CS 164, Fall 2006The Pyth Programming Language (version of 2 Nov 2006)P. N. Hilfinger

Our project this year is a compiler for a dialect of the popular language PYTHON. PYTHON itself is usually categorized as a *scripting language,* meaning either that it is intended to implement extensions to your computer's command language, or that it is intended to implement "glue" programs that accomplish most of their work by invoking other self-contained programs.

You will often hear PYTHON described as an "interpreted language," in contrast to a "compiled language." To the limited extent this statement is meaningful, it is false. First, as you'll see in this course, the adjectives "interpreted" and "compiled" do not properly modify programming languages, but rather *implementations* of programming languages. Any language that can be implemented by an interpreter can be compiled, and vice-versa. There are C compilers, C interpreters, Lisp compilers, Lisp interpreters, Java compilers, and Java interpreters. There indeed are PYTHON interpreters, but this semester, we will implement a compiler for the PYTH dialect of PYTHON.

Although this document is self-contained, I think that any well-trained professional programmer would do well to be familiar with PYTHON itself, a useful language whose design is refreshingly clean—in marked contrast to other scripting languages, especially PERL. A reference document is available on-line (there is a link from the class homepage) or in book form (David M. Beazley, *Python Essential Reference, Third Edition,* New Riders Publishing, 2006, ISBN: 0672328623).

1 Lexical Structure

Typically, we factor the description a programming language's syntax into two parts: a *lexical description,* which defines a set of possible *tokens* or atomic symbols formed from the text of a program (called its *source code* or *source*), and a *grammar* that describes the possible properly formed constructs (or *sentences*), that may be created from sequences of such tokens. It is not essential to make this separation, but the practice has proven useful over the years, and there are tools and techniques for converting each of these descriptions into translation programs.

1.1 Comments, Whitespace, and Ends of Lines

The source code of a PYTH program consists of the tokens described below possibly interspersed with whitespace (blanks, tabs, formfeeds), blank lines, and comments. Notionally, the conversion of source into tokens proceeds from the beginning to the end of the source in sequence, at each point forming the longest token or separator allowed by the rules. For example, the empty program

```
# print 42
```

is interpreted as a blank line consisting of a single comment and the end of line, even though '`print`' and '`42`' can otherwise be interpreted as valid tokens. Likewise,

```
if42 = 31
```

is treated as three tokens '`if42`', '`=`', and '`31`' even though '`if`', '`42`', '`3`' and '`1`' could all be valid tokens. This interpretation is sometimes called the *maximal munch rule.* An implication of this is that whitespace between two tokens is necessary only when the two tokens could be interpreted differently if concatenated.

A comment consists of the hash character ('`#`', sometimes called an *octothorpe*) followed by all characters up to, but not including the end of the line. A blank line consists of any amount of whitespace[1], possibly followed by a comment, and the end of the line.

The end of a line is itself a token (or part of one); that is, in contrast to most programming languages, it is not generally treated as whitespace, but as a terminating character, like the semicolon in C or Java. There are three exceptions:

1. A backslash (\) immediately followed by the end of a line and not inside a comment or string is treated as a space, so that backslash acts as a continuation character.

2. Inside a pair of balanced bracketing tokens (`(` and `)`, `{` and `}`, or `[` and `]`), ends of lines are treated as spaces.

3. Inside triple-quoted string literals (§5.6), end-of-line characters are treated as themselves.

For example, the two statements

```
print 12, 13
print 12, \
13
```

are equivalent, as are

```
a = 3 * (b + c) - x[ i ]
a = 3 * (b
+ c - x[
i
]
```

1.2 Indentation

PYTH and PYTHON are unusual among programming languages in that their statement-bracketing construct uses *indentation*[2]. You are probably familiar with how other conventional programming languages group multiple statements together into what are effectively single statements with brackets such as Pascal's **begin**...**end** or C's and Java's `{...}`. In

[1] I'm speaking formally here, so when I say "any amount" I really mean any amount, including zero.

[2] I have it on good authority that this design was the choice of Mrs. Karin Dewar, the (non-programming) wife of Prof. Robert B. K. Dewar of NYU's Courant Institute, when she was called in to mediate (in her nightgown) by the designers of PYTHON's ancestor language, who could not agree on which of several alternative constructs to use.

PYTH, there are two special opening and closing brackets often called INDENT and DEDENT, which are supposed to balance each other, like `{...}` braces in Java. Any non-blank line that is indented by more than the last non-blank line (if any) is treated as if it started with an INDENT token. If a non-blank line is indented by less than the previous non-blank line, then it is treated as if it started with one DEDENT token for each unbalanced INDENT token on a preceding line with the same or more indentation. It is a syntax error if there is no preceding unbalanced INDENT on a line with the same indentation. So, for example,

```
if x:
    print 1       # Begins with INDENT
    if y:
        print 2   # Begins with INDENT
 # Blank line, ignored
    else:         # Begins with DEDENT
        print 3   # Begins with INDENT
print 4           # Begins with DEDENT DEDENT
```

On the other hand, this program is illegal:

```
if x:
    print 1
  print 2         # Error: inconsistent indentation
```

To avoid special cases at the beginning and end, the program is treated as if it started and ended with a completely unindented **pass** statement. For purposes of determining the amount a line is indented, a horizontal tab character is equivalent to the smallest non-zero number of spaces that causes the indentation of the current line so far to be a multiple of eight spaces. Thus, zero to seven spaces followed by a tab is equivalent to eight spaces.

The indentation of continuation lines is ignored (that is, it's treated as ordinary whitespace and not considered in inserting implicit INDENTs and DEDENTs). Thus

```
if x:
    print 1, \
  2
    x = (3 *
y) + 4
```

is perfectly legal.

1.3 Identifiers and Reserved Words

An *identifier* is a token that consists of letters, digits, and underscores (_), and does not start with a digit. Identifiers are mostly used as names of things. However, certain of them have special roles in the syntax, and may not be used as names. These *reserved words* are as follows:

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

Words marked with an asterisk are not used in PYTH, but are reserved because of their use in PYTHON.

1.4 Literals

A *literal* is any construct that stands for a value determined entirely by the text of that literal. For example, the numeral `42` stands for an integral value that can be determined from its two digits in the usual way.

1.4.1 Integer literals

PYTH's integer literals are essentially the same as those of Java:

- A sequence of digits that does not start with '0' and is in the range 0..2147483648 is a decimal numeral. The numeral 2147483648 represents the value -2147483648 (due to PYTH's Java-like modular arithmetic)[3].
- A sequence of digits that starts with '0' represents an octal numeral, and must be in the range $0..2^{32}-1$ ($0..37777777777_8$). Octal numerals greater than $2^{31}-1$ represent negative numbers as in Java (for example, 37777777777 represents -1, since $-1 \equiv 2^{32}-1 \bmod 2^{32}$.
- A token of the form `0x`H or `0X`H, where H is a sequence of one or more hexadecimal (base 16) digits (0–9, a–f, A–F), represents a hexadecimal numeral. H must be in the range $0..2^{32}-1$ ($0..\texttt{ffffffff}_{16}$). As for octal literals, numerals greater than $2^{31}-1$ represent negative numbers.

1.4.2 Floating-point literals

A floating-point literal is a decimal fraction or a numeral in modified scientific notation. In ordinary mathematical or engineering contexts, such literals would denote rational numbers; in PYTH, they represent rational *approximations* to these numbers. They are approximate, in general, because computer floating-point arithmetic generally uses binary fractions internally, and not all base-10 fractions can be exactly represented with finite base-2 fractions. For

[3] You might ask why I didn't simply make 2147483647 the largest decimal numeral, as in other languages, since that is the largest positive integer representable in PYTH. The reason is that such a rule makes it awkward to write the most negative number. In PYTH(as in Java), there are no negative numerals, just positive numerals that have the negation operator applied to them: '-4' is actually two tokens. With the PYTH rule, you may write the most negative number as -2147483648 if you want, since in modulo 2^32 arithmetic, 2147483648 is its own negative.

example, 0.1 in base 2 is the repeating numeral 0.00011001100110011. . . , and if floating-point numbers are limited to 52 bits of significance (a common number), then what you write as 0.1 is actually

$$0.09999999999999997779553950749686919152736663818359375,$$

which is close enough for most purposes.

Floating-point literals in PYTH come in any of the following forms (the following all denote the same number):

```
123.0
123.
1.23e2   # Means 1.23 × 10^2
1.23E2
0.123e3
.123e3
1230.0e-1
1230e-1
```

That is, either a sequence of digits containing one decimal point, or a sequence of digits containing at most one decimal point followed by an 'e' or 'E', an optional sign, and an integer numeral, which is always treated as decimal.

