* is true, clean up the docstring's indentation with *inspect.cleandoc()*.

Changed in version 3.5: `AsyncFunctionDef` is now supported.

`ast.fix_missing_locations`(*node*)

When you compile a node tree with *compile()*, the compiler expects `lineno` and `col_offset` attributes for every node that supports them. This is rather tedious to fill in for generated nodes, so this helper adds these attributes recursively where not already set, by setting them to the values of the parent node. It works recursively starting at *node*.

`ast.increment_lineno`(*node*, *n=1*)

Increment the line number of each node in the tree starting at *node* by *n*. This is useful to "move code"

to a different location in a file.

ast.copy_location(*new_node*, *old_node*)
Copy source location (`lineno` and `col_offset`) from *old_node* to *new_node* if possible, and return *new_node*.

ast.iter_fields(*node*)
Yield a tuple of `(fieldname, value)` for each field in `node._fields` that is present on *node*.

ast.iter_child_nodes(*node*)
Yield all direct child nodes of *node*, that is, all fields that are nodes and all items of fields that are lists of nodes.

ast.walk(*node*)
Recursively yield all descendant nodes in the tree starting at *node* (including *node* itself), in no specified order. This is useful if you only want to modify nodes in place and don't care about the context.

class ast.NodeVisitor
A node visitor base class that walks the abstract syntax tree and calls a visitor function for every node found. This function may return a value which is forwarded by the *visit()* method.

This class is meant to be subclassed, with the subclass adding visitor methods.

visit(*node*)
Visit a node. The default implementation calls the method called `self.visit_`*classname* where *classname* is the name of the node class, or *generic_visit()* if that method doesn't exist.

generic_visit(*node*)
This visitor calls *visit()* on all children of the node.

Note that child nodes of nodes that have a custom visitor method won't be visited unless the visitor calls *generic_visit()* or visits them itself.

Don't use the *NodeVisitor* if you want to apply changes to nodes during traversal. For this a special visitor exists (*NodeTransformer*) that allows modifications.

class ast.NodeTransformer
A *NodeVisitor* subclass that walks the abstract syntax tree and allows modification of nodes.

The *NodeTransformer* will walk the AST and use the return value of the visitor methods to replace or remove the old node. If the return value of the visitor method is `None`, the node will be removed from its location, otherwise it is replaced with the return value. The return value may be the original node in which case no replacement takes place.

Here is an example transformer that rewrites all occurrences of name lookups (`foo`) to `data['foo']`:

```
class RewriteName(NodeTransformer):

    def visit_Name(self, node):
        return copy_location(Subscript(
            value=Name(id='data', ctx=Load()),
            slice=Index(value=Str(s=node.id)),
            ctx=node.ctx
        ), node)
```

Keep in mind that if the node you're operating on has child nodes you must either transform the child nodes yourself or call the `generic_visit()` method for the node first.

For nodes that were part of a collection of statements (that applies to all statement nodes), the visitor may also return a list of nodes rather than just a single node.

Usually you use the transformer like this:

```
node = YourTransformer().visit(node)
```

ast.dump(*node*, *annotate_fields=True*, *include_attributes=False*)
Return a formatted dump of the tree in *node*. This is mainly useful for debugging purposes. The returned string will show the names and the values for fields. This makes the code impossible to evaluate, so if evaluation is wanted *annotate_fields* must be set to `False`. Attributes such as line numbers and column offsets are not dumped by default. If this is wanted, *include_attributes* can be set to `True`.

See also:

Green Tree Snakes, an external documentation resource, has good details on working with Python ASTs.

33.3 `symtable` — Access to the compiler's symbol tables

Source code: Lib/symtable.py

Symbol tables are generated by the compiler from AST just before bytecode is generated. The symbol table is responsible for calculating the scope of every identifier in the code. *symtable* provides an interface to examine these tables.

33.3.1 Generating Symbol Tables

symtable.symtable(*code*, *filename*, *compile_type*)
Return the toplevel *SymbolTable* for the Python source *code*. *filename* is the name of the file containing the code. *compile_type* is like the *mode* argument to *compile()*.

33.3.2 Examining Symbol Tables

class symtable.SymbolTable
A namespace table for a block. The constructor is not public.

get_type()
Return the type of the symbol table. Possible values are `'class'`, `'module'`, and `'function'`.

get_id()
Return the table's identifier.

get_name()
Return the table's name. This is the name of the class if the table is for a class, the name of the function if the table is for a function, or `'top'` if the table is global (*get_type()* returns `'module'`).

get_lineno()
Return the number of the first line in the block this table represents.

is_optimized()
Return `True` if the locals in this table can be optimized.

is_nested()
Return `True` if the block is a nested class or function.

has_children()
Return `True` if the block has nested namespaces within it. These can be obtained with *get_children()*.

has_exec()
Return `True` if the block uses `exec`.

get_identifiers()
Return a list of names of symbols in this table.

lookup(*name*)
Lookup *name* in the table and return a *Symbol* instance.

get_symbols()
Return a list of *Symbol* instances for names in the table.

get_children()
Return a list of the nested symbol tables.

class symtable.Function
A namespace for a function or method. This class inherits *SymbolTable*.

get_parameters()
Return a tuple containing names of parameters to this function.

get_locals()
Return a tuple containing names of locals in this function.

get_globals()
Return a tuple containing names of globals in this function.

get_frees()
Return a tuple containing names of free variables in this function.

class symtable.Class
A namespace of a class. This class inherits *SymbolTable*.

get_methods()
Return a tuple containing the names of methods declared in the class.

class symtable.Symbol
An entry in a *SymbolTable* corresponding to an identifier in the source. The constructor is not public.

get_name()
Return the symbol's name.

is_referenced()
Return `True` if the symbol is used in its block.

is_imported()
Return `True` if the symbol is created from an import statement.

is_parameter()
Return `True` if the symbol is a parameter.

is_global()
Return `True` if the symbol is global.

is_declared_global()
Return `True` if the symbol is declared global with a global statement.

is_local()
Return `True` if the symbol is local to its block.

is_free()
Return `True` if the symbol is referenced in its block, but not assigned to.

is_assigned()
Return `True` if the symbol is assigned to in its block.

`is_namespace()`
Return `True` if name binding introduces new namespace.

If the name is used as the target of a function or class statement, this will be true.

For example:

```
>>> table = symtable.symtable("def some_func(): pass", "string", "exec")
>>> table.lookup("some_func").is_namespace()
True
```

Note that a single name can be bound to multiple objects. If the result is `True`, the name may also be bound to other objects, like an int or list, that does not introduce a new namespace.

`get_namespaces()`
Return a list of namespaces bound to this name.

`get_namespace()`
Return the namespace bound to this name. If more than one namespace is bound, *ValueError* is raised.

33.4 `symbol` — Constants used with Python parse trees

Source code: Lib/symbol.py

This module provides constants which represent the numeric values of internal nodes of the parse tree. Unlike most Python constants, these use lower-case names. Refer to the file `Grammar/Grammar` in the Python distribution for the definitions of the names in the context of the language grammar. The specific numeric values which the names map to may change between Python versions.

This module also provides one additional data object:

`symbol.sym_name`
Dictionary mapping the numeric values of the constants defined in this module back to name strings, allowing more human-readable representation of parse trees to be generated.

33.5 `token` — Constants used with Python parse trees

Source code: Lib/token.py

This module provides constants which represent the numeric values of leaf nodes of the parse tree (terminal tokens). Refer to the file `Grammar/Grammar` in the Python distribution for the definitions of the names in the context of the language grammar. The specific numeric values which the names map to may change between Python versions.

The module also provides a mapping from numeric codes to names and some functions. The functions mirror definitions in the Python C header files.

`token.tok_name`
Dictionary mapping the numeric values of the constants defined in this module back to name strings, allowing more human-readable representation of parse trees to be generated.

`token.ISTERMINAL(`*x*`)`
Return true for terminal token values.

token.ISNONTERMINAL(*x*)
: Return true for non-terminal token values.

token.ISEOF(*x*)
: Return true if *x* is the marker indicating the end of input.

