Basic Syntax and Arithmetic in Python and Ada

by:

Graham Morehead

October 10, 2008

COS 301, University of Maine

Abstract

Basic syntax and arithmetic in Python and Ada are discussed and compared, including statement separators, block structure, and operators. A specific example shows the garnering of floating point input from the command line, rounding, and evaluation of an expression using basic arithmetic operators.

Table of Contents

Basic Syntax and Arithmetic in Python and Ada

1. Python and Ada are different

The heading above states the obvious, but it doesn't overstate the fact that Python and Ada are quite distant from each other on the syntax spectrum. Python has an intuitive forgiving syntax. It is easy to compose. Ada by comparison is arcane and unforgiving. In Python there are often multiple ways to write the same thing. In Ada there is usually only one right way. Looser syntax rules mean that Python developers can write much more quickly, which is a main goal of Python. Tighter syntax means that the Ada compiler has fewer possibilities to consider (fewer than if it had looser rules -- not fewer than Python). Such tightness can help when verifying correctness, which is a main goal of Ada. The downside is that Ada is difficult to compose.

2. Syntactic Structure

2.a. Statement Separators

Ada is a more typically structured language in that whitespace doesn't play a meaningful role. Each successive statement is separated by a semi-colon. If each statement happens to be on a separate line, it is easier to read, but it doesn't change how it is tokenized. Python, on the other hand, uses a newline to separate each statement. Having each command on a separate line makes sense to human eyes, and that fits with the philosophy behind Python. The semi-colon is optional on the end of Python statements. They are only mandatory when a programmer chooses to place more than one statement on a given line [1,2,3].

The semi-colon is a better separator from a compiler writer's point of view, but it's not enough for a human. Fom a developer's point of view, the newline is necessary to keep everything from agglutinating into one big jumble. Once everything is on it's own line, the semi-colon is only there for the compiler's benefit. Python users seem to appreciate its optionality. The only complication comes when one statement must extend to multiple lines. Since the newline char has meaning in Python, it must be escaped with a backslash. The fact that such extension reduces readability should encourage programmers to write shorter statements when possible [1,2,3]. See Appendix 1 for a brief overview of the differences in Statement Separators between Python and Ada

2.b. Block Structure

Programmers have long been indenting their nested blocks as a matter of convention because code written this way is easier to cognize. One glance can tell you where the code blocks begin and end, and how they are nested in relation to each other. Even though this convention is not enforced in most languages, almost all programmers write this way. Ada is usually indented, but the Ada tokenizer ignores all indents. The Tokenizer looks a block's *start* statement and *end* statement to determine where a Block starts and finishes. A Block usually starts with a conditional such as "if," "while," or a definition such as "procedure". An *end* statement ends each block with a phrase such as "end if," or "end <procedure name>." The end statement also resolves the "dangling else" problem for Ada. Python, however, uses varying amount of whitespace to signal block nesting. The philosophy behind Python block structure is intuitive and human-oriented. If it looks nested, it is. Python makes the indentation mandatory. The information carried in the whitespace-length is not lost in the Python tokenizer. This information is passed to the parser so that it can build a parse tree. Indentation has taken the role typically played by curly braces or *start/end* statements [1,2,3]. See Appendix 2 for an overview.

3. Data Types

A general difference between the data types in Python and Ada is that in Python, less needs to be specified at compile time. In Ada, most data types should have a range specified. Unlike languages like C, where an int is 4 bytes, an Integer in Ada will have a size dependent upon the range that was specified in code. A Float size in memory will depend on the number of significant digits, another attribute specified in the code. Many size related problems are found at compile time because of these range restrictions. This narrowing of possible input values can also dramatically reduce the search space for correctness verification analysis. On the downside, Ada developers must know the range of each variable ahead of time. Determining these values can be a time-consuming process. In contrast, Python allows variable size to grow as needed with no onus on the programmer to allow for such growth [20,21].

The more complex data types show the same divergence in priorities as do the basic ones. Python has composite types such as tuples and lists allow for their members to be of any type, and to be referenced and assigned with ease through a number of syntactic-sugary notations. Ada has composite types such as Protected and Task which aid in correctly coding for concurrency [20,21]. See Appendix 5 for an overview.

4. Operators

4.a. Precedence

Operator precedence does not differ greatly between Ada and Python. There are a few more operators in Python than there are in Ada, but the ones that they share mostly share the same precedence levels, and they share them with other common languages as well, such as Java and C. One difference that does stick out, however, is the relatively high precedence in Ada given to the 'Not' operator. In Python it sits with the other logical operators. In Ada it has a higher precedence than even the exponential operator (**) [16.17]. See Appendix 3 for an overview.