1.4.3 String literals

Strings in PYTH are sequences of bytes. Their literals are written as ASCII text surrounded by any of several different kinds of quotation marks:

Literal	Meaning
`"A 'string' in double quotes"`	`A 'string' in double quotes`
`'A "string" in single quotes'`	`A "string" in single quotes`
`"A \"string\" in double quotes"`	`A "string" in double quotes`
`'A \'string\' in single quotes'`	`A 'string' in single quotes`
`'''A 'string' in triple single quotes'''`	`A 'string' in triple single quotes`
`"""A "string" in triple double quotes"""`	`A "string" in triple double quotes`
`"A string\nthat contains a new line"`	`A string` `that contains a new line`
`'''A multi-line` `string in triple single quotes'''`	`A multi-line` `string in triple single quotes`
`"""A multi-line` `string in triple double quotes"""`	`A multi-line` `string in triple double quotes`

As these examples suggest, string literals generally represent the literal sequence of characters composing them, minus the matching beginning and ending quotes, which may be any of the symbols double quote, single quote, triple double quote, triple single quote. The exceptions are the following two-character *escape sequences,* each of which denotes a single character:

```
\a          Bell (Control-G, encoded as the byte value 7)
\b          Backspace (Control-H, byte value 8)
\t          Tab (Control-I, byte value 9)
\n          New line (Control-J, byte value 10)
\r          Return (Control-M, byte value 13)
\e          Escape (byte value 27)
\0          Null (byte value 0)
\v          Vertical tab (byte value 11)
\f          Form feed (Control-L, byte value 12)
\"          Double quotation mark
\'          Single quotation mark (apostrophe)
\\          Backslash
```

Inside single quotes, the double quote need not be escaped, and inside double quotes, the single quote need not be escaped. Triple-quoted strings may contain newlines, whereas their single-quoted counterparts may not (and have to use `\n` instead). All strings end at the first occurrence of an unescaped matching quote. Thus, `"""""x"".."""` means `""x""`. An *unrecognized escape sequence*—a backslash followed by any character other than those above—is unchanged, so `\q` represents the sequence `\q`.

A *raw string* is a string literal prefixed with 'r' or 'R'. In these strings, all backslash sequences are treated as unrecognized escape sequences. Thus, `r'\n'` represents the two-character string `\n`, and `r'''\''''` represents `\'` (there is no way to write a raw string literal that ends in a single backslash).

Finally, two adjacent string literals (separated, if at all, only by whitespace) represent the concatenation of the two strings. Thus

`"One" ' ' "String"`	*represents*	`One String`
`"exon""erate"`	*represents*	`exonerate`
`"Use " r'''\n''' ' for newline'`	*represents*	`Use \n for newline`

2 Statements

A *statement* is a language construct that "does something." This is as opposed to an *expression,* which is a language construct that denotes a value (or a computation that produces a value). Statements and expressions are closely related, and some expressions can double as statements (they are "evaluated for their side-effects" as opposed to their values). We often distinguish *simple statements* from *compound statements.* The latter contain other statements, so that their syntactic definition is therefore generally recursive. A *control statement* is a kind of statement that controls when and whether other statements are executed. A *declaration* is a kind of statement that defines new names.

A *statement,* per se, consists either of a *statement list*, a single compound statement, a single **import** statement (2.5), or a type declaration (4.2), A statement list is a sequence of one or more simple statements separated by semicolons, optionally followed by a semicolon, and then terminated by a newline. The simple statements are **pass** (§2.1.1), **print** statements

(§2.1.3), expression statements (§2.1.2), assignments (§2.1.4 and §2.1.5), **break** and **continue** statements (§2.3.2), and **return** statements (§2.4).

A *program* in PYTH consists of a sequence of zero or more statements.

2.1 Non-control statements

2.1.1 Pass

A **pass** statement does nothing. This probably does not sound useful, but one does occasionally need to write, for example, a statement that doesn't need to do anything in places where PYTH requires there to be a statement:

```
def close ():
    pass
```

or you just might think that it looks better to be explicit about cases in which nothing needs to happen:

```
if x < 0:
    pass
elif x < 10:
    y = f (x)
elif ...
```

2.1.2 Expression statements

Any non-empty expression list (see §5.4) may be used as a statement (as may an empty pair of parentheses, although as a statement, it is entirely useless). When used as a statement, its value is ignored, and the expressions are evaluated for their side-effects alone. So typically, you'll see function calls like this:

```
clear(theBoard)
```

2.1.3 Print

A **print** statement is a convenient way to output text. The simple form:

`print` $\texttt{expression}_1$`, ...`$\texttt{expression}_k$

prints external representations of its arguments, separated by spaces, on the standard output, followed by an end of line. If $k = 0$, therefore, as in:

```
print
```

the program simply starts a new line. The external representations used are those produced by the `str` method, which is defined on all types (§4.1).

Following the last expression with a comma:

`print` $\texttt{expression}_1$`, ...expression`k`,`

($k \geq 1$) does the same thing, but suppresses the end of line. For all but the first output on a line, furthermore, both forms of **print** first output a space. As a result,

```
print 1,
print 2,
print 3
```

prints

```
1 2 3
```

just like

```
print 1, 2, 3
```

You can direct output to a file other than the standard output using the `>>` operator:

```
f = open ("results.txt", "r")     # f points to a "file object"
print >> f, "The answer is", x
```

Again, in the absence of arguments, this just ends the current line in `f`, and with a trailing comma, it suppresses the end-of-line. The standard output is represented by a file called `sys.stdout`, so that the following are equivalent:

```
print 1, 2, 3
print >>sys.stdout, 1, 2, 3
```

The file `sys.stderr` represents the standard error output (conventionally used for error messages from the program).

2.1.4 Assignment

In its simplest form, an assignment looks like this:

variable `=` *expression*

The value of the expression is stored in the variable.

The right side of an assignment may be a list of expressions separated by commas (and possibly followed by one), which is just an abbreviation for a tuple (see §5.4):

```
x = 1, 2, 3        # short for x = (1, 2, 3)
y = 1,             # short for y = (1,)
```

cause `x` to reference the tuple `(1,2,3)` and `y` to reference the one-element tuple `(1,)`.

The left side of an assignment must be something assignable to (an *lvalue* in C/C++ parlance). This can mean a simple variable or parameter, as in the examples above. It can also be a dereferenced field of an object, as in

```
x = Account ()     # x now references an object of type Account
x.balance = 0
```

an indexed element of a list, as in

```
A = [ 1, 2, 3]
A[2] = 4           # The list now contains [1, 2, 4]
```

or a *slice* of a list (see also §5.3), as in

```
A = [ 1, 2, 3, 4 ]
A[1:3] = [ 5, 6 ]  # The list now contains [ 1, 5, 6, 4 ]
A[3:4] = [ ]       # The list now contains [ 1, 5, 6 ]
A[0:0] = (-1, 0)   # The list now contains [ -1, 0, 1, 5, 6 ]
A[3:] = [9]        # Same as A[3:5] = [9]; the list now
                   # contains [ -1, 0, 1, 9 ]
```

Actually, these last cases are themselves a shorthand ("syntactic sugar")[4]:

Assignment	**Meaning**
`A[1] = E`	`__setitem__(A,1,E)`
`A[3:4] = S`	`__setslice__(A, 3, 4, S)`

Things get interesting when the *left* side of an assignment is a tuple of assignable entities. In this case, the value on the right side must itself reference a value that can be treated as a sequence (tuple, list, dictionary) with the same number of elements. A simple example:

```
(x, y, z) = 1, 2, 3   # Equivalent to (x, y, z) = (1, 2, 3)
x, y, z = 1, 2, 3     # Same thing
```

which assigns 1, 2, and 3, respectively to `x`, `y`, and `z`. The right side of the assignment need not be an explicit sequence, as long as its value is a sequence at execution time. For example,

```
q = (1, 2, 3)
x, y, z = q
```

has the same effect on `x`, `y`, and `z` as the previous assignments. To distinguish a one-element sequence on the left side from a simple assignment, use a trailing comma:

```
x, = (1,)
```

assigns the integer 1 to `x`, while

```
x = (1,)
```

assigns the one-element sequence containing the single element 1 to `x`. (In general, one may always have a trailing comma after a list of things forming some kind of sequence *display,* but it only makes a difference in the one-element case.)

The left-side sequences may be nested, as in

```
(x, (y, z[3])) = q
```

which requires that `q` contain two elements, the second of which is a sequence containing two items.

[4] *Syntactic sugar* refers to built-in language syntax that is equivalent to some less convenient or less familiar form. Thus, we say that `A[1]` "is syntactic sugar" for, or "is the sugared version of," or "is sugaring of" `__get__(A,1)`. This usage has led some to use the term "syntactic alum" for distasteful attempts at syntactic sugaring or (from Brian Reid) "syntactic ketchup" for unnecessary sugaring.