The token constants are:

token.ENDMARKER
token.NAME
token.NUMBER
token.STRING
token.NEWLINE
token.INDENT
token.DEDENT
token.LPAR
token.RPAR
token.LSQB
token.RSQB
token.COLON
token.COMMA
token.SEMI
token.PLUS
token.MINUS
token.STAR
token.SLASH
token.VBAR
token.AMPER
token.LESS
token.GREATER
token.EQUAL
token.DOT
token.PERCENT
token.LBRACE
token.RBRACE
token.EQEQUAL
token.NOTEQUAL
token.LESSEQUAL
token.GREATEREQUAL
token.TILDE
token.CIRCUMFLEX
token.LEFTSHIFT
token.RIGHTSHIFT
token.DOUBLESTAR
token.PLUSEQUAL
token.MINEQUAL
token.STAREQUAL
token.SLASHEQUAL
token.PERCENTEQUAL
token.AMPEREQUAL
token.VBAREQUAL
token.CIRCUMFLEXEQUAL
token.LEFTSHIFTEQUAL
token.RIGHTSHIFTEQUAL
token.DOUBLESTAREQUAL
token.DOUBLESLASH

token.DOUBLESLASHEQUAL
token.AT
token.ATEQUAL
token.RARROW
token.ELLIPSIS
token.OP
token.ERRORTOKEN
token.N_TOKENS
token.NT_OFFSET

The following token type values aren't used by the C tokenizer but are needed for the *tokenize* module.

token.COMMENT
Token value used to indicate a comment.

token.NL
Token value used to indicate a non-terminating newline. The *NEWLINE* token indicates the end of a logical line of Python code; NL tokens are generated when a logical line of code is continued over multiple physical lines.

token.ENCODING
Token value that indicates the encoding used to decode the source bytes into text. The first token returned by *tokenize.tokenize()* will always be an ENCODING token.

Changed in version 3.5: Added AWAIT and ASYNC tokens.

Changed in version 3.7: Added *COMMENT*, *NL* and *ENCODING* tokens.

Changed in version 3.7: Removed AWAIT and ASYNC tokens. "async" and "await" are now tokenized as *NAME* tokens.

33.6 keyword — Testing for Python keywords

Source code: Lib/keyword.py

This module allows a Python program to determine if a string is a keyword.

keyword.iskeyword(*s*)
Return true if *s* is a Python keyword.

keyword.kwlist
Sequence containing all the keywords defined for the interpreter. If any keywords are defined to only be active when particular *__future__* statements are in effect, these will be included as well.

33.7 tokenize — Tokenizer for Python source

Source code: Lib/tokenize.py

The *tokenize* module provides a lexical scanner for Python source code, implemented in Python. The scanner in this module returns comments as tokens as well, making it useful for implementing "pretty-printers," including colorizers for on-screen displays.

To simplify token stream handling, all operator and delimiter tokens and *Ellipsis* are returned using the generic *OP* token type. The exact type can be determined by checking the exact_type property on the *named tuple* returned from *tokenize.tokenize()*.

33.7.1 Tokenizing Input

The primary entry point is a *generator*:

`tokenize.tokenize`(*readline*)

The *tokenize()* generator requires one argument, *readline*, which must be a callable object which provides the same interface as the *io.IOBase.readline()* method of file objects. Each call to the function should return one line of input as bytes.

The generator produces 5-tuples with these members: the token type; the token string; a 2-tuple `(srow, scol)` of ints specifying the row and column where the token begins in the source; a 2-tuple `(erow, ecol)` of ints specifying the row and column where the token ends in the source; and the line on which the token was found. The line passed (the last tuple item) is the *logical* line; continuation lines are included. The 5 tuple is returned as a *named tuple* with the field names: `type string start end line`.

The returned *named tuple* has an additional property named `exact_type` that contains the exact operator type for *OP* tokens. For all other token types `exact_type` equals the named tuple `type` field.

Changed in version 3.1: Added support for named tuples.

Changed in version 3.3: Added support for `exact_type`.

tokenize() determines the source encoding of the file by looking for a UTF-8 BOM or encoding cookie, according to **PEP 263**.

All constants from the *token* module are also exported from *tokenize*.

Another function is provided to reverse the tokenization process. This is useful for creating tools that tokenize a script, modify the token stream, and write back the modified script.

`tokenize.untokenize`(*iterable*)

Converts tokens back into Python source code. The *iterable* must return sequences with at least two elements, the token type and the token string. Any additional sequence elements are ignored.

The reconstructed script is returned as a single string. The result is guaranteed to tokenize back to match the input so that the conversion is lossless and round-trips are assured. The guarantee applies only to the token type and token string as the spacing between tokens (column positions) may change.

It returns bytes, encoded using the *ENCODING* token, which is the first token sequence output by *tokenize()*.

tokenize() needs to detect the encoding of source files it tokenizes. The function it uses to do this is available:

`tokenize.detect_encoding`(*readline*)

The *detect_encoding()* function is used to detect the encoding that should be used to decode a Python source file. It requires one argument, readline, in the same way as the *tokenize()* generator.

It will call readline a maximum of twice, and return the encoding used (as a string) and a list of any lines (not decoded from bytes) it has read in.

It detects the encoding from the presence of a UTF-8 BOM or an encoding cookie as specified in **PEP 263**. If both a BOM and a cookie are present, but disagree, a *SyntaxError* will be raised. Note that if the BOM is found, `'utf-8-sig'` will be returned as an encoding.

If no encoding is specified, then the default of `'utf-8'` will be returned.

Use *open()* to open Python source files: it uses *detect_encoding()* to detect the file encoding.

`tokenize.open`(*filename*)

Open a file in read only mode using the encoding detected by *detect_encoding()*.

New in version 3.2.

exception tokenize.TokenError
Raised when either a docstring or expression that may be split over several lines is not completed anywhere in the file, for example:

```
"""Beginning of
docstring
```

or:

```
[1,
 2,
 3
```

Note that unclosed single-quoted strings do not cause an error to be raised. They are tokenized as *ERRORTOKEN*, followed by the tokenization of their contents.

33.7.2 Command-Line Usage

New in version 3.3.

The *tokenize* module can be executed as a script from the command line. It is as simple as:

```
python -m tokenize [-e] [filename.py]
```

The following options are accepted:

-h, --help
show this help message and exit

-e, --exact
display token names using the exact type

If `filename.py` is specified its contents are tokenized to stdout. Otherwise, tokenization is performed on stdin.

33.7.3 Examples

Example of a script rewriter that transforms float literals into Decimal objects:

```
from tokenize import tokenize, untokenize, NUMBER, STRING, NAME, OP
from io import BytesIO

def decistmt(s):
    """Substitute Decimals for floats in a string of statements.

    >>> from decimal import Decimal
    >>> s = 'print(+21.3e-5*-.1234/81.7)'
    >>> decistmt(s)
    "print (+Decimal ('21.3e-5')*-Decimal ('.1234')/Decimal ('81.7'))"

    The format of the exponent is inherited from the platform C library.
    Known cases are "e-007" (Windows) and "e-07" (not Windows).  Since
    we're only showing 12 digits, and the 13th isn't close to 5, the
    rest of the output should be platform-independent.
```

(continues on next page)

(continued from previous page)

```
    >>> exec(s)  #doctest: +ELLIPSIS
    -3.21716034272e-0...7

    Output from calculations with Decimal should be identical across all
    platforms.

    >>> exec(decistmt(s))
    -3.217160342717258261933904529E-7
    """
    result = []
    g = tokenize(BytesIO(s.encode('utf-8')).readline)  # tokenize the string
    for toknum, tokval, _, _, _ in g:
        if toknum == NUMBER and '.' in tokval:  # replace NUMBER tokens
            result.extend([
                (NAME, 'Decimal'),
                (OP, '('),
                (STRING, repr(tokval)),
                (OP, ')')
            ])
        else:
            result.append((toknum, tokval))
    return untokenize(result).decode('utf-8')
```

Example of tokenizing from the command line. The script:

```
def say_hello():
    print("Hello, World!")

say_hello()
```

will be tokenized to the following output where the first column is the range of the line/column coordinates where the token is found, the second column is the name of the token, and the final column is the value of the token (if any)

```
$ python -m tokenize hello.py
0,0-0,0:            ENCODING       'utf-8'
1,0-1,3:            NAME           'def'
1,4-1,13:           NAME           'say_hello'
1,13-1,14:          OP             '('
1,14-1,15:          OP             ')'
1,15-1,16:          OP             ':'
1,16-1,17:          NEWLINE        '\n'
2,0-2,4:            INDENT         '    '
2,4-2,9:            NAME           'print'
2,9-2,10:           OP             '('
2,10-2,25:          STRING         '"Hello, World!"'
2,25-2,26:          OP             ')'
2,26-2,27:          NEWLINE        '\n'
3,0-3,1:            NL             '\n'
4,0-4,0:            DEDENT         ''
4,0-4,9:            NAME           'say_hello'
4,9-4,10:           OP             '('
```

