The History of Python blog articles

Introduction and Overview

by Guido van Rossum
Tuesday, January 13, 2009

Introduction

Python is currently one of the most popular dynamic programming languages, along with Perl, Tcl, PHP, and newcomer Ruby. Although it is often viewed as a "scripting" language, it is really a general purpose programming language along the lines of Lisp or Smalltalk (as are the others, by the way). Today, Python is used for everything from throw-away scripts to large scalable web servers that provide uninterrupted service 24x7. It is used for GUI and database programming, client- and server-side web programming, and application testing. It is used by scientists writing applications for the world's fastest supercomputers and by children first learning to program.
In this blog, I will shine the spotlight on Python's history. In particular, how Python was developed, major influences in its design, mistakes made, lessons learned, and future directions for the language.

Acknowledgment: I am indebted to Dave Beazley for many of the better sentences in this blog. (For more on the origins of this blog, see my other blog.)

A Bird's Eye View of Python

When one is first exposed to Python, they are often struck by way that Python code looks, at least on the surface, similar to code written in other conventional programming languages such as C or Pascal. This is no accident---the syntax of Python borrows heavily from C. For instance, many of Python's keywords (if, else, while, for, etc.) are the same as in C, Python identifiers have the same naming rules as C, and most of the standard operators have the same meaning as C. Of course, Python is obviously not C and one major area where it differs is that instead of using braces for statement grouping, it uses indentation. For example, instead of writing statements in C like this

```
if (a < b) {
   max = b;
} else {
   max = a;
}
```

Python just dispenses with the braces altogether (along with the trailing semicolons for good measure) and uses the following structure

```
if a < b:
   max = b
else:
   max = a
```

The other major area where Python differs from C-like languages is in its use of dynamic typing. In C, variables must always be explicitly declared and given a specific type such as int or double. This information is then used to perform static compile-time checks of the program as well as for allocating memory locations used for storing the variable’s value. In Python, variables are simply names that refer to objects. Variables do not need to be declared before they are assigned and they can even change type in the middle of a program. Like other dynamic languages, all type-checking is performed at run-time by an interpreter instead of during a separate compilation step.

Python’s primitive built-in data types include Booleans, numbers (machine integers, arbitrary-precision integers, and real and complex floating point numbers), and strings (8-bit and Unicode). These are all immutable types, meaning that values are represented by objects that cannot be modified after their creation. Compound built-in data types include tuples (immutable arrays), lists (resizable arrays) and dictionaries (hash tables).

For organizing programs, Python supports packages (groups of modules and/or packages), modules (related code grouped together in a single source file), classes, methods and functions. For flow control, it provides if/else, while, and a high-level for statement that loops over any “iterable” object. For error handling, Python uses (non-resumable) exceptions. A raise statement throws an exception, and try/except/finally statements specify exception handlers. Built-in operations throw exceptions when error conditions are encountered.

In Python, all objects that can be named are said to be "first class." This means that functions, classes, methods, modules, and all other named objects can be freely passed around, inspected, and placed in various data structures (e.g., lists or dictionaries) at run-time. And speaking of objects, Python also has full support for object-oriented programming including user-defined classes, inheritance, and run-time binding of methods.

Python has an extensive standard library, which is one of the main reasons for its popularity. The standard library has more than 100 modules and is always evolving. Some of these modules include regular expression matching, standard mathematical functions, threads, operating systems interfaces, network programming, standard internet protocols (HTTP,FTP, SMTP, etc.), email handling, XML processing, HTML parsing, and a GUI toolkit (Tcl/Tk).

In addition, there is a very large supply of third-party modules and packages, most of which are also open source. Here one finds web frameworks (too many to list!), more GUI toolkits, efficient numerical libraries (including wrappers for many popular Fortran packages), interfaces to relational databases (Oracle, MySQL, and others), SWIG (a tool for making arbitrary C++ libraries available as Python modules), and much more.

