A brief introduction to Python and RavenSpark(Ada)

by:

Graham Morehead

(First Revision)

October 17, 2008

COS 301, University of Maine

Abstract

A brief history of RavenSpark(Ada) and Python are discussed. Salient features are highlighted and compared. A decimal to binary algorithm is implemented in both. The style, strengths, and weaknesses of the respective projects are analyzed and used to compare the languages.

Table of Contents

A brief introduction to Python and RavenSpark(Ada)

1. Understanding of Python and Ada

In some respects Python and RavenSpark seem similar. They are both high-level, object-oriented, and strongly-typed languages with some mechanism for modularity. However, their differences are marked. Python is easy to learn. RavenSpark is ostensibly optimized for correctness checking, making it a very difficult language to learn. RavenSpark is merely a subset of Ada. It is Ada with the Ravenscar tasking profile[8], a pragma specification, so from this point on the paper will refer to Ada instead of RavenSpark. The specifics of RavenSpark are matter for a more in-depth paper.

Ada is difficult to learn, partly because of stricter rules and partly because of the dearth of easy-to-read online resources. Python is quite easy to learn. When a problem arises It is easier to find sample code and answers to questions than in similar situations with Ada. For a longtime programmer, Python appears to be much more intuitive. It was designed to be easy to learn. Ada was not. Ada was designed with quality as its highest priority. The Ada compiler attempts to catch more errors at compile time than are caught in other languages.

The investigation seeks a high-level understanding of Python and Ada. Each language has its own strengths and weaknesses and its own history. The paper will discuss some differences between the two and some salient features that were found during the investigation.

2. History of Ada

In 1977, Jean Ichbiah directed a team that developed an embedded language called “Green”. It

was influenced most heavily by Pascal, but different revisions also share qualities with Algol, C++, Smalltalk and Java. He submitted Green to the Department of Defense (DoD) as an entry in a competition. It was accepted in 1978 and he continued to lead in its development as it was renamed to "Ada,"[10] in memory of the widely-recognized first computer programmer, Ada Lovelace[9].

The philosophy behind Ada emphasizes high reliability, maintainability, readability over writeability, and reusability. Compile-time error-checking is maximized[3]. This philosophy made it appropriate for avionics and weapons systems. By 1995 over 50 million lines of Ada code were begin maintained by the DoD, with much more planned. Because its compilers can be validated it was the language of choice for mission-critical systems. It was required for all new DoD software starting in 1987, but starting in 1995 it was only required for non-critical systems[6].

In 1983 The American National Standards Institute (ANSI) produced a standard definition of the language. The International Standards Organization (ISO) produced their corresponding document in 1987. Since then it updated in 1995 and 2005. In January 1988 the Ada Board drew up plans for "Ada 9X," which became Ada 95. Christine M. Anderson, then Project Manager, ensured that the needs of the general Ada community were considered, not just those of the DoD. The team managing this effort came from The Ada Board, the Ada Joint Program Office in the DoD, ANSI, and ISO. They addressed weaknesses in interfacing, extensions, libraries, and tasking[5].

Ada 83 was able to interface with other languages and had a good foundation of modularity, but they felt more was needed. Ada 83 was unable to interface with other software whenever it required the passing of procedures as parameters. The team felt that a flexible approach was needed that would also allow the secure manipulation of references. They wanted to add greater ability to extend a program without modifying any proven components. They wanted different library packages to have greater visibility into each other's types, reducing the need for large packages, and they wanted Ada 95 to take advantage of recent developments in scheduling theory[5]. Once they were finished, Ada 95

was almost a new language. They added polymorphism through certain types, a hierarchical library system, protected objects for better real-time control, and more functions in the predefined library and annexes[11].

The 2005 update was a much more modest effort. Technically it is an “amendment,” not a “revision.” Mostly volunteers contributed to this effort with some support from the Ada Resource Association and Ada-Europe. Although the definition consisted of several separately-released documents, and the ISO publication was released in 2007, the name Ada 2005 was kept. The most important improvements concerned the Object Oriented model, flexibility of access types (similar to pointers), visibility into packages, tasking, and real-time operation[11]. Despite the growth in volunteer support, Ada has yet to find wide acceptance. If it weren't for the large effort in 1995, Ada might have become a niche language as was feared by some at the DoD[6].

