Python

Victor Bos

"And now for something completely different..." Python is a scripting language with clear syntax and semantics, support for object orientation, and an extensive standard library. In contrast with many other scripting languages Python code is readable and, therefore, reusable. This makes Pyhton a useful tool for software development, since it can be used to implement prototypes as well as production versions of applications.

Introduction

Python is a *scripting* or *extension* language similar to Perl [12], Tcl/Tk [8]. In his foreword to *Programming Python* [6] Python's creator Guido van Rossum wrote (See [11]): *"I decided to write an interpreter for the new scripting language I had been thinking about lately: a descendant of ABC that would appeal to Unix/C hackers."* The ABC language, [5], has never become popular, which is partly caused by its peculiar syntax, but it was well designed. In addition to ABC, Python was influenced by Modula-3, an object oriented descendant of Pascal meant for system programming, see[7]. This resulted in a scripting language with clear syntax (which is not common for scripting languages) and powerful language constructs.

Furthermore, Python comes with an extensive standard library that provides the programmer access to a huge set of routines. Therefore, a Python programmer usually does not have to spend much time to implementation details of standard routines like matching a regular expression on a string, or accessing operating system functionality to create processes, pipes, etc. Instead, a Python programmer can just look up the relevant Python *modules* in the standard library and use them to solve her/his problem.

The language

The syntax of Python is quite standard, as will be shown in examples throughout this article. However, there are some controversial aspects. *Indentation* of groups of statements is one of them. If a group starts on a new line, all its statements should be indented by the same number of columns. For example, a while loop is written as:

```
while i<n and f(i)<f(n):
    a[i] = f(i)
    i = i + 1
```

The statements `a[i] = f(i)` and `i = i + 1` form a group. Since indentation is used to indicate groups, no group delimiters like `{` and `}` or `begin`–`end` are needed. Programmers unfamiliar with Python might find this irritating, however, it is not a drawback. Experienced programmers (in no matter what language) have usually adopted their own style of indentation for groups of statements. Since Python does not prescribe the number of columns of indentation, these people can keep on using their own style in Python. Furthermore, the code does not get messed up with group delimiters.

Build-in data structures Python has the following data structures build-in: integers, floats, strings, tuples, lists, dictionaries, and functions. There are no booleans, which is a shortcoming not only of Python but of most scripting languages. Integers, floats, and strings are standard data structures which we will not discuss here. A tuple is an *immutable*

sequence of elements, that is, it is a sequence of which the elements cannot change once the tuple is created. Lists are mutable sequences of elements; elements can be added to and removed from lists. A very powerful build-in data structure is the *dictionary*. A dictionary is a look-up table or associative array containing key-value pairs. Hashing is used to look up a key in a dictionary which means dictionaries have fast access times. Finally, functions are first-class objects in Python. Therefore, Python programs can be a mix of functional and imperative programs. A *lambda-syntax*, known from many functional programming languages, is used to denote anonymous function. For example, the function that adds two elements could be written in Python as `lambda x, y: x + y`. Since this is a normal object, it can be assigned to variables, as will be shown later.

Universal object model Python has a *universal object model*, which means that every piece of data in a Python program is an object. As usual in object oriented programming languages, an object has attributes that define the state of the object and methods to allow other objects to perform operations on the object. For example, the following Python code defines a class `Point` of objects with an x and a y coordinate and a method `dist` to compute the distance between two objects.

```
class Point:

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def dist(self, other):
        dx = self.x - other.x
        dy = self.y - other.y
        return (dx**2 + dy**2) ** .5
```

In the definition of `dist`, the two argument `**` operator is used; `x ** y` raises `x` to the power `y`.