A major appeal of Python (and other dynamic programming languages for that matter) is that seemingly complicated tasks can often be expressed with very little code. As an example, here is a simple Python script that fetches a web page, scans it looking for URL references, and prints the first 10 of those.

```
# Scan the web looking for references

import re
import urllib

regex = re.compile(r'href="([^"]+)"')

def matcher(url, max=10):
   "Print the first several URL references in a given url."
```

```
    data = urllib.urlopen(url).read()
    hits = regex.findall(data)
    for hit in hits[:max]:
        print urllib.basejoin(url, hit)

matcher("http://python.org")
```

This program can easily be modified to make a web crawler, and indeed Scott Hassan has told me that he wrote Google's first web crawler in Python. Today, Google employs millions of lines of Python code to manage many aspects of its operations, from build automation to ad management (Disclaimer: I am currently a Google employee.)

Underneath the covers, Python is typically implemented using a combination of a bytecode compiler and interpreter. Compilation is implicitly performed as modules are loaded, and several language primitives require the compiler to be available at run-time. Although Python's de-facto standard implementation is written in C, and available for every imaginable hardware/software platform, several other implementations have become popular. Jython is a version that runs on the JVM and has seamless Java integration. IronPython is a version for the Microsoft .NET platform that has similar integration with other languages running on .NET. PyPy is an optimizing Python compiler/interpreter written in Python (still a research project, being undertaken with EU funding). There's also Stackless Python, a variant of the C implementation that reduces reliance on the C stack for function/method calls, to allow co-routines, continuations, and microthreads.

Python's Design Philosophy

by Guido van Rossum
Tuesday, January 13, 2009

Later blog entries will dive into the gory details of Python's history. However, before I do that, I would like to elaborate on the philosophical guidelines that helped me make decisions while designing and implementing Python.

First of all, Python was originally conceived as a one-person "skunkworks" project – there was no official budget, and I wanted results quickly, in part so that I could convince management to support the project (in which I was fairly successful). This led to a number of timesaving rules:

- Borrow ideas from elsewhere whenever it makes sense.
- "Things should be as simple as possible, but no simpler." (Einstein)
- Do one thing well (The "UNIX philosophy").
- Don't fret too much about performance--plan to optimize later when needed.
- Don't fight the environment and go with the flow.
- Don't try for perfection because "good enough" is often just that.
- (Hence) it's okay to cut corners sometimes, especially if you can do it right later.

Other principles weren't intended as timesavers. Sometimes they were quite the opposite:

- The Python implementation should not be tied to a particular platform. It's okay if some functionality is not always available, but the core should work everywhere.
- Don't bother users with details that the machine can handle (I didn't always follow this rule and some of the of the disastrous consequences are described in later sections).
- Support and encourage platform-independent user code, but don't cut off access to platform capabilities or properties (This is in sharp contrast to Java.)
- A large complex system should have multiple levels of extensibility. This maximizes the opportunities for users, sophisticated or not, to help themselves.
- Errors should not be fatal. That is, user code should be able to recover from error conditions as long as the virtual machine is still functional.
- At the same time, errors should not pass silently (These last two items naturally led to the decision to use exceptions throughout the implementation.)

- A bug in the user’s Python code should not be allowed to lead to undefined behavior of the Python interpreter; a core dump is never the user’s fault.

Finally, I had various ideas about good programming language design, which were largely imprinted on me by the ABC group where I had my first real experience with language implementation and design. These ideas are the hardest to put into words, as they mostly revolved around subjective concepts like elegance, simplicity and readability.

Although I will discuss more of ABC's influence on Python a little later, I’d like to mention one readability rule specifically: punctuation characters should be used conservatively, in line with their common use in written English or high-school algebra. Exceptions are made when a particular notation is a long-standing tradition in programming languages, such as “x*y” for multiplication, “a[i]” for array subscription, or “x.foo” for attribute selection, but Python does not use “$” to indicate variables, nor “!” to indicate operations with side effects.

Tim Peters, a long time Python user who eventually became its most prolific and tenacious core developer, attempted to capture my unstated design principles in what he calls the “Zen of Python.” I quote it here in its entirety:

- Beautiful is better than ugly.