4.b. Associativity

Ada operators have much simpler associativity than those in Python. In Ada, several operators have left associativity, and several don't have any associativity. None have right associativity. The overall effect again is a tightening of the grammar constructs within Ada. This stringency seems to be intended for the benefit of the compiler developer, not the average Ada programmer [18].

Python has both left and right associativity, and something called “chain” associativity. Under both left and right associativity, the phrase “a < b < c” is relatively useless in most languages. When observed it is usually a rookie mistake, but not in Python. The '<' operator has “chain” associativity, which means that the above phrase translates to “(a < b) && (b < c)”, which is the typical meaning of this phrase in algebra. A number of Python operators have chain associativity, opening up Python code to sundry statements not likely in other languages [19]. See Appendix 4 for an overview.

5. Code Sample

Two pieces of sample code are provided. They produce similar output for identical input. Their respective output numbers are equal in value, but they differ in format. Floats in Ada output in the format xEy as opposed to the simpler Python output. The goal of the sample code is to show each language's approach to reading command line input, averaging, and other arithmetic.

For the Python code see Appendix 6. For the Python output see Appendix 7.

For the Ada code see Appendix 8. For the Ada output see Appendix 9.

5.a. Obtaining Input

In both cases, after a module is included or imported, the argument vector (similar to C) can be read and it's elements can be cast into whatever variables are needed; floats, in this case. Besides Python's easier syntax, this step does not differ in complexity between the two implementations.

5.b. Averaging and Rounding

Both languages are able to accomplish simple mathematics (such as averaging) with ease, but Ada requires more steps because of its more restrictive grammar. The Python code is less than half the length of the Ada code. Besides the five floats declared for the input, five more were declared in the Ada code just to hold the rounded versions. With greater understanding it could be possible to ensconce the rounding statements inside the print statements as they are in the Python code, but it is not easy.

Both Ada and Python show “Banker's” rounding. From 'n.5' they both round up to 'n+1'. Native rounding functions are used instead of merely casting to integers. In each case a custom procedure would be the only way to utilize the more equinimitous “nearest even” type of rounding. Ada and Python each have their version of the 'Floor' and 'Ceiling' functions, but these do nothing more than 'truncate(n)' and 'truncate(n)+1', respectively. They are not rounding methods.

5.c. Expression Evaluation

The following expression is calculated in both languages:

$$(a - b^3 * (c + d) / (e -a) + 1) - (a + e)^2$$

where the variables a, b, c, d, and e are the floats from the first step. This equation was equally simple to implement in both code samples. The evaluation takes place on one line. The only difference is that in Python it is evaluated on the same line that prints it. Both languages appear tuned to accomplish basic arithmetic.

6. A Difference in Priorities

The code project and the details laid out in the Appendices show a marked delineation between the priorities of the creator(s) of each language. Ada forces programmers to jump through various hoops to accomplish simple tasks while Python allows programmers a number of intuitive ways to implement algorithms in few statements. Python programs allow variables to grow as needed, where as Ada allows programmers to set the range of variables, thus giving them a greater granularity of control over the size of memory usage and overall complexity of the machine code. It is abundantly clear from the topics discussed above that Python was created to make the developer comfortable and prolific, while Ada was written for an audience that cared more about correctness and concurrency and less about the length of each development cycle. Both languages have significant strengths that should be considered before choosing one for development.

Bibliography

Note: not all of the following were referenced in the above paper, but are being used in the overall project.

[1] M. Pilgrim, *Dive Into Python*, 20 May 2004,
http://diveintopython.org
(Useful for learning Python basics).

[2] J. Elkner, A. Downey, C. Meyers, *How to Think Like a Computer Scientist: Learning with Python*, 2nd Edition
http://openbookproject.net//thinkCSpy/index.xhtml
(Useful for learning Python basics).

[3] R. Riehle, *Ada Distilled: An Introduction to Ada Programming for Experienced Computer Programmers*, July 2003, AdaWorks Software Engineering
http://www.adapower.com/launch.php?URL=http%3A%2F%2Fwww.adapower.com%2Fpdfs%2FAdaDistilled07-27-2003.pdf
(Useful for learning Ada basics).

[4] C. Ausnit-Hood et al., *Ada 95 Quality and Style*, 1995, Software Productivity Consortium
(Useful for Ada examples and some history).

[5] J. Barnes, *Ada 95 Rationale*, 1995, Intermetrics Inc.
(Useful for Ada examples and some history).