The example shows at least two syntactic peculiarities. First of all, Python has special syntax for special methods like the `__init__` method. The special syntax, which in my opinion is quite ugly, is an identifier that starts and ends with two underscores. The `__init__` method is special, since it is a constructor of the `Point` class and will be called whenever a point is created, for example, the point $(1, 2)$ is created by calling `Point(1,2)`. Note that the coordinate arguments of `__init__` have default values, `x=0` and `y=0`, so the point $(0, 0)$ could be created by calling `Point()`. Other special methods are used to overload operators and build-in functions. For example, the `__add__` method can be used to overload the `+` operator. By extending the `Point` class with the following definition of `__add__`, we can write `p1 + p2` in order to add the points `p1` and `p2`.

```
def __add__(self, other):
    return Point(self.x + other.x,
                 self.y + other.y)
```

The other strange part of the examples above is the `self`-parameter of `__init__`, `dist`, and `__add__`. This parameter is a self-reference to the object on which the method is invoked. Whereas in most object-oriented languages there is usually no need to make the reference to an object itself explicit, in Python it is. Furthermore, the self-reference is always the first parameter of the method. By convention it is called `self`, but the programmer is free to choose another identifier.

So, in a constructor (`__init__`) `self` refers to the object that is created and in a normal method (`dist` or `__add__`) `self` refers to the object on which the method is invoked. In some programming languages, `this` is used instead of `self`, e.g., C++ [9] and Java [1].

Unlike many object-oriented programming languages, the set of attributes and the set of methods of an object are not constant during its lifetime. For example, the following code creates a `Point` object, changes its x-coordinate, and adds a color attribute.

```
p = Point()
p.x = p.x + 4
p.color = "yellow"
```

Programming styles Python supports three programming styles: procedural, object-oriented, and functional programming. Furthermore, these styles can be mixed arbitrarily. Of course, an unrestricted mix of these three styles will not improve readability and maintainability of the program and it is therefore wise to stick to one style as much as possible. However, programming styles are meant to ease programming and not to restrict the freedom of the programmer. Therefore, if in a given situation one particular style is not adequate, it should be possible to switch to another style. Python supports programming using multiple styles, whereas a

pure functional language or a pure object oriented language does not.

The following Python listing is an example showing the three programming styles. First we take the `Point` class again and extend it with the special method `__str__`. This method will be called if a `Point` object should be represented by a string, e.g., in order to print it. After the class definition, two functions are defined: `closerToOrig` and `findMax`. The functions are not part of the `Point` class, because their indentation is not the same as the indentation of the class body. The function `closerToOrig` takes two points and determines if the first is closer to the origin, i.e., `Point(0,0)`, than the second. The `findMax` function is a generic function that takes a non-empty list of elements and a compare function `lessthan`. The compare function determines if its first argument is less than its second argument. Note that `closerToOrig` is such a compare function.

```
class Point:

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def dist(self, other):
        dx = self.x - other.x
        dy = self.y - other.y
        return (dx**2 + dy**2) ** .5

    def __str__(self):
        return ("(" + str(self.x) +
                ", " + str(self.y) +
                ")")

def closerToOrig(p0,p1):
    return (p0.dist(Point(0,0)) <
            p1.dist(Point(0,0)))

def findMax(list, lessthan):
    if len(list)>0:
        m = list[0]
        for i in list[1:]:
            if lessthan(m, i):
                m = i
        return m
    else:
        print "No max in empty list"
```

Given a list of elements and a suitable compare function on the elements, `findMax` finds a maximal element in the list with respect to the compare function. For instance, given a list of points, `findMax` can be used to determine a point that is at least as far from the origin as all other points. For example, consider the following Python code. On the first line, a list `l` of three points is created. On the second line, this list is printed. The `map` function takes a function and a list and applies the function on each element in the list. The function `str` returns a string representation of its argument. If applied to a `Point`, it calls the special method `__str__` defined above. The third line creates a list of numbers representing the distance of the points in list `l` to the origin. Finally, the fifth line calls the `findMax` function with arguments `l` and `closerToOrig` in order to find a point in `l` that is at least as for from the origin as all other points in `l`.

```
l = [Point(3,4), Point(), Point(2,1)]
print map(str, l)
d = map(lambda x: x.dist(Point()), l)
print map(str, d)
m = findMax(l, closerToOrig)
print m
```

The output of this Python code is:

```
['(3, 4)', '(0, 0)', '(2, 1)']
['5.0', '0.0', '2.2360679775']
(3, 4)
```

Standard library

Python comes with an extensive standard library organized in *modules* and *packages*. Furthermore, the standard library is mostly platform independent. People familiar with Java will recognize much of the functionality, like network programming, threads, and a standard windowing toolkit. In addition, it includes modules that define Perl-like regular expressions and powerful string operations. In this section, I will discuss some functionality of Python's standard library. For more detailed information, see [6, 2].

