Python Reference Manual

Guido van Rossum
Dept. CST, CWI, P.O. Box 94079
1090 GB Amsterdam, The Netherlands
E-mail: guido@cwi.nl

4 May 1994
Release 1.0.2

Abstract

Python is a simple, yet powerful, interpreted programming language that bridges the gap between C and shell programming, and is thus ideally suited for "throw-away programming" and rapid prototyping. Its syntax is put together from constructs borrowed from a variety of other languages; most prominent are influences from ABC, C, Modula-3 and Icon.

The Python interpreter is easily extended with new functions and data types implemented in C. Python is also suitable as an extension language for highly customizable C applications such as editors or window managers.

Python is available for various operating systems, amongst which several flavors of UNIX, Amoeba, the Apple Macintosh O.S., and MS-DOS.

This reference manual describes the syntax and "core semantics" of the language. It is terse, but attempts to be exact and complete. The semantics of non-essential built-in object types and of the built-in functions and modules are described in the *Python Library Reference*. For an informal introduction to the language, see the *Python Tutorial*.

Contents

Chapter 1

Introduction

This reference manual describes the Python programming language. It is not intended as a tutorial.

While I am trying to be as precise as possible, I chose to use English rather than formal specifications for everything except syntax and lexical analysis. This should make the document better understandable to the average reader, but will leave room for ambiguities. Consequently, if you were coming from Mars and tried to re-implement Python from this document alone, you might have to guess things and in fact you would probably end up implementing quite a different language. On the other hand, if you are using Python and wonder what the precise rules about a particular area of the language are, you should definitely be able to find them here.

It is dangerous to add too many implementation details to a language reference document — the implementation may change, and other implementations of the same language may work differently. On the other hand, there is currently only one Python implementation, and its particular quirks are sometimes worth being mentioned, especially where the implementation imposes additional limitations. Therefore, you'll find short "implementation notes" sprinkled throughout the text.

Every Python implementation comes with a number of built-in and standard modules. These are not documented here, but in the separate *Python Library Reference* document. A few built-in modules are mentioned when they interact in a significant way with the language definition.

1.1 Notation

The descriptions of lexical analysis and syntax use a modified BNF grammar notation. This uses the following style of definition:

```
name:           lc_letter (lc_letter | "_")*
lc_letter:      "a"..."z"
```

The first line says that a `name` is an `lc_letter` followed by a sequence of zero or more `lc_letter`s and underscores. An `lc_letter` in turn is any of the single characters 'a' through 'z'. (This rule is actually adhered to for the names defined in lexical and grammar rules in this document.)

Each rule begins with a name (which is the name defined by the rule) and a colon. A vertical bar (`|`) is used to separate alternatives; it is the least binding operator in this notation. A star (`*`) means zero or more repetitions of the preceding item; likewise, a plus (`+`) means one or more repetitions, and a phrase enclosed in square brackets (`[ ]`) means zero or one occurrences (in other words, the enclosed phrase is optional). The `*` and `+` operators bind as tightly as possible; parentheses are used for grouping. Literal strings are enclosed in double quotes. White space is only meaningful to separate tokens. Rules are normally contained on a single line; rules with many alternatives may be formatted alternatively with each line after the first beginning with a vertical bar.

In lexical definitions (as the example above), two more conventions are used: Two literal characters separated by three dots mean a choice of any single character in the given (inclusive) range of ASCII characters. A phrase between angular brackets (`<...>`) gives an informal description of the symbol defined; e.g. this could be used to describe the notion of 'control character' if needed.

Even though the notation used is almost the same, there is a big difference between the meaning of lexical and syntactic definitions: a lexical definition operates on the individual characters of the input source, while a syntax definition operates on the stream of tokens generated by the lexical analysis. All uses of BNF in the next chapter ("Lexical Analysis") are lexical definitions; uses in subsequent chapters are syntactic definitions.

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.

2.1 Line structure

A Python program is divided in a number of logical lines. The end of a logical line is represented by the token NEWLINE. Statements cannot cross logical line boundaries except where NEWLINE is allowed by the syntax (e.g. between statements in compound statements).

2.1.1 Comments

A comment starts with a hash character (`#`) that is not part of a string literal, and ends at the end of the physical line. A comment always signifies the end of the logical line. Comments are ignored by the syntax.

2.1.2 Explicit line joining

Two or more physical lines may be joined into logical lines using backslash characters (\), as follows: when a physical line ends in a backslash that is not part of a string literal or comment, it is joined with the following forming a single logical line, deleting the backslash and the following end-of-line character. For example:

```
if 1900 < year < 2100 and 1 <= month <= 12 \
   and 1 <= day <= 31 and 0 <= hour < 24 \
   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 (but it does continue a string literal, see below).

2.1.3 Implicit line joining

Expressions in parentheses, square brackets or curly braces can be split over more than one physical line without using backslashes. For example:

```
month_names = ['Januari', 'Februari', 'Maart',      # These are the
               'April',   'Mei',      'Juni',       # Dutch names
               'Juli',    'Augustus', 'September',  # for the months
               'Oktober', 'November', 'December']   # of the year
```

Implicitly continued lines can carry comments. The indentation of the continuation lines is not important. Blank continuation lines are allowed.

2.1.4 Blank lines

A logical line that contains only spaces, tabs, and possibly a comment, is ignored (i.e., no NEWLINE token is generated), except that during interactive input of statements, an entirely blank logical line terminates a multi-line statement.

2.1.5 Indentation

Leading whitespace (spaces and tabs) at the beginning of a logical line is used to compute the indentation level of the line, which in turn is used to determine the grouping of statements.

First, tabs are replaced (from left to right) by one to eight spaces such that the total number of characters up to there is a multiple of eight (this is intended to be the same rule as used by UNIX). The total number of spaces preceding the first non-blank character then determines the line's indentation. Indentation cannot be split over multiple physical lines using backslashes.

The indentation levels of consecutive lines are used to generate INDENT and DEDENT tokens, using a stack, as follows.

Before the first line of the file is read, a single zero is pushed on the stack; this will never be popped off again. The numbers pushed on the stack will always be strictly increasing from bottom to top. At the beginning of each logical line, the line's indentation level is compared to the top of the stack. If it is equal, nothing happens. If it is larger, it is pushed on the stack, and one INDENT token is generated. If it is smaller, it *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 confusingly) indented piece of Python code:

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

```
if len(l) <= 1:
              return [l]
r = []
for i in range(len(l)):
         s = l[:i] + l[i+1:]
         p = perm(s)
         for x in p:
          r.append(l[i:i+1] + x)
return r
```

The following example shows various indentation errors:

```
def perm(l):                       # error: first line indented
for i in range(len(l)):            # error: not indented
    s = l[:i] + l[i+1:]
        p = perm(l[:i] + l[i+1:])   # error: unexpected indent
        for x in p:
                r.append(l[i:i+1] + x)
            return r                # error: inconsistent dedent
```

(Actually, the first three errors are detected by the parser; only the last error is found by the lexical analyzer — the indentation of `return r` does not match a level popped off the stack.)

2.2 Other tokens

Besides NEWLINE, INDENT and DEDENT, the following categories of tokens exist: identifiers, keywords, literals, operators, and delimiters. Spaces and tabs are not tokens, but serve to delimit tokens. Where ambiguity exists, a token comprises the longest possible string that forms a legal token, when read from left to right.

2.3 Identifiers

Identifiers (also referred to as names) are described by the following lexical definitions:

```
identifier:     (letter|"_") (letter|digit|"_")*
letter:         lowercase | uppercase
lowercase:      "a"..."z"
uppercase:      "A"..."Z"
digit:          "0"..."9"
```

Identifiers are unlimited in length. Case is significant.

2.3.1 Keywords

The following identifiers are used as reserved words, or *keywords* of the language, and cannot be used as ordinary identifiers. They must be spelled exactly as written here:

```
and         del         for         in          print
break       elif        from        is          raise
class       else        global      not         return
continue    except      if          or          try
def         finally     import      pass        while
```

2.4 Literals

Literals are notations for constant values of some built-in types.

2.4.1 String literals

String literals are described by the following lexical definitions:

```
stringliteral:   shortstring | longstring
shortstring:     "'" shortstringitem* "'" | '"' shortstringitem* '"'
longstring:      "'''" longstringitem* "'''" | '"""' longstringitem* '"""'
shortstringitem: shortstringchar | escapeseq
shortstringchar: <any ASCII character except "\" or newline or the quote>
longstringchar:  <any ASCII character except "\">
escapeseq:       "\" <any ASCII character>
```

In “long strings” (strings surrounded by sets of three quotes), unescaped newlines and quotes are allowed (and are retained), except that three unescaped quotes in a row terminate the string. (A “quote” is the character used to open the string, i.e. either `'` or `"`.)

Escape sequences in strings are interpreted according to rules similar to those used by Standard C. The recognized escape sequences are:

`\`*newline*	Ignored
`\\`	Backslash (`\`)
`\'`	Single quote (`'`)
`\"`	Double quote (`"`)
`\a`	ASCII Bell (BEL)
`\b`	ASCII Backspace (BS)
`\f`	ASCII Formfeed (FF)
`\n`	ASCII Linefeed (LF)
`\r`	ASCII Carriage Return (CR)
`\t`	ASCII Horizontal Tab (TAB)
`\v`	ASCII Vertical Tab (VT)
`\`*ooo*	ASCII character with octal value *ooo*
`\x`*xx...*	ASCII character with hex value *xx...*

In strict compatibility with Standard C, up to three octal digits are accepted, but an unlimited number of hex digits is taken to be part of the hex escape (and then the lower 8 bits of the resulting hex number are used in all current implementations...).

All unrecognized escape sequences are left in the string unchanged, i.e., *the backslash is left in the string.* (This behavior is useful when debugging: if an escape sequence is mistyped, the resulting output is more easily recognized as broken. It also helps a great deal for string literals used as regular expressions or otherwise passed to other modules that do their own escape handling.)

2.4.2 Numeric literals

There are three types of numeric literals: plain integers, long integers, and floating point numbers.

Integer and long integer literals are described by the following lexical definitions:

```
longinteger:    integer ("l"|"L")
integer:        decimalinteger | octinteger | hexinteger
decimalinteger: nonzerodigit digit* | "0"
octinteger:     "0" octdigit+
hexinteger:     "0" ("x"|"X") hexdigit+

nonzerodigit:   "1"..."9"
octdigit:       "0"..."7"
hexdigit:        digit|"a"..."f"|"A"..."F"
```

Although both lower case 'l' and upper case 'L' are allowed as suffix for long integers, it is strongly recommended to always use 'L', since the letter 'l' looks too much like the digit '1'.

Plain integer decimal literals must be at most $2^{31}-1$ (i.e., the largest positive integer, assuming 32-bit arithmetic). Plain octal and hexadecimal literals may be as large as $2^{32}-1$, but values larger than $2^{31}-1$ are converted to a negative value by subtracting 2^{32}. There is no limit for long integer literals.

Some examples of plain and long integer literals:

```
7     2147483647                      0177     0x80000000
3L    79228162514264337593543950336L  0377L    0x100000000L
```

Floating point literals are described by the following lexical definitions:

```
floatnumber:    pointfloat | exponentfloat
pointfloat:     [intpart] fraction | intpart "."
exponentfloat:  (intpart | pointfloat) exponent
intpart:        digit+
fraction:       "." digit+
exponent:       ("e"|"E") ["+"|"-"] digit+
```

The allowed range of floating point literals is implementation-dependent.

Some examples of floating point literals:

```
3.14    10.    .001    1e100    3.14e-10
```

Note that numeric literals do not include a sign; a phrase like `-1` is actually an expression composed of the operator `-` and the literal `1`.

2.5 Operators

The following tokens are operators:

```
+       -       *       /       %
<<      >>      &       |       ^       ~
<       ==      >       <=      <>      !=      >=
```

The comparison operators `<>` and `!=` are alternate spellings of the same operator.

2.6 Delimiters

The following tokens serve as delimiters or otherwise have a special meaning:

```
(       )       [       ]       {       }
;       ,       :       .       `       =
```

The following printing ASCII characters are not used in Python. Their occurrence outside string literals and comments is an unconditional error:

```
@       $       "       ?
```

They may be used by future versions of the language though!

Chapter 3

Data model

3.1 Objects, values and types

Objects are Python's abstraction for data. All data in a Python program is represented by objects or by relations between objects. (In a sense, and in conformance to Von Neumann's model of a "stored program computer", code is also represented by objects.)

Every object has an identity, a type and a value. An object's *identity* never changes once it has been created; you may think of it as the object's address in memory. An object's *type* is also unchangeable. It determines the operations that an object supports (e.g. "does it have a length?") and also defines the possible values for objects of that type. The *value* of some objects can change. Objects whose value can change are said to be *mutable*; objects whose value is unchangeable once they are created are called *immutable*. The type determines an object's (im)mutability.

Objects are never explicitly destroyed; however, when they become unreachable they may be garbage-collected. An implementation is allowed to delay garbage collection or omit it altogether — it is a matter of implementation quality how garbage collection is implemented, as long as no objects are collected that are still reachable. (Implementation note: the current implementation uses a reference-counting scheme which collects most objects as soon as they become unreachable, but never collects garbage containing circular references.)

Note that the use of the implementation's tracing or debugging facilities may keep objects alive that would normally be collectable.

Some objects contain references to "external" resources such as open files or windows. It is understood that these resources are freed when the object is garbage-collected, but since garbage collection is not guaranteed to happen, such objects also provide an explicit way to release the external resource, usually a `close` method. Programs are strongly recommended to always explicitly close such objects.

Some objects contain references to other objects; these are called *containers*. Examples of containers are tuples, lists and dictionaries. The references are part of a container's value. In most cases, when we talk about the value of a container, we imply the values, not the identities of the contained objects; however, when we talk about the (im)mutability of a container, only the identities of the immediately contained objects are implied. (So, if an immutable container contains a reference to a mutable object, its value changes if that mutable object is changed.)

Types affect almost all aspects of objects' lives. Even the meaning of object identity is affected in some sense: for immutable types, operations that compute new values may actually return a reference to any existing object with the same type and value, while for mutable objects this is not allowed. E.g. after

