Chapter 6

Expressions

“I can calculate the motion of heavenly bodies, but not the madness of people.”
— Isaac Newton

“rode hard and put up wet”
— Texan expression

Computing starts to become interesting when operators and types combine to form expressions. Throughout science, there are expressions about equalities or inequalities, typically called formulas or equations. Scientific breakthroughs sometimes have concise technical statements expressed by such formulas. Students learn how to calculate with expressions (usually algebraic expressions), but the same general way of manipulating symbols happens with logic, balancing chemical reaction equations, deducing voltages for a circuit and so on. Every student learns how, with the aid of a calculator, to find the $x$ satisfying $x = 8/(5^2 - 3 \times 7)$. Early on, students learn rules of what to do first, how to proceed from one step to the next, in order to solve such problems. Similarly, programming languages need syntax rules that guide interpretation of source code, which is the topic of this chapter. A Python program might contain an expression

$\text{not True or not False and True}$

Without syntax rules, this expression seems ambiguous. Programs can also mix operators and arguments having a variety of types; here, syntax rules (like grammar in English) can improve programming style, so that programs written by one person are more easily understood by someone else.

Sequential Reduction

Let us reconsider, in “slow motion”, how an arithmetic expression is reduced to a final answer. The following table shows the original expression and lines

<table>
<thead>
<tr>
<th>Original Expression</th>
<th>Reduced Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = 8/(5^2 - 3 \times 7)$</td>
<td>$x = 8/(25 - 21)$</td>
</tr>
</tbody>
</table>
Expressions

Expressions

that make one simplification to the previous line, until the last line is a single
number.

\[
1-2-3-4 \quad \rightarrow \quad -1-3-4 \\
-1-3-4 \quad \rightarrow \quad -4-4 \\
-4-4 \quad \rightarrow \quad -8
\]

The *order of evaluation* for these steps is *left-to-right*, that is, each step cal-
culates the leftmost operation in the expression to get the next line. This is
Python’s normal way of reducing, or *evaluating* expressions. So long as an ex-
pression has an operator in it, there is further evaluation work to do. It’s easy
to change the order of evaluation by inserting parentheses:

\[
1-(2-3)-4 \quad \rightarrow \quad 1--1-4 \\
-1--1-4 \quad \rightarrow \quad 0-4 \\
0-4 \quad \rightarrow \quad -4
\]

When part of an expression is enclosed in parentheses, Python evaluates that
part first, but otherwise the order of evaluation remains left to right. Unfor-
tunately, Python does not show this detailed, sequential reduction of an ex-
pression: Python appears to instantly evaluate the expression and return an
answer. However, it can sometimes be a useful exercise to manually go through
an evaluation one step at a time. You may already suspect that Python has an
algorithm for evaluation.

Well-Formed Expressions

Using some “rules of construction”, a clear definition of the syntax of expressions
emerges. The rules, taken together, define *well-formed expressions*. These are
expressions that follow the syntax, but may still be incorrect. The rules define
expressions that superficially look reasonable: parentheses match, operators have
arguments, and so on. Yet, according to the rules, “9/(3-2-1)” is a well-formed
expression, even though Python would output a *ZeroDivisionError* message
when trying to evaluate it. This is a general phenomenon of programming lan-
guages: so-called *syntax errors* are found by compilers, before the program ever
runs; whereas *runtime errors* are discovered later, when the program executes.
Think of the well-formed formula rules as a gatekeeper for correct programs.
The rules may not find all the bugs, but can find some obvious ones. An-
other nice property of the syntax rules is that programming language editors
(which are essentially word processors specialized to programs) can be aware
of the rules, finding errors even as you use the keyboard to write source code.
This works somewhat like spell-checkers that watch what you type, suggesting
whether or not a word is suspicious according to its dictionary.
The rules that follow are deceptively powerful. The rules start with simple statements, but build on each other in perhaps surprising ways. It’s best to read them first, then look at examples of expressions, going back to the rules if questions arise. Rules R6 and R7 look more complicated, but the pattern is easy to understand after looking at a few examples.

**Rule R1**
Every Python value, be it numeric, character, string, tuple, list, dictionary, boolean, etc, is a well-formed expression.

**Rule R2**
If $E$ is a well-formed expression, then $(E)$ is a well-formed expression.

**Rule R3**
Given any unary operator $\circ$, if $E$ is a well-formed expression, then $\circ E$ is a well-formed expression.

**Rule R4**
Given any binary operator $\circ$, and two well-formed expressions $L$ and $R$, then $L \circ R$ is a well-formed expression.

**Rule R5** (Indexing Syntax) If $S$ is a well-formed expression and $E$ is a well-formed expression, then $S[E]$ is a well-formed expression.