Internet Internet programming is one of the most important application domains of Python. One of the reasons for Python's popularity is that the standard library provides functionality by which both server and client side Internet applications can be written. For example, the modules `urlparse` and `mimetools` provide functionality to manipulate url strings and mime encoded messages, respectively. In addition to these modules, there are modules to process HTML, XML, and SGML documents,

modules that provide HTTP servers, and modules to write CGI scripts. The fact that many CGI scripts are written in Python and that there exist full size web-applications, like Zope (http://www.zope.org/), shows that Python is popular among internet application programmers.

Operating system services Python has build in functionality to read and write files. In addition, the standard library offers functionality to handle files and directories, sub-processes, streams, and pipes. The sub-processes need not be Python programs, but can be any program that runs on your system. In this way, Python can be used to control different applications or as a communication means between different applications.

Profiling Python comes with a *deterministic profiler*. The online Python reference describes deterministic profiling as follows:

Deterministic profiling is meant to reflect the fact that all function call, function return, and exception events are monitored, and precise timings are made for the intervals between these events (during which time the user's code is executing). In contrast, statistical profiling randomly samples the effective instruction pointer, and deduces where time is being spent. The latter technique traditionally involves less overhead (as the code does not need to be instrumented), but provides only relative indications of where time is being spent.

A profiler is an important tool for an extensible scripting language, since it enables software developers to analyze an application thoroughly and make the right decisions about which routines are time critical and should be implemented in a system programming language, and which routines are less time critical and can therefore be written in the scripting language. Below, I will explain the possible role of the Python profiler in a software development process.

Serialization Serialization is the transformation of a (run-time) data structure into a sequence of bytes such that it is possible to recover the original data structure from the sequence of bytes. In Python's standard library, several modules exist to serialize arbitrary objects. Furthermore, serialization is platform independent. Therefore, it is quite easy to store the current state of an application as a sequence of bytes in a file, transfer it to another computer (which also runs Python), and to continue with the application in the same state on that computer. Usually, serialization is applied not to complete applications, but to some crucial data structures of the applications that should be available the next time the application is executed.

Threads Python supports multi-threaded applications. The threading modules resemble to some extent the threading mechanism of Java. Multi threading is very useful for writing server applications. For example, an HTTP server is usually written using multiple threads. In its main loop it waits for a client to make a connection. As soon as a client makes a request, the server creates a new thread that processes the request. During the processing of the new thread, the main loop is ready to accept a new request.

Windowing toolkit A widely seen application of Python is for writing graphical user interfaces. Since the standard library has a windowing toolkit, named by `tkinter` and derived from Tcl/Tk's UI widgets, writing a user interface in Python has the advantage of being platform independent.

Python glue

One of the goals of Python is to act as a glue language that connects different applications and libraries. To be more precise, Python was developed to be used in an open environment in which Python programs could be integrated with non-Python programs. Therefore, Python was developed to be embed-able as well as extensible and interfaces of how to embed and extend Python are well documented, see http://www.python.org/doc/current/ext/ext.html, Chapters 14 and 15 of [6], or Appendix B of [2]. As a glue language, Python greatly facilitates reuse of existing code, for example, see [3].

Embedding Python means integrating the Python interpreter in another application such that Python programs can be run from within the other application. This effective adds all of Python's scripting power to the hosting application. Extending

Python means integrating applications or libraries in the Python interpreter such that its is available from within Python programs. It is possible to embed and extend Python at the same time. As usual, such a union based on equality can be very fruitful.

If Python is used to glue applications and libraries together, care should be taken that it does not replace techniques especially designed to act as an interface between software components. In fact, using Python as a glue language and using a standardized interface technique should be orthogonal design decisions. For example, if the application is supposed to be available at some *object market*, see [10], its interface should be defined using a standardized interface technique, e.g., CORBA or XML, instead of Python.

So, if there are good arguments to use CORBA in a situation where Python is not used for integration, then it should still be used if Python is used for integration. This claim can be turned around as well: if Python can be used for integration, then using a standardized interface technique is probably too much overhead. As is explained below, integrating existing code with Python requires the interface (C/C++ header files) of the code to be available which can be problematic in a commercial environment. However, in that case, integration without Python is at least as big a problem.