```
a = 1; b = 1; c = []; d = []
```

`a` and `b` may or may not refer to the same object with the value one, depending on the implementation, but `c` and `d` are guaranteed to refer to two different, unique, newly created empty lists.

3.2 The standard type hierarchy

Below is a list of the types that are built into Python. Extension modules written in C can define additional types. Future versions of Python may add types to the type hierarchy (e.g. rational or complex numbers, efficiently stored arrays of integers, etc.).

Some of the type descriptions below contain a paragraph listing 'special attributes'. These are attributes that provide access to the implementation and are not intended for general use. Their definition may change in the future. There are also some 'generic' special attributes, not listed with the individual objects: `__methods__` is a list of the method names of a built-in object, if it has any; `__members__` is a list of the data attribute names of a built-in object, if it has any.

None This type has a single value. There is a single object with this value. This object is accessed through the built-in name `None`. It is returned from functions that don't explicitly return an object.

Numbers These are created by numeric literals and returned as results by arithmetic operators and arithmetic built-in functions. Numeric objects are immutable; once created their value never changes. Python numbers are of course strongly related to mathematical numbers, but subject to the limitations of numerical representation in computers.

Python distinguishes between integers and floating point numbers:

Integers These represent elements from the mathematical set of whole numbers.
There are two types of integers:

Plain integers These represent numbers in the range -2^{31} through $2^{31}-1$. (The range may be larger on machines with a larger natural word size, but not smaller.) When the result of an operation falls outside this range, the exception `OverflowError` is raised. For the purpose of shift and mask operations, integers are assumed to have a binary, 2's complement notation using 32 or more bits, and hiding no bits from the user (i.e., all 2^{32} different bit patterns correspond to different values).

Long integers These represent numbers in an unlimited range, subject to available (virtual) memory only. For the purpose of shift and mask operations, a binary representation is assumed, and negative numbers are represented in a variant of 2's complement which gives the illusion of an infinite string of sign bits extending to the left.

The rules for integer representation are intended to give the most meaningful interpretation of shift and mask operations involving negative integers and the least surprises when switching between the plain and long integer domains. For any operation except left shift, if it yields a result in the plain integer domain without causing overflow, it will yield the same result in the long integer domain or when using mixed operands.

Floating point numbers These represent machine-level double precision floating point numbers. You are at the mercy of the underlying machine architecture and C implementation for the accepted range and handling of overflow.

Sequences These represent finite ordered sets indexed by natural numbers. The built-in function `len()` returns the number of elements of a sequence. When this number is n, the index set contains the numbers $0, 1, \ldots, n-1$. Element `i` of sequence `a` is selected by `a[i]`.

Sequences also support slicing: `a[i:j]` selects all elements with index k such that $i <= k < j$. When used as an expression, a slice is a sequence of the same type — this implies that the index set is renumbered so that it starts at 0 again.

Sequences are distinguished according to their mutability:

Immutable sequences An object of an immutable sequence type cannot change once it is created. (If the object contains references to other objects, these other objects may be mutable and may be changed; however the collection of objects directly referenced by an immutable object cannot change.)

The following types are immutable sequences:

Strings The elements of a string are characters. There is no separate character type; a character is represented by a string of one element. Characters represent (at least) 8-bit bytes. The built-in functions `chr()` and `ord()` convert between characters and nonnegative integers representing the byte values. Bytes with the values 0-127 represent the corresponding ASCII values. The string data type is also used to represent arrays of bytes, e.g. to hold data read from a file.
(On systems whose native character set is not ASCII, strings may use EBCDIC in their internal representation, provided the functions `chr()` and `ord()` implement a mapping between ASCII and EBCDIC, and string comparison preserves the ASCII order. Or perhaps someone can propose a better rule?)

Tuples The elements of a tuple are arbitrary Python objects. Tuples of two or more elements are formed by comma-separated lists of expressions. A tuple of one element (a 'singleton') can be formed by affixing a comma to an expression (an expression by itself does not create a tuple, since parentheses must be usable for grouping of expressions). An empty tuple can be formed by enclosing 'nothing' in parentheses.

Mutable sequences Mutable sequences can be changed after they are created. The subscription and slicing notations can be used as the target of assignment and `del` (delete) statements.

There is currently a single mutable sequence type:

Lists The elements of a list are arbitrary Python objects. Lists are formed by placing a comma-separated list of expressions in square brackets. (Note that there are no special cases needed to form lists of length 0 or 1.)

Mapping types These represent finite sets of objects indexed by arbitrary index sets. The subscript notation `a[k]` selects the element indexed by `k` from the mapping `a`; this can be used in expressions and as the target of assignments or `del` statements. The built-in function `len()` returns the number of elements in a mapping.

There is currently a single mapping type:

Dictionaries These represent finite sets of objects indexed by almost arbitrary values. The only types of values not acceptable as keys are values containing lists or dictionaries or other mutable types that are compared by value rather than by object identity — the reason being that the implementation requires that a key's hash value be constant. Numeric types used for keys obey the normal rules for numeric comparison: if two numbers compare equal (e.g. 1 and 1.0) then they can be used interchangeably to index the same dictionary entry.

Dictionaries are mutable; they are created by the `{...}` notation (see section 5.2.5).

Callable types These are the types to which the function call (invocation) operation, written as `function(argument, argument, ...)`, can be applied:

User-defined functions A user-defined function object is created by a function definition (see section 7.5). It should be called with an argument list containing the same number of items as the function's formal parameter list.

Special read-only attributes: `func_code` is the code object representing the compiled function body, and `func_globals` is (a reference to) the dictionary that holds the function's global variables — it implements the global name space of the module in which the function was defined.

User-defined methods A user-defined method (a.k.a. *object closure*) is a pair of a class instance object and a user-defined function. It should be called with an argument list containing one item less than the number of items in the function's formal parameter list. When called, the class instance becomes the first argument, and the call arguments are shifted one to the right.

Special read-only attributes: `im_self` is the class instance object, `im_func` is the function object.

Built-in functions A built-in function object is a wrapper around a C function. Examples of built-in functions are `len` and `math.sin`. There are no special attributes. The number and type of the arguments are determined by the C function.

Built-in methods This is really a different disguise of a built-in function, this time containing an object passed to the C function as an implicit extra argument. An example of a built-in method is `list.append` if `list` is a list object.

Classes Class objects are described below. When a class object is called as a parameterless function, a new class instance (also described below) is created and returned. The class's initialization function is not called — this is the responsibility of the caller. It is illegal to call a class object with one or more arguments.

Modules Modules are imported by the `import` statement (see section 6.10). A module object is a container for a module's name space, which is a dictionary (the same dictionary as referenced by the `func_globals` attribute of functions defined in the module). Module attribute references are translated to lookups in this dictionary. A module object does not contain the code object used to initialize the module (since it isn't needed once the initialization is done).

Attribute assignment update the module's name space dictionary.

Special read-only attributes: `__dict__` yields the module's name space as a dictionary object; `__name__` yields the module's name as a string object.

Classes Class objects are created by class definitions (see section 7.6). A class is a container for a dictionary containing the class's name space. Class attribute references are translated to lookups in this dictionary. When an attribute name is not found there, the attribute search continues in the base classes. The search is depth-first, left-to-right in the order of their occurrence in the base class list.

Class attribute assignments update the class's dictionary, never the dictionary of a base class.

A class can be called as a parameterless function to yield a class instance (see above).

Special read-only attributes: `__dict__` yields the dictionary containing the class's name space; `__bases__` yields a tuple (possibly empty or a singleton) containing the base classes, in the order of their occurrence in the base class list.

Class instances A class instance is created by calling a class object as a parameterless function. A class instance has a dictionary in which attribute references are searched. When an attribute is not found there, and the instance's class has an attribute by that name, and that class attribute is a user-defined function (and in no other cases), the instance attribute reference yields a user-defined method object (see above) constructed from the instance and the function.

Attribute assignments update the instance's dictionary.

Class instances can pretend to be numbers, sequences, or mappings if they have methods with certain special names. These are described in section 3.3.

Special read-only attributes: `__dict__` yields the attribute dictionary; `__class__` yields the instance's class.

Files A file object represents an open file. (It is a wrapper around a C `stdio` file pointer.) File objects are created by the `open()` built-in function, and also by `posix.popen()` and the `makefile` method of socket objects. `sys.stdin`, `sys.stdout` and `sys.stderr` are file objects corresponding the the interpreter's standard input, output and error streams. See the Python Library Reference for methods of file objects and other details.

Internal types A few types used internally by the interpreter are exposed to the user. Their definition may change with future versions of the interpreter, but they are mentioned here for completeness.

Code objects Code objects represent executable code. The difference between a code object and a function object is that the function object contains an explicit reference to the function's context (the module in which it was defined) which a code object contains no context. There is no way to execute a bare code object.

Special read-only attributes: `co_code` is a string representing the sequence of instructions; `co_consts` is a list of literals used by the code; `co_names` is a list of names (strings) used by the code; `co_filename` is the filename from which the code was compiled. (To find out the line numbers, you would have to decode the instructions; the standard library module `dis` contains an example of how to do this.)

Frame objects Frame objects represent execution frames. They may occur in traceback objects (see below).

Special read-only attributes: `f_back` is to the previous stack frame (towards the caller), or `None` if this is the bottom stack frame; `f_code` is the code object being executed in this frame; `f_globals` is the dictionary used to look up global variables; `f_locals` is used for local variables; `f_lineno` gives the line number and `f_lasti` gives the precise instruction (this is an index into the instruction string of the code object).

Traceback objects Traceback objects represent a stack trace of an exception. A traceback object is created when an exception occurs. When the search for an exception handler unwinds the execution stack, at each unwound level a traceback object is inserted in front of the current traceback. When an exception handler is entered (see also section 7.4), the stack trace is made available to the program as `sys.exc_traceback`. When the program contains no suitable handler, the stack trace is written (nicely formatted) to the standard error stream; if the interpreter is interactive, it is also made available to the user as `sys.last_traceback`.

Special read-only attributes: `tb_next` is the next level in the stack trace (towards the frame where the exception occurred), or `None` if there is no next level; `tb_frame` points to the execution frame of the current level; `tb_lineno` gives the line number where the exception occurred; `tb_lasti` indicates the precise instruction. The line number and last instruction in the traceback may differ from the line number of its frame object if the exception occurred in a `try` statement with no matching `except` clause or with a `finally` clause.

3.3 Special method names

A class can implement certain operations that are invoked by special syntax (such as subscription or arithmetic operations) by defining methods with special names. For instance, if a class defines a method named `__getitem__`, and `x` is an instance of this class, then `x[i]` is equivalent to `x.__getitem__(i)`. (The reverse is not true — if `x` is a list object, `x.__getitem__(i)` is not equivalent to `x[i]`.)

Except for `__repr__`, `__str__` and `__cmp__`, attempts to execute an operation raise an exception when no appropriate method is defined. For `__repr__` and `__cmp__`, the traditional interpretations are used in this case. For `__str__`, the `__repr__` method is used.

3.3.1 Special methods for any type

`__init__(self, args...)` Called when the instance is created. The arguments are those passed to the class constructor expression. If a base class has an `__init__` method the derived class's `__init__` method must explicitly call it to ensure proper initialization of the base class part of the instance.

`__del__(self)` Called when the instance is about to be destroyed. If a base class has an `__del__` method the derived class's `__del__` method must explicitly call it to ensure proper deletion of the base class part of the instance. Note that it is possible for the `__del__` method to postpone destruction of the instance by creating a new reference to it. It may then be called at a later time when this new reference is deleted. Also note that it is not guaranteed that `__del__` methods are called for objects that still exist when the interpreter exits.

`__repr__(self)` Called by the `repr()` built-in function and by conversions (reverse quotes) to compute the string representation of an object.

`__str__(self)` Called by the `str()` built-in function and by the `print` statement compute the string representation of an object.

`__cmp__(self, other)` Called by all comparison operations. Should return -1 if `self < other`, 0 if `self == other`, +1 if `self > other`. If no `__cmp__` operation is defined, class instances are compared by object identity ("address"). (Implementation note: due to limitations in the interpreter, exceptions raised by comparisons are ignored, and the objects will be considered equal in this case.)

`__hash__(self)` Called by dictionary operations and by the built-in function `hash()`. Should return a 32-bit integer usable as a hash value for dictionary operations. The only required property is that objects which compare equal have the same hash value; it is advised to somehow mix together (e.g. using exclusing or) the hash values for the components of the object that also play a part in comparison of objects. If a class does not define a `__cmp__` method it should not define a `__hash__` operation either; if it defines `__cmp__` but not `__hash__` its instances will not be usable as dictionary keys. If a class defines mutable objects and implements a `__cmp__` method it should not implement `__hash__`, since the dictionary implementation assumes that a key's hash value is a constant.

3.3.2 Special methods for sequence and mapping types

`__len__(self)` Called to implement the built-in function `len()`. Should return the length of the object, an integer `>=` 0. Also, an object whose `__len__()` method returns 0 is considered to be false in a Boolean context.

`__getitem__(self, key)` Called to implement evaluation of `self[key]`. Note that the special interpretation of negative keys (if the class wishes to emulate a sequence type) is up to the `__getitem__` method.

`__setitem__(self, key, value)` Called to implement assignment to `self[key]`. Same note as for `__getitem__`.

`__delitem__(self, key)` Called to implement deletion of `self[key]`. Same note as for `__getitem__`.

3.3.3 Special methods for sequence types

`__getslice__(self, i, j)` Called to implement evaluation of `self[i:j]`. Note that missing `i` or `j` are replaced by 0 or `len(self)`, respectively, and `len(self)` has been added (once) to originally negative `i` or `j` by the time this function is called (unlike for `__getitem__`).

`__setslice__(self, i, j, sequence)` Called to implement assignment to `self[i:j]`. Same notes as for `__getslice__`.

`__delslice__(self, i, j)` Called to implement deletion of `self[i:j]`. Same notes as for `__getslice__`.

3.3.4 Special methods for numeric types

`__add__(self, other)`
`__sub__(self, other)`
`__mul__(self, other)`
`__div__(self, other)`
`__mod__(self, other)`
`__divmod__(self, other)`
`__pow__(self, other)`
`__lshift__(self, other)`
`__rshift__(self, other)`
`__and__(self, other)`
`__xor__(self, other)`
`__or__(self, other)`
Called to implement the binary arithmetic operations (`+`, `-`, `*`, `/`, `%`, `divmod()`, `pow()`, `<<`, `>>`, `&`, `^`, `|`).

`__neg__(self)`
`__pos__(self)`
`__abs__(self)`

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

`__nonzero__(self)` Called to implement boolean testing; should return 0 or 1. An alternative name for this method is `__len__`.

`__coerce__(self, other)` Called to implement "mixed-mode" numeric arithmetic. Should either return a tuple containing self and other converted to a common numeric type, or None if no way of conversion is known. 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).

Note that this method is not called to coerce the arguments to `+` and `*`, because these are also used to implement sequence concatenation and repetition, respectively. Also note that, for the same reason, in `n*x`, where `n` is a built-in number and `x` is an instance, a call to `x.__mul__(n)` is made.[1]

`__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.

[1] The interpreter should really distinguish between user-defined classes implementing sequences, mappings or numbers, but currently it doesn't — hence this strange exception.

Chapter 4

Execution model

4.1 Code blocks, execution frames, and name spaces

A *code block* is a piece of Python program text that can be executed as a unit, such as a module, a class definition or a function body. Some code blocks (like modules) are executed only once, others (like function bodies) may be executed many times. Code block may textually contain other code blocks. Code blocks may invoke other code blocks (that may or may not be textually contained in them) as part of their execution, e.g. by invoking (calling) a function.

The following are code blocks: A module is a code block. A function body is a code block. A class definition is a code block. Each command typed interactively is a separate code block; a script file is a code block. The string argument passed to the built-in function `eval` and to the `exec` statement are code blocks. And finally, the expression read and evaluated by the built-in function `input` is a code block.

A code block is executed in an execution frame. An *execution frame* contains some administrative information (used for debugging), determines where and how execution continues after the code block's execution has completed, and (perhaps most importantly) defines two name spaces, the local and the global name space, that affect execution of the code block.

A *name space* is a mapping from names (identifiers) to objects. A particular name space may be referenced by more than one execution frame, and from other places as well. Adding a name to a name space is called *binding* a name (to an object); changing the mapping of a name is called *rebinding*; removing a name is *unbinding*. Name spaces are functionally equivalent to dictionaries.

The *local name space* of an execution frame determines the default place where names are defined and searched. The *global name space* determines the place where names listed in `global` statements are defined and searched, and where names that are not explicitly bound in the current code block are searched.

Whether a name is local or global in a code block is determined by static inspection of the source text for the code block: in the absence of `global` statements, a name that is bound anywhere in the code block is local in the entire code block; all other names are considered global. The `global` statement forces global interpretation of selected names throughout the code block. The following constructs bind names: formal parameters, `import` statements, class and function definitions (these bind the class or function name), and targets that are identifiers if occurring in an assignment, `for` loop header, or `except` clause header.

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

When a global name is not found in the global name space, it is searched in the list of "built-in" names (which is actually the global name space of the module `__builtin__`). When a name is not found at all, the `NameError` exception is raised.[1]

The following table lists the meaning of the local and global name space for various types of code blocks. The name space for a particular module is automatically created when the module is first referenced.

Code block type	Global name space	Local name space	Notes
Module	n.s. for this module	same as global	
Script	n.s. for `__main__`	same as global	
Interactive command	n.s. for `__main__`	same as global	
Class definition	global n.s. of containing block	new n.s.	
Function body	global n.s. of containing block	new n.s.	
String passed to `exec` statement	global n.s. of cobtaining block	local n.s. of containing block	(1)
String passed to `eval()`	global n.s. of caller	local n.s. of caller	(1)
File read by `execfile()`	global n.s. of caller	local n.s. of caller	(1)
Expression read by `input`	global n.s. of caller	local n.s. of caller	

Notes:

n.s. means *name space*

(1) The global and local name space for these can be overridden with optional extra arguments.

4.2 Exceptions

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

The Python interpreter raises an exception when it detects an run-time error (such as division by zero). A Python program can also explicitly raise an exception with the `raise` statement. Exception handlers are specified with the `try...except` statement.

[1]If the code block contains `exec` statement or the construct `from ... import *`, the semantics of names not explicitly mentioned in a `global` statement change subtly: name lookup first searches the local name space, then the global one, then the built-in one.

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

When an exception is not handled at all, the interpreter terminates execution of the program, or returns to its interactive main loop.

Exceptions are identified by string objects. Two different string objects with the same value identify different exceptions.

When an exception is raised, an object (maybe `None`) is passed as the exception's "parameter"; this object does not affect the selection of an exception handler, but is passed to the selected exception handler as additional information.

See also the description of the `try` and `raise` statements.

Chapter 5

Expressions and conditions

Note: In this and the following chapters, extended BNF notation will be used to describe syntax, not lexical analysis.

This chapter explains the meaning of the elements of expressions and conditions. Conditions are a superset of expressions, and a condition may be used wherever an expression is required by enclosing it in parentheses. The only places where expressions are used in the syntax instead of conditions is in expression statements and on the right-hand side of assignment statements; this catches some nasty bugs like accidentally writing `x == 1` instead of `x = 1`.

The comma plays several roles in Python's syntax. It is usually an operator with a lower precedence than all others, but occasionally serves other purposes as well; e.g. it separates function arguments, is used in list and dictionary constructors, and has special semantics in `print` statements.

When (one alternative of) a syntax rule has the form