(continues on next page)

(continued from previous page)

```
4,10-4,11:          OP             ')'
4,11-4,12:          NEWLINE        '\n'
5,0-5,0:            ENDMARKER      ''
```

The exact token type names can be displayed using the `-e` option:

```
$ python -m tokenize -e hello.py
0,0-0,0:            ENCODING       'utf-8'
1,0-1,3:            NAME           'def'
1,4-1,13:           NAME           'say_hello'
1,13-1,14:          LPAR           '('
1,14-1,15:          RPAR           ')'
1,15-1,16:          COLON          ':'
1,16-1,17:          NEWLINE        '\n'
2,0-2,4:            INDENT         '    '
2,4-2,9:            NAME           'print'
2,9-2,10:           LPAR           '('
2,10-2,25:          STRING         '"Hello, World!"'
2,25-2,26:          RPAR           ')'
2,26-2,27:          NEWLINE        '\n'
3,0-3,1:            NL             '\n'
4,0-4,0:            DEDENT         ''
4,0-4,9:            NAME           'say_hello'
4,9-4,10:           LPAR           '('
4,10-4,11:          RPAR           ')'
4,11-4,12:          NEWLINE        '\n'
5,0-5,0:            ENDMARKER      ''
```

33.8 tabnanny — Detection of ambiguous indentation

Source code: Lib/tabnanny.py

For the time being this module is intended to be called as a script. However it is possible to import it into an IDE and use the function *check()* described below.

Note: The API provided by this module is likely to change in future releases; such changes may not be backward compatible.

`tabnanny.check(`*file_or_dir*`)`

If *file_or_dir* is a directory and not a symbolic link, then recursively descend the directory tree named by *file_or_dir*, checking all `.py` files along the way. If *file_or_dir* is an ordinary Python source file, it is checked for whitespace related problems. The diagnostic messages are written to standard output using the *print()* function.

`tabnanny.verbose`

Flag indicating whether to print verbose messages. This is incremented by the `-v` option if called as a script.

`tabnanny.filename_only`

Flag indicating whether to print only the filenames of files containing whitespace related problems. This is set to true by the `-q` option if called as a script.

exception tabnanny.**NannyNag**
Raised by *process_tokens()* if detecting an ambiguous indent. Captured and handled in *check()*.

tabnanny.**process_tokens**(*tokens*)
This function is used by *check()* to process tokens generated by the *tokenize* module.

See also:

Module *tokenize* Lexical scanner for Python source code.

33.9 pyclbr — Python class browser support

Source code: Lib/pyclbr.py

The *pyclbr* module provides limited information about the functions, classes, and methods defined in a Python-coded module. The information is sufficient to implement a module browser. The information is extracted from the Python source code rather than by importing the module, so this module is safe to use with untrusted code. This restriction makes it impossible to use this module with modules not implemented in Python, including all standard and optional extension modules.

pyclbr.**readmodule**(*module*, *path=None*)
Return a dictionary mapping module-level class names to class descriptors. If possible, descriptors for imported base classes are included. Parameter *module* is a string with the name of the module to read; it may be the name of a module within a package. If given, *path* is a sequence of directory paths prepended to sys.path, which is used to locate the module source code.

pyclbr.**readmodule_ex**(*module*, *path=None*)
Return a dictionary-based tree containing a function or class descriptors for each function and class defined in the module with a def or class statement. The returned dictionary maps module-level function and class names to their descriptors. Nested objects are entered into the children dictionary of their parent. As with readmodule, *module* names the module to be read and *path* is prepended to sys.path. If the module being read is a package, the returned dictionary has a key '__path__' whose value is a list containing the package search path.

New in version 3.7: Descriptors for nested definitions. They are accessed through the new children attibute. Each has a new parent attribute.

The descriptors returned by these functions are instances of Function and Class classes. Users are not expected to create instances of these classes.

33.9.1 Function Objects

Class Function instances describe functions defined by def statements. They have the following attributes:

Function.**file**
Name of the file in which the function is defined.

Function.**module**
The name of the module defining the function described.

Function.**name**
The name of the function.

Function.**lineno**
The line number in the file where the definition starts.

`Function.parent`
For top-level functions, None. For nested functions, the parent.

New in version 3.7.

`Function.children`
A dictionary mapping names to descriptors for nested functions and classes.

New in version 3.7.

33.9.2 Class Objects

Class `Class` instances describe classes defined by class statements. They have the same attributes as Functions and two more.

`Class.file`
Name of the file in which the class is defined.

`Class.module`
The name of the module defining the class described.

`Class.name`
The name of the class.

`Class.lineno`
The line number in the file where the definition starts.

`Class.parent`
For top-level classes, None. For nested classes, the parent.

New in version 3.7.

`Class.children`
A dictionary mapping names to descriptors for nested functions and classes.

New in version 3.7.

`Class.super`
A list of `Class` objects which describe the immediate base classes of the class being described. Classes which are named as superclasses but which are not discoverable by *readmodule_ex()* are listed as a string with the class name instead of as `Class` objects.

`Class.methods`
A dictionary mapping method names to line numbers. This can be derived from the newer children dictionary, but remains for back-compatibility.

33.10 py_compile — Compile Python source files

Source code: Lib/py_compile.py

The *py_compile* module provides a function to generate a byte-code file from a source file, and another function used when the module source file is invoked as a script.

Though not often needed, this function can be useful when installing modules for shared use, especially if some of the users may not have permission to write the byte-code cache files in the directory containing the source code.

exception py_compile.**PyCompileError**
Exception raised when an error occurs while attempting to compile the file.

py_compile.compile(*file*, *cfile=None*, *dfile=None*, *doraise=False*, *optimize=-1*, *invalidation_mode=PycInvalidationMode.TIMESTAMP*)

Compile a source file to byte-code and write out the byte-code cache file. The source code is loaded from the file named *file*. The byte-code is written to *cfile*, which defaults to the **PEP 3147**/**PEP 488** path, ending in `.pyc`. For example, if *file* is `/foo/bar/baz.py` *cfile* will default to `/foo/bar/__pycache__/baz.cpython-32.pyc` for Python 3.2. If *dfile* is specified, it is used as the name of the source file in error messages when instead of *file*. If *doraise* is true, a *PyCompileError* is raised when an error is encountered while compiling *file*. If *doraise* is false (the default), an error string is written to `sys.stderr`, but no exception is raised. This function returns the path to byte-compiled file, i.e. whatever *cfile* value was used.

If the path that *cfile* becomes (either explicitly specified or computed) is a symlink or non-regular file, *FileExistsError* will be raised. This is to act as a warning that import will turn those paths into regular files if it is allowed to write byte-compiled files to those paths. This is a side-effect of import using file renaming to place the final byte-compiled file into place to prevent concurrent file writing issues.

optimize controls the optimization level and is passed to the built-in *compile()* function. The default of `-1` selects the optimization level of the current interpreter.

invalidation_mode should be a member of the *PycInvalidationMode* enum and controls how the generated bytecode cache is invalidated at runtime. The default is *PycInvalidationMode.CHECKED_HASH* if the `SOURCE_DATE_EPOCH` environment variable is set, otherwise the default is *PycInvalidationMode.TIMESTAMP*.

Changed in version 3.2: Changed default value of *cfile* to be **PEP 3147**-compliant. Previous default was *file* + `'c'` (`'o'` if optimization was enabled). Also added the *optimize* parameter.

Changed in version 3.4: Changed code to use *importlib* for the byte-code cache file writing. This means file creation/writing semantics now match what *importlib* does, e.g. permissions, write-and-move semantics, etc. Also added the caveat that *FileExistsError* is raised if *cfile* is a symlink or non-regular file.

Changed in version 3.7: The *invalidation_mode* parameter was added as specified in **PEP 552**. If the `SOURCE_DATE_EPOCH` environment variable is set, *invalidation_mode* will be forced to *PycInvalidationMode.CHECKED_HASH*.