3. History of Python

In December 1998, Guido van Rossum was looking for a way to occupy himself over Christmas break. He chose to start working an interpreter for a new scripting language he had been thinking about. Feeling a bit irreverent, he named it “Python” after “Monty Python.” It would be a descendant of ABC, which was itself inspired by SETL. He wanted to write a language that would be powerful enough to appeal to Unix/C hackers, but be easy to learn. Within the next year, he submitted a proposal to the Defense Advanced Research Projects Agency (DARPA) outlining his goals for this new language. It would be easy to learn and read, as powerful as competing languages, open source, and suitable for everyday tasks[13].

The greatest emphasis was on programmer-time efficiency. The programming community appears to agree with Guido's priority. Since 1999 his language has been adopted widely. Python

interpreters have been written for many disparate platforms including BeOS, QNX, Nokia phones, and the Nintendo Gamecube[12]. It's not just a Unix creature. In the same open-source spirit that guided the language itself, the Python community started running the "PyCon" Conference in 2003. Historically held in Texas or Washington D.C, it is now being held in several countries[14].

Despite the fact that there is no formal definition of Python, many people have contributed their time to its advancement. Instead of a specification, there are a handful of widely-accepted implementations, the most important of which is CPython, written in C. There is also JPython, written in Java. There are a number of "unofficial" ports as well to platforms such as DOS, Symbian, and Amiga[12]. Python has gathered adherents from a wide swath of the programming community; something Ada has not done.

4. Comparing Ada to Python

The goals for each language limn a direction for discussing the differences between them. I was equally unknowledgable about both languages when beginning this project. The following paragraphs describe my first impressions.

4.1 Programmer Efficiency

The creator of Python wanted it to be easy to use. The creators of Ada wanted their language to be easy to read and find bugs. Python is definitely easier to learn than Ada. A decimal to binary converter was written in Ada and Python. It took a matter of hours to learn enough Python to write it, but the equivalent in Ada took days. Every step took longer because it was more complicated. Even something as simple as creating and using an array required a lot more reading and experimenting in Ada than it did in Python. The Ada compiler (GNAT) had one thing over the python interpreter in that

it gave not only the line but the character of each problem. Even so, debugging took much longer than in Python. It can only be assumed that in a large codebase Ada's stricter rules would help avoid more insidious runtime bugs, but at least on small projects Python is more programmer-efficient. Python's wide acceptance is probably due in large part to the great ease with which it can be learned.

4.2 Reliability

With a desire to use Ada in mission-critical systems, some Ada compilers are supposedly equipped to validate the code they compile. The whole nature of the language should weed out most bugs, and correctness checking should find yet more. All considered, code written in Ada should be the next best thing to real-world testing. In contrast, Python doesn't do nearly as many checks. It is partly for this reason that it is so much easier to write. Types are a simple example of Python's relaxed restrictions. In Python types are dynamic while in Ada they are static. With other grammar constructs there is typically more than one way to do something in Python while in Ada there is usually only one. All of this latitude could potentially open up Python code to a lot more bugs. Surprisingly, more bugs were encountered in the Ada development than in the Python.

A tiny project like the 'decimal2binary' project mentioned above is not a real measure of either language, but for what it's worth, the Python code proved more reliable than the Ada code. At stake was exactly the sort of reliability Ada was created for.

The sort of bug that the Ada compiler is designed to find is a common one where a programmer has implemented an algorithm that may seem correct even after spot-checking, but will produce unpredictable results under certain input conditions because they either exceed the limit on a variable or forgot to check for some other limit within the program. It was exactly this kind of bug that first appeared in the Ada implementation. The compiler didn't find it or even produce a warning.