2.1.5 Augmented assignments

As in Java, the operators `+=`, `-=`, etc., have the effect of applying an operator to an assignable entity (e.g., variable, array element) and then assigning the result back into the same entity. For example,

```
x += f(x)
```

is equivalent to

```
x = x + f(x)
```

However, the identity of the entity that is assigned to is calculated just once so as to be the same place from which the prior value is fetched. Thus, in

```
a[g()] += f(x)
```

`g` will be called only once, as if we had written:

```
tmp1 = a
tmp2 = g()
tmp1[tmp] = tmp2[tmp] + f(x)
```

where `tmp1` and `tmp2` are variables that are only used here. As a result, even if `g` has a side effect of changing the value of `a` or of causing the next call to `g` to return a different value, only one element of the array will be involved.

In fact, all of the augmented assignments translate to the application of certain standard methods. For example, when you write

```
x += f(x)
```

you essentially get

```
x = __iadd__(x, f(x))
```

Table 1 gives all the operation names for augmented assignments.

2.2 Grouping statements

Compound statements by definition contain smaller constituent statements. Syntactically, each of these constituents is a statement, but one often needs the effect of a sequence of statements, rather than just one. Typical programming languages provide some kind grouping (or *block*) construct for this purpose. In PYTH, this construct is called a *suite,* which comes in two flavors. The first form of suite is the *statement list.* For example:

```
if h > 0: dx = x/h; dy = y/h
else: print "singularity"
```

The second form consists of an end-of-line followed by an INDENT (see §1.2), one or more statements, each terminated by an end-of-line, and a matching DEDENT. For example:

Table 1: Special method names for augmented assignment

Operation	**Equivalent method call**
`x += y`	`x = __iadd__(x,y)`
`x -= y`	`x = __isub__(x,y)`
`x *= y`	`x = __imul__(x,y)`
`x /= y`	`x = __idiv__(x,y)`
`x %= y`	`x = __imod__(x,y)`
`x **= y`	`x = __ipow__(x,y)`
`x &= y`	`x = __iand__(x,y)`
`x |= y`	`x = __ior__(x,y)`
`x ^= y`	`x = __ixor__(x,y)`
`x <<= y`	`x = __ilshift__(x,y)`
`x >>= y`	`x = __irshift__(x,y)`

```
if h > 0:
    dx = x/h
    dy = y/h
else:
    print "singularity"
```

In a statement like this, with two suites, there is no need to use the same form for both; for example:

```
if h > 0:
    dx = x/h
    dy = y/h
else: print "singularity"
```

2.3 Control statements

2.3.1 If statements

This statement executes exactly one of several possible statements depending on the values of some condition tests. The general form is

`if` *expression*$_1$`:` *suite*$_1$
`elif` *expression*$_2$`:` *suite*$_2$
`...`
`elif` *expression*$_k$`:` *suite*$_k$
`else:` *suite*$_{k+1}$

where $k \geq 1$ (so that there need not be any **elif** clauses) and the **else** clause is optional. A missing **else** clause is equivalent to

```
else: pass
```

Each $expression_i$ is evaluated in order. The suite corresponding to the first expression that yields a true value (see §5.13) is executed, after which the whole **if** statement is finished. If no expression yields a true value, the **else** clause is executed.

2.3.2 While, break, and continue statements

The **while** statement executes a body repeatedly, under control of an iteration condition:

```
while expression: suite1
else: suite2
```

This repeatedly evaluates *expression* and executes $suite_1$ as long as the expression yields a true value (see §5.13). When the expression yields a false value, $suite_2$ is executed. The **else** clause is optional, and when missing defaults to **else: pass**.

The control statement

```
break
```

terminates execution of the nearest enclosing loop that contains the **break** in its loop body ($suite_1$ for **while** loops) and that is also contained inside the same class or function definitions (if any) that immediately encloses the **break** statement. It is an error for **break** to appear anywhere else. For example, these **break**s are illegal:

```
def f(n):
    if c > 0: break          # ERROR: not in a loop
    while True:
       def test (x):
           if x >= 0: break  # ERROR: 'def test' intervenes
       g(c)
       ...
```

Execution of **break** skips the **else** clause of the loop it terminates. For example,

```
n = 0
while n < len (A):
    if A[n] == X: break
else:
    return None
print n
```

prints the index at which `X` occurs in `A`, or returns a null value from whatever function contains this code.

The control statement

```
continue
```

may only occur where **break** may occur. Its effect is to skip to the end of the current iteration of the loop body. For example,

```
while i < N-1:
    i += 1
    if A[i] != 0:
        continue
    print "Computing A[%d]:" % (i),
    A[i] = f(i)
```

skips the print and assignment statements if `A[i]` is not 0.

The **break** and **continue** statements are also valid inside **for** loops, described next.

2.3.3 For loops

The **for** loop is simply a shorthand for a particular kind of **while** loop[5]. You write

`for` *left-hand side* `in` *expression list*`:` *suite*$_1$
`else:` *suite*$_2$

where *left-hand side* is anything you can put on the left of an assignment statement (see §2.1.4). This is functionally equivalent to the following:

```
LST = (expression list)
I = 0
while I+1 < len (LST):
    left-hand side = __getindex__ (LST, I)
    I += 1
    suite1  # (but indented appropriately)
else: suite2
```

where `LST` and `I` are replaced by some new variables that are not used anywhere else in the program. As usual, the **else** clause may be omitted, in which case *suite*$_2$ defaults to **pass**.

For example:

```
A = [ 1, 2, 3 ]  # An array
for i in A:
  print i
```

The `__getindex__` method is defined for built-in sequence types and programmers can define this method for new classes as well (just as Java programmers can define classes that implement `java.util.Iterable` and work with Java's **for** loop).

PYTH has a built-in type `xrange` that allows you to write loops over numbers:

```
for i in xrange (0,3): print i,
```

prints '0 1 2'. In effect, the value of `xrange(`L,U`)` is the sequence of integers, j, such that $L \leq j < U$.

[5] PYTH's **for** statement differs significantly from PYTHON's, which makes use of generators and exceptions.

2.4 Return statements

The statement

`return` *optional_expression_list*

must be a part of a function body. The *expression_list* defaults to `None` if not specified. It is evaluated and this value is returned as the value of the current call to the innermost enclosing function.

2.5 Import statements

The statement

`import` *identifier*

causes the contents of a file called *identifier*`.py` to be substituted for the statement, if that file has not previously been imported. The statement has no effect if *identifier*`.py` has already been imported. It may only occur at the outer level of the program and not within a suite, including those of class declarations or function declarations. The compiler searches for this file in the directory containing the program's source, plus additional directories in a standard list.

At the end of a *indentifier*`.py`, all open INDENT brackets are implicitly closed with DEDENTS, as if the file were followed by an unindented **pass** statement.

3 Scoping and Declarations

In the study of programming languages, a *declaration* is something that introduces a meaning or *binding* for an identifier. See §6 for information about declaring classes. This section discusses the other kinds of declaration.

3.1 Scopes and declarative regions

The *scope* of a declaration is the segment of program text or execution in which that declaration's binding is the one that defines the identifier. PYTH uses *static scoping,* which means that the scope of a declaration is fixed, and does not depend on what statements in the program get executed. A *declarative region* is a section of text that confines the scope of the declarations within it.

For example, in the Java declaration

```
class A {                 // 0
   private int x;         // 1
   void f (int x) {       // 2
      if (x > 0) {        // 3
         int y = ...;     // 4
         f (y);           // 5
```

```
        } else {            // 6
           String y = ...;  // 7
           g (y);           // 8
        }
     }                      // 9
     void g () { ... }      // 10
}                           // 11
```

there are declarative regions for the body of `A` (lines 1–10), the body of `f` (lines 2–8), the block that forms the 'then' part of the **if** statement (lines 4–5), and the block that forms the **else** clause (lines 7–8). The scope of the declaration of `y` at 4 is lines 4–5 (beginning at the declaration and ending with the declarative region that contains that declaration). The scope of the declaration of `y` at 7 is lines 7–8. The scopes of the declaration of `f` and `g` are line 1–10 of `A`, plus certain locations elsewhere following a '.' (as in `q.f (3)`). The scope of the declaration of `x` at 1 is lines 1 and 9–10; the "hole" in the middle is caused by the fact that the declaration of `x` at 2 *hides* or *shadows* the declaration at 1.

In Pyth, the declarative regions are the program as a whole, plus the body of each class and the parameter list and body (together) of each function definition. In general, any declaration's scope extends throughout the entire declarative region that immediately encloses it. For example, in

```
def f (x, y):
    while x > 0:
       if y == 0:
           print z      // A
       else:
           z = x+1      // B
           x = g(z)     // C
           y -= 1

def g (y): ...          // D
```

the local variable `z` is created by the assignment statement B and is the same variable referred to in A. The function `g` referred to at C is the same one defined at D.