[6] B. Boehm et al., *Ada and Beyond*, 1997, National Academy Press
(Useful for Ada history and information about community).

[7] A. Burns, A. Wellings, *Concurrency in Ada*, 1995, Cambridge University Press
(Useful for Ada examples and some history).

[8] J. Jones and commenters, "Does Python have a concurrency problem?," blog, October 5, 2005
http://www.oreillynet.com/onlamp/blog/2005/10/does_python_have_a_concurrency.html
(Description of a weakness in Python).

[9] Wikipedia articles,
http://en.wikipedia.org/wiki/Ada_(programming_language)
http://en.wikipedia.org/wiki/RavenSPARK
http://en.wikipedia.org/wiki/Ravenscar_profile
(Useful for Ada history).

[10] Wikipedia article,
http://en.wikipedia.org/wiki/Jean_Ichbiah
(Useful for Ada history).

[11] J. Barnes, *Rationale for Ada 2005*
http://www.adaic.org/standards/05rat/html/Rat-TOC.html
(Useful for Ada examples and dome history).

[12] Wikipedia articles
http://en.wikipedia.org/wiki/Python_(programming_language)
http://en.wikipedia.org/wiki/CPython
(Useful for Python history and characteristics).

[13] Wikipedia article
http://en.wikipedia.org/wiki/Guido_van_Rossum
(Useful for Python history).

[14] Official Website for Python
http://www.python.org/about
http://www.python.org/community/pycon
(Useful for information about the Python community).

[15] Wikipedia article
http://en.wikipedia.org/wiki/Comparison_of_programming_languages_(syntax)
(Useful for listing characteristics of each language).

[16] Wikipedia article
http://www.ibiblio.org/g2swap/byteofpython/read/operator-precedence.html
(Useful for showing Python operator precedence)

[17] Wikipedia article
http://en.wikibooks.org/wiki/Ada_Programming/Operators
(Useful for information about Ada operators)

[18] Course material
http://www.cs.bilkent.edu.tr/~kdincer/teaching/spring1999/bu-bil222-pl/lectures/pdf-files/bil222-chp6-905-04.pdf
(Useful for Ada operator associativity information)

[19] A. Martelli, excerpt from: *Python in a Nutshell*, 2003, O'Reilly Media Inc.,
http://www.onlamp.com/python/excerpt/PythonPocketRef/examples/python.pdf
(Useful as an overview of the Python language)

[20] *PowerAda User's Guide*, OC Systems,
http://www.ocsystems.com/user_guide/powerada/html/powerada-64.html
(Useful for information about Ada data types)

[21] *Python Tutorial*, Penzilla.net
http://www.penzilla.net/tutorials/python/datatypes/
(Useful for information about Python data types)

Appendix 1: Statement Separators

<u>PYTHON</u>

1. To separate Statements on separate lines : *newline*
   (A *semi-colon* is also acceptable at the end of a line, but is not mandatory)

2. To separate Statements on same line : *semi-colon*
   (i.e.: *newline* and *semi-colon* are almost interchangeable in this context)

3. To extend a statement past a *newline* : Escape the newline with a backslash (\)

[1,2]

<u>ADA</u>

1. To separate Statements on separate lines : *semi-colon newline*
   (*semi-colon* mandatory)

2. To separate Statements on same line : *semi-colon*

3. To extend a statement past a *newline* : Just keep typing -- '\n' ignored by the Ada compiler

[3]

Appendix 2: Block Structure

PYTHON

A Block is comprised of a Head statement (e.g.: if, while, def, etc.), followed by a Body of one or more statements that are nested under the Head. If Head returns non-zero, the Body will start to execute.

The format is the following:

```
     <Head Statement>:                          (Note the colon)
          <Body Statement 1>
          <Body Statement 2>
          ...
          <Body Last Statement>
     <Statement outside Block>
```

The indentation above is meaningful. Python uses indentation to parse the Block. The Block ends when the indentation ends.

EXAMPLE:

```
if x==1:
     print 'x is one'
     print 'We are still in the Block'
print 'Now we are outside the Block'
```

It is acceptable to put two Body statements on the same line if they are separated by a semi-colon. They could even be put on the same line as the Head, immediately after the colon [1,2].

ADA

A Block is comprised of a Head statement (e.g.: if, while, procedure, etc.), followed by a Body of one or more statements that are nested under the Head, and a final End statement to close the Block. If Head returns non-zero, the Body will start to execute.