Changed in version 3.7.2: The `SOURCE_DATE_EPOCH` environment variable no longer overrides the value of the *invalidation_mode* argument, and determines its default value instead.

class py_compile.PycInvalidationMode

A enumeration of possible methods the interpreter can use to determine whether a bytecode file is up to date with a source file. The `.pyc` file indicates the desired invalidation mode in its header. See pyc-invalidation for more information on how Python invalidates `.pyc` files at runtime.

New in version 3.7.

`TIMESTAMP`
: The `.pyc` file includes the timestamp and size of the source file, which Python will compare against the metadata of the source file at runtime to determine if the `.pyc` file needs to be regenerated.

`CHECKED_HASH`
: The `.pyc` file includes a hash of the source file content, which Python will compare against the source at runtime to determine if the `.pyc` file needs to be regenerated.

`UNCHECKED_HASH`
: Like *CHECKED_HASH*, the `.pyc` file includes a hash of the source file content. However, Python will at runtime assume the `.pyc` file is up to date and not validate the `.pyc` against the source file at all.

  This option is useful when the `.pycs` are kept up to date by some system external to Python like a build system.

py_compile.main(*args=None*)
: Compile several source files. The files named in *args* (or on the command line, if *args* is None) are compiled and the resulting byte-code is cached in the normal manner. This function does not search a directory structure to locate source files; it only compiles files named explicitly. If '-' is the only parameter in args, the list of files is taken from standard input.

  Changed in version 3.2: Added support for '-'.

When this module is run as a script, the *main()* is used to compile all the files named on the command line. The exit status is nonzero if one of the files could not be compiled.

See also:

Module *compileall* Utilities to compile all Python source files in a directory tree.

33.11 compileall — Byte-compile Python libraries

Source code: Lib/compileall.py

This module provides some utility functions to support installing Python libraries. These functions compile Python source files in a directory tree. This module can be used to create the cached byte-code files at library installation time, which makes them available for use even by users who don't have write permission to the library directories.

33.11.1 Command-line use

This module can work as a script (using **python -m compileall**) to compile Python sources.

directory ...
file ...
: Positional arguments are files to compile or directories that contain source files, traversed recursively. If no argument is given, behave as if the command line was **-l <directories from sys.path>**.

-l
: Do not recurse into subdirectories, only compile source code files directly contained in the named or implied directories.

-f
: Force rebuild even if timestamps are up-to-date.

-q
: Do not print the list of files compiled. If passed once, error messages will still be printed. If passed twice (**-qq**), all output is suppressed.

-d destdir
: Directory prepended to the path to each file being compiled. This will appear in compilation time tracebacks, and is also compiled in to the byte-code file, where it will be used in tracebacks and other messages in cases where the source file does not exist at the time the byte-code file is executed.

-x regex
: regex is used to search the full path to each file considered for compilation, and if the regex produces a match, the file is skipped.

-i list
: Read the file **list** and add each line that it contains to the list of files and directories to compile. If **list** is **-**, read lines from **stdin**.

`-b`
: Write the byte-code files to their legacy locations and names, which may overwrite byte-code files created by another version of Python. The default is to write files to their **PEP 3147** locations and names, which allows byte-code files from multiple versions of Python to coexist.

`-r`
: Control the maximum recursion level for subdirectories. If this is given, then `-l` option will not be taken into account. **`python -m compileall <directory> -r 0`** is equivalent to **`python -m compileall <directory> -l`**.

`-j N`
: Use *N* workers to compile the files within the given directory. If `0` is used, then the result of *os.cpu_count()* will be used.

`--invalidation-mode [timestamp|checked-hash|unchecked-hash]`
: Control how the generated byte-code files are invalidated at runtime. The `timestamp` value, means that `.pyc` files with the source timestamp and size embedded will be generated. The `checked-hash` and `unchecked-hash` values cause hash-based pycs to be generated. Hash-based pycs embed a hash of the source file contents rather than a timestamp. See pyc-invalidation for more information on how Python validates bytecode cache files at runtime. The default is `timestamp` if the `SOURCE_DATE_EPOCH` environment variable is not set, and `checked-hash` if the `SOURCE_DATE_EPOCH` environment variable is set.

Changed in version 3.2: Added the `-i`, `-b` and `-h` options.

Changed in version 3.5: Added the `-j`, `-r`, and `-qq` options. `-q` option was changed to a multilevel value. `-b` will always produce a byte-code file ending in `.pyc`, never `.pyo`.

Changed in version 3.7: Added the `--invalidation-mode` parameter.

There is no command-line option to control the optimization level used by the *compile()* function, because the Python interpreter itself already provides the option: **`python -O -m compileall`**.

33.11.2 Public functions

`compileall.compile_dir(`*dir*, *maxlevels=10*, *ddir=None*, *force=False*, *rx=None*, *quiet=0*, *legacy=False*, *optimize=-1*, *workers=1*, *invalidation_mode=py_compile.PycInvalidationMode.TIMESTAMP*`)`
: Recursively descend the directory tree named by *dir*, compiling all `.py` files along the way. Return a true value if all the files compiled successfully, and a false value otherwise.

  The *maxlevels* parameter is used to limit the depth of the recursion; it defaults to `10`.

  If *ddir* is given, it is prepended to the path to each file being compiled for use in compilation time tracebacks, and is also compiled in to the byte-code file, where it will be used in tracebacks and other messages in cases where the source file does not exist at the time the byte-code file is executed.

  If *force* is true, modules are re-compiled even if the timestamps are up to date.

  If *rx* is given, its search method is called on the complete path to each file considered for compilation, and if it returns a true value, the file is skipped.

  If *quiet* is `False` or `0` (the default), the filenames and other information are printed to standard out. Set to `1`, only errors are printed. Set to `2`, all output is suppressed.

  If *legacy* is true, byte-code files are written to their legacy locations and names, which may overwrite byte-code files created by another version of Python. The default is to write files to their **PEP 3147** locations and names, which allows byte-code files from multiple versions of Python to coexist.

  optimize specifies the optimization level for the compiler. It is passed to the built-in *compile()* function.

The argument *workers* specifies how many workers are used to compile files in parallel. The default is to not use multiple workers. If the platform can't use multiple workers and *workers* argument is given, then sequential compilation will be used as a fallback. If *workers* is lower than `0`, a *ValueError* will be raised.

invalidation_mode should be a member of the *py_compile.PycInvalidationMode* enum and controls how the generated pycs are invalidated at runtime.

Changed in version 3.2: Added the *legacy* and *optimize* parameter.

Changed in version 3.5: Added the *workers* parameter.

Changed in version 3.5: *quiet* parameter was changed to a multilevel value.

Changed in version 3.5: The *legacy* parameter only writes out `.pyc` files, not `.pyo` files no matter what the value of *optimize* is.

Changed in version 3.6: Accepts a *path-like object*.

Changed in version 3.7: The *invalidation_mode* parameter was added.

`compileall.compile_file`(*fullname*, *ddir=None*, *force=False*, *rx=None*, *quiet=0*, *legacy=False*, *optimize=-1*, *invalidation_mode=py_compile.PycInvalidationMode.TIMESTAMP*)

Compile the file with path *fullname*. Return a true value if the file compiled successfully, and a false value otherwise.

If *ddir* is given, it is prepended to the path to the file being compiled for use in compilation time tracebacks, and is also compiled in to the byte-code file, where it will be used in tracebacks and other messages in cases where the source file does not exist at the time the byte-code file is executed.

If *rx* is given, its search method is passed the full path name to the file being compiled, and if it returns a true value, the file is not compiled and `True` is returned.

If *quiet* is `False` or `0` (the default), the filenames and other information are printed to standard out. Set to `1`, only errors are printed. Set to `2`, all output is suppressed.

If *legacy* is true, byte-code files are written to their legacy locations and names, which may overwrite byte-code files created by another version of Python. The default is to write files to their **PEP 3147** locations and names, which allows byte-code files from multiple versions of Python to coexist.

optimize specifies the optimization level for the compiler. It is passed to the built-in *compile()* function.

invalidation_mode should be a member of the *py_compile.PycInvalidationMode* enum and controls how the generated pycs are invalidated at runtime.

New in version 3.2.

Changed in version 3.5: *quiet* parameter was changed to a multilevel value.

Changed in version 3.5: The *legacy* parameter only writes out `.pyc` files, not `.pyo` files no matter what the value of *optimize* is.

Changed in version 3.7: The *invalidation_mode* parameter was added.