- Explicit is better than implicit.

- Simple is better than complex.

- Complex is better than complicated.

- Flat is better than nested.

- Sparse is better than dense.

- Readability counts.

- Special cases aren't special enough to break the rules.

- Although practicality beats purity.

- Errors should never pass silently.

- Unless explicitly silenced.

- In the face of ambiguity, refuse the temptation to guess.

- There should be one-- and preferably only one --obvious way to do it.

- Although that way may not be obvious at first unless you're Dutch.

- Now is better than never.
- Although never is often better than right now.
- If the implementation is hard to explain, it's a bad idea.
- If the implementation is easy to explain, it may be a good idea.
- Namespaces are one honking great idea -- let's do more of those!

Although my experience with ABC greatly influenced Python, the ABC group had a few design principles that were radically different from Python’s. In many ways, Python is a conscious departure from these:

- The ABC group strived for perfection. For example, they used tree-based data structure algorithms that were proven to be optimal for asymptotically large collections (but were not so great for small collections).

- The ABC group wanted to isolate the user, as completely as possible, from the “big, bad world of computers” out there. Not only should there be no limit on the range of numbers, the length of strings, or the size of collections (other than the total memory available), but users should also not be required to deal with files, disks, “saving”, or other programs. ABC should be the only tool they ever needed. This desire also caused the ABC group to create a complete integrated editing environment, unique to ABC (There was an escape possible from ABC’s environment, for sure, but it was mostly an afterthought, and not accessible directly from the language.)

- The ABC group assumed that the users had no prior computer experience (or were willing to forget it). Thus, alternative terminology was introduced that was considered more “newbie-friendly” than standard programming terms. For example, procedures were called “how-tos” and variables “locations”.

- The ABC group designed ABC without an evolutionary path in mind, and without expecting user participation in the design of the language. ABC was created as a closed system, as flawless as its designers could make it. Users were not encouraged to “look under the hood”. Although there was talk of opening up parts of the implementation to advanced users in later stages of the project, this was never realized.

In many ways, the design philosophy I used when creating Python is probably one of the main reasons for its ultimate success. Rather than striving for perfection, early adopters found that Python worked "well enough" for their purposes. As the user-base grew, suggestions for improvement were gradually incorporated into the language. As we will seen in later sections, many of these improvements have involved substantial changes and reworking of core parts of the language. Even today, Python continues to evolve.

A Brief Timeline of Python

by Guido van Rossum
Tuesday, January 20, 2009

The development of Python occurred at a time when many other dynamic (and open-source) programming languages such as Tcl, Perl, and (much later) Ruby were also being actively developed and gaining popularity. To help put Python in its proper historical perspective, the following list shows the release history of Python. The earliest dates are approximate as I didn't consistently record all events:

Release Date	**Version**
December, 1989	Implementation started
1990	Internal releases at CWI
February 20, 1991	0.9.0 (released to alt.sources)
February, 1991	0.9.1
Autumn, 1991	0.9.2
December 24, 1991	0.9.4
January 2, 1992	0.9.5 (Macintosh only)
April 6, 1992	0.9.6
Unknown, 1992	0.9.7beta
January 9, 1993	0.9.8
July 29, 1993	0.9.9
January 26, 1994	1.0.0
February 15, 1994	1.0.2
May 4, 1994	1.0.3
July 14, 1994	1.0.4
October 11, 1994	1.1
November 10, 1994	1.1.1
April 13, 1995	1.2
October 13, 1995	1.3
October 25, 1996	1.4
January 3, 1998	1.5
October 31, 1998	1.5.1
April 13, 1999	1.5.2
September 5, 2000	1.6
October 16, 2000	2.0
April 17, 2001	2.1
December 21, 2001	2.2
July 29, 2003	2.3
November 30, 2004	2.4
September 16, 2006	2.5
October 1, 2008	2.6
December 3, 2008	3.0

I've added hyperlinks to the releases that are still being advertised on python.org at this time. Note that many releases were followed by several micro-releases, e.g. 2.0.1; I haven't bothered to include these in the table as otherwise it would become too long. Source tarball of very old releases are also still accessible, here: http://www.python.org/ftp/python/src/. Various ancient binary releases and other historical artefacts can still be found by going one level up from there.