A prerequisite of extending Python with a given library is that the interface of the library is defined in C-header files or that the source code is available in C or C++. This is a limitation, since there are useful libraries out there for which no C-header files exists. However, for almost any subject there exist C and C++ libraries as well or if the source is available in, say, Fortran, then writing a C-header file for it is not too difficult. Furthermore, if Python should be integrated with Java applications, one should consider using *Jython*: a Python implementation written in Java, see http://www.jython.org. It is said that Jython-Java integration is better than the conventional Python-C/C++ integration, since no recompilation of Java code is needed due to Java's reflection API. However, since I have no experience with Jython, I will only focus on the Python-C/C++ combination.

Extending Python with an existing library effectively means that a wrapper for the library has to be created and together with the wrapper, the library has to be turned in an object file that can be loaded dynamically e.g., shared libraries or DLLs, or that is linked statically with the Python interpreter. The wrapper should take care of the translation between data structures of Python and the data structures of the library. The conversion between C/C++ and Python data structures is documented extensively and, therefore, after some reading, not difficult.

Tools have been developed that create wrappers automatically. SWIG is one of such tools and stands for *Simplified Wrapper and Interface Generator*, see http://www.swig.org/. SWIG is not just a tool to create wrappers and interfaces for Python, it can also generate interfaces for other languages, e.g., Perl and Tcl/Tk. SWIG comes with extensive documentation and the SWIG user guide (available on http://www.swig.org/doc.html) has devoted one chapter to the combination of SWIG and Python.

Example of a Python Extension Since extensibility of Python is one of its most powerful features, I spend the remainder of this section to describe my experience with extending Python with an 'off-the-shelve' BDD library. A BDD (*binary decision diagram*) is a data structure to store boolean functions [4] space efficiently. For this article, it is not necessary to explain BDDs, but it suffices to give some examples of what can be done with BDDs. First of all, BDDs manipulate boolean function symbolically. For example, given a BDD for two boolean functions f_0 and f_1, there are BDD operations to compute a BDD for the function $\mathit{and}(f_0, f_1)$ defined by

$$\mathit{and}(f_0, f_1)(b) = f_0(b) \wedge f_1(b).$$

There are also operations to compute other common boolean operations, like $\vee$, $\rightarrow$, etc. In addition to these symbolic operations on boolean functions, a BDD library provides routines to determine if a boolean function (represented by a BDD) can return *true* for some concrete values of its arguments. That is, there are routines that determine if a boolean formula can be satisfied. Given that almost any problem defined formally can be translated into a problem defined in boolean formulas, BDD libraries can be used an many areas. Historically, BDDs have been applied mostly to tasks in digital system design, verification, and testing.

The BDD library I chose is called *BuDDy* and its

source code is freely available. It can be downloaded from http://www.itu.dk/research/buddy/. There is no good reason why I chose this BDD package; it just happened to be the first package I found that was freely available and installed without problems on my machine. BuDDy is written in C and has some additional definitions to use it in C++.

Extending Python with BuDDy was not a complicated task, thanks to SWIG. The main difficulties were in dealing with pointer arguments and function pointers, since SWIG does not process them automatically. So, in these cases I had to write some extra code in a so-called SWIG interface file. After that, SWIG generates the wrappers which could be compiled and linked with the original BuDDy code into a Python module. Note that the BuDDy code is left unchanged

So, the functionality of BuDDy is now available to Python programs. However, it is at a somewhat low level; python programs directly call C functions to generate and manipulate BDDs. Furthermore, since garbage collection of objects created by BuDDy is left to the programmer, the Python code quickly becomes a unreadable mess of function calls and temporary variables. Note that this is more a problem of BuDDy than of Python; the C-examples that come with BuDDy exhibit the same mess of function calls and temporary variables. To make it better accessible, BuDDy has a C++ class that takes care of automatic garbage collection and overloads some operators such that function calls can be written as operator applications. I did the same in Python and wrote a class that defines BDDs as normal Python objects. Also, I overloaded some Python operators in the same way the C++ class did. As a result, the Python code is at least as readable as the C++ code. For example, the following listing shows some lines of C++ code of an implementation of the N-queens problem in C++ using BuDDy (here, `X` is a two dimensional array of bdds and `a`, `b`, `c`, and `d` are bdd variables):

```
bdd a=bddtrue,
    b=bddtrue,
    c=bddtrue,
    d=bddtrue;
int k,l;

/* No one in the same column */
for (l=0 ; l<N ; l++)
   if (l != j)
      a = a & (X[i][j]
               >> !X[i][l]);
```