No name may be given two declarations immediately within the same declarative region. Local variable declarations, parameter declarations, and **defs** may be hidden in nested declarative regions. For example, with

```
def f():
    x = 3
    def g (x):
       print x
    g(12)
```

a call to `f` prints 12.

Names of types, predefined or otherwise, may not be redeclared anywhere in any declarative region.

3.2 Local variables

Local variables are declared in a given declarative region if there are any assignments to them (possibly implicit) and they are not formal parameters of the immediately enclosing function. Multiple assignments in the same declarative region count as one declaration. For example, in the following PYTH program, `x` is local to the body of `f`, `y` is a formal parameter (the assignment to it does not create a new variable), and `SIZE` is "local" to the program as a whole:

```
def f (y):
    y = g(y)
    x = y+12
    print y
SIZE = 13
```

The declaration, in other words, is implicit. The scope of a local declaration of `x` includes the entire body of the declarative region containing it. Thus, when a variable is not assigned to in a particular declarative region, its definition is "inherited" (although not in the object-oriented sense) from the enclosing context. For example,

```
def f ():
    x = 12
    y = 3
    def g ():
        y = 7
        print x, y,
    g()
    print x, y
f()
```

prints `12 7 12 3`.

Sometimes, however, you mean for an assignment within a function body to refer to an outer variable. The **global** declaration allows you to do so, to a limited extent:

```
x = 12
def tryToSetIt (y):
    x = y
def reallySetIt (y):
    global x
    x = y
tryToSetIt (42)
print x         # Prints 12
reallySetIt (42)
print x         # Prints 42
```

The identifiers (there may be a comma-separated list) in a **global** declaration must be defined at the outer level of the program (i.e., outside any **def**'s or class declarations). The scope

of the **global** declaration is the entire function that contains it, not including any nested function definitions. A **global** declaration at the outer level is rather useless, but it does still require that its variables be defined somewhere at the outer level.

It follows that a function can only set its own local variables or those at the outer (global) level. If it is nested inside another function, it cannot set that outer function's variables. The reason is simple: either the variable it assigns to is declared **global**, in which case it refers to an outer-level variable, or else is declared local by the assignment itself.

The scope of a local-variable declaration is the entire declarative region in which it is assigned to at least once. Hence, it is possible to reference the variable before its assignment. Prior to assignment, its value is `None`[6].

3.3 Declaring constants

The declaration

`def` *name* `=` *expression*

is a constant declaration that defines *name* to denote the value of the *expression*. The expressions in constant declarations are evaluated in the usual execution order, as if each declaration were an ordinary statement. The scope of the name, however, will typically start before that, in accordance with the usual scoping rules. Prior to executing the declaration, the value of *name* is `None`. One cannot assign to anything defined with **def**.

3.4 Declaring functions and formal parameters

Functions may be declared with *function declarations,* which are specialized **def** statements, as in

```
def f (x):
   return x+1
```

The identifiers listed in parentheses are the formal parameters of the function being defined. Their scope is the body of the function.

Functions declared this way immediately inside a class body are *instance methods,* which means that they get called in a special way (see 5.10). Preceding the declaration with the keyword **class**, as in

```
class def g ():
   return 42
```

defines a *class method* (or *static method*), which is basically just an ordinary function. (See §6 for the significance of **class def**, which is used inside classes only.)

It is also possible to define variables or ordinary constants whose values are functions, to pass functions as parameters, return them as values or store them in other data structures. For example:

[6]This treatment is different from Pyth's mostly to avoid introducing still another kind of value—the "undefined value."

```
def f (x,y): ...
def synonymForF = f
functionVar = f
typedFunctionVar : (All, All) -> All
typedFunctionVar = f
listOfFunctions = [synonymForF]
```

3.5 Importing foreign functions

PYTH programs can define functions implemented in "foreign" languages (generally C), by using a special **import** statement as the sole constituent of their body[7]:

```
def Pyth name (parameter list):
    import "foreign name"
```

The foreign-name string must be a name recognizable to the linker. Calls to function *Pyth name* will call the given foreign function, passing the indicated parameters. The precise calling conventions are implementation-dependent.

4 Values and types

As for most programming languages, every value manipulated in PYTH has a type. These types are part of a hierarchy (as in Java), and each has a written denotation. The predefined types are described in Table 2.

A type may be a *subtype* of another; the inverse relation is *supertype*. The subtype relation is reflexive (each type is a subtype of itself) transitive (all my supertypes are supertypes of my subtypes as well), and antisymmetric (if T is a subtype of T' and vice-verse, then $T = T'$). All types are subtypes of `Any`, and `Void` is a subtype of all types (so that the "hierarchy," properly speaking, is what is known as a *lattice*).

All class types (see §6) are subtypes of their declared parent types, and of `Object`.

A function type `(`$T_1, \ldots, T_n$`)->`T_0 is only a subtype of `Any` and of any other function type `(`$T'_1, \ldots, T'_n$`)->`T'_0 where T_0 is a subtype of T'_0 and for every i, T'_i is a subtype of T_i (yes, that's right: it's reversed for the arguments. We say that subtyping in *covariant* in the return type and *contravariant* in the argument types).

The other predefined types in Table 2 have no other subtypes or supertypes.

Values in PYTH, as in Java, are generally *references* to objects. These objects are either *immutable*—meaning that once created, their contents (state) may not be changed—or *mutable.* For example, since list objects are mutable and tuple objects are not,

```
L = [1,2,3]
Q = L
L[1] = 0   # Legal, L and Q now both reference the same list
           # object with contents [1,0,3]
```

[7] Actually, most of PYTH's predefined functions are themselves "foreign" code and use these special imports.

Table 2: Predefined types in PYTH

Name	**Description**
Any	The supertype of all types. Every value is an Any.
Void	The type of the null object, None.
Object	A supertype of all user-defined classes.
Int	Signed, 32-bit integers in the range -2^{31} to $2^{31}-1$.
Float	Double-precision floating-point numbers.
Bool	Logical values (True or False).
String	Character strings.
Tuple	Immutable tuples (e.g., `(1, 2, 3)`).
Xrange	Results of the xrange function (ranges of Ints).
List	Mutable (modifiable) sequences (e.g., `[1,2,3]`).
Dict	Mapping (e.g., `{1: 'A', 2: 'B'}`).
File	External files.
$(T_1,\ldots,T_k)$`->`T_0	The type of all functions that take $k \geq 0$ arguments of types $T_1,\ldots,T_k$ and returns a value of type T_0. Notationally, `->`T_0 defaults to `->Void` if omitted.

```
T = (1, 2, 3)
T[1] = 0   # Runtime error.
```

However, the sequence of assignments

```
T = (1, 2)
R = T
T = (3, 4) # Legal.  R now references a tuple object with contents
           # (1,2) and T references one with contents (3,4)
```

Again, this is because, as in Java, the variable `T` contains a reference to these tuple objects. The second assignment to `T` does *not* change the first tuple's state; it merely changes *which* tuple `T` is pointing to.

By their nature, immutable objects have an interesting property: one can pretend that their variables do *not* contain references to objects, and instead contain the objects themselves. After all, as the examples above show, after `T` is assigned to `R`, there is nothing you can do `T` that changes the contents of the tuple pointed to by `R`, just as in ordinary Java, once you assign one integer variable to another, any operations on the first variable have no effect on the second. In typical PYTH implementations, we take advantage of this fact, so that `Int`- and `Float`-valued variables, for example, do not contain pointers, and doing arithmetic does not require allocating new object storage.

4.1 The types Any and Void

The special value `None` belongs to all types. Its type is `Void`, the subtype of all types. In effect, this type inherits all instance methods, including ones in new classes you define, provides implementations for a few, and "implements" all the rest to cause errors.

All values belong to type `Any` (it is a supertype of all types), but one cannot create a value whose dynamic type is `Any`. The methods defined for `Any` include all the predefined methods, but with certain exceptions, shown in Table 3, the default implementations of all these methods is to cause an error[8].

4.2 Declaring types

A type declaration has the form

name `:` *type*

It applies to the declaration of *name* (variable, constant, function) that is immediately within the current declarative region. The given *type* becomes the static type of *name* wherever the

[8]This is in marked contrast to Java, where, for example, the predefined String methods are defined only on Strings, and will give compile-time errors when attempted on things whose static type is Object. As a result of PYTH's rule, many errors that would be caught at compile-time in Java are only caught during execution in PYTH. The reason for this is to increase the resemblance between PYTHand its parent language PYTHON, which is entirely dynamically typed. New method names defined by the user in PYTH, by contrast, are *not* defined in class `Any`, and so (unlike PYTHON, but like Java), you need type declarations to inform the compiler that these new method exist in a particular value.

declaration applies. If *name* is a variable or field, only values that have a subtype of the given type may be assigned to *name.* The types of the formal parameters of a function may be declared outside the function like this:

```
augment : (Int, Int) -> Int
def augment (a, b): ...
```

which causes `a` and `b` to have static type `Int`.