Personal History - part 1, CWI

by Guido van Rossum
Tuesday, January 20, 2009

Python's early development started at a research institute in Amsterdam called CWI, which is a Dutch acronym for a phrase that translates into English as Centre for Mathematics and Computer Science. CWI is an interesting place; funded by the Dutch government's Department of Education and other research grants, it conducts academic-level research into computer science and mathematics. At any given time there are plenty of Ph.D. students wandering about and old-timers in the profession may still remember its original name, the Mathematical Centre. Under this name, it was perhaps most famous for the invention of Algol 68.

I started working at CWI in late 1982, fresh out of university, as a programmer in the ABC group led by Lambert Meertens and Steven Pemberton. After 4 or 5 years the ABC project was terminated due to the lack of obvious success and I moved to CWI's Amoeba group led by Sape Mullender. Amoeba was a micro-kernel-based distributed system being jointly developed by CWI and the Vrije Universiteit of Amsterdam, under leadership of Andrew Tanenbaum. In 1991 Sape left CWI for a professorship at the University of Twente and I ended up in the newly formed CWI multimedia group led by Dick Bulterman.

Python is a direct product of my experience at CWI. As I explain later, ABC gave me the key inspiration for Python, Amoeba the immediate motivation, and the multimedia group fostered its growth. However, so far as I know, no funds at CWI were ever officially earmarked for its development. Instead, it merely evolved as an important tool for use in both the Amoeba and multimedia groups.

My original motivation for creating Python was the perceived need for a higher level language in the Amoeba project. I realized that the development of system administration utilities in C was taking too long. Moreover, doing these in the Bourne shell wouldn't work for a variety of reasons. The most important one was that as a distributed micro-kernel system with a radically new design, Amoeba's primitive operations were very different (and finer-grain) than the traditional primitive operations available in the Bourne shell. So there was a need for a language that would "bridge the gap between C and the shell." For a long time, this was Python's main catchphrase.

At this point, you might ask "why not port an existing language?" In my view, there weren't a lot of suitable languages around at that time. I was familiar with Perl 3, but it was even more tied to Unix than the Bourne shell. I also didn't like Perl's syntax--my tastes in programming language syntax were strongly influenced by languages like Algol 60, Pascal, Algol 68 (all of which I had learned early on), and last but not least, ABC, on which I'd spent four years of my life. So, I decided to design a language of my own which would borrow everything I liked from ABC while at the same time fixing all its problems (as I perceived them).

The first problem I decided to fix was the name! As it happened, the ABC team had some trouble picking a name for its language. The original name for the language, *B*, had to be abandoned because of confusion with another language named B, that was older and better known. In any case, *B* was meant as a working title only (the joke was that *B* was the name of the variable containing the name of the language--hence the italics). The team had a public contest to come up with a new name, but none of the submissions made the cut, and in the end, the internal back up candidate prevailed. The name was meant to convey the idea that the language made programming "as simple as ABC", but it never convinced me all that much.

So, rather than over-analyzing the naming problem, I decided to under-analyze it. I picked the first thing that came to mind, which happened to be Monty Python's Flying Circus, one of my favorite comedy troupes. The reference felt suitably irreverent for what was essentially a "skunkworks project". The word "Python" was also catchy, a bit edgy, and at the same time, it fit in the tradition of naming languages after famous people, like Pascal, Ada, and Eiffel. The Monty Python team may not be famous for their advancement of science or technology, but they are certainly a geek favorite. It also fit in with a tradition in the CWI Amoeba group to name programs after TV shows.

For many years I resisted attempts to associate the language with snakes. I finally gave up when O'Reilly wanted to put a snake on the front of their first Python book "Programming Python". It was an O'Reilly tradition to use animal pictures, and if it had to be an animal, it might as well be a snake.