```
name:           othername
```

and no semantics are given, the semantics of this form of `name` are the same as for `othername`.

5.1 Arithmetic conversions

When a description of an arithmetic operator below uses the phrase "the numeric arguments are converted to a common type", this both means that if either argument is not a number, a `TypeError` exception is raised, and that otherwise the following conversions are applied:

- first, if either argument is a floating point number, the other is converted to floating point;
- else, if either argument is a long integer, the other is converted to long integer;

- otherwise, both must be plain integers and no conversion is necessary.

5.2 Atoms

Atoms are the most basic elements of expressions. Forms enclosed in reverse quotes or in parentheses, brackets or braces are also categorized syntactically as atoms. The syntax for atoms is:

```
atom:           identifier | literal | enclosure
enclosure:      parenth_form | list_display | dict_display | string_conversion
```

5.2.1 Identifiers (Names)

An identifier occurring as an atom is a reference to a local, global or built-in name binding. If a name is assigned to anywhere in a code block (even in unreachable code), and is not mentioned in a `global` statement in that code block, then it refers to a local name throughout that code block. When it is not assigned to anywhere in the block, or when it is assigned to but also explicitly listed in a `global` statement, it refers to a global name if one exists, else to a built-in name (and this binding may dynamically change).

When the name is bound to an object, evaluation of the atom yields that object. When a name is not bound, an attempt to evaluate it raises a `NameError` exception.

5.2.2 Literals

Python knows string and numeric literals:

```
literal:        stringliteral | integer | longinteger | floatnumber
```

Evaluation of a literal yields an object of the given type (string, integer, long integer, floating point number) with the given value. The value may be approximated in the case of floating point literals. See section 2.4 for details.

All literals correspond to immutable data types, and hence the object's identity is less important than its value. Multiple evaluations of literals with the same value (either the same occurrence in the program text or a different occurrence) may obtain the same object or a different object with the same value.

(In the original implementation, all literals in the same code block with the same type and value yield the same object.)

5.2.3 Parenthesized forms

A parenthesized form is an optional condition list enclosed in parentheses:

```
parenth_form:      "(" [condition_list] ")"
```

A parenthesized condition list yields whatever that condition list yields.

An empty pair of parentheses yields an empty tuple object. Since tuples are immutable, the rules for literals apply here.

(Note that tuples are not formed by the parentheses, but rather by use of the comma operator. The exception is the empty tuple, for which parentheses *are* required — allowing unparenthesized "nothing" in expressions would causes ambiguities and allow common typos to pass uncaught.)

5.2.4 List displays

A list display is a possibly empty series of conditions enclosed in square brackets:

```
list_display:   "[" [condition_list] "]"
```

A list display yields a new list object.

If it has no condition list, the list object has no items. Otherwise, the elements of the condition list are evaluated from left to right and inserted in the list object in that order.

5.2.5 Dictionary displays

A dictionary display is a possibly empty series of key/datum pairs enclosed in curly braces:

```
dict_display:   "{" [key_datum_list] "}"
key_datum_list: key_datum ("," key_datum)* [","]
key_datum:      condition ":" condition
```

A dictionary display yields a new dictionary object.

The key/datum pairs are evaluated from left to right to define the entries of the dictionary: each key object is used as a key into the dictionary to store the corresponding datum.

Restrictions on the types of the key values are listed earlier in section 3.2. Clashes between duplicate keys are not detected; the last datum (textually rightmost in the display) stored for a given key value prevails.

5.2.6 String conversions

A string conversion is a condition list enclosed in reverse (or backward) quotes:

```
string_conversion: "‘" condition_list "‘"
```

A string conversion evaluates the contained condition list and converts the resulting object into a string according to rules specific to its type.

If the object is a string, a number, `None`, or a tuple, list or dictionary containing only objects whose type is one of these, the resulting string is a valid Python expression which can be passed to the built-in function `eval()` to yield an expression with the same value (or an approximation, if floating point numbers are involved).

(In particular, converting a string adds quotes around it and converts "funny" characters to escape sequences that are safe to print.)

It is illegal to attempt to convert recursive objects (e.g. lists or dictionaries that contain a reference to themselves, directly or indirectly.)

5.3 Primaries

Primaries represent the most tightly bound operations of the language. Their syntax is:

```
primary:        atom | attributeref | subscription | slicing | call
```

5.3.1 Attribute references

An attribute reference is a primary followed by a period and a name:

```
attributeref:   primary "." identifier
```

The primary must evaluate to an object of a type that supports attribute references, e.g. a module or a list. This object is then asked to produce the attribute whose name is the identifier. If this attribute is not available, the exception `AttributeError` is raised. Otherwise, the type and value of the object produced is determined by the object. Multiple evaluations of the same attribute reference may yield different objects.

5.3.2 Subscriptions

A subscription selects an item of a sequence (string, tuple or list) or mapping (dictionary) object:

```
subscription:   primary "[" condition "]"
```

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

If it is a mapping, the condition 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 it is a sequence, the condition 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 resulting value must be a nonnegative integer smaller than the number of items in the sequence, and the subscription selects the item whose index is that value (counting from zero).

A string's items are characters. A character is not a separate data type but a string of exactly one character.

5.3.3 Slicings

A slicing (or slice) selects a range of items in a sequence (string, tuple or list) object:

```
slicing:        primary "[" [condition] ":" [condition] "]"
```

The primary must evaluate to a sequence object. The lower and upper bound expressions, if present, must evaluate to plain integers; defaults are zero and the sequence's length, respectively. If either bound is negative, the sequence's length is added to it. The slicing now selects all items with index k such that $i <= k < j$ where i and j are the specified lower and upper bounds. This may be an empty sequence. It is not an error if i or j lie outside the range of valid indexes (such items don't exist so they aren't selected).

5.3.4 Calls

A call calls a callable object (e.g. a function) with a possibly empty series of arguments:

```
call:           primary "(" [condition_list] ")"
```

The primary must evaluate to a callable object (user-defined functions, built-in functions, methods of built-in objects, class objects, and methods of class instances are callable). If it is a class, the argument list must be empty; otherwise, the arguments are evaluated.

A call always returns some value, possibly `None`, unless it raises an exception. How this value is computed depends on the type of the callable object. If it is:

a user-defined function: the code block for the function is executed, passing it the argument list. The first thing the code block will do is bind the formal parameters to the arguments; this is described in section 7.5. When the code block executes a `return` statement, this specifies the return value of the function call.

a built-in function or method: the result is up to the interpreter; see the library reference manual for the descriptions of built-in functions and methods.

a class object: a new instance of that class is returned.

a class instance method: the corresponding user-defined function is called, with an argument list that is one longer than the argument list of the call: the instance becomes the first argument.

5.4 Unary arithmetic operations

All unary arithmetic (and bit-wise) operations have the same priority:

```
u_expr:         primary | "-" u_expr | "+" u_expr | "~" u_expr
```

The unary "-" (minus) operator yields the negation of its numeric argument.

The unary "+" (plus) operator yields its numeric argument unchanged.

The unary "~" (invert) operator yields the bit-wise inversion of its plain or long integer argument. The bit-wise inversion of `x` is defined as `-(x+1)`.

In all three cases, if the argument does not have the proper type, a `TypeError` exception is raised.

5.5 Binary arithmetic operations

The binary arithmetic operations have the conventional priority levels. Note that some of these operations also apply to certain non-numeric types. There is no "power" operator, so there are only two levels, one for multiplicative operators and one for additive operators:

```
m_expr:         u_expr | m_expr "*" u_expr
              | m_expr "/" u_expr | m_expr "%" u_expr
a_expr:         m_expr | aexpr "+" m_expr | aexpr "-" m_expr
```

The "*" (multiplication) operator yields the product of its arguments. The arguments must either both be numbers, or one argument must be a plain integer and the other must be a sequence. In the former case, the numbers are converted to a common type and then multiplied together. In the latter case, sequence repetition is performed; a negative repetition factor yields an empty sequence.

The "/" (division) operator yields the quotient of its arguments. The numeric arguments are first converted to a common type. Plain or long integer division yields an integer of the same type; the result is that of mathematical division with the 'floor' function applied to the result. Division by zero raises the `ZeroDivisionError` exception.

The "%" (modulo) operator yields the remainder from the division of the first argument by the second. The numeric arguments are first converted to a common type. A zero right argument raises the `ZeroDivisionError` exception. The arguments may be floating point numbers, e.g. `3.14 % 0.7` equals `0.34`. The modulo operator always yields a result with the same sign as its second operand (or zero); the absolute value of the result is strictly smaller than the second operand.

The integer division and modulo operators are connected by the following identity: `x == (x/y)*y + (x%y)`. Integer division and modulo are also connected with the built-in function `divmod()`: `divmod(x, y) == (x/y, x%y)`. These identities don't hold for floating point numbers; there a similar identity holds where `x/y` is replaced by `floor(x/y)`).

The "+" (addition) operator yields the sum of its arguments. The arguments must either both be numbers, or both sequences of the same type. In the former case, the numbers are converted to a common type and then added together. In the latter case, the sequences are concatenated.

The "-" (subtraction) operator yields the difference of its arguments. The numeric arguments are first converted to a common type.

5.6 Shifting operations

The shifting operations have lower priority than the arithmetic operations:

```
shift_expr:     a_expr | shift_expr ( "<<" | ">>" ) a_expr
```

These operators accept plain or long integers as arguments. The arguments are converted to a common type. They shift the first argument to the left or right by the number of bits given by the second argument.

A right shift by n bits is defined as division by 2^n. A left shift by n bits is defined as multiplication with 2^n; for plain integers there is no overflow check so this drops bits and flip the sign if the result is not less than 2^{31} in absolute value.

Negative shift counts raise a `ValueError` exception.

5.7 Binary bit-wise operations

Each of the three bitwise operations has a different priority level:

```
and_expr:       shift_expr | and_expr "&" shift_expr
xor_expr:       and_expr | xor_expr "^" and_expr
or_expr:       xor_expr | or_expr "|" xor_expr
```

The "&" operator yields the bitwise AND of its arguments, which must be plain or long integers. The arguments are converted to a common type.

The "^" operator yields the bitwise XOR (exclusive OR) of its arguments, which must be plain or long integers. The arguments are converted to a common type.

The "|" operator yields the bitwise (inclusive) OR of its arguments, which must be plain or long integers. The arguments are converted to a common type.

5.8 Comparisons

Contrary to C, all comparison operations in Python have the same priority, which is lower than that of any arithmetic, shifting or bitwise operation. Also contrary to C, expressions like `a < b < c` have the interpretation that is conventional in mathematics:

```
comparison:     or_expr (comp_operator or_expr)*
comp_operator:  "<"|">"|"=="|">="|"<="|"<>"|"!="|"is" ["not"]|["not"] "in"
```

Comparisons yield integer values: 1 for true, 0 for false.

Comparisons can be chained arbitrarily, e.g. $x < y <= z$ is equivalent to $x < y$ `and` $y <= z$, except that y is evaluated only once (but in both cases z is not evaluated at all when $x < y$ is found to be false).

Formally, $e_0 op_1 e_1 op_2 e_2 ... e_{n-1} op_n e_n$ is equivalent to $e_0 op_1 e_1$ `and` $e_1 op_2 e_2$ `and` ... `and` $e_{n-1} op_n e_n$, except that each expression is evaluated at most once.

Note that $e_0 op_1 e_1 op_2 e_2$ does not imply any kind of comparison between e_0 and e_2, e.g. $x < y > z$ is perfectly legal.

The forms `<>` and `!=` are equivalent; for consistency with C, `!=` is preferred; where `!=` is mentioned below `<>` is also implied.

The operators `"<", ">", "==", ">=", "<="`, and `"!="` compare the values of two objects. The objects needn't have the same type. If both are numbers, they are coverted to a common type. Otherwise, objects of different types *always* compare unequal, and are ordered consistently but arbitrarily.

(This unusual definition of comparison is done to simplify the definition of operations like sorting and the `in` and `not in` operators.)

Comparison of objects of the same type depends on the type:

- Numbers are compared arithmetically.
- Strings are compared lexicographically using the numeric equivalents (the result of the built-in function `ord`) of their characters.
- Tuples and lists are compared lexicographically using comparison of corresponding items.
- Mappings (dictionaries) are compared through lexicographic comparison of their sorted (key, value) lists.[1]
- Most other types compare unequal unless they are the same object; the choice whether one object is considered smaller or larger than another one is made arbitrarily but consistently within one execution of a program.

The operators `in` and `not in` test for sequence membership: if y is a sequence, x `in` y is true if and only if there exists an index i such that $x = y[i]$. x `not in` y yields the inverse truth value. The exception `TypeError` is raised when y is not a sequence, or when y is a string and x is not a string of length one.[2]

The operators `is` and `is not` test for object identity: x `is` y is true if and only if x and y are the same object. x `is not` y yields the inverse truth value.

[1]This is expensive since it requires sorting the keys first, but about the only sensible definition. An earlier version of Python compared dictionaries by identity only, but this caused surprises because people expected to be able to test a dictionary for emptiness by comparing it to {}.

[2]The latter restriction is sometimes a nuisance.

5.9 Boolean operations

Boolean operations have the lowest priority of all Python operations:

```
condition:      or_test | lambda_form
or_test:        and_test | or_test "or" and_test
and_test:       not_test | and_test "and" not_test
not_test:       comparison | "not" not_test
lambda_form: "lambda" [parameter_list]: condition
```

In the context of Boolean operations, and also when conditions are used by control flow statements, the following values are interpreted as false: `None`, numeric zero of all types, empty sequences (strings, tuples and lists), and empty mappings (dictionaries). All other values are interpreted as true.

The operator `not` yields 1 if its argument is false, 0 otherwise.

The condition x `and` y first evaluates x; if x is false, its value is returned; otherwise, y is evaluated and the resulting value is returned.

The condition x `or` y first evaluates x; if x is true, its value is returned; otherwise, y is evaluated and the resulting value is returned.

(Note that `and` and `or` do not restrict the value and type they return to 0 and 1, but rather return the last evaluated argument. This is sometimes useful, e.g. if `s` is a string that should be replaced by a default value if it is empty, the expression `s or 'foo'` yields the desired value. Because `not` has to invent a value anyway, it does not bother to return a value of the same type as its argument, so e.g. `not 'foo'` yields `0`, not `''`.)

Lambda forms (lambda expressions) have the same syntactic position as conditions. They are a shorthand to create anonymous functions; the expression `lambda` *arguments*`:` *condition* yields a function object that behaves virtually identical to one defined with `def` *name* `(`*arguments*`): return` *condition*. See section 7.5 for the syntax of parameter lists. Note that functions created with lambda forms cannot contain statements.

5.10 Expression lists and condition lists

```
expr_list:      or_expr ("," or_expr)* [","]
cond_list:      condition ("," condition)* [","]
```

The only difference between expression lists and condition lists is the lowest priority of operators that can be used in them without being enclosed in parentheses; condition lists allow all operators, while expression lists don't allow comparisons and Boolean operators (they do allow bitwise and shift operators though).

Expression lists are used in expression statements and assignments; condition lists are used everywhere else where a list of comma-separated values is required.

An expression (condition) list containing at least one comma yields a tuple. The length of the tuple is the number of expressions (conditions) in the list. The expressions (conditions) are evaluated from left to right. (Conditions lists are used syntactically is a few places where no tuple is constructed but a list of values is needed nevertheless.)

The trailing comma is required only to create a single tuple (a.k.a. a *singleton*); it is optional in all other cases. A single expression (condition) without a trailing comma doesn't create a tuple, but rather yields the value of that expression (condition).

(To create an empty tuple, use an empty pair of parentheses: `()`.)

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
               | assignment_stmt
               | pass_stmt
               | del_stmt
               | print_stmt
               | return_stmt
               | raise_stmt
               | break_stmt
               | continue_stmt
               | import_stmt
               | global_stmt
               | access_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`):

```
expression_stmt: expression_list
```

An expression statement evaluates the expression list (which may be a single expression). If the value is not `None`, it is converted to a string using the rules for string conversions (expressions in reverse quotes), and the resulting string is written to standard output (see section 6.5) on a line by itself.

(The exception for `None` is made so that procedure calls, which are syntactically equivalent to expressions, do not cause any output. A tuple with only `None` items is written normally.)

6.2 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 section 5.3 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 a target is part of a mutable 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 section 3.2).

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 target list is a comma-separated list of targets: the object must be a tuple with the same number of items as the list contains targets, and the items are assigned, from left to right, to the corresponding targets.

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

- If the target is an identifier (name):
  - If the name does not occur in a `global` statement in the current code block: the name is bound to the object in the current local name space.
  - Otherwise: the name is bound to the object in the current global name space.

  The name is rebound if it was already bound.
- If the target is a target list enclosed in parentheses: the object is assigned to that target list as described above.
- If the target is a target list enclosed in square brackets: the object must be a list with the same number of items as the target list contains targets, and its items are assigned, from left to right, to the corresponding targets.

- If the target is an attribute reference: The primary expression in the reference is evaluated. It should yield an object with assignable attributes; if this is not the case, `TypeError` is raised. That object is then asked to assign the assigned object to the given attribute; if it cannot perform the assignment, it raises an exception (usually but not necessarily `AttributeError`).

- If the target is a subscription: The primary expression in the reference is evaluated. It should yield either a mutable sequence (list) object or a mapping (dictionary) object. Next, the subscript expression is evaluated.

  If the primary is a mutable sequence object (a list), the subscript must yield a plain integer. If it is negative, the sequence's length is added to it. The resulting value must be a nonnegative integer less than the sequence's length, and the sequence is asked to assign the assigned object to its item with that index. If the index is out of range, `IndexError` is raised (assignment to a subscripted sequence cannot add new items to a list).

  If the primary is a mapping (dictionary) object, the subscript must have a type compatible with the mapping's key type, and the mapping is then asked to to create a key/datum pair which maps the subscript to the assigned object. This can either replace an existing key/value pair with the same key value, or insert a new key/value pair (if no key with the same value existed).

- If the target is a slicing: The primary expression in the reference is evaluated. It should yield a mutable sequence object (e.g. a list). The assigned object should be a sequence object of the same type. Next, the lower and upper bound expressions are evaluated, insofar they are present; defaults are zero and the sequence's length. The bounds should evaluate to (small) integers. If either bound is negative, the sequence's length is added to it. The resulting bounds are clipped to lie between zero and the sequence's length, inclusive. Finally, the sequence object is asked to replace the slice with the items of the assigned sequence. The length of the slice may be different from the length of the assigned sequence, thus changing the length of the target sequence, if the object allows it.

(In the original implementation, the syntax for targets is taken to be the same as for expressions, and invalid syntax is rejected during the code generation phase, causing less detailed error messages.)

6.3 The `pass` statement

```
pass_stmt:      "pass"
```

`pass` is a null operation — when it is executed, nothing happens. It is useful as a placeholder when a statement is required syntactically, but no code needs to be executed, for example:

```
def f(arg): pass    # a function that does nothing (yet)

class C: pass       # an class with no methods (yet)
```

6.4 The del statement

```
del_stmt:       "del" target_list
```

Deletion is recursively defined very similar to the way assignment is defined. Rather that spelling it out in full details, here are some hints.

Deletion of a target list recursively deletes each target, from left to right.

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 `global` statement in the same code block.

Deletion of attribute references, subscriptions and slicings is passed to the primary object involved; deletion of a slicing is in general equivalent to assignment of an empty slice of the right type (but even this is determined by the sliced object).

6.5 The print statement