`compileall.compile_path`(*skip_curdir=True*, *maxlevels=0*, *force=False*, *quiet=0*, *legacy=False*, *optimize=-1*, *invalidation_mode=py_compile.PycInvalidationMode.TIMESTAMP*)

Byte-compile all the `.py` files found along `sys.path`. Return a true value if all the files compiled successfully, and a false value otherwise.

If *skip_curdir* is true (the default), the current directory is not included in the search. All other parameters are passed to the *compile_dir()* function. Note that unlike the other compile functions, `maxlevels` defaults to `0`.

Changed in version 3.2: Added the *legacy* and *optimize* parameter.

Changed in version 3.5: *quiet* parameter was changed to a multilevel value.

Changed in version 3.5: The *legacy* parameter only writes out `.pyc` files, not `.pyo` files no matter what the value of *optimize* is.

Changed in version 3.7: The *invalidation_mode* parameter was added.

To force a recompile of all the `.py` files in the `Lib/` subdirectory and all its subdirectories:

```
import compileall

compileall.compile_dir('Lib/', force=True)

# Perform same compilation, excluding files in .svn directories.
import re
compileall.compile_dir('Lib/', rx=re.compile(r'[/\\][.]svn'), force=True)

# pathlib.Path objects can also be used.
import pathlib
compileall.compile_dir(pathlib.Path('Lib/'), force=True)
```

See also:

Module *py_compile* Byte-compile a single source file.

33.12 `dis` — Disassembler for Python bytecode

Source code: Lib/dis.py

The *dis* module supports the analysis of CPython *bytecode* by disassembling it. The CPython bytecode which this module takes as an input is defined in the file `Include/opcode.h` and used by the compiler and the interpreter.

CPython implementation detail: Bytecode is an implementation detail of the CPython interpreter. No guarantees are made that bytecode will not be added, removed, or changed between versions of Python. Use of this module should not be considered to work across Python VMs or Python releases.

Changed in version 3.6: Use 2 bytes for each instruction. Previously the number of bytes varied by instruction.

Example: Given the function `myfunc()`:

```
def myfunc(alist):
    return len(alist)
```

the following command can be used to display the disassembly of `myfunc()`:

```
>>> dis.dis(myfunc)
  2           0 LOAD_GLOBAL              0 (len)
              2 LOAD_FAST                0 (alist)
              4 CALL_FUNCTION            1
              6 RETURN_VALUE
```

(The “2” is a line number).

33.12.1 Bytecode analysis

New in version 3.4.

The bytecode analysis API allows pieces of Python code to be wrapped in a *Bytecode* object that provides easy access to details of the compiled code.

class dis.Bytecode(*x*, ***, *first_line=None*, *current_offset=None*)

Analyse the bytecode corresponding to a function, generator, asynchronous generator, coroutine, method, string of source code, or a code object (as returned by *compile()*).

This is a convenience wrapper around many of the functions listed below, most notably *get_instructions()*, as iterating over a *Bytecode* instance yields the bytecode operations as *Instruction* instances.

If *first_line* is not `None`, it indicates the line number that should be reported for the first source line in the disassembled code. Otherwise, the source line information (if any) is taken directly from the disassembled code object.

If *current_offset* is not `None`, it refers to an instruction offset in the disassembled code. Setting this means *dis()* will display a "current instruction" marker against the specified opcode.

classmethod from_traceback(*tb*)

Construct a *Bytecode* instance from the given traceback, setting *current_offset* to the instruction responsible for the exception.

codeobj

The compiled code object.

first_line

The first source line of the code object (if available)

dis()

Return a formatted view of the bytecode operations (the same as printed by *dis.dis()*, but returned as a multi-line string).

info()

Return a formatted multi-line string with detailed information about the code object, like *code_info()*.

Changed in version 3.7: This can now handle coroutine and asynchronous generator objects.

Example:

```
>>> bytecode = dis.Bytecode(myfunc)
>>> for instr in bytecode:
...     print(instr.opname)
...
LOAD_GLOBAL
LOAD_FAST
CALL_FUNCTION
RETURN_VALUE
```

33.12.2 Analysis functions

The *dis* module also defines the following analysis functions that convert the input directly to the desired output. They can be useful if only a single operation is being performed, so the intermediate analysis object isn't useful:

dis.code_info(*x*)
Return a formatted multi-line string with detailed code object information for the supplied function, generator, asynchronous generator, coroutine, method, source code string or code object.

Note that the exact contents of code info strings are highly implementation dependent and they may change arbitrarily across Python VMs or Python releases.

New in version 3.2.

Changed in version 3.7: This can now handle coroutine and asynchronous generator objects.

dis.show_code(*x*, *, *file=None*)
Print detailed code object information for the supplied function, method, source code string or code object to *file* (or `sys.stdout` if *file* is not specified).

This is a convenient shorthand for `print(code_info(x), file=file)`, intended for interactive exploration at the interpreter prompt.

New in version 3.2.

Changed in version 3.4: Added *file* parameter.

dis.dis(*x=None*, *, *file=None*, *depth=None*)
Disassemble the *x* object. *x* can denote either a module, a class, a method, a function, a generator, an asynchronous generator, a coroutine, a code object, a string of source code or a byte sequence of raw bytecode. For a module, it disassembles all functions. For a class, it disassembles all methods (including class and static methods). For a code object or sequence of raw bytecode, it prints one line per bytecode instruction. It also recursively disassembles nested code objects (the code of comprehensions, generator expressions and nested functions, and the code used for building nested classes). Strings are first compiled to code objects with the *compile()* built-in function before being disassembled. If no object is provided, this function disassembles the last traceback.

The disassembly is written as text to the supplied *file* argument if provided and to `sys.stdout` otherwise.

The maximal depth of recursion is limited by *depth* unless it is `None`. `depth=0` means no recursion.

Changed in version 3.4: Added *file* parameter.

Changed in version 3.7: Implemented recursive disassembling and added *depth* parameter.

Changed in version 3.7: This can now handle coroutine and asynchronous generator objects.

dis.distb(*tb=None*, *, *file=None*)
Disassemble the top-of-stack function of a traceback, using the last traceback if none was passed. The instruction causing the exception is indicated.

The disassembly is written as text to the supplied *file* argument if provided and to `sys.stdout` otherwise.

Changed in version 3.4: Added *file* parameter.

dis.disassemble(*code*, *lasti=-1*, *, *file=None*)
dis.disco(*code*, *lasti=-1*, *, *file=None*)
Disassemble a code object, indicating the last instruction if *lasti* was provided. The output is divided in the following columns:

1. the line number, for the first instruction of each line
2. the current instruction, indicated as `-->`,
3. a labelled instruction, indicated with `>>`,
4. the address of the instruction,
5. the operation code name,

6. operation parameters, and
7. interpretation of the parameters in parentheses.

The parameter interpretation recognizes local and global variable names, constant values, branch targets, and compare operators.

The disassembly is written as text to the supplied *file* argument if provided and to `sys.stdout` otherwise.

Changed in version 3.4: Added *file* parameter.

`dis.get_instructions`(*x*, ***, *first_line=None*)
Return an iterator over the instructions in the supplied function, method, source code string or code object.

The iterator generates a series of *Instruction* named tuples giving the details of each operation in the supplied code.

If *first_line* is not `None`, it indicates the line number that should be reported for the first source line in the disassembled code. Otherwise, the source line information (if any) is taken directly from the disassembled code object.

New in version 3.4.

`dis.findlinestarts`(*code*)
This generator function uses the `co_firstlineno` and `co_lnotab` attributes of the code object *code* to find the offsets which are starts of lines in the source code. They are generated as `(offset, lineno)` pairs. See Objects/lnotab_notes.txt for the `co_lnotab` format and how to decode it.

Changed in version 3.6: Line numbers can be decreasing. Before, they were always increasing.

`dis.findlabels`(*code*)
Detect all offsets in the code object *code* which are jump targets, and return a list of these offsets.

`dis.stack_effect`(*opcode*[, *oparg*])
Compute the stack effect of *opcode* with argument *oparg*.

New in version 3.4.

33.12.3 Python Bytecode Instructions

The *get_instructions()* function and *Bytecode* class provide details of bytecode instructions as *Instruction* instances:

`class dis.Instruction`
Details for a bytecode operation

`opcode`
numeric code for operation, corresponding to the opcode values listed below and the bytecode values in the *Opcode collections*.