**Rule R6** (Function Syntax) If $f$ is a function name, and $E_1, E_2, E_3, \ldots$, are all well-formed expressions, then $f(), f(E_1), f(E_1,E_2), f(E_1,E_2,E_3), \ldots$ are also well-formed expressions.

**Rule R7** (Method Syntax) If $f$ is a method name, $S$ is a well-formed expression, and $E_1, E_2, E_3, \ldots$, are all well-formed expressions, then $S.f(), S.f(E_1), S.f(E_1,E_2), S.f(E_1,E_2,E_3), \ldots$ are also well-formed expressions.

This list of rules is incomplete. A more formal, complete list of rules would have provisions for lists, tuples, dictionaries, sets, strings, and other features introduced in later chapters. The rules for the types of Chapter 4 can be understood by intuition because of the many examples already presented. The difference here is that expressions can be used in lists. For well-formed expressions $E_1$, $E_2$, $E_3$, the list $[E_1, E_2, E_3]$ is also a well-formed expression; similarly, expressions can be used in tuples, dictionaries, and so on.

**Examples**
The following examples show that expressions must be well-formed to be valid Python, though there are cases where it is not enough to be well-formed. The first line is not well-formed because only Rules R2, R6 and R7 introduce parentheses, and always in pairs surrounding well-formed expressions: "xyz"[2 is not a well-formed expression because Rule R5 demands that brackets occur in pairs.
Operator Priorities

Even when an expression is well-formed and is valid in Python, meaning that it runs without halting and outputting an error message, the result might not be what we expect. The problem is that, without a further kind of rule, there can be some ambiguity about the order of sequentially reducing an expression to a final result. Consider this mixture of string concatenation and indexing:

"mobility"+"patterns"[0]

Without knowing better, you might think this should reduce to 'm', reasoning that evaluation goes left to right, so the intermediate result would be "mobilitypatterns"[0], which returns 'm'. But, instead, Python returns "mobilityp", because the indexing operation occurred first, before the string concatenation. If you wanted the first order of operations, then you would need to use:

("mobility"+"patterns")[0]

Using parentheses forces the order of evaluation, so the answer here is 'm'. What is needed, in order to make Python's evaluation of expressions predictable, is to know about all the exceptions to the normal, left-to-right, evaluation process. Figure 6.1 shows the priority of Python operators, with the highest priority operator at the top of the table.

Examples As done earlier in the chapter, working through examples sequentially, one reduction per step, illustrates Python's order of evaluation. The examples labels each step by the priority of an operator in the table of Figure 6.1 Each line makes one reduction step to get the next line.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Well-formed</th>
<th>Valid Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>len(&quot;xyz&quot;[2])</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>5+&quot;AB&quot;[9]</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>*+3,4</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>&quot;abc&quot;[2]*len(&quot;abc&quot;)</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>2.5 (3e9 - -4E8)</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>[1+2,True or not False]</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>
Expressions

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>function application</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$f(\cdots)$</td>
<td>function application</td>
</tr>
<tr>
<td>2</td>
<td>$E_1[E_2]$</td>
<td>index (lookup)</td>
</tr>
<tr>
<td>3</td>
<td>$E.f(\cdots)$</td>
<td>method call</td>
</tr>
<tr>
<td>4</td>
<td>**</td>
<td>exponentiation</td>
</tr>
<tr>
<td>5</td>
<td>$-E$</td>
<td>change sign</td>
</tr>
<tr>
<td>6</td>
<td>$\ast, /, \div, \mod$</td>
<td>multiplication, division, remainder</td>
</tr>
<tr>
<td>7</td>
<td>$+, -$</td>
<td>addition, subtraction</td>
</tr>
<tr>
<td>8</td>
<td>$&lt;, \leq, &gt;, \geq, &lt;&gt;, !=$</td>
<td>comparison operators</td>
</tr>
<tr>
<td>9</td>
<td>in, not in</td>
<td>membership</td>
</tr>
<tr>
<td>10</td>
<td>not $E$</td>
<td>logical negation</td>
</tr>
<tr>
<td>11</td>
<td>and</td>
<td>logical conjunct</td>
</tr>
<tr>
<td>12</td>
<td>or</td>
<td>logical disjunct</td>
</tr>
</tbody>
</table>

Figure 6.1: Python operator priorities

The next section of the chapter has a more detailed explanation of how Python evaluates and reduces expressions, but the example above hints at how things occur: there is a series of steps, each step simplifying the expression, and each step consults the table of Figure 6.1. Extra parentheses can override Python’s operator priority; the same holds for brackets (used to define lists or indexing) and for curly braces (used to define dictionaries). Consider