With the naming issue settled, I started working on Python in late December 1989, and had a working version in the first months of 1990. I didn't keep notes, but I remember vividly that the first piece of code I wrote for Python's implementation was a simple LL(1) parser generator I called "pgen." This parser generator is still part of the Python source distribution and probably the least changed of all the code. This early version of Python was used by a number of people at CWI, mostly, but not exclusively in the Amoeba group during 1990. Key developers besides myself were my officemates, programmers Sjoerd Mullender (Sape's younger brother) and Jack Jansen (who remained one of the lead developers of the Macintosh port for many years after I left CWI).

On February 20, 1991, I first released Python to the world in the alt.sources newsgroup (as 21 uuencoded parts that had to be joined together and uudecoded to form a compressed tar file). This version was labeled 0.9.0, and released under a license that was an almost verbatim copy of the MIT license used by the X11 project at the time, substituting "Stichting Mathematisch Centrum", CWI's parent organization, as the responsible legal entity. So, like almost everything I've written, Python was open source before the term was even invented by Eric Raymond and Bruce Perens in late 1997.

There was immediately a lot of feedback and with this encouragement I kept a steady stream of releases coming for the next few years. I started to use CVS to track changes and to allow easier sharing of coding responsibilities with Sjoerd and Jack (Coincidentally, CVS was originally developed as a set of shell scripts by Dick Grune, who was an early member of the ABC group). I wrote a FAQ, which was regularly posted to some newsgroup, as was customary for FAQs in those days before the web, started a mailing list, and in March 1993 the comp.lang.python newsgroup was created with my encouragement but without my direct involvement. The newsgroup and mailing list were joined via a bidirectional gateway that still exists, although it is now implemented as a feature of mailman – the dominant open source mailing list manager, itself written in Python.

In the summer of 1994, the newsgroup was buzzing with a thread titled "If Guido was hit by a bus?" about the dependency of the growing Python community on my personal contributions. This culminated in an invitation from Michael McLay for me to spend two months as a guest researcher at NIST, the US National Institute for Standards and Technology, formerly the National Bureau of Standards, in Gaithersburg, Maryland. Michael had a number of "customers" at NIST who were interested in using Python for a variety of standards-related projects and the budget for my stay there was motivated by the need to help them improve their Python skills, as well as possibly improving Python for their needs.

The first Python workshop was held while I was there in November 1994, with NIST programmer Ken Manheimer providing important assistance and encouragement. Of the approximately 20 attendees, about half are still active participants in the Python community and a few have become major open source project leaders themselves (Jim Fulton of Zope and Barry Warsaw of GNU mailman). With NIST's support I also gave a keynote for about 400 people at the Usenix Little Languages conference in Santa Fe, organized by Tom Christiansen, an open-minded Perl advocate who introduced me to Perl creator Larry Wall and Tcl/Tk author John Ousterhout.

Next installment: how I got a job in the US...

Personal History - part 2, CNRI and beyond

by Guido van Rossum
Tuesday, January 27, 2009

The Python workshop (see previous posting) resulted in a job offer to come work on mobile agents at CNRI (the Corporation for National Research Initiatives). CNRI is a non-profit research lab in Reston, Virginia. I joined in April 1995. CNRI’s director, Bob Kahn, was the first to point out to me how much Python has in common with Lisp, despite being completely different at a superficial (syntactic) level. Python work at CNRI was funded indirectly by a DARPA grant for mobile agent research. Although there was DARPA support for projects that used Python, there was not much direct support for language development itself.

At CNRI, I led and helped hire a small team of developers to build a mobile agent system in pure Python. The initial team members were Roger Masse and Barry Warsaw who were bitten by the Python bug at the Python workshop at NIST. In addition, we hired Python community members Ken Manheimer and Fred Drake. Jeremy Hylton, an MIT graduate originally hired to work on text retrieval, also joined the team. The team was initially managed by Ted Strollo and later on by Al Vezza.

