Python Reference Manual

Guido van Rossum
Corporation for National Research Initiatives (CNRI)

Table of Contents

Table of Contents

CHAPTER 1: I

CHAPTER 2: LEXICAL ANALYSIS

A Python program is read by a *parser*. Input to the parser is a stream of *tokens*, generated by the *lexical analyzer*. This chapter describes how the lexical analyzer breaks a file into tokens.

Python uses the 7-bit ASCII character set for program text and string literals. 8-bit characters may be

```
   and 0 <= minute < 60 and 0 <= second < 60:   # Looks like a valid date
        return 1
```

A line ending in a backslash cannot carry a comment. A backslash does not continue a comment. A backslash does not continue a token except for string literals (i.e., tokens other than string literals cannot be split

must be one of the numbers occurring on the stack; all numbers on the stack that are larger are popped off, and for each number popped off a DEDENT token is generated. At the end of the file, a DEDENT token is generated for each number remaining on the stack that is larger than zero.

Here is an example of a correctly (though confuaingly) indented piece of Python code:

```
def perm(l):
        # Compute the list of all permutations of l
```

is the character used to open the string, i.e. either ' or ".)

Unless an ‘r’ or ‘R’ prefix is present, escape sequences in strings
similar to those used by Standard C. The recognized escape sequen

Unlike Standard C, all unrecognized escape sequences are left in the stri
When an ‘r’ or ‘R’ prefix is present, backslashes are still used to quote the fo
the resulting output is more easily recognized as broken.)

ber are used in all current implementations.)

```
r"\n"
```

2.4.1.1 String literal concatenation

Multiple adjacent string literals (delisf 0d by whf 0space), possibly using different quoting conventions, are allowed, and their meaning is the same as their concatenation. Thus, `"hello" 'world'`

Note that the integer part of a floating point number cannot look like an octal integer. The allowed range of floating point literals is implementation-dependent. Some examples of floating point literals:3

2

I

i

A
s
c (S
a

N - i
o -’ 1.

2

T

+
<
<

T

3.2 The standard type hierarchy

no reason to complicate the language with two kinds of floating point numbers.

Complex numbers These represent complex numbers as a pair of machine-level double precision floating point numbers. The same caveats apply as for floating point numbers. The

There is currently a single intrinsic mapping type:

Dictionaries These represent finite sets of objects indexed by nearly arbitrary values. The only types of values not acceptable as keys are values containing lists or dictionaries or other mutable v42(that)-243oare compared rather than object —
son being that fhe efficient implementation of dictionaries requires a key’s value fo re-

When a bound user-defined method object is called, the underlying function (`im_func`) is called, inserting the class instance (`im_self`) in front of the argument list. For instance, when `C` is a class which contains a definition for a function `f`, and `x` is an instance of `C`, calling `x.f(1)` is equivalent to calling `C.f(x, 1)`.

Note that the transformation from function object to (unbound or bound) method object happens each time the attribute is retrieved from the class or instance. In some cases, a fruitful optimization is to assign the attribute to a local variable and call that local vari-

Special read-only attributes: `co_argcount` is the number of positional arguments (including arguments with default values); `co_nlocals` is the number of local variables used by the function (including arguments); `co_varnames` is a tuple containing the names of the local variables (starting with the argument names); `co_code` is a string representing the sequence of bytecode instructions; `co_consts` is a tuple containing the literals used by the bytecode; `co_names` is a tuple containing the names used by the bytecode; `co_filename` is the filename from which the code was compiled; `co_flags` is an integer encoding a number of flags for the interpreter. The following flag bits are defined: bit 2 is set if the function uses the "`*arguments`" syntax to accept an arbitrary number of positional arguments; bit 3 is set if the function uses the "`**keywords`" syntax to accept arbitrary keyword arguments; other bits are used internally or reserved for future use. The first item in `co_consts` is the documentation string of the function, or `None` if undefined. To find out the first line number of a function, you have to disassemble the bytecode instructions; the standard library module `codehack` defines a function `getlineno()` that returns the first line number of a code object.

Frame objects Frame objects represent execution frames. They may occur in traceback ob-

`__setslice__(self, i, j, sequence)` Called to implement assignment to `self[i:j]`. The `sequence` argument can have any type. The return value should be `None`. Same notes for `i` and `j` as for`Call(__getslice__)Tj /F4 1 Tf 7.2 0 TD (.)Tj /F6 1 Tf -17.781`

`__invert__(self)` Called to implement the unary arithmetic operations (`-`, `+`, `abs()` and `~`).

`__int__(self)`
`__long__(self)`
`__float__(self)` Called to implement the built-in functions `int()`, `long()` and `float()`. Should return a value of the appropriate type.

`__oct__(self)`
`__hex__(self)` Called to implement the built-in functions `oct()` and `hex()`. Should return a string value.