$\{3+4: \text{True}, \ 50/2: \text{"quarter"}\}[12-5]$

According to the operator priorities, (2) dictionary lookup has highest priority for evaluation. But, the dictionary items have expressions for keys, and these are within the \{\} symbols; similarly, there is an expression within the [\] for the lookup value, and that also has higher priority to reduce first. The actual order of evaluation would therefore be

$\{3+4: \text{True}, \ 50/2: \text{"quarter"}\}[12-5] \rightarrow \text{left-to-right}$
$\{7: \text{True}, \ 50/2: \text{"quarter"}\}[12-5] \rightarrow \text{left-to-right}$
$\{7: \text{True}, \ 25: \text{"quarter"}\}[12-5] \rightarrow \text{left-to-right}$
$\{7: \text{True}, \ 25: \text{"quarter"}\}[7] \rightarrow \text{(2)}$
$\text{True}$

You might notice above that the first three steps, done left-to-right, could instead be done in any order, because they are “independent” expressions. Mentally, this is the simplest way to think about parentheses (or braces and brackets): they surround expressions that have to be evaluated before the operators outside of the parentheses (braces, or brackets).

Typical of Python usage is a comparison expression with function application, arithmetic or string operations in it:
Expressions

\[
\text{len('t'+\text{E})}>2 \text{ and not "x" in "axis" } \rightarrow \text{ parentheses}
\]
\[
\text{len("t E")}>2 \text{ and not "x" in "axis" } \rightarrow (1)
\]
\[
3>2 \text{ and not "x" in "axis" } \rightarrow (8)
\]
\[
\text{True and not "x" in "axis" } \rightarrow (9)
\]
\[
\text{True and False } \rightarrow (10)
\]
\[
\text{False}
\]

Many software professionals are not fully cognizant of operator priorities. Having to work with numerous programming languages, system tools, database packages, and a variety of mobile computing devices (plus keep up with all the latest trends in computing), even professionals cannot be expected to recall obscure details of a particular programming language. Thus, some programmers might rewrite the example above as

\[
(\text{len('t'+\text{E})}>2) \text{ and (not ("x" in "axis"))}
\]

Using the added parentheses, it’s obvious what is evaluated first, second, and so on. The and is clearly the last operation to be evaluated. If you’re uncertain or perhaps just want to make expressions more readable, consider adding extra parentheses. Whether you can them or not is up to you: in all areas of life, expression can be a matter of personal style.

**Python’s Algorithm**

Given the list of rules earlier in this chapter, how does Python evaluate an expression? There are different ways to explain this, and we use a simplified explanation here. Repeating from earlier the general rule is left-to-right reduction of an expression to a final value. However, as Python evaluates from left to right, it “looks ahead” before reduces an operator. Consider the expression

\[
2-3-4*2**5+1
\]

Going left-to-right, the first operator is subtraction: does Python therefore immediately reduce 2-3 to \(-1\)? No! First, Python observes that the operator following 3 is another minus operator; now, since the operator on the left of 3 and on the right of 3 have the same priority, Python can safely reduce 2-3 to \(-1\) and get

\[
-1-4*2**5+1
\]

as a partially reduced expression. The next operation to consider is reducing \(-1-4\) to \(-5\): can this be done? No! First Python observes that the operator to the right of 4 is \(*\). Comparing multiplication to minus in the priority table
of Figure 6.1, we see that multiplication has higher priority. Hence, Python should reduce $4 \times 2$ before reducing the minus sign to the left of $4$. So is that what Python does next? No! Before reducing $4 \times 2$, Python looks to the right of $2$ and finds the operator $**$. Again, we consult the table of Figure 6.1. It turns out that $**$ has higher priority than $\times$, so Python should first reduce $2**5$ before working on the multiplication. Is that what Python does next? No! Python looks to the right of $5$ to see if there is another operator (if there were no more operators, we’d finally be at the end of the story). To the right of $5$ is the plus operator. The plus operator has lower priority than $**$ in the table. Therefore, finally, Python can get to work actually reducing an operation to a value. To summarize where things stand at this point in the evaluation, Python has a “backlog” of two jobs to do,

1. reduce $2**5$, getting $32$
2. then reduce $4 \times 32$, getting $128$

The partially reduced expression thus becomes

$$-1-128+1$$

Once again, Python can consider reducing the minus operator to the right of the first $1$ in this expression. Is that what Python does? No! First, look ahead to compare this minus operator to the operator on the right of $128$. It’s a plus operator. In the table of Figure 6.1, the minus and plus have the same priority. What’s Python to do in this event of equal priority operators? The answer is simple, just use left-to-right evaluation. Python therefore reduces $-1-128$ to $-129$. The partially reduced expression becomes

$$-129+1$$

One more step gets the final value, $128$.