In the absence of an explicit type declaration, functions declared with function declarations (see §3.4):

```
def f(a_1, ..., a_n): ...
```

have type `(Any,...,Any)->Any` for non-instance methods and `(`T`,Any,...,Any)->Any` for instance methods, where T is the enclosing class name (see §6) In either case, there are n argument types. All other declarations, by default, ascribe a static type of `Any` to the declared entity.

4.3 Static type consistency

The PYTHON language is *dynamically typed,* meaning that at execution time, each value carries with it information that identifies its type. It is possible for a program to specify an operation on a certain variable that turns out to be meaningless when actually executed, because the particular value of that variable happens to have the wrong type at the time. Java, by contrast, is *statically typed,* meaning that at translation time, and apart from any execution, the compiler has enough information to know whether there might be type errors upon execution, and rejects programs where this is the case. PYTH is a sort of compromise. It is statically typed, but the default implementation of many operations is to raise an error. This allows many PYTHON programs to run pretty much unchanged in PYTH, but at the time, it weakens the compile-time error detection that PYTH can do relative to languages such as Java or C++.

PYTH's dynamic type rule (enforced at execution time, if necessary) is basically that the actual value assigned to any variable, attribute, parameter, or constant, or returned from any function must have a subtype of the static type declared for that variable, attribute, etc.

Its static type rule (enforced by the compiler) is that the static type, T_v, of the expression assigned to any variable, attribute, parameter, or constant, or returned from any function and the static type, T_d, declared for that variable, attribute, etc., must be *compatible.* We say that types T_v and T_d are compatible if either one of them is a subtype of the other—in other words, if a value of type T_v *might* have a type that "fits" in a T_d. (This is where PYTH differs from Java, which requires that T_v be a subtype of T_d.) For example,

```
x : List       # x has static type List.
...
y = x          # y has (by default) static type All, so this is OK.
x = y          # This is also OK, because y MIGHT contain a List
x = 3          # Error: Int not a subtype of List, nor vice-versa
```

5 Expressions

Expressions can be *evaluated* so as to *yield* values. In the process of these evaluations, they may also have various *side effects* on program variables and perhaps things external to the program (such as the contents of your display). In §1.4 we saw *literals,* the simplest expressions, denoting values of type `Int`, `Float`, and `Str`.

5.1 Operator expressions

Table 4 lists the operators and the equivalent calls grouped by decreasing *operator precedence.* The precedence of an operator determines how to implicitly parenthesize certain combinations of operators[9]. Because `*` has higher precedence than `+`, for example, the expression '`x+y*z`' means '`x+(y*z)`', and not '`(x+y)*z`'. With one exception, all operators are *left associative:* that is, in cases of ambiguity in an expression involving two operators of equal precedence, PYTH usually groups left to right, so that '`x-y-z`' is interpreted as '`(x-y)-z`'. The exception is `**`; that is, '`x**y**z`' is interpreted as '`x**(y**z)`'.

As indicated in Table 4, most PYTH operators are actually syntactic sugar for certain standard function calls, also given in Table 4. For example, the expression `-x*y**3` is exactly equivalent to

```
__mul__(__neg__(x), __pow__(y,3))
```

As a result, you can extend the definitions of operators to new classes that you define by simply defining functions in those classes with names like `__add__`. Unfortunately, PYTH shares a common problem with this technique: while you can make `myFoo+3` mean whatever you want, where `myFoo` contains an object of a class you define, you can't make `3+myFoo` mean what you want.

5.2 Operations on numbers

The types `Int` and `Float` correspond to the Java types `int` and `double`. Table 5 describes the available operators. Their operands must be numbers, except as noted, although violations of this rule will not in general be discovered until the program is executed. The results of these operations, unless otherwise noted, have the same type as the operands. If one operand is `Int` and the other `Float`, the integer operand is first converted to the nearest corresponding `Float` value. Integer arithmetic is performed modulo 2^{32}, as in Java.

5.3 Sequences

The built-in sequence classes—`Tuple`, `List`, `String`, and `Xrange`—implement operations for fetching members of a set or sequence of items by an integer index (0-based). These classes

[9]Many authors (including, alas, David Beazley, author of the *Python Essential Reference*) mischaracterize operator precedence as determining "order of evaluation," saying things like "`*` is evaluated before `+`." But in the example `x+y*z`, it is the *operands* of `*` and `+` that differ—you add or multiply different things in the two different groupings. In fact, the order-of-evaluation description is simply wrong in the case of complex expressions. For example, in `x*y+u*v+s*t`, it is simply not the case that programming languages require `s*t` to be evaluated before `x*y+u*v`.

all implement the operations summarized in sugared form in Table 6 (see Table 4 for the unsugared, function-call equivalents).

5.4 Tuples

Tuples are immutable sequences of zero or more component values. Their components can be of any combination of types. They are created by *expression lists* that are either empty and surrounded in parentheses or that contain at least one comma. An expression list is a list of zero or more expressions that are each followed by a comma, with the possible exception of the last. To represent a tuple, the list must be surrounded in parentheses if there are zero expressions in it or it appears in some larger expression.

```
()
1, 2, 3
1, 2, 3,
1,
```

are expression lists that evaluate to tuples. In the last example, the trailing comma is necessary to indicate a value consisting of a one-element tuple as opposed to the simple integer 1. Expression lists that are part of list displays (see §5.5 below) or argument lists of function calls don't count as tuples. An expression list consisting of a single expression with no trailing comma simply yields the value of that expression. When necessary, you can surround a tuple in parentheses to avoid ambiguity:

```
f(1,2,3)    # Call function f with three integer-valued arguments
g((1,2,3))  # Call function g with one tuple-valued argument
```

5.5 Lists

Lists are mutable sequences. Like tuples, their components can be of any types. Unlike tuples, the values of individual components may be changed and the length of a list object may change. A *list display* creates a new list. It has the form

```
[ optional expression list ]
```

Because lists are mutable, they support various operations to modify either their length or contents, as detailed in Table 7.

One can assign to individual elements or slices, as indicated in §2.1.4:

```
L = [1,2,3]
Q = L
L[1] = 4           # Q == L == [1,4,3]
L[-1] = 0          # Q == L == [1,4,0]
L[1:3] = (11, 12)  # Q == L == [1,11,12]
L[0:2] = [42]      # Q == L == [42,12]
L[1:1] = [19,20]   # Q == L == [42,19,20,12]
```

Again, these assignments are equivalent to calls on `__setitem__` and `__setslice__`.

5.6 Strings

Strings are immutable sequences of characters. PYTH doesn't actually have a distinct type "character." Instead, characters are themselves one-element strings! Thus,

```
"abc"[1] == "abc"[1][0] == "abc"[1][0][0][0] == "b"
```

Table 8 shows the extra operations defined on strings, beyond the general sequence operations already given in Table 6.

5.7 Xranges

Xranges are immutable sequences of consecutive integers. They are created with the `xrange` function—`xrange`(i, j) is the sequence of all integers $\geq i$ and $< j$, so that `xrange(1,10)` is the sequence of integers from 1 to 9. As a result,

```
for i in xrange(0, N):
    doSomething(i)
```

is the PYTHy way of writing

```
for (i = 0; i < N; i += 1) {
    doSomething(i);
}
```

5.8 Dicts

Dicts are dictionaries—mutable mappings from values of (almost) any types to values of any types. Their operations look similar to those of sequences, with some subtle differences. To create a Dict, use a *dictionary display:*

`{` K_1 `:` V_1`, ...,` K_n `:` V_n `}`

where $n \geq 0$, returns a new Dict object whose `__getitem__` operation returns the value of V_i whenever given a *key* of K_i, and causes an error if given a key that is not one of the K_i. The assignment (`__setitem__`) method replaces or sets the value returned by `__getitem__`. For example,

```
d = { 1 : 10, "a" : 20, 3 : 40 }
print d[1]    # Prints 10
print d['a']  # Prints 20
#print d[0]   # ERROR
d[1] = "q"
d[0] = 42
print d[1]    # Prints q
print d[0]    # Prints 42
```

The keys used may be of any type that implements the `__hash__` operation, which is essentially the same as Java's `hashCode` method. This operation is defined for all types, but causes an "unimplemented" error when applied to mutable predefined objects (lists or dictionaries) or immutable objects that contain them.

A **for** loop over a Dict produces its currently defined keys. For example,

```
myDict = { 14:2, "a": 3, True: (1, 7) }
for x in myDict:
    print x, "maps to", myDict[x]
```

will print

```
14 maps to 2
a maps to 3
True maps to (1, 7)
```

(but not necessarily in that order).

Table 9 list other operations on Dicts.

5.9 Selection

The dot (.) operator indicates *selection* of a named entity defined in some class or object.