`opname`
human readable name for operation

`arg`
numeric argument to operation (if any), otherwise `None`

`argval`
resolved arg value (if known), otherwise same as arg

`argrepr`
human readable description of operation argument

offset
start index of operation within bytecode sequence

starts_line
line started by this opcode (if any), otherwise `None`

is_jump_target
`True` if other code jumps to here, otherwise `False`

New in version 3.4.

The Python compiler currently generates the following bytecode instructions.

General instructions

`NOP`
Do nothing code. Used as a placeholder by the bytecode optimizer.

`POP_TOP`
Removes the top-of-stack (TOS) item.

`ROT_TWO`
Swaps the two top-most stack items.

`ROT_THREE`
Lifts second and third stack item one position up, moves top down to position three.

`DUP_TOP`
Duplicates the reference on top of the stack.

New in version 3.2.

`DUP_TOP_TWO`
Duplicates the two references on top of the stack, leaving them in the same order.

New in version 3.2.

Unary operations

Unary operations take the top of the stack, apply the operation, and push the result back on the stack.

`UNARY_POSITIVE`
Implements `TOS = +TOS`.

`UNARY_NEGATIVE`
Implements `TOS = -TOS`.

`UNARY_NOT`
Implements `TOS = not TOS`.

`UNARY_INVERT`
Implements `TOS = ~TOS`.

`GET_ITER`
Implements `TOS = iter(TOS)`.

`GET_YIELD_FROM_ITER`
If `TOS` is a *generator iterator* or *coroutine* object it is left as is. Otherwise, implements `TOS = iter(TOS)`.

New in version 3.5.

Binary operations

Binary operations remove the top of the stack (TOS) and the second top-most stack item (TOS1) from the stack. They perform the operation, and put the result back on the stack.

`BINARY_POWER`
Implements `TOS = TOS1 ** TOS`.

`BINARY_MULTIPLY`
Implements `TOS = TOS1 * TOS`.

`BINARY_MATRIX_MULTIPLY`
Implements `TOS = TOS1 @ TOS`.

New in version 3.5.

`BINARY_FLOOR_DIVIDE`
Implements `TOS = TOS1 // TOS`.

`BINARY_TRUE_DIVIDE`
Implements `TOS = TOS1 / TOS`.

`BINARY_MODULO`
Implements `TOS = TOS1 % TOS`.

`BINARY_ADD`
Implements `TOS = TOS1 + TOS`.

`BINARY_SUBTRACT`
Implements `TOS = TOS1 - TOS`.

`BINARY_SUBSCR`
Implements `TOS = TOS1[TOS]`.

`BINARY_LSHIFT`
Implements `TOS = TOS1 << TOS`.

`BINARY_RSHIFT`
Implements `TOS = TOS1 >> TOS`.

`BINARY_AND`
Implements `TOS = TOS1 & TOS`.

`BINARY_XOR`
Implements `TOS = TOS1 ^ TOS`.

`BINARY_OR`
Implements `TOS = TOS1 | TOS`.

In-place operations

In-place operations are like binary operations, in that they remove TOS and TOS1, and push the result back on the stack, but the operation is done in-place when TOS1 supports it, and the resulting TOS may be (but does not have to be) the original TOS1.

`INPLACE_POWER`
Implements in-place `TOS = TOS1 ** TOS`.

`INPLACE_MULTIPLY`
Implements in-place `TOS = TOS1 * TOS`.

`INPLACE_MATRIX_MULTIPLY`
Implements in-place `TOS = TOS1 @ TOS`.

New in version 3.5.

`INPLACE_FLOOR_DIVIDE`
Implements in-place `TOS = TOS1 // TOS`.

`INPLACE_TRUE_DIVIDE`
Implements in-place `TOS = TOS1 / TOS`.

`INPLACE_MODULO`
Implements in-place `TOS = TOS1 % TOS`.

INPLACE_ADD
Implements in-place TOS = TOS1 + TOS.

INPLACE_SUBTRACT
Implements in-place TOS = TOS1 - TOS.

INPLACE_LSHIFT
Implements in-place TOS = TOS1 << TOS.

INPLACE_RSHIFT
Implements in-place TOS = TOS1 >> TOS.

INPLACE_AND
Implements in-place TOS = TOS1 & TOS.

INPLACE_XOR
Implements in-place TOS = TOS1 ^ TOS.

INPLACE_OR
Implements in-place TOS = TOS1 | TOS.

STORE_SUBSCR
Implements TOS1[TOS] = TOS2.

DELETE_SUBSCR
Implements del TOS1[TOS].

Coroutine opcodes

GET_AWAITABLE
Implements TOS = get_awaitable(TOS), where get_awaitable(o) returns o if o is a coroutine object or a generator object with the CO_ITERABLE_COROUTINE flag, or resolves o.__await__.

New in version 3.5.

GET_AITER
Implements TOS = TOS.__aiter__().

New in version 3.5.

Changed in version 3.7: Returning awaitable objects from __aiter__ is no longer supported.

GET_ANEXT
Implements PUSH(get_awaitable(TOS.__anext__())). See GET_AWAITABLE for details about get_awaitable

New in version 3.5.

BEFORE_ASYNC_WITH
Resolves __aenter__ and __aexit__ from the object on top of the stack. Pushes __aexit__ and result of __aenter__() to the stack.

New in version 3.5.

SETUP_ASYNC_WITH
Creates a new frame object.

New in version 3.5.

Miscellaneous opcodes

PRINT_EXPR
Implements the expression statement for the interactive mode. TOS is removed from the stack and printed. In non-interactive mode, an expression statement is terminated with *POP_TOP*.

BREAK_LOOP
Terminates a loop due to a break statement.

`CONTINUE_LOOP`(*target*)
Continues a loop due to a `continue` statement. *target* is the address to jump to (which should be a *FOR_ITER* instruction).

`SET_ADD`(*i*)
Calls `set.add(TOS1[-i], TOS)`. Used to implement set comprehensions.

`LIST_APPEND`(*i*)
Calls `list.append(TOS[-i], TOS)`. Used to implement list comprehensions.

`MAP_ADD`(*i*)
Calls `dict.setitem(TOS1[-i], TOS, TOS1)`. Used to implement dict comprehensions.

New in version 3.1.

For all of the *SET_ADD*, *LIST_APPEND* and *MAP_ADD* instructions, while the added value or key/value pair is popped off, the container object remains on the stack so that it is available for further iterations of the loop.

`RETURN_VALUE`
Returns with TOS to the caller of the function.

`YIELD_VALUE`
Pops TOS and yields it from a *generator*.

`YIELD_FROM`
Pops TOS and delegates to it as a subiterator from a *generator*.

New in version 3.3.

`SETUP_ANNOTATIONS`
Checks whether `__annotations__` is defined in `locals()`, if not it is set up to an empty `dict`. This opcode is only emitted if a class or module body contains *variable annotations* statically.

New in version 3.6.

`IMPORT_STAR`
Loads all symbols not starting with `'_'` directly from the module TOS to the local namespace. The module is popped after loading all names. This opcode implements `from module import *`.

`POP_BLOCK`
Removes one block from the block stack. Per frame, there is a stack of blocks, denoting nested loops, try statements, and such.

`POP_EXCEPT`
Removes one block from the block stack. The popped block must be an exception handler block, as implicitly created when entering an except handler. In addition to popping extraneous values from the frame stack, the last three popped values are used to restore the exception state.

`END_FINALLY`
Terminates a `finally` clause. The interpreter recalls whether the exception has to be re-raised, or whether the function returns, and continues with the outer-next block.

`LOAD_BUILD_CLASS`
Pushes `builtins.__build_class__()` onto the stack. It is later called by *CALL_FUNCTION* to construct a class.

`SETUP_WITH`(*delta*)
This opcode performs several operations before a with block starts. First, it loads `__exit__()` from the context manager and pushes it onto the stack for later use by `WITH_CLEANUP`. Then, `__enter__()` is called, and a finally block pointing to *delta* is pushed. Finally, the result of calling the enter method is pushed onto the stack. The next opcode will either ignore it (*POP_TOP*), or store it in (a) variable(s) (*STORE_FAST*, *STORE_NAME*, or *UNPACK_SEQUENCE*).

New in version 3.2.

WITH_CLEANUP_START
Cleans up the stack when a `with` statement block exits. TOS is the context manager's `__exit__()` bound method. Below TOS are 1–3 values indicating how/why the finally clause was entered:

- SECOND = `None`
- (SECOND, THIRD) = (`WHY_{RETURN,CONTINUE}`), retval
- SECOND = `WHY_*`; no retval below it
- (SECOND, THIRD, FOURTH) = exc_info()

In the last case, `TOS(SECOND, THIRD, FOURTH)` is called, otherwise `TOS(None, None, None)`. Pushes SECOND and result of the call to the stack.