The corresponding lines of Python code for the N-queens problem reads:

```
a = bddtrue
b = bddtrue
c = bddtrue
d = bddtrue

# No one in the same column
for l in range(0,N):
    if (l != j):
        a = a & (X[i][j]
                 >>  -X[i][l])
```

Software development with Python

Sometimes, scripting languages are said to be good for prototyping, but not for real application development. A prototype bears the associations of 'quick and dirty' and 'to be thrown away.' However, Python is more than just a prototype language. Due to its clear syntax and its universal object model, reuse of Python programs is a very attractive option. Therefore, a substantial part of Python code of a prototype of an application could very well end up in the code of the final application.

So, what is Python's role in a software development process? First of all, it can be used for prototyping; like any scripting language, it enables programmers to write quickly a mock up of an application in order to analyze the feasibility of the project.

A simplified and Python centered, view on software development could be described as follows. Firstly, determine user requirements of the application and build a prototype in Python. Next, a development cycle is started that consist of assessment of the prototype, estimation of costs of improving the prototype, and finally a decision whether to improve the prototype or to abort the cycle and declare the current prototype the final application.

During each cycle, assessment of the prototype can lead to new or more precise requirements. Analysis of the prototype shows, among others, computation intensive code, which could be implemented in a system programming language. The Python profiler is very useful to detect computation intensive code. If, after some runs of the development cycle, all computation intensive code is implemented in a system programming language, there is probably not much more speed to gain. At that time, it

is a waste of time to translate the remaining Python code into a system programming language.

Conclusions

In this article I have discussed the Python language. Python is a scripting language with clear syntax and an extensive standard library. It supports, but does not enforce, procedural, functional, and object oriented programming styles. Unlike many other scripting languages, Pyhton code is readable and, therefore, reusable. Reusability is even more supported by Python's platform independence. Python can play an important role in software development, since it is a powerful tool for prototyping as well as for implementing the final application. If the application contains computational intensive code, which will be too slow if programmed in a scripting language like Python, the extension interface of Python makes it very easy to implement this code in a system programming language like C/C++. Furthermore, together with its embedding interface, the extension interface of Python enables efficient integration with existing applications and libraries.

References

[1] Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, 2nd edition, 1997.

[2] David M. Beazley. *Python Essential Reference*. New Riders, 2000.

[3] David M. Beazley and Peter S. Lomdahl. Feeding a large-scale physics application to python. In *Proceedings of the 6th International Python Conference*, San Jose, California, October 1997.

[4] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[5] Leo Geurts, Lambert Meertens, and Steven Pemberton. *The ABC Programmer's Handbook.* Prentice-Hall, 1990. To be republished by the CWI. See also http://www.cwi.nl/ steven/abc/.

[6] Mark Lutz. *Programming Python.* O'Reilly & Associates, first edition, October 1996.

[7] Greg Nelson, editor. *System Programming with Modula-3.* Series in Innovative Technology. Prentice Hall, 1991.

[8] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[9] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, special edition, 2000.

[10] Clemens Szyperski. *Component Software, Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[11] Guido van Rossum. Foreword for Programming Python, May 1996. See [6]. Also available on http://www.python.org/doc/essays/foreword.html.

[12] Larry Wall, Tom Christiansen, and Randal L. Schwarz. *Programming Perl.* O'Reilly & Associates, 2nd edition, 1996.

Biography Since January 1998, Victor Bos is a PhD student at the Eindhoven University of Technology. He is involved in formal methods research at the computer science department. His current interest lies in applying formal method techniques to industrial engineering and therefore he works closely together with the Systems Engineering Group at the department of Mechanical Engineering. He was an OOTI from 1996–1998. In December 1995, he received his masters degree in computer science at the University of Groningen.