The format in Ada is the following:

```
     <Head Statement>
          <Body Statement 1>;
          <Body Statement 2>;
          ...
          <Body Last Statement>;
     <End statement>;
     <Statement outside Block>;
```

Every statement except the Head must terminate with a semi-colon. It is acceptable to remove all newlines and tabs from the format above. The whitespace is only meaningful when it serves the lexer to disambiguate elements of a statement. 'Body Statement 1' could have begun one space after the Head Statement [3].

EXAMPLE:

```
   loop
      Gnat.IO.Put("Inside the loop");
      X := X + 1;
      exit when X > 10;
   end loop;
   Gnat.IO.Put("Outside the loop");
```

Appendix 3: Operator Precedence

Note: Both of the lists below are sorted from the lowest precedence to the highest, even within a line when not commutative.

PYTHON

lambda	Anonymous function creator
or, and, not	Logical operators
in, not in	Membership tests
is, is not	Identity tests
<, <=, >, >=, !=, ==	Value Comparisons
|, ^, &, <<, >>	Bitwise operators
+, -	Binary arithmetic
*, /, %	Other Binary operators
+, -	Unary arithmetic
~x	Bitwise NOT
**	Exponential
x.attribute	Attribute reference
x[index]	Subscription
f(arguments ...)	Function call
(expressions, ...)	Binding or tuple display
[expressions, ...]	List display
{key:datum, ...}	Dictionary display
\`expressions, ...\`	String conversion

[16]

ADA

and, or, xor	Logical operators
/=, =, <, <=, >, >=	Relational operators
+, -, &	Binary arithmetic
+, -	Unary arithmetic
*, /, mod, rem	Other Binary operators
**	Exponential
not	Logical Not
abs	Absolute Value
and then, or else	shortcuts
in, not in	Membership tests

[17]

Appendix 4: Operator Associativity

PYTHON

lambda	NA
or, and, not	Left
in, not in	Chain
is, is not	Chain
<, <=, >, >=, !=, ==	Chain
|, ^, &, <<, >>	Left
+, - (Binary)	Left
*, /, %	Left
+, - (Unary)	Not Associative (NA)
~x	NA
**	Right
x.attribute	Left
x[index]	Left
f(arguments ...)	Left
(expressions, ...)	NA
[expressions, ...]	NA
{key:datum, ...}	NA
\`expressions, ...\`	NA

[19]

ADA

and, or, xor	Left
/=, =, <, <=, >, >=	Left
+, -, & (Binary)	Left
+, - (Unary)	NA
*, /, mod, rem	Left
**	NA
not	NA
abs	NA

[18]

Appendix 5: Basic Native Data Types

PYTHON

Basic types:

int	4 bytes
long	8 bytes or more as needed
float	4 bytes or more as needed
str	as needed

The more complex types are combinations of the above. They are:

tuple	(Immutable group of possibly inhomogeneous elements)
list	(Mutable group of possibly inhomogeneous elements)
dict	(Hash keyed on str's, values can be any type)

[21]

ADA

"Elementary Types" are the basic native types in Ada. They are:

Signed INTEGER	defined via 'range' value
Unsigned INTEGER (aka Modular)	defined via 'range' value, has wraparound
BOOLEAN	1 byte
CHARACTER	1 byte
FLOAT	defined via no.of digits
ACCESS (similar to pointer)	4 bytes
ENUMERATION	as needed

The more complex types are "Composite Types" in Ada. They are:

array
record (similar to struct)
protected (supplies concurrency control)
task (similar to thread)
[20]

Appendix 6: Ada Code Example

```
with Ada.Text_IO, Ada.Float_Text_Io, Gnat.IO, Ada.Command_Line;
use  Ada.Text_IO, Ada.Float_Text_Io, Gnat.IO, Ada.Command_Line;

procedure two is
   -- INPUT: five floats
   -- OUTPUT: average, roundings, etc.

   -- Accept five floats from command-line
   a:Float:= Float'Value(Argument(1));
   b:Float:= Float'Value(Argument(2));
   c:Float:= Float'Value(Argument(3));
   d:Float:= Float'Value(Argument(4));
   e:Float:= Float'Value(Argument(5));

   -- Rounded versions
   aR:Float:= 0.0;
   bR:Float:= 0.0;
   cR:Float:= 0.0;
   dR:Float:= 0.0;
   eR:Float:= 0.0;

   -- Other Variables
   Sum:Float:= 0.0;
   Average:Float:= 0.0;
   Expression:Float:= 0.0;

   -- create package for output
   package F_IO is new  Ada.Text_IO.Float_IO (Float);

begin

   -- Calculate Average
   Sum := a+b+c+d+e;
   Average := Sum / 5.0;

   Ada.Text_IO.New_Line;
   Gnat.IO.Put("Average:");
   F_IO.Put(Average); -- Print Average
   Ada.Text_IO.New_Line;

   -- Round the floats
   aR:= Float'Rounding(a);
   bR:= Float'Rounding(b);
   cR:= Float'Rounding(c);
   dR:= Float'Rounding(d);
   eR:= Float'Rounding(e);

   -- Print rounding results
   F_IO.Put(a);
   Gnat.IO.Put_Line( " rounds to" & aR'img );
   F_IO.Put(b);
   Gnat.IO.Put_Line( " rounds to" & bR'img );
   F_IO.Put(c);
   Gnat.IO.Put_Line( " rounds to" & cR'img );
   F_IO.Put(d);
   Gnat.IO.Put_Line( " rounds to" & dR'img );
   F_IO.Put(e);
   Gnat.IO.Put_Line( " rounds to" & eR'img );

   -- Calculate Expression
   Expression:= (a-(b*b*b)*(c+d)/(e-a)+1.0)-(a+e)*(a+e);
   Gnat.IO.Put("Expression evaluates to:");
   F_IO.Put(Expression);
   Ada.Text_IO.New_Line;
   Ada.Text_IO.New_Line;

end two;
```