```
print_stmt:     "print" [ condition ("," condition)* [","] ]
```

`print` evaluates each condition in turn and writes the resulting object to standard output (see below). If an object is not a string, it is first converted to a string using the rules for string conversions. The (resulting or original) string is then written. A space is written before each object is (converted and) written, unless the output system believes it is positioned at the beginning of a line. This is the case: (1) when no characters have yet been written to standard output; or (2) when the last character written to standard output is `\n`; or (3) when the last write operation on standard output was not a `print` statement. (In some cases it may be functional to write an empty string to standard output for this reason.)

A `"\n"` character is written at the end, unless the `print` statement ends with a comma. This is the only action if the statement contains just the keyword `print`.

Standard output is defined as the file object named `stdout` in the built-in module `sys`. If no such object exists, or if it is not a writable file, a `RuntimeError` exception is raised. (The original implementation attempts to write to the system's original standard output instead, but this is not safe, and should be fixed.)

6.6 The return statement

```
return_stmt:    "return" [condition_list]
```

`return` may only occur syntactically nested in a function definition, not within a nested class definition.

If a condition list is present, it is evaluated, else `None` is substituted.

`return` leaves the current function call with the condition list (or `None`) as return value.

When `return` passes control out of a `try` statement with a `finally` clause, that finally clause is executed before really leaving the function.

6.7 The `raise` statement

```
raise_stmt:     "raise" condition ["," condition]
```

`raise` evaluates its first condition, which must yield a string object. If there is a second condition, this is evaluated, else `None` is substituted.

It then raises the exception identified by the first object, with the second one (or `None`) as its parameter.

6.8 The `break` statement

```
break_stmt:     "break"
```

`break` may only occur syntactically nested in a `for` or `while` loop, but not nested in a function or class definition within that loop.

It terminates the nearest enclosing loop, skipping the optional `else` clause if the loop has one.

If a `for` loop is terminated by `break`, the loop control target keeps its current value.

When `break` passes control out of a `try` statement with a `finally` clause, that finally clause is executed before really leaving the loop.

6.9 The `continue` statement

```
continue_stmt:  "continue"
```

`continue` may only occur syntactically nested in a `for` or `while` loop, but not nested in a function or class definition or `try` statement within that loop.[1]

It continues with the next cycle of the nearest enclosing loop.

6.10 The `import` statement

```
import_stmt:     "import" identifier ("," identifier)*
               | "from" identifier "import" identifier ("," identifier)*
               | "from" identifier "import" "*"
```

[1]Except that it may currently occur within an `except` clause.

Import statements are executed in two steps: (1) find a module, and initialize it if necessary; (2) define a name or names in the local name space (of the scope where the `import` statement occurs). The first form (without `from`) repeats these steps for each identifier in the list, the `from` form performs them once, with the first identifier specifying the module name.

The system maintains a table of modules that have been initialized, indexed by module name. (The current implementation makes this table accessible as `sys.modules`.) When a module name is found in this table, step (1) is finished. If not, a search for a module definition is started. This first looks for a built-in module definition, and if no built-in module if the given name is found, it searches a user-specified list of directories for a file whose name is the module name with extension `".py"`. (The current implementation uses the list of strings `sys.path` as the search path; it is initialized from the shell environment variable `$PYTHONPATH`, with an installation-dependent default.)

If a built-in module is found, its built-in initialization code is executed and step (1) is finished. If no matching file is found, `ImportError` is raised. If a file is found, it is parsed, yielding an executable code block. If a syntax error occurs, `SyntaxError` is raised. Otherwise, an empty module of the given name is created and inserted in the module table, and then the code block is executed in the context of this module. Exceptions during this execution terminate step (1).

When step (1) finishes without raising an exception, step (2) can begin.

The first form of `import` statement binds the module name in the local name space to the module object, and then goes on to import the next identifier, if any. The `from` from does not bind the module name: it goes through the list of identifiers, looks each one of them up in the module found in step (1), and binds the name in the local name space to the object thus found. If a name is not found, `ImportError` is raised. If the list of identifiers is replaced by a star (`*`), all names defined in the module are bound, except those beginning with an underscore(`_`).

Names bound by import statements may not occur in `global` statements in the same scope.

The `from` form with `*` may only occur in a module scope.

(The current implementation does not enforce the latter two restrictions, but programs should not abuse this freedom, as future implementations may enforce them or silently change the meaning of the program.)

6.11 The `global` statement

```
global_stmt:    "global" identifier ("," identifier)*
```

The `global` statement is a declaration which holds for the entire current scope. It means that the listed identifiers are to be interpreted as globals. While *using* global names is automatic if they are not defined in the local scope, *assigning* to global names would be impossible without `global`.

Names listed in a `global` statement must not be used in the same scope before that `global` statement is executed.

Names listed in a `global` statement must not be defined as formal parameters or in a `for` loop control target, `class` definition, function definition, or `import` statement.

(The current implementation does not enforce the latter two restrictions, but programs should not abuse this freedom, as future implementations may enforce them or silently change the meaning of the program.)

6.12 The `access` statement

```
access_stmt:    "access" ...
```

This statement will be used in the future to control access to instance and class variables. Currently its syntax and effects are undefined; however the keyword `access` is a reserved word for the parser.

6.13 The `exec` statement

```
exec_stmt:    "exec" expression ["in" expression ["," expression]]
```

This statement supports dynamic execution of Python code. The first expression should evaluate to either a string, an open file object, or a code object. If it is a string, the string is parsed as a suite of Python statements which is then executed (unless a syntax error occurs). If it is an open file, the file is parsed until EOF and executed. If it is a code object, it is simply executed.

In all cases, if the optional parts are omitted, the code is executed in the current scope. If only the first expression after `in` is specified, it should be a dictionary, which will be used for both the global and the local variables. If two expressions are given, both must be dictionaries and they are used for the global and local variables, respectively.

Note: dynamic evaluation of expressions is supported by the built-in function `eval`.

Chapter 7

Compound statements

Compound statements contain (groups of) other statements; they affect or control the execution of those other statements in some way. In general, compound statements span multiple lines, although in simple incarnations a whole compound statement may be contained in one line.

The `if`, `while` and `for` statements implement traditional control flow constructs. `try` specifies exception handlers and/or cleanup code for a group of statements. Function and class definitions are also syntactically compound statements.

Compound statements consist of one or more 'clauses'. A clause consists of a header and a 'suite'. The clause headers of a particular compound statement are all at the same indentation level. Each clause header begins with a uniquely identifying keyword and ends with a colon. A suite is a group of statements controlled by a clause. A suite can be one or more semicolon-separated simple statements on the same line as the header, following the header's colon, or it can be one or more indented statements on subsequent lines. Only the latter form of suite can contain nested compound statements; the following is illegal, mostly because it wouldn't be clear to which `if` clause a following `else` clause would belong:

```
if test1: if test2: print x
```

Also note that the semicolon binds tighter than the colon in this context, so that in the following example, either all or none of the `print` statements are executed:

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

Summarizing:

```
compound_stmt:  if_stmt | while_stmt | for_stmt
              | try_stmt | funcdef | classdef
suite:          stmt_list NEWLINE | NEWLINE INDENT statement+ DEDENT
statement:      stmt_list NEWLINE | compound_stmt
stmt_list:      simple_stmt (";" simple_stmt)* [";"]
```

Note that statements always end in a `NEWLINE` possibly followed by a `DEDENT`.

Also note that optional continuation clauses always begin with a keyword that cannot start a statement, thus there are no ambiguities (the 'dangling `else`' problem is solved in Python by requiring nested `if` statements to be indented).

The formatting of the grammar rules in the following sections places each clause on a separate line for clarity.

7.1 The `if` statement

The `if` statement is used for conditional execution:

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

It selects exactly one of the suites by evaluating the conditions one by one until one is found to be true (see section 5.9 for the definition of true and false); then that suite is executed (and no other part of the `if` statement is executed or evaluated). If all conditions are false, the suite of the `else` clause, if present, is executed.

7.2 The `while` statement

The `while` statement is used for repeated execution as long as a condition is true:

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

This repeatedly tests the condition and, if it is true, executes the first suite; if the condition is false (which may be the first time it is tested) the suite of the `else` clause, if present, is executed and the loop terminates.

A `break` statement executed in the first suite terminates the loop without executing the `else` clause's suite. A `continue` statement executed in the first suite skips the rest of the suite and goes back to testing the condition.

7.3 The `for` statement

The `for` statement is used to iterate over the elements of a sequence (string, tuple or list):

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

The condition list is evaluated once; it should yield a sequence. The suite is then executed once for each item in the sequence, in the order of ascending indices. Each item in turn is assigned to the target list using the standard rules for assignments, and then the suite is executed. When the items are exhausted (which is immediately when the sequence is empty), the suite in the `else` clause, if present, is executed, and the loop terminates.

A `break` statement executed in the first suite terminates the loop without executing the `else` clause's suite. A `continue` statement executed in the first suite skips the rest of the suite and continues with the next item, or with the `else` clause if there was no next item.

The suite may assign to the variable(s) in the target list; this does not affect the next item assigned to it.

The target list is not deleted when the loop is finished, but if the sequence is empty, it will not have been assigned to at all by the loop.

Hint: the built-in function `range()` returns a sequence of integers suitable to emulate the effect of Pascal's `for i := a to b do`; e.g. `range(3)` returns the list `[0, 1, 2]`.

Warning: There is a subtlety when the sequence is being modified by the loop (this can only occur for mutable sequences, i.e. lists). An internal counter is used to keep track of which item is used next, and this is incremented on each iteration. When this counter has reached the length of the sequence the loop terminates. This means that if the suite deletes the current (or a previous) item from the sequence, the next item will be skipped (since it gets the index of the current item which has already been treated). Likewise, if the suite inserts an item in the sequence before the current item, the current item will be treated again the next time through the loop. This can lead to nasty bugs that can be avoided by making a temporary copy using a slice of the whole sequence, e.g.