In the first Ada implementation of *decimal2binary*, whenever the input was above a certain

number the output became unpredictable. It was perplexing at first because there were no errors at compile time or even at run time. The input to the process is a decimal and the output (printed to the screen) is supposed to be binary. Since the output is just being printed to the screen, the internal type didn't matter. The output being printed was an Integer that just happened to have only 1s and 0s. The value of this Integer was overwhelmed once the input reached somewhere around 1500. An input of 1536 produced an output of -1884901888; total nonsense. Besides being grossly inaccurate, the output shouldn't have had anything in it but 1s and 0s. It became obvious that the output Integer was being assigned values that were above its max. This is a typical situation where the programmer of a high-level language comes face-to-face with the vagaries of low-level limitations like the size of the memory alloted to a variable. It is a situation that wasn't supposed to occur without warnings in Ada. It was my mistake, of course, to assign a value to the Integer that was above its max. The question is, why didn't the Ada compiler complain? Why didn't the executable complain at runtime? Instead of exiting with an error or even a warning it continued to produce nonsense.

The code was rewritten to hold the output in a Stack object instead of an Integer. It can now take input as high as 2.1e9 and still provide a reasonable answer. This executable will throw a "CONSTRAINT" error when its input is overwhelmed. This constraint error was not explicitly coded. It is automatically provided to help the programmer. Where was this type of constraint error in the first implementation?

There were no such problems encountered in the Python implementation. No constraint limits were found for the Python version. Several tests are shown in the appendix, including the Python output for a googol (10e100). The following inputs are not included in the appendix but were tried as well: 10e1000, 10e3000, 10e8000. It's not feasible to verify them manually, of course, but the output looked appropriate at first glance. After 10e8000, the limits of the Linux command line stopped further progress because the numbers were being entered literally instead of by scientific notation.

The “max” error in the Ada code was clearly my fault (The limitations of an Integer are evident), but so are most errors. If Ada was designed to help programmers avoid such problems, why didn't an error occur when numbers above the Integer's max were reached? The fact that a novice programmer was easily able to compile code that was clearly wrong leads to speculation that more complex errors could go uncovered as well.

4.3 Object-code efficiency and speed

Python is a scripting language, whereas Ada is compiled like C, so it's difficult to compare their respective efficiencies in this area. The Python version of the project can be compiled into a '.pyc' file which turns out to be much smaller than the compiled Ada project, but it can only be run by invoking the Python interpreter, so it is easy to make it smaller than a true executable. More complex projects will show which one is faster, but Ada should be faster. First, it has no interpreter layer. Second, it was designed for embedded real-time computing.

Ada has a large advantage over Python for concurrent applications. Ada has intrinsic support for concurrency even in its first version, which introduced the high-level idea of a “rendezvous” for interprocess communication[7]. Python, on the other hand, was not designed with concurrency in mind. It appears to be an afterthought and it doesn't work perfectly. The main problem is that CPython has a Global Interpreter Lock (GIL) which only allows one op-code of bytecode to execute at a time. Sharing the GIL across cores is inefficient enough that only one core is ever maximized. Python without the GIL reportedly runs even slower[8].

4.4 Other differences

These other differences are useful in classifying both languages:

SUBJECT	PYTHON	ADA
Paradigms	object oriented, imperative, functional	imperative, object-oriented
Typing	strong, dynamic	strong, static
Whitespace meaningful	Yes	No
Resolution of dangling else	by indent	by “end if”
Parameterized Types	No	Yes
Duck Typing	Yes	No
Case-sensitive	Yes	No
Curly braces and sq.brackets	Yes	No
Influenced by	ABC, SETL, ALGOL 68, C, Haskell, Icon, Lisp, Modula-3, Perl, Java	ALGOL 68, Pascal, C++ (Ada 95), Smalltalk (Ada 95), Java (Ada 2005)

[1,9,12,13]

5. Decimal to Binary conversion

Converting decimals to binary was chosen as a first foray into these languages. The algorithm is simple:

```
input integer X
while (X > 0)
     print X mod 2
     X = X / 2
     move cursor left
```

Moving the cursor left after printing is not easy in most languages, so it was simulated by storing the output in a stack and then printing in reverse order. Another strategy is to store the output in an integer that happens to have only 1s and 0s. The weakness of this second strategy was highlighted earlier, so both implementations use a stack. Each implementation takes input from the command line, initializes a stack array, and then runs through two loops. Loop 1 fills up the array with the output values. Loop 2 prints the values in reverse order. Implementing the same algorithm in both languages shows so far that both are equally expressive, but Python is more laconic and easy to use while coding in Ada was arduous.