If T is the name of a type (including a class), then $T.a$ refers to the definition of a in T. This declaration of a will either be predefined, an attribute or **def**ed constant in the definition of T (when T is a user-defined class), or a definition inherited by T.

Otherwise, a selection has the form $E.a$ and E is an expression (i.e., something with a value). In this case, the search for a definition is essentially the same as that for $T.a$, where T is the static type of E. When the definition found is an instance variable, then the selection refers to the instance of that variable in the object referenced by the value of E. It may be assigned to or fetched from:

```
class AClass:
   instanceVar = 3      # Declares an instance variable
   ...
x : AClass
x = AClass()            # Create a new instance
x.instanceVar = 13
print x.instanceVar     # Prints 13
```

In this case (a is an instance variable), it is a runtime error for E to yield the value `None`.

If a refers to an instance method, the selection is illegal (a compile-time error) except when used as the target method in a method call.

When a does not refer to an instance variable, the selection behaves like $T.a$. When the selection is then used as the function in a function call (e.g., `E.a(3)`, there is additional special treatment, described in §5.10.

5.10 Function calls

Function calls have the familiar prefix syntax: $f(a_1, \ldots, a_n)$, for $n \geq 0$. Here f must be a function-valued expression. The grammar and precedence rules (§5.1) imply that f can only be an identifier, selection, function call (as in `incr(f)(3)`), array index or slice (as in g[3](4)), or bracketed expression of some kind (parenthesized expression, display, or string literal). The requirement that f be a function rules out displays and strings.

More specifically, the type of f in $f(a_1, \ldots, a_n)$ must be $(T_1, \ldots, T_n) -> T_0$, and each T_i must be a subtype of a_i. The type of the returned value must be a subtype of T_0.

There are two important special cases.

A. When f is a simple identifier (i.e, no dots), there is no definition of f in scope, and $n \geq 1$, the function that is called is `(`a_1`).`f. So, if `foo` is not otherwise defined, `foo(x)` becomes `((x).foo)(x)`.

B. When f is a selection of the form $E.g$, E is a value (not a type), and g is defined as an instance method (see §3.4), a copy of the value of E is inserted as the first parameter of g. That is, $E.g(a_1, \ldots, a_n)$ becomes in effect $(E.g)(E, a_1, \ldots, a_n)$, where E is evaluated only once.

It's probably not immediately obvious, but these make PYTH instance methods behave a lot like Java's. In Java, you might write `x.f(y)`, where instance method `f` is defined

```
void f (int p) { ... }
```

This calls `f` and passes two parameters: the value of `y` becomes the value of the explicit formal parameter `p`, and the value of `x` becomes the value of the implicit parameter `this`. In PYTH, the implicit parameter is explicit, but otherwise the effect is the same: the call is the same as for Java, and the definition of `f` might be

```
def f (self, p): ...
```

On the other hand, rule A above is a simply a convenience that allows us to write `len(myString)` and have it mean `myString.len ()`.

5.11 Restrictions on function values

A function value is *nested* if the function declaration that creates it is itself within the body of another function declaration. For reasons we'll get into (but you are welcome to figure out for yourselves), there are certain restrictions on nested function values that can only be enforced at execution time. Specifically, nested function values may not be returned from functions, nor may they be stored in instance variables, class variables, global variables (not nested in any function declarations), tuples, lists, or dicts.

Functions defined at the outer level (outside any class or enclosing function declaration) and class methods are therefore not subject to this restriction. Instance methods, however, are subject to another, harsher, restriction: an instance method may be selected from a value only as the function value in a method call (§5.10).

For example,

```
glob = None
def f (x):
    global globalVar

    def g ():
       return x+1

    callIt (g)           # OK: passed as parameter
    tmp = g              # OK: assigned to local variable
    tmp = [ g ]          # RUNTIME ERROR: stored in list
    glob = g             # RUNTIME ERROR: glob is global
    glob = f             # OK: f not nested
    AClass.aVar = g      # RUNTIME ERROR: assigned to class variable
    inst : AClass
    inst = AClass ()
    tmp = inst.meth      # ERROR: selecting a method without calling
    inst.aVar = g        # ERROR: assigning to instance variable.
    return g             # RUNTIME ERROR: returned
class AClass (Object):
    aVar = None
    def meth (): ...
```

5.12 Comparisons

The comparison operators—`<`, `>`, `<=`, `>=`, `==`, `!=`—are special in that they evaluate their operands in a unique way. In PYTH, you can write

```
if 0 <= f(x) < 10: ...
```

and it will mean what you expect: the value of `f(x)` is between 0 and 9, inclusive. In other words, a sequence of comparisons $E_1 <_1 E_2 <_2 \cdots E_n$, where the $<_i$ are comparison operators, is evaluated according to the following algorithm:

```
Set L ← E_1 and i ← 2.
while i ≤ n:
    Set R ← E_i
    if not L <_i R:
        Comparison yields False; stop
    Set L ← R and i ← i + 1
Comparison yields True
```

The interesting points here are that (1) evaluation stops when one comparison yields false, and (2) each E_i is evaluated only once. The individual comparisons themselves are simply calls to the comparison methods listed in Table 4, and can therefore be defined for new classes.

5.13 Boolean values and logical operations

PYTH has two literal constant values of type `Bool`: `True` and `False`, with the obvious meanings.

Several contexts in PYTH—**if** and **while** statements, for example—require a "truth value." By definition, the following values are *false values:*

- The `Bool` value `False`;
- The value `None`;
- The `Int` value 0;
- The `Float` values 0.0 and −0.0;
- Any value of type `String`, `List`, `Tuple`, `Xrange`, or `Dict` for which the value of `len(·)` is 0.

All other predefined values are true values.

The predefined function `truth(·)` is declared on all types. It converts true values to `True` and false ones to `False`. Its default value is `True`, but you can override it for any class you introduce.

The logical operators `and`, `or`, and `not`, described in Table 10 are exceptions to the rule in that they do not correspond to functions, unlike most PYTH operators. Each of them evaluates only enough of its operands to determine its final value.

5.14 Files

The type `File` supports a few simple operations for reading and writing bytes from and to an external file. As described in §2.1.3, the **print** statement has some special syntax for designating an output file as destination. Table 11 describes the other available operations on `Files`.

5.15 Miscellaneous functions

The library class `sys` must be imported to be used:

```
import sys
```

It provides the definitions shown in Table 12.

6 Classes

PYTH has classes vaguely resembling Java's[10]. The declaration:

`class` *class-name* `(`*parent-class-name*`):` *suite*

[10]Beware, however, that they differ markedly from those of PYTHON. PYTHON's classes are much more dynamic than PYTH's

defines a new class named *class-name* that extends (subtypes, inherits from) the class *parent-class-name* (both names are simple identifiers). The *parent-class-name* must be the predefined type `Object` or must appear previously in the program text. Class definitions may only appear at the outer level of a program—not inside a suite (so not only are there no nested or local classes, but you cannot define a class conditionally by putting its definition in an **if** statement).

All user-defined classes must eventually inherit from the type `Object` (that is, the only valid parent types are `Object` and user-defined class types; predefined types such as `List`, `Any`, and `Void` are therefore not extendable).

6.1 Instance variables and class variables

Instance variables (also known as *attributes*) are declared by being assigned to in the body of a class declaration, but not inside a function definition. For example,

```
class A(Object):
   x = f(3)     # If q is an A, then q.x is defined.
   y = x + 7    # Likewise, q.y
```

The scope of these declarations is the body of `A`, not including any **def**s in `A`, plus all places where they are made visible by selection (see §5.9)—that is, in expressions like E`.x`, where E is a value whose static type is a subtype of `A` (this is just like Java).

Pyth (like Python) handles instance variables rather differently from Java. When the declaration of `A` above is executed, variables named `A.x` and `A.y` get created, and initialized by the assignment statements given. After this declaration, you can create new instances of `A` by "calling" `A`:

```
newA = A()
```

This expression creates a new object of type `A`, containing new instances of all the instance variables defined in `A` and all instance variables in `A`'s supertypes (just like Java). However, the initial values of these variables are *copied* from `A.x`, `A.y`, and similarly from the supertypes. This means that the assignments to `x` and `y` in the class definition for `A` occur only once (not, as in Java, on every instance creation). In fact, after

```
A.x = 42
q: A
q = A()
```

the value of `q.x` will have been initialized to 42! In Java terms, there are *both* instance variables named `x` and `y`, *and* corresponding static variables named `A.x` and `A.y` that supply their initial values.

Just as elsewhere, you can define constants (including functions) in classes using **def**. Static constant variables are defined by

`def` *name* `=` E

Since it is static, you refer to it either as *T.name,* where T is the enclosing type (or a subtype), or *x.name,* where x is an instance of T (or a subtype).

If something is defined as an instance variable (or inherited), it may not be redefined with **def** and vice-versa.