A curious illustration of Python’s left-to-right order of evaluation occurs with boolean operators $or$, and. Python optimizes the way that these operators are evaluated by skipping reduction when it won’t matter. Suppose Python is asked to evaluate

$$True \ or \ \not{}$$

where “$\not{}$” is some expression. It turns out that whether “$\not{}$” is $True$ or it is $False$ doesn’t matter: the final result will be $True$, just based on the simple definition of “$or$” in Python. Therefore, Python skips even trying to evaluate “$\not{}$” (so long as it is a well-formed expression). Here’s some evidence of this fact, using interactive Python:
Expressions

>>> (3>2) or (3>(2/0))
True
>>> (3>(2/0)) or (3>2)
ZeroDivisionError: integer division or modulo by zero

Evaluating left-to-right, Python gets the partially reduced expression

True or (3>(2/0))

Python can immediately reduce this to True, since it doesn’t matter what value would be on the right side of the or operator. But, when we switch around the two sides, the left-to-right rule asks Python to evaluate (3>(2/0)) first, and this is an error.

A similar example would be an expression

7>1 and "t" in "it" and 4>2+2 and 7>3/0

According to the priority table in Figure 6.1, all operators (>, in, +, /) are higher priority than and. However, going left-to-right, whenever Python might encounter something of the form

False and 8

then Python can immediately conclude the reduction to False is correct, regardless of the value of “8”, by the definition of and. Partial evaluations for the expression above are

7>1 and "t" in "it" and 4>2+2 and 7>3/0 ➔
True and "t" in "it" and 4>2+2 and 7>3/0 ➔
True and True and 4>2+2 and 7>3/0 ➔
True and True and True and 4>2+2 and 7>3/0 ➔
True and True and True and True and 4>2+2 and 7>3/0 ➔
True and True and True and True and True and 4>4 and 7>3/0 ➔
True and True and True and True and True and True and False and 7>3/0 ➔
False and 7>3/0 ➔

False

Terminology Review

Jargon introduced in this chapter includes: expressions, evaluation, order of evaluation, well-formed expressions, run-time errors, syntax errors, left-to-right, operator priorities, reduction, syntax rules.

Copyright © 2010 Ted R. Herman, all rights reserved.
Expressions

Exercises

(1) What would be the result of Python evaluating the following?

1. "Hello",[21,4]
2. "abcd" + ("mnop")
4. "amazing" >= "astounding"
5. "and" in "or" or "or" != "and"
6. "california"["f"
7. "divide"/2
8. "easy" in ("yes we ease"*2)
9. "gb" in (10*"boing")
10. "hand" + "traffic"[2*5-4]
11. "hexadecimal"/2
12. "orange"[(2,9,-2,5)[2]]
13. "w" in "Iowa" and (5!=4*3-7 or "k" not in "Hawk")
14. "xmo/2+57
15. ("and" in "or") or ("or" != "and")
16. ("hq"*2,("a"<"b")and True)
17. (((10<2*5) or (7<7*7))",a"*2
18. ((not ((10-2)==2**3) and (1e5 > 3000))
19. (-1,-2,-3,-4)[-2]
20. (-2**3,"0"+"2")
21. (0*"x") in "Iowa"
22. (1,2)[0]
23. (256**0.5)**0. (256**0.5)**0.5
25. (not True, "to"+"rn", 72/6)[2-4]
26. (type(4*4),type("a"*2),type(3.0*2))
27. -(3**4) <= (-3)**(4
28. 1*2+3*4*5/10
29. not (True and False)
30. 1+2*7**2
31. 1.-4**
32. 105/2 == 52.5*2.0 (Python2)
33. 10>7 or "m">"b" and True
34. 15*(3-2*(12+6/(4+8))
35. 15//2+1 (Python3)
36. 2 in (21,5-1)
37. 2*((3-1)*"ox")
38. 2*2+9+2*3-5
39. 3*(2+(8*(1+(6+7)*2))))
40. 4*(-3+2+2)*'stop'
41. 5-4-3-2
42. 6/((10*4)+3)) (Python2)
43. 8.1e6-10000 <= (2**3)*(10**6)
44. 89//2+1e2 (Python3)
45. True and ((5!=2) or (not False))
46. True and (False or not False)
47. type( 8>10 )
48. 8.1e6-10000 <= (2**3)*(10**6)

(2) What should be in the two blanks so that Python would evaluate
the following expression to be 25?

((10,11,12),(4,5,25),(7,19,21))[____][____]

(3) What value should be in the blank so that Python will evaluate this
expression to be True?

("four","five","six","seven","eight")[____] in "one hundred sixty"

(4) What value should be in the blank so that Python will evaluate this
expression to be 19?

(30-19+2,-5+3*8,"19",True)[____]

(5) What value should be in the blank so that Python will evaluate this
expression to be False?

(5>=5,"a"<"m",6==2*3,"f" in "swim",True)[ 2 - ____ ]