Appendix 7: Ada Output

```
Average: 3.50000E+00
 1.50000E+00 rounds to 2.00000E+00
 2.40000E+00 rounds to 2.00000E+00
 3.60000E+00 rounds to 4.00000E+00
 4.50000E+00 rounds to 5.00000E+00
 5.50000E+00 rounds to 6.00000E+00
Expression evaluates to:-7.44936E+01


Average: 1.20000E+00
 1.00000E+00 rounds to 1.00000E+00
 1.00000E+00 rounds to 1.00000E+00
 1.00000E+00 rounds to 1.00000E+00
 1.00000E+00 rounds to 1.00000E+00
 2.00000E+00 rounds to 2.00000E+00
Expression evaluates to:-9.00000E+00


Average: 2.20000E+00
 5.00000E+00 rounds to 5.00000E+00
 2.00000E+00 rounds to 2.00000E+00
 3.00000E+00 rounds to 3.00000E+00
 1.00000E+00 rounds to 1.00000E+00
 0.00000E+00 rounds to 0.00000E+00
Expression evaluates to:-1.26000E+01


Average: 2.67100E+00
 5.78000E+00 rounds to 6.00000E+00
 2.24000E+00 rounds to 2.00000E+00
 3.33000E+00 rounds to 3.00000E+00
 1.50500E+00 rounds to 2.00000E+00
 5.00000E-01 rounds to 1.00000E+00
Expression evaluates to:-2.23662E+01
```

Appendix 8: Python Code Example

```
#!/usr/bin/python
import string, sys

# Read command line args
a = float(sys.argv[1])
b = float(sys.argv[2])
c = float(sys.argv[3])
d = float(sys.argv[4])
e = float(sys.argv[5])

# Calculate Average
Average = (a+b+c+d+e)/5
print  # just for newline
print "Average: %f" % (Average)
print " %f rounds to %f" % (a,round(a))
print " %f rounds to %f" % (b,round(b))
print " %f rounds to %f" % (c,round(c))
print " %f rounds to %f" % (d,round(d))
print " %f rounds to %f" % (e,round(e))

# Evaluate Expression
print  "Expression  evaluates  to  %f"  %  ((a-(b*b*b)*(c+d)/(e-a)+1.0)-
(a+e)*(a+e))
print
```

Appendix 9: Python Output

```
Average: 3.500000
 1.500000 rounds to 2.000000
 2.400000 rounds to 2.000000
 3.600000 rounds to 4.000000
 4.500000 rounds to 5.000000
 5.500000 rounds to 6.000000
Expression evaluates to -74.493600


Average: 1.200000
 1.000000 rounds to 1.000000
 1.000000 rounds to 1.000000
 1.000000 rounds to 1.000000
 1.000000 rounds to 1.000000
 2.000000 rounds to 2.000000
Expression evaluates to -9.000000


Average: 2.200000
 5.000000 rounds to 5.000000
 2.000000 rounds to 2.000000
 3.000000 rounds to 3.000000
 1.000000 rounds to 1.000000
 0.000000 rounds to 0.000000
Expression evaluates to -12.600000


Average: 2.671000
 5.780000 rounds to 6.000000
 2.240000 rounds to 2.000000
 3.330000 rounds to 3.000000
 1.505000 rounds to 2.000000
 0.500000 rounds to 1.000000
Expression evaluates to -22.366238
```