This team helped me create and maintain additional Python community infrastructure such as the python.org website, the CVS server, and the mailing lists for various Python Special Interest Groups. Python releases 1.3 through 1.6 came out of CNRI. For many years Python 1.5.2 was the most popular and most stable version.

GNU mailman was also born here: we originally used a Perl tool called Majordomo, but Ken Manheimer found it unmaintainable and looked for a Python solution. He found out about something written in Python by John Viega and took over maintenance. When Ken left CNRI for Digital Creations, Barry Warsaw took over, and convinced the Free Software Foundation to adopt it as its official mailing list tool. Hence Barry licensed it under the GPL (GNU Public License).

The Python workshops continued, at first twice a year, but due to the growth and increased logistical efforts they soon evolved into yearly events. These were first run by whoever wanted to host them, like NIST (the first one), USGS (the second and third one) and LLNL (the fourth one, and the start of the yearly series). Eventually CNRI took over the organization, and later (together with the WWW and IETF conferences) this was spun off as a commercial effort, Fortec. Attendance quickly rose to several hundreds. When Fortec faded away a while after I left CNRI, the International Python Conference was folded into O'Reilly's Open Source Conference (OSCON), but at the same time the Python Software Foundation (see below) started a new series of grassroots conferences named PyCon.

We also created the first (loose) organization around Python at CNRI. In response to efforts by Mike McLay and Paul Everitt to create a "Python Foundation", which ended up in the quicksand of bylaw drafting, Bob Kahn offered to create the "Python Software Activity", which would not be an independent legal entity but simply a group of people working under CNRI's legal (non-profit) umbrella. The PSA was successful in rallying the energy of a large group of committed Python users, but its lack of independence limited its effectiveness.

CNRI also used DARPA money to fund the development of JPython (later shortened to Jython), a Python implementation in and for Java. Jim Hugunin initially created JPython while doing graduate work at MIT. He then convinced CNRI to hire him to complete the work (or perhaps CNRI convinced Jim to join -- it happened while I was on vacation). When Jim left CNRI less than two years later to join the AspectJ project at Xerox PARC, Barry Warsaw continued the JPython development. (Much later, Jim would also author IronPython, the Python port to Microsoft's .NET. Jim also wrote the first version of Numeric Python.)

Other projects at CNRI also started to use Python. Several new core Python developers came out of this, in particular Andrew Kuchling, Neil Schemenauer, and Greg Ward, who worked for the MEMS Exchange project. (Andrew had contributed to Python even before joining CNRI; his first major project was the Python Cryptography Toolkit, a third party library that made many fundamental cryptological algorithms available to Python users.)

On the wings of Python's success, CNRI tried to come up with a model to fund Python development more directly than via DARPA research grants. We created the Python Consortium, modeled after the X Consortium, with a minimum entrance fee of $20,000. However, apart from one group at Hewlett-Packard, we didn't get much traction, and eventually the consortium died of anemia. Another attempt to find funding was Computer Programming for Everybody (CP4E), which received some DARPA funding. However, the funding wasn't enough for the whole team, and it turned out that there was a whole old-boys network involved in getting actually most of the money spread over several years. That was not something I enjoyed, and I started looking for other options.

Eventually, in early 2000, the dot-com boom, which hadn’t quite collapsed yet, convinced me and three other members of the CNRI Python team (Barry Warsaw, Jeremy Hylton, and Fred Drake) to join BeOpen.com, a California startup that was recruiting open source developers. Tim Peters, a key Python community member, also joined us.

In anticipation of the transition to BeOpen.com, a difficult question was the future ownership of Python. CNRI insisted on changing the license and requested that we release Python 1.6 with this new license. The old license used while I was still at CWI had been a version of the MIT license. The releases previously made at CNRI used a slightly modified version of that license, with basically one sentence added where CNRI disclaimed most responsibilities. The 1.6 license however was a long wordy piece of lawyerese crafted by CNRI's lawyers.