```
for x in a[:]:
    if x < 0: a.remove(x)
```

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" [condition ["," 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 cannot be mixed.

The `try...except` form specifies one or more exception handlers (the `except` clauses). When no exception occurs in the `try` clause, no exception handler is executed. When an exception occurs in the `try` suite, a search for an exception handler is started. This inspects the except clauses in turn until one is found that matches the exception. A condition-less except clause, if present, must be last; it matches any exception. For an except clause with a condition, that condition is evaluated, and the clause matches the exception if the resulting object is “compatible” with the exception. An object is compatible with an exception if it is either the object that identifies the exception or it is a tuple containing an item that is compatible with the exception. Note that the object identities must match, i.e. it must be the same object, not just an object with the same value.

If no except clause matches the exception, the search for an exception handler continues in the surrounding code and on the invocation stack.

If the evaluation of a condition in the header of an except clause raises an exception, the original search for a handler is cancelled and a search starts for the new exception in the surrounding code and on the call stack (it is treated as if the entire `try` statement raised the exception).

When a matching except clause is found, the exception's parameter is assigned to the target specified in that except clause, if present, and the except clause's suite is executed. When the end of this suite is reached, execution continues normally after the entire try statement. (This means that if two nested handlers exist for the same exception, and the exception occurs in the try clause of the inner handler, the outer handler will not handle the exception.)

Before an except clause's suite is executed, details about the exception are assigned to three variables in the `sys` module: `sys.exc_type` receives the object identifying the exception; `sys.exc_value` receives the exception's parameter; `sys.exc_traceback` receives a traceback object (see section 3.2) identifying the point in the program where the exception occurred.

The optional `else` clause is executed when no exception occurs in the `try` clause. Exceptions in the `else` clause are not handled by the preceding `except` clauses.

The `try...finally` form specifies a ‘cleanup’ handler. The `try` clause is executed. When no exception occurs, the `finally` clause is executed. When an exception occurs in the `try` clause, the exception is temporarily saved, the `finally` clause is executed, and then the saved exception is re-raised. If the `finally` clause raises another exception or executes a `return`, `break` or `continue` statement, the saved exception is lost.

When a `return` or `break` statement is executed in the `try` suite of a `try...finally` statement, the `finally` clause is also executed ‘on the way out’. A `continue` statement is illegal in the `try` clause. (The reason is a problem with the current implementation — this restriction may be lifted in the future).

7.5 Function definitions

A function definition defines a user-defined function object (see section 3.2):

```
funcdef:        "def" funcname "(" [parameter_list] ")" ":" suite
parameter_list: (defparameter ",")* ("*" identifier | defparameter [","])
defparameter:   parameter ["=" condition]
```

```
sublist:        parameter ("," parameter)* [","]
parameter:      identifier | "(" sublist ")"
funcname:       identifier
```

A function definition is an executable statement. Its execution binds the function name in the current local name space to a function object (a wrapper around the executable code for the function). This function object contains a reference to the current global name space as the global name space to be used when the function is called.

The function definition does not execute the function body; this gets executed only when the function is called.

When one or more top-level parameters have the form *parameter = condition*, the function is said to have "default parameter values". Default parameter values are evaluated when the function definition is executed. For a parameter with a default value, the correponding argument may be omitted from a call, in which case the parameter's default value is substituted. If a parameter has a default value, all following parameters must also have a default value — this is a syntactic restriction that is not expressed by the grammar.[1]

Function call semantics are described in section 5.3.4. When a user-defined function is called, first missing arguments for which a default value exists are supplied; then the arguments (a.k.a. actual parameters) are bound to the (formal) parameters, as follows:

- If there are no formal parameters, there must be no arguments.
- If the formal parameter list does not end in a star followed by an identifier, there must be exactly as many arguments as there are parameters in the formal parameter list (at the top level); the arguments are assigned to the formal parameters one by one. Note that the presence or absence of a trailing comma at the top level in either the formal or the actual parameter list makes no difference. The assignment to a formal parameter is performed as if the parameter occurs on the left hand side of an assignment statement whose right hand side's value is that of the argument.
- If the formal parameter list ends in a star followed by an identifier, preceded by zero or more comma-followed parameters, there must be at least as many arguments as there are parameters preceding the star. Call this number N. The first N arguments are assigned to the corresponding formal parameters in the way descibed above. A tuple containing the remaining arguments, if any, is then assigned to the identifier following the star. This variable will always be a tuple: if there are no extra arguments, its value is `()`, if there is just one extra argument, it is a singleton tuple.

Note that the 'variable length parameter list' feature only works at the top level of the parameter list; individual parameters use a model corresponding more closely to that of ordinary assignment. While the latter model is generally preferable, because of the greater type safety it offers (wrong-sized tuples aren't silently mistreated), variable length parameter lists are a sufficiently accepted practice in most programming languages that a compromise has been worked out. (And anyway, assignment has no equivalent for empty argument lists.)

[1] Currently this is not checked; instead, `def f(a=1,b)` is interpreted as `def f(a=1,b=None)`.

It is also possible to create anonymous functions (functions not bound to a name), for immediate use in expressions. This uses lambda forms, described in section 5.9.

7.6 Class definitions

A class definition defines a class object (see section 3.2):

```
classdef:       "class" classname [inheritance] ":" suite
inheritance:    "(" [condition_list] ")"
classname:      identifier
```

A class definition is an executable statement. It first evaluates the inheritance list, if present. Each item in the inheritance list should evaluate to a class object. The class's suite is then executed in a new execution frame (see section 4.1), using a newly created local name space and the original global name space. (Usually, the suite contains only function definitions.) When the class's suite finishes execution, its execution frame is discarded but its local name space is saved. A class object is then created using the inheritance list for the base classes and the saved local name space for the attribute dictionary. The class name is bound to this class object in the original local name space.

Chapter 8

Top-level components

The Python interpreter can get its input from a number of sources: from a script passed to it as standard input or as program argument, typed in interactively, from a module source file, etc. This chapter gives the syntax used in these cases.

8.1 Complete Python programs

While a language specification need not prescribe how the language interpreter is invoked, it is useful to have a notion of a complete Python program. A complete Python program is executed in a minimally initialized environment: all built-in and standard modules are available, but none have been initialized, except for `sys` (various system services), `__builtin__` (built-in functions, exceptions and `None`) and `__main__`. The latter is used to provide the local and global name space for execution of the complete program.

The syntax for a complete Python program is that for file input, described in the next section.

The interpreter may also be invoked in interactive mode; in this case, it does not read and execute a complete program but reads and executes one statement (possibly compound) at a time. The initial environment is identical to that of a complete program; each statement is executed in the name space of `__main__`.

Under UNIX, a complete program can be passed to the interpreter in three forms: with the **-c** *string* command line option, as a file passed as the first command line argument, or as standard input. If the file or standard input is a tty device, the interpreter enters interactive mode; otherwise, it executes the file as a complete program.

8.2 File input

All input read from non-interactive files has the same form:

```
file_input:     (NEWLINE | statement)*
```

This syntax is used in the following situations:

- when parsing a complete Python program (from a file or from a string);
- when parsing a module;
- when parsing a string passed to the `exec` statement;

8.3 Interactive input

Input in interactive mode is parsed using the following grammar:

```
interactive_input: [stmt_list] NEWLINE | compound_stmt NEWLINE
```

Note that a (top-level) compound statement must be followed by a blank line in interactive mode; this is needed to help the parser detect the end of the input.

8.4 Expression input

There are two forms of expression input. Both ignore leading whitespace.

The string argument to `eval()` must have the following form:

```
eval_input:      condition_list NEWLINE*
```

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

```
input_input:     condition_list NEWLINE
```

Note: to read 'raw' input line without interpretation, you can use the built-in function `raw_input()` or the `readline()` method of file objects.

Index