The appendices include the code for each implementation and the output first from the Ada version and then from the Python. The output for both is equivalent until the Ada code starts exiting with a constraint error.

6. Conclusion

Not surprisingly, Ada was more difficult to learn. It was surprising, however, just how

challenging it was to accomplish relatively simple tasks. Casting the command line arguments from strings to integers, for instance, was not easy. A Float was considered for use as well because it uses more memory than an Integer, but floats are incompatible with the 'mod' function in Ada. Creating a stack array was worse. There are so many restrictions around creating an unbounded array that the idea was eventually rejected for a large bounded one. The array was never filled it because of the aforementioned input constraint error.

The difficulty in learning Ada is exacerbated by the scarcity of good online resources. There are numerous sites dedicated to Ada, but there aren't as many useful ones as there are for Python. There are fewer blogs and Q&A pages. Ada books are relatively impenetrable, and the language is not intuitive to someone with a heavy Perl background.

The Ada version of decimal2binary took several days to develop. The Python version took several hours. The Ada version was susceptible to various issues and weaknesses. The Python version proved robust. Although this paper highlighted strengths of both languages, it sheds doubt on the DoD's claim that Ada provides a lower life-cycle cost of 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).

Appendix 1: Ada Implementation

Note: comments preceded by "--"

```
-- include for I/O
With Gnat.IO; Use Gnat.IO;
-- include to read from command line args
With Ada.Command_line; Use Ada.Command_Line;

procedure decimal2binary is
   -- INPUT: integer on command line
   -- OUTPUT: the integer in binary
   Input:String:=Argument(1);
   X:Integer:=Integer'Value(Argument(1));
   -- Create stack to hold binary values
   Stack:array (Natural range 1 .. 999) of Integer;
   Top:Natural:=1;

begin
   loop
      -- Take modulo and divide by 2
      Stack(Top) := X mod 2;
      Top := Top + 1;
      X := (X / 2);
      exit when X < 1;
   end loop;

   loop
      -- Print binary values in reverse order
      Top := Top - 1;
      exit when Top < 1;
      Gnat.IO.Put (Stack(Top));  -- Print one digit
   end loop;

   New_Line;
end decimal2binary;
```

Appendix 2: Python Implementation

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

x = int(sys.argv[1]); # Read command line arg
Stack = [];  # Declare an array

# Loop to find binary
while x > 0:
    y = x % 2;
    Stack.extend([y]);
    x = x / 2;

# Loop to print binary
while Stack:
    z = str(Stack.pop(-1));
    sys.stdout.write(z);

print; # just for newline
```

Appendix 3: Execution

```
graham> decimal2binary.py 1
1
graham> decimal2binary.py 2
10
graham> decimal2binary.py 3
11
graham> decimal2binary.py 7
111
graham> decimal2binary.py 29
11101
graham> decimal2binary.py 3000
101110111000
graham> decimal2binary.py 2200000000
10000011001000010101011000000000
graham> decimal2binary.py 10000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000
10010010010011010110100100101100101001100001101111100111010110000101100100111100001001100010011001110011000
00010111111001110001010110011110010000010001110001000010001101001111100101010101011001001000011000010001
01010100000101110100011110001000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000
graham>
graham> cd ../ada
graham> decimal2binary 1
1
graham> decimal2binary 2
10
graham> decimal2binary 3
11
graham> decimal2binary 7
111
graham> decimal2binary 29
11101
graham> decimal2binary 3000
101110111000
graham> decimal2binary.py 2200000000
decimal2binary.py: Command not found.
graham> decimal2binary 2200000000

raised CONSTRAINT_ERROR : s-valint.adb:74 explicit raise
graham> decimal2binary 2100000000
11111010010101101110101000000000
graham>
```