WITH_CLEANUP_FINISH
Pops exception type and result of 'exit' function call from the stack.

If the stack represents an exception, *and* the function call returns a 'true' value, this information is "zapped" and replaced with a single `WHY_SILENCED` to prevent *END_FINALLY* from re-raising the exception. (But non-local gotos will still be resumed.)

All of the following opcodes use their arguments.

STORE_NAME(*namei*)
Implements `name = TOS`. *namei* is the index of *name* in the attribute `co_names` of the code object. The compiler tries to use *STORE_FAST* or *STORE_GLOBAL* if possible.

DELETE_NAME(*namei*)
Implements `del name`, where *namei* is the index into `co_names` attribute of the code object.

UNPACK_SEQUENCE(*count*)
Unpacks TOS into *count* individual values, which are put onto the stack right-to-left.

UNPACK_EX(*counts*)
Implements assignment with a starred target: Unpacks an iterable in TOS into individual values, where the total number of values can be smaller than the number of items in the iterable: one of the new values will be a list of all leftover items.

The low byte of *counts* is the number of values before the list value, the high byte of *counts* the number of values after it. The resulting values are put onto the stack right-to-left.

STORE_ATTR(*namei*)
Implements `TOS.name = TOS1`, where *namei* is the index of name in `co_names`.

DELETE_ATTR(*namei*)
Implements `del TOS.name`, using *namei* as index into `co_names`.

STORE_GLOBAL(*namei*)
Works as *STORE_NAME*, but stores the name as a global.

DELETE_GLOBAL(*namei*)
Works as *DELETE_NAME*, but deletes a global name.

LOAD_CONST(*consti*)
Pushes `co_consts[consti]` onto the stack.

LOAD_NAME(*namei*)
Pushes the value associated with `co_names[namei]` onto the stack.

BUILD_TUPLE(*count*)
Creates a tuple consuming *count* items from the stack, and pushes the resulting tuple onto the stack.

BUILD_LIST(*count*)
Works as *BUILD_TUPLE*, but creates a list.

BUILD_SET(*count*)
Works as *BUILD_TUPLE*, but creates a set.

BUILD_MAP(*count*)
Pushes a new dictionary object onto the stack. Pops `2 * count` items so that the dictionary holds *count* entries: `{..., TOS3: TOS2, TOS1: TOS}`.

Changed in version 3.5: The dictionary is created from stack items instead of creating an empty dictionary pre-sized to hold *count* items.

BUILD_CONST_KEY_MAP(*count*)
The version of *BUILD_MAP* specialized for constant keys. *count* values are consumed from the stack. The top element on the stack contains a tuple of keys.

New in version 3.6.

BUILD_STRING(*count*)
Concatenates *count* strings from the stack and pushes the resulting string onto the stack.

New in version 3.6.

BUILD_TUPLE_UNPACK(*count*)
Pops *count* iterables from the stack, joins them in a single tuple, and pushes the result. Implements iterable unpacking in tuple displays `(*x, *y, *z)`.

New in version 3.5.

BUILD_TUPLE_UNPACK_WITH_CALL(*count*)
This is similar to *BUILD_TUPLE_UNPACK*, but is used for `f(*x, *y, *z)` call syntax. The stack item at position `count + 1` should be the corresponding callable `f`.

New in version 3.6.

BUILD_LIST_UNPACK(*count*)
This is similar to *BUILD_TUPLE_UNPACK*, but pushes a list instead of tuple. Implements iterable unpacking in list displays `[*x, *y, *z]`.

New in version 3.5.

BUILD_SET_UNPACK(*count*)
This is similar to *BUILD_TUPLE_UNPACK*, but pushes a set instead of tuple. Implements iterable unpacking in set displays `{*x, *y, *z}`.

New in version 3.5.

BUILD_MAP_UNPACK(*count*)
Pops *count* mappings from the stack, merges them into a single dictionary, and pushes the result. Implements dictionary unpacking in dictionary displays `{**x, **y, **z}`.

New in version 3.5.

BUILD_MAP_UNPACK_WITH_CALL(*count*)
This is similar to *BUILD_MAP_UNPACK*, but is used for `f(**x, **y, **z)` call syntax. The stack item at position `count + 2` should be the corresponding callable `f`.

New in version 3.5.

Changed in version 3.6: The position of the callable is determined by adding 2 to the opcode argument instead of encoding it in the second byte of the argument.

LOAD_ATTR(*namei*)
Replaces TOS with `getattr(TOS, co_names[namei])`.

COMPARE_OP(*opname*)
Performs a Boolean operation. The operation name can be found in `cmp_op[opname]`.

`IMPORT_NAME`(*namei*)
Imports the module `co_names[namei]`. TOS and TOS1 are popped and provide the *fromlist* and *level* arguments of *__import__()*. The module object is pushed onto the stack. The current namespace is not affected: for a proper import statement, a subsequent *STORE_FAST* instruction modifies the namespace.

`IMPORT_FROM`(*namei*)
Loads the attribute `co_names[namei]` from the module found in TOS. The resulting object is pushed onto the stack, to be subsequently stored by a *STORE_FAST* instruction.

`JUMP_FORWARD`(*delta*)
Increments bytecode counter by *delta*.

`POP_JUMP_IF_TRUE`(*target*)
If TOS is true, sets the bytecode counter to *target*. TOS is popped.

New in version 3.1.

`POP_JUMP_IF_FALSE`(*target*)
If TOS is false, sets the bytecode counter to *target*. TOS is popped.

New in version 3.1.

`JUMP_IF_TRUE_OR_POP`(*target*)
If TOS is true, sets the bytecode counter to *target* and leaves TOS on the stack. Otherwise (TOS is false), TOS is popped.

New in version 3.1.

`JUMP_IF_FALSE_OR_POP`(*target*)
If TOS is false, sets the bytecode counter to *target* and leaves TOS on the stack. Otherwise (TOS is true), TOS is popped.

New in version 3.1.

`JUMP_ABSOLUTE`(*target*)
Set bytecode counter to *target*.

`FOR_ITER`(*delta*)
TOS is an *iterator*. Call its *__next__()* method. If this yields a new value, push it on the stack (leaving the iterator below it). If the iterator indicates it is exhausted TOS is popped, and the byte code counter is incremented by *delta*.

`LOAD_GLOBAL`(*namei*)
Loads the global named `co_names[namei]` onto the stack.

`SETUP_LOOP`(*delta*)
Pushes a block for a loop onto the block stack. The block spans from the current instruction with a size of *delta* bytes.

`SETUP_EXCEPT`(*delta*)
Pushes a try block from a try-except clause onto the block stack. *delta* points to the first except block.

`SETUP_FINALLY`(*delta*)
Pushes a try block from a try-except clause onto the block stack. *delta* points to the finally block.

`LOAD_FAST`(*var_num*)
Pushes a reference to the local `co_varnames[var_num]` onto the stack.

`STORE_FAST`(*var_num*)
Stores TOS into the local `co_varnames[var_num]`.

`DELETE_FAST`(*var_num*)
Deletes local `co_varnames[var_num]`.

`LOAD_CLOSURE`(*i*)
Pushes a reference to the cell contained in slot *i* of the cell and free variable storage. The name of the variable is `co_cellvars[i]` if *i* is less than the length of *co_cellvars*. Otherwise it is `co_freevars[i - len(co_cellvars)]`.

`LOAD_DEREF`(*i*)
Loads the cell contained in slot *i* of the cell and free variable storage. Pushes a reference to the object the cell contains on the stack.

`LOAD_CLASSDEREF`(*i*)
Much like *LOAD_DEREF* but first checks the locals dictionary before consulting the cell. This is used for loading free variables in class bodies.

New in version 3.4.

`STORE_DEREF`(*i*)
Stores TOS into the cell contained in slot *i* of the cell and free variable storage.

`DELETE_DEREF`(*i*)
Empties the cell contained in slot *i* of the cell and free variable storage. Used by the `del` statement.

New in version 3.2.

`RAISE_VARARGS`(*argc*)
Raises an exception. *argc* indicates the number of arguments to the raise statement, ranging from 0 to 3. The handler will find the traceback as TOS2, the parameter as TOS1, and the exception as TOS.