6.2 Instance methods and class methods

Instance methods are defined by

```
def name (parameters): ...
```

All instance methods must have at least one parameter. The convention (not required) is to name it `self`, as in

```
def clear (self) : ...
```

The reason for the convention is that ordinarily, the first parameter of an instance method plays the same role as the quantity variously called 'self' (Smalltalk) or 'this' (Java, C++).

A slight variation of this introduces static (class) methods (which are allowed to be parameterless):

```
class def name (parameters): ...
```

This may appear only in a class definition. It may be referred to as *T.name,* where T is the name of the enclosing class (or subtype of it).

The special class method `__init__` is used to initialize instances (it is a class method whether or not declared as "`class def`". It is similar to a Java constructor. Its parameter list specifies the parameters that must be supplied to allocate a new object. That is, the allocation expression

```
A(1,2)
```

first creates a new `A`, call it *new*, copies in the values of its instance variables from the corresponding static variables in `A`, and then calls `A.__init__(`*new*`,1,2)`. If you don't supply a definition of `__init__`, PYTH will create one that looks like this:

```
def __init__ (self): P.__init__(self)
```

where `P` is the immediate supertype of `A`. Unlike Java, PYTH does *not* otherwise call the parent's constructor automatically; the programmer is supposed to do this[11], so that we would write `A`'s constructor like this:

```
def __init__ (self, a, b):
    P.__init__ (self)
    ...
```

6.3 Inheritance, overriding, and hiding

As in typical object-oriented languages, classes inherit the members defined in their superclasses. In PYTH, This means slightly different things in different cases[12].

[11] No, that's not "safe," but it does follow the practice of PYTHON.

[12] Here again, be warned that PYTH's rules are significantly different from PYTHON's.

Static variables and constants. As described in §6.1, definitions such as

```
class A(Object):
   myVar = 3
   ...
```

creates a static variable named `A.myVar` from which instance variables in instances of `A` are copied. When a class `B` inherits from `A`, the name `B.myVar` becomes a synonym for `A.myVar`; no new static variable is created regardless of whether `myVar` is assigned to in the body of `B`. There may not be a type declaration for `myVar` in the body of `B`. It is also illegal to introduce any kind of **def** of `myVar` in any subtype of `A`.

Likewise, if we have defined

```
class A(Object):
   def myConst = 17;
```

and `B` extends `A`, then `B.myConst` is a synonym for `A.myConst`. The class `B` may not have its own definition of `myConst`, nor may it assign to `myConst`.

Instance variables. Every class inherits one instance variable for each static variable it inherits. It follows that there is no overriding or hiding of instance variables; it a parent class has an instance variable `x` of type `T`, then its descendants have one instance variable `x` or type `T`.

The `__init__` method. The special method `__init__` is always a static method, regardless of whether it is declared with `def` or `class def`. It is *not* inherited. In classes where it is not defined, PYTH implicitly introduces a parameterless `__init__` method, as described in §6.2.

Other class methods. If `A` defines a class (static) method other than `__init__` (using the syntax

```
class def f(x): ...
```

) then `B.f` is a synonym for `A.f` in all subtypes of `A` that don't redefine `f`. A class may only redefine a class method with another class method.

Instance methods. If class `A` defines an instance method:

```
class A(Object):
   def myMethod(self, x): ...
```

then its subtype may either inherit this method (by not redefining it), or they may *override* it. If subtype `B` of `A` inherits `myMethod` and `b` is an instance of `B`, then `b.myMethod(3)` (or `myMethod(b,3)`) call `A`'s method.

Otherwise, subtype `B` may override `A`'s definition by providing its own **def** of it. `B` may not redefine `myMethod` with a **class def**, a constant declaration, or an assignment. The overriding

definition must have the same number of parameters as that of `A`. Its first parameter must have the same type as the containing class, and its other parameters must have the same types as in the parent class. Its return type must be a subtype of the return type of the overridden method. (The overriding class may provide a type declaration for the method, as long as it conforms to these restrictions.)

6.4 Example

Figure 1 shows a two-class hierarchy. The class `Parent` defines a constant `allObjects`, which is supposed to be a list of all objects of type `Parent` or its subtypes, and an integer instance variable `x`. It has two instance methods for incrementing `x` and for "working." It defines a class method that applies `incr` to all `Parent` objects on its list, and a constructor that sets an initial value for `x` and adds the newly created object to the list of all `Parent` objects[13].

The subtype `Child` adds an instance variable `S`, inherits the `incr` method unchanged and overrides the `work` method. The new `work` method first calls the overridden method in `Parent` (in Java, you would call `super.work` to do this), and then adds its own actions. The constructor for `Child`, as its first action, explicitly calls the parent type's constructor, insuring that each `Child` object will be added to the list of all `Parent` objects.

[13] You might be momentarily puzzled by the fact that we declared `allObjects` as a constant list and then go changing it. However, `allObjects` *is* constant—a constant pointer to be precise. It is the state of the object it constantly points to that changes.

Figure 1: A simple class hierarchy

```
class Parent (Object):
    x : Int
    x = 0

    allObjects : List
    def allObjects = []

    __init__ : (Parent, Int) -> Void
    def __init__ (self, x0):
        self.x = x0
        allObjects += [self]
        return self

    incr : (Parent, Int) -> Void
    def incr (self, incr):
        self.x += incr

    def work (self):
        print "The current value is " \
          + str(self.x)

    incrAll : Int -> Void
    class def incrAll (x)
        for p in allObjects: p.incr (x)
```

```
class Child (Parent):
    S : String
    S = ""

    __init__: (Child, Int, String) -> Void
    def __init__ (self, x0, s0):
        Parent.__init__ (self, x0)
        S = s0

    def work (self):
        Parent.work (self)
        print "The current name is " + str(self.s)
```

Table 3: Operations with nontrivial default implementations for all most types.

Operation	**Meaning**
`str(`A`)`	A (printable) String representation of A. Analogous to `toString` in Java. The default implementation produces a string that identifies the type and identity of the value.
`__hash__(`A`)`	An `Int` value suitable for hashing. Analogous to `hashValue` in Java. Normally, the value should be constant over the lifetime of the object, and consistent with the `==` operation (that is, objects that are `==` should have equal hash values). The default implementation returns an integer derived from the identity of the object.
truth(A)	Returns `True` if A is a true value (see §5.13) and `False` otherwise.
$A \prec B$	where $\prec$ is any of the comparison operations (`==`, `<=`, etc.), returns a truth value (see §5.13) indicating order. By default, the `==` and `!=` operators are implemented as `is` and `is not`. All these, when applied between a pair of values of different type (not both numerical), either cause an error, or yield the same truth value for all values with the same pair of types. By convention, if you define any of these for a class you introduce, you should maintain this property.
A `is` B	Identity test, returning `True` or `False`. True iff A and B are the same object. All objects are identical to themselves, and different from objects of other types. For types `Int`, `Float`, `String`, `Bool`, and `Void`, the **is** operator is the same as the `==` operator. Otherwise, objects created by two different operations are not necessarily identical.
A `is not` B	Same as `not (`A `is` B`)`.

Table 4: PYTH expression operators and corresponding function names. Horizontal lines separate operators of different precedences, highest precedence first.

Operator	**Corresponding functions**	**Predefined meaning**
$(E_1, \ldots)$		Tuple display (§5.4) or parenthesization
$[E_1, \ldots]$		List display (§5.5)
$\{K_1 : V_1, \ldots\}$		Dictionary display (§5.8)
E . ID		Attribute (field) selection; E may be an expression or type name (§5.9)
E`[`I`]`	`__getitem__`(E, I)	Array indexing (§5.3)
E`[`I `:` J`]`	`__getslice__`(E, I, J)	Array slice (§5.3)
$\mathcal{F}$`(`$A_1, \ldots, A_k$`)`		Function call (§5.10)
`**`	`__pow__`	Exponentiation, right-associative (§5.2)
Unary `+`, `-`, `~`	`__pos__`, `__neg__`, `__invert__`	Arithmetic unary operations (§5.2)
`*`, `/`, `%`	`__mul__`, `__div__`, `__mod__`	Multiplication or repetition, division, modulo (§5.2)
`+`, `-`,	`__add__`, `__sub__`	Addition or catenation, subtraction (§5.2)
`<<`, `>>`	`__lshift__`, `__rshift__`	Arithmetic shifting (§5.2)
`&`	`__and__`	Bitwise and (§5.2)
`^`	`__xor__`	Bitwise xor (§5.2)
`|`	`__or__`	Bitwise or (§5.2)
`<`, `<=`	`__lt__`, `__le__`	Comparisons (special, see §5.12)
`>`, `>=`	`__gt__`, `__ge__`	
`==`, `!=`	`__eq__`, `__ne__`	
in	`__contains__`	
not in		X `not in` Y iff `not (`X `in` Y`)`
is		Identity (Table 3)
is not		X `is not` Y iff `not (`X `is` Y`)`
not		Logical negation (§5.13)
and		Logical and (special, see §5.13)
or		Logical or (special, see §5.13

Table 5: Operations on numbers, part I.