`__coerce__(self, other)` Called to implement "mixed-mode" numeric arithmetic. Should either return a 2-tuple containing `self` and `other` converted to a common numeric type, or `None` if no conversion is possible. When the common type would be the type of `other`, it is sufficient to return `None`, since the interpreter will also ask the other object to attempt a coercion (but sometimes, if the implementation of the other type cannot be changed, it is useful to do the conversion to the other type here).

Coercion rules: to evaluate x *op* y, the following steps are taken (where `__op__` and `__rop__` are the method names corresponding to *op*, e.g. if *op* is '+', `__add__` and `__radd__` are used). If an exception occurs at any point, the evaluation is abandoned and exception handling takes over.

and
op

4.2 Exceptions

CHAPTER 5: EXPRESSIONS

This chapter explains the meaning of the elements of expressions in Python.

Syntax notes:

- Expressions

The primary must evaluate to an object of a sequence or mapping type.

If the primary is a mapping, the expression list must evaluate to an object whose value is one of the keys of the mapping, and the subscription selects the value in the mapping that corresponds to that key.

If the primary is a sequence, the expression (list) must evaluate to a plain integer. If this value is negative, the length of the sequence is added to it (so that, e.g. `x[-1]` selects the last item of `x`.) The re-

A call always returns some value, possibly `None`, unless it raises an exception. How this value is,0 Tc 0 Tw (com

```
m_expr:          u_expr | m_expr "*" u_expr
 .2 TD ( .2 TD .2 TD .29   u_expr | m_expr%"*" u_expr)TT* (a(m_expr:       m u_exp am
```

```
def name (arguments):
    return expression
```

See “Function definitions” on page 50 for the syntax of parameter lists. Note that functions created with lambda forms cannot contain statements.

5.11 Expression lists

```
expression_list:      expression ("," expression)* [","]
```

An expression list containing at least one comma yields a tuple. The length of the tuple is the number

5.12 Summary

The following table summarizes the operator precedences in Python, from lowest precedence (least bindingortohighstrecedencelmotOeratornthhave-28o (th)-28o (sam)-2830 ecedence

CHAPTER 6: SIMPLE STATEMENTS

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

```
simple_stmt:    expression_stmt
              | assert_stmt
              | assignment_stmt
              | pass_stmt
              | del_stmt
              | print_stmt
              | return_stmt
              | raise_stmt
              | break_stmt
              | continue_stmt
              | import_stmt
              | global_stmt
              | exec_stmt
```

6.1 Expression statements

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

```
expression_stmt: expression_list
```

An expression statement evaluates the expression list (which may be a single expression). In interactive mode, if the value is not `None`, it is converted to a seftive mode, if th Theist convertedto rlinntax6 1 T35.09[(649(

ator emits no code for an `assert` statement when optimization is requested at compile time. Note that it is unnecessary to include the source code for the expression that failed in the error message; it will be displayed as part of the stack trace.

6.3 Assignment statements

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

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

(See “Primaries” on page 29 for the syntax definitions for the last three symbols.)

An assignment statement evaluates the expression list (remember that this can be a single expression or a comma-separated list, the latter yielding a tuple) and assigns the single resulting object to each of the target lists, from left to right.

Assignment is defined recursively depending on the form of the target (list). When atarget (listis part of atargetmutable object (an attribute reference, subscription or slicing), the mutable object must ultimately perform the assignment and decide about its validity, and may raise an exception if the assignment is unacceptable. The rules observed by various types and the exceptions raised are given with the definition of the object types (See “The standard type hierarchy” on page 12.)

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

- If the target list is a single target: the object is assigned to that target.
- If the (l06tlist is of s: the must the number of items as there are s in the list, and the items are assigned, from left to right, to the corresponding s.This rule relaxed since Python 1.5; in earlier versions, the object had to be a

Deletion of a name removes the binding of that name (which must exist) from the local or global name space, depending on whether the name occurs in a

7.4 The `try` statement

The `try` statement specifies exception handlers and/or cleanup code for a group of statements:

```
try_stmt:       try_exc_stmt | try_fin_stmt
try_exc_stmt:   "try" ":" suite
                ("except" [expression ["," target]] ":" suite)+
                ["else" ":" suite]
try_fin_stmt:   "try" ":" suite
                "finally" ":" suite
```

There are two forms of `try` statement: `try...except` and `try...finally`. These forms can-

It is also possible to create anonymous functions (functions not initially bound to a name), for immediate use in expressions. This uses lambda forms, described in section “Boolean operations” on

CHAPTER 8: TOP-LEVEL COMPONENTS

8.4 Expression input

There are two forms of expression input. Both ignore leading whitespace. The string argument to `eval()` mus. have the following form:

```
eval_input:     expression_lis. NEWLINE*
```

The input line read by `input()` mus. have the following form:

```
input_input:    expression_lis. NEWLINE
```

Note:expinp_s86.67 Ttiu86.67 ‘raw’86.67 inputlinewithoutinterpretation,86.67 you86.67 can86.67 use86.67 the `raw_input()` function the `Ttiuline()` method of file objects.

E

I

O

INDEX