`CALL_FUNCTION`(*argc*)
Calls a callable object with positional arguments. *argc* indicates the number of positional arguments. The top of the stack contains positional arguments, with the right-most argument on top. Below the arguments is a callable object to call. `CALL_FUNCTION` pops all arguments and the callable object off the stack, calls the callable object with those arguments, and pushes the return value returned by the callable object.

Changed in version 3.6: This opcode is used only for calls with positional arguments.

`CALL_FUNCTION_KW`(*argc*)
Calls a callable object with positional (if any) and keyword arguments. *argc* indicates the total number of positional and keyword arguments. The top element on the stack contains a tuple of keyword argument names. Below that are keyword arguments in the order corresponding to the tuple. Below that are positional arguments, with the right-most parameter on top. Below the arguments is a callable object to call. `CALL_FUNCTION_KW` pops all arguments and the callable object off the stack, calls the callable object with those arguments, and pushes the return value returned by the callable object.

Changed in version 3.6: Keyword arguments are packed in a tuple instead of a dictionary, *argc* indicates the total number of arguments.

`CALL_FUNCTION_EX`(*flags*)
Calls a callable object with variable set of positional and keyword arguments. If the lowest bit of *flags* is set, the top of the stack contains a mapping object containing additional keyword arguments. Below that is an iterable object containing positional arguments and a callable object to call. *BUILD_MAP_UNPACK_WITH_CALL* and *BUILD_TUPLE_UNPACK_WITH_CALL* can be used for merging multiple mapping objects and iterables containing arguments. Before the callable is called, the mapping object and iterable object are each "unpacked" and their contents passed in as keyword and positional arguments respectively. `CALL_FUNCTION_EX` pops all arguments and the callable object off the stack, calls the callable object with those arguments, and pushes the return value returned by the callable object.

New in version 3.6.

`LOAD_METHOD`(*namei*)
Loads a method named `co_names[namei]` from TOS object. TOS is popped and method and TOS

are pushed when interpreter can call unbound method directly. TOS will be used as the first argument (`self`) by *CALL_METHOD*. Otherwise, `NULL` and method is pushed (method is bound method or something else).

New in version 3.7.

`CALL_METHOD`(*argc*)
Calls a method. *argc* is number of positional arguments. Keyword arguments are not supported. This opcode is designed to be used with *LOAD_METHOD*. Positional arguments are on top of the stack. Below them, two items described in *LOAD_METHOD* on the stack. All of them are popped and return value is pushed.

New in version 3.7.

`MAKE_FUNCTION`(*argc*)
Pushes a new function object on the stack. From bottom to top, the consumed stack must consist of values if the argument carries a specified flag value

- `0x01` a tuple of default values for positional-only and positional-or-keyword parameters in positional order
- `0x02` a dictionary of keyword-only parameters' default values
- `0x04` an annotation dictionary
- `0x08` a tuple containing cells for free variables, making a closure
- the code associated with the function (at TOS1)
- the *qualified name* of the function (at TOS)

`BUILD_SLICE`(*argc*)
Pushes a slice object on the stack. *argc* must be 2 or 3. If it is 2, `slice(TOS1, TOS)` is pushed; if it is 3, `slice(TOS2, TOS1, TOS)` is pushed. See the *slice()* built-in function for more information.

`EXTENDED_ARG`(*ext*)
Prefixes any opcode which has an argument too big to fit into the default two bytes. *ext* holds two additional bytes which, taken together with the subsequent opcode's argument, comprise a four-byte argument, *ext* being the two most-significant bytes.

`FORMAT_VALUE`(*flags*)
Used for implementing formatted literal strings (f-strings). Pops an optional *fmt_spec* from the stack, then a required *value*. *flags* is interpreted as follows:

- `(flags & 0x03) == 0x00`: *value* is formatted as-is.
- `(flags & 0x03) == 0x01`: call *str()* on *value* before formatting it.
- `(flags & 0x03) == 0x02`: call *repr()* on *value* before formatting it.
- `(flags & 0x03) == 0x03`: call *ascii()* on *value* before formatting it.
- `(flags & 0x04) == 0x04`: pop *fmt_spec* from the stack and use it, else use an empty *fmt_spec*.

Formatting is performed using `PyObject_Format()`. The result is pushed on the stack.

New in version 3.6.

`HAVE_ARGUMENT`
This is not really an opcode. It identifies the dividing line between opcodes which don't use their argument and those that do (`< HAVE_ARGUMENT` and `>= HAVE_ARGUMENT`, respectively).

Changed in version 3.6: Now every instruction has an argument, but opcodes `< HAVE_ARGUMENT` ignore it. Before, only opcodes `>= HAVE_ARGUMENT` had an argument.

33.12.4 Opcode collections

These collections are provided for automatic introspection of bytecode instructions:

dis.opname
Sequence of operation names, indexable using the bytecode.

dis.opmap
Dictionary mapping operation names to bytecodes.

dis.cmp_op
Sequence of all compare operation names.

dis.hasconst
Sequence of bytecodes that access a constant.

dis.hasfree
Sequence of bytecodes that access a free variable (note that 'free' in this context refers to names in the current scope that are referenced by inner scopes or names in outer scopes that are referenced from this scope. It does *not* include references to global or builtin scopes).

dis.hasname
Sequence of bytecodes that access an attribute by name.

dis.hasjrel
Sequence of bytecodes that have a relative jump target.

dis.hasjabs
Sequence of bytecodes that have an absolute jump target.

dis.haslocal
Sequence of bytecodes that access a local variable.

dis.hascompare
Sequence of bytecodes of Boolean operations.

33.13 pickletools — Tools for pickle developers

Source code: Lib/pickletools.py

This module contains various constants relating to the intimate details of the *pickle* module, some lengthy comments about the implementation, and a few useful functions for analyzing pickled data. The contents of this module are useful for Python core developers who are working on the *pickle*; ordinary users of the *pickle* module probably won't find the *pickletools* module relevant.

33.13.1 Command line usage

New in version 3.2.

When invoked from the command line, `python -m pickletools` will disassemble the contents of one or more pickle files. Note that if you want to see the Python object stored in the pickle rather than the details of pickle format, you may want to use `-m pickle` instead. However, when the pickle file that you want to examine comes from an untrusted source, `-m pickletools` is a safer option because it does not execute pickle bytecode.

For example, with a tuple `(1, 2)` pickled in file `x.pickle`:

```
$ python -m pickle x.pickle
(1, 2)

$ python -m pickletools x.pickle
    0: \x80 PROTO      3
    2: K    BININT1    1
    4: K    BININT1    2
    6: \x86 TUPLE2
    7: q    BINPUT     0
    9: .    STOP
highest protocol among opcodes = 2
```

Command line options

-a, --annotate
Annotate each line with a short opcode description.

-o, --output=<file>
Name of a file where the output should be written.

-l, --indentlevel=<num>
The number of blanks by which to indent a new MARK level.

-m, --memo
When multiple objects are disassembled, preserve memo between disassemblies.

-p, --preamble=<preamble>
When more than one pickle file are specified, print given preamble before each disassembly.

33.13.2 Programmatic Interface

pickletools.dis(*pickle*, *out=None*, *memo=None*, *indentlevel=4*, *annotate=0*)
Outputs a symbolic disassembly of the pickle to the file-like object *out*, defaulting to `sys.stdout`. *pickle* can be a string or a file-like object. *memo* can be a Python dictionary that will be used as the pickle's memo; it can be used to perform disassemblies across multiple pickles created by the same pickler. Successive levels, indicated by `MARK` opcodes in the stream, are indented by *indentlevel* spaces. If a nonzero value is given to *annotate*, each opcode in the output is annotated with a short description. The value of *annotate* is used as a hint for the column where annotation should start.

New in version 3.2: The *annotate* argument.

pickletools.genops(*pickle*)
Provides an *iterator* over all of the opcodes in a pickle, returning a sequence of `(opcode, arg, pos)` triples. *opcode* is an instance of an `OpcodeInfo` class; *arg* is the decoded value, as a Python object, of the opcode's argument; *pos* is the position at which this opcode is located. *pickle* can be a string or a file-like object.

pickletools.optimize(*picklestring*)
Returns a new equivalent pickle string after eliminating unused `PUT` opcodes. The optimized pickle is shorter, takes less transmission time, requires less storage space, and unpickles more efficiently.