Operation	**Description**
`+`E	The identity.
`-`E	Negation.
`abs(`E`)`	Absolute value.
`str(`E`)`	A string containing an integer or floating-point numeral (depending on the type of E) that denotes E. Thus `str(3.25)` is `'3.25'`.
`int(`E`)`	E converted to an integer. E may be an `Int`, in which case the operation is the identity; a `Float`, in which case it yields the integer resulting from truncating the fraction of E; or a string, in which case the result is the integer denoted by the string, interpreted as a signed decimal integer literal (it is an error in that case if string E does not contain a decimal integer literal).
`float(`E`)`	E converted to a `Float`. E may be a `Float`, in which case the operation is the identity; an `Int`, in which case it yields the nearest `Float` value to the operand; or a string, in which case the result is the value denoted by the string, interpreted as a signed floating-point literal (it is an error in that case if string E does not contain such a literal).
E_1 `+` E_2	The sum of E_1 and E_2.
E_1 `-` E_2	The difference of E_1 and E_2.
E_1 `*` E_2	The product of E_1 and E_2.
E_1 `/` E_2	The quotient of E_1 and E_2. For two integer operands, yields $\lfloor E_1/E_2 \rfloor$. *WARNING:* This is different from Java and C, which truncate integer divisions. For example, in Pyth, -1/3 is -1, not 0.
E_1 `%` E_2	Modulus. For integer operands, the integer, r, whose sign is the same as E_2, such that $0 \leq \lvert r \rvert < \lvert E_2 \rvert$ and $k \cdot E_2 + r = E_1$ for some integer k. For floating-point operands, the formula is the same, but r is a `Float` and the equality is "up to round-off error," as usual for floating-point operations.
E_1 `**` E_2	E_1 raised to the E_2 power. E_2 must be an `Int` (it is not converted, nor is E_1). The type of the result is the same as that of E_1. It is an error for E_2 to be negative if E_1 is an `Int`, or if it is 0.

Table 5: Operations on numbers, part II.

Operation	**Description**
$E_1 \prec E_2$	where $\prec$ is one of the comparison operators, has the usual arithmetic meaning. If E_2 is not a number, the result is always true or always false, depending only on the dynamic type of E_2.
E_1 `&` E_2	The bitwise and of the 32-bit representations of `Int`s E_1 and E_2.
E_1 `^` E_2	The bitwise exclusive or of the 32-bit representations of `Int`s E_1 and E_2.
E_1 `|` E_2	The bitwise or of the 32-bit representations of `Int`s E_1 and E_2.
`~` E	The bitwise complement of the 32-bit representation of `Int` E.
E_1 `<<` E_2	E_1 shifted left by E_2 mod 32 bits (modulo 2^{32}, as usual). The operands must be `Int`s, and $E_2 \geq 0$.
E_1 `>>` E_2	E_1 shifted right by E_2 mod 32 bits (i.e., the sign bit is replicated E_2 times on the left, dropping E_2 bits on the right). The operands must be `Int`s, and $E_2 \geq 0$.

Table 6: Common operations on sequence classes.

Expression	**Description**
`len(`S`)`	Number of items in collection (≥ 0).
S`[`i`]`	Element indexed by i in collection S. Indices must be integers $-\text{len}(S) \leq i < \text{len}(S)$. If $i < 0$, then S`[`i`]` is equivalent to S`[`i`+len(`S`)]` (i.e., i^{th} from the end).
S`[`i`:`j`]`	A slice (consecutive subsequence) having the same type as S and containing all elements or S with indices k, where $i' \leq k < j'$. Here $i' = i$ if $i \geq 0$ or $i' = i + \text{len}(S)$ if $i < 0$ and $j' = \min(j, \text{len}(S))$. So if `x` is the string `"abc"`, then `x[0:3]` and `x[0:100]` are both `"abc"`, `x[0:2]` is `"ab"`, and `x[-2:3]` is `"bc"`.
S`[`i`:]`	Same as S`[`i`:len(`S`)]` (where S is evaluated once).
S_1 `+` S_2	Concatenation of S_1 and S_2. S_1 and S_2 must have the same type. The result is a new sequence with that same type consisting of the elements of S_1 followed by those of S_2.
S `*` n`,` n `*` S	where n is an integer, concatenates $\max(n, 0)$ copies of S into a new sequence of the same type as S.
E_1 `==` E_2	True iff E_1 and E_2 are sequences of the same type and length with corresponding members `==`.
E_1 `!=` E_2	Same as `not (`E_1 `==` E_2`)`.
$E_1 \prec E_2$	where $\prec$ is one of the comparison operators (`<`, `>`, `<=`, `>=`) is lexicographic comparison when the operands are sequences of the same type. That is, the ordering is determined by the first corresponding members that are not equal. If they all are equal, but one operand is shorter, then it compares less than. If the operands have different types, the result is consistently either true or false depending only on the types of the operands (e.g., either all strings are less than all lists or greater than all lists), in such a way that the comparisons are always transitive.
x `in` S	True iff x=S`[`i`]` for some i, else False.
x `not in` S	Same as `not` x `in` S.

Table 7: Operations on lists.

Operation	**Description**
`L[i] = E`	Element assignment (see §2.1.4).
`L[i:j] = E`	Slice assignment (see §2.1.4).
`L[i:] = E`	Same as `L[i:len(L)]`, but `L` is evaluated only once.
`append(L, E)`	Same as `L(-1:) = [E]`.

`list(S)`	Create a new list from `S`. Same as the value of `L` after
	`L = []`
	`for x in S:`
	`  L.append (x)`

$E_1 \prec E_2$	where $\prec$ is one of the usual comparison operators. See §5.12.

Table 8: Additional operations on Strings

Operation	**Description**
str(S)	The identity operation.
isalpha(S)	True iff all characters in string S are letters.
isdigit(S)	True iff all characters in string S are digits.
islower(S)	True iff all characters in string S are lower-case.
isupper(S)	True iff all characters in string S are upper-case.
lower(S)	The value of S with all upper-case characters replaced by lower-case characters.
upper(S)	The value of S with all lower-case characters replaced by upper-case characters.

Table 9: Operations on dictionaries

Operation	**Description**
d`[`k`]`	The value associated with key k. Causes an error if k is not a key in d.
`__delitem__(`d`,` k`)`	Remove key k from d.
d`[`k`] =` V	Set the value associated with key k to V.
`len(`d`)`	The number of distinct keys in d.
k `in` d	True iff k is a key in d.
k `not in` d	Same as `not (`k `in` d`)`.
`clear(`d`)`	Remove all keys from d.
`get (`d`,` k`,` v`)`	If k is in d, then $d[k]$, and otherwise v.

Table 10: Logical operators

Operator	**Description**
`True`	The true value of type `Bool`.
`False`	The false value of type `Bool`.
`truth(`X`)`	For any value X, returns `True` if X is a true value, `False` otherwise.
E_1 `and` E_2	Yields the value of E_1 if it is a false value, and otherwise yields E_2. E_2 is not evaluated if E_1 yields a false value.
E_1 `or` E_2	Yields E_1 if E_1 yields a true value, and otherwise yields E_2. E_2 is not evaluated if E_1 yields a true value.
`not` E	Yields `False` if E yields a true value, and otherwise `True`.

Table 11: Operations on Files.

Operation	**Description**
`open(`S`, `M`)`	Creates a new File that reads from or writes to an external file named S (a string). The second argument, M, is a string indicating the *mode,* and is one of `"r"` (result is an input file), `"w"` (result is an output file, and the external file is first erased), or `"a"` (result is an output file, and all output is appended to whatever was there before).
`close(`F`)`	Closes the File F, making further operations on F illegal.
`read(`F`, `N`)`	Where F is an input File and N is an `Int`, reads data from F and returns it as a String. If $N \geq 0$, reads at most N bytes (fewer if an end-of-file is encountered first). Otherwise, reads and returns the entire file up to the end of file.
`readline(`F`)`	Returns one line from input File F. Reads and returns all characters up to an including the next end of line, or the entire rest of the file if there is no next end-of-line.
`write(`F`, `S`)`	Writes String S to output File F.

Table 12: The `sys` library module.

Operation	**Description**
`sys.exit(`N`)`	Where N is an integer, exits the program with status code N (0 means normal termination).
`sys.stdin`	A variable initially containing a File reading from the standard input.
`sys.stdout`	A variable initially containing a File that writes to the standard output.
`sys.stderr`	A variable initially containing a File that writes to the standard error output.
`sys.argv`	A list containing the command-line options passed to the program. Each element is a string.

Index