We had several long wrestling discussions with Richard Stallman and Eben Moglen of the Free Software Foundation about some parts of this new license. They feared it would be incompatible with the GPL, and hence threaten the viability of GNU mailman, which had by now become an essential tool for the FSF. With the help of Eric Raymond, changes to the CNRI Python license were made that satisfied both the FSF and CNRI, but the resulting language is not easy to understand. The only good thing I can say about it is that (again thanks to Eric Raymond's help) it has the seal of approval of the Open Source Initiative as a genuine open source license. Only slight modifications were made to the text of the license to reflect the two successive changes of ownership, first BeOpen.com and then the Python Software Foundation, but in essence the handiwork of CNRI's lawyers still stands.

Like so many startups at the time, the BeOpen.com business plan failed rather spectacularly. It left behind a large debt, some serious doubts about the role played by some of the company's officers, and some very disillusioned developers besides my own team.

Luckily year my team, by now known as PythonLabs, was pretty hot, and we were hired as a unit by Digital Creations, one of the first companies to use Python. (Ken Manheimer had preceded us there a few years before.) Digital Creations soon renamed itself Zope Corporation after its main open source product, the web content management system Zope. Zope’s founders Paul Everitt and Rob Page had attended the very first Python workshop at NIST in 1994, as did its CTO, Jim Fulton.

History could easily have gone very differently: besides Digital Creations, we were also considering offers from VA Linux and ActiveState. VA Linux was then a rising star on the stock market, but eventually its stock price (which had made Eric Raymond a multi-millionaire on paper) collapsed rather dramatically. Looking back I think ActiveState would not have been a bad choice, despite the controversial personality of its founder Dick Hardt, if it hadn't been located in Canada.

In 2001 we created the Python Software Foundation, a non-profit organization, whose initial members were the main contributing Python developers at that time. Eric Raymond was one of the founding members. I'll have to write more about this period another time.

Microsoft Ships Python Code... in 1996

by Greg Stein
Wednesday, January 28, 2009

My thanks go to Guido for allowing me to share my own history of Python!

I'll save my introduction to Python for another post, but the end result was its introduction into a startup that I co-founded in 1991 with several people. We were working on a large client/server system to handle Business-to-Consumer electronic shopping. Custom TCP protocols operating over the old X.25 network, and all that. Old school.

In 1995, we realized, contrary to our earlier beliefs, that more consumers actually *were* on the Internet, and that we needed a system for our customers (the vendors) to reach Internet-based consumers. I was tasked to figure out our approach, and selected Python as my prototyping tool.

Our first problem was moving to an entirely browser-based solution. Our custom client was no longer viable, so we needed a new shopping experience for the consumer, and server infrastructure to support that. At that time, talking to a web browser meant writing CGI scripts for the Apache and Netscape HTTP servers. Using CGI, I connected to our existing server backend to process orders, maintain the shopping basket, and to fetch product information. These CGI scripts produced plain, vanilla HTML (no AJAX in 1995!).

This approach was less-than-ideal since each request took time to spin up a new CGI process. The responsiveness was very poor. Then, in December 1995, while attending the Python Workshop in Washington, DC, I was introduced to some Apache and Netscape modules (from Digital Creations, who are best known for Zope) which ran persistently within the server process. These modules used an RPC system called ILU to communicate with backend, long-running processes. With this system in place, the CGI forking overhead disappeared and the shopping experience was now quite enjoyable! We started to turn the prototype into real code. The further we went with it, the better it looked and more people jumped onto the project. Development moved **very** fast over the next few months (thanks Python!).

In January 1996, Microsoft knocked on our door. Their internal effort at creating an electronic commerce system was floundering, and they needed people that knew the industry (we'd been doing electronic commerce for several years by that point) and somebody who was nimble. We continued to develop the software during the spring while negotiations occurred, and then the acquisition finalized in June 1996.

Once we arrived at Microsoft with our small pile of Python code, we had to figure out how to ship the product on Windows NT. The team we joined had lots of Windows experience and built an IIS plugin to communicate over named pipes to the backend servers, which were NT Services with our Python server code embedded. With a mad sprint starting in July, we shipped Microsoft Merchant Server 1.0 in October, 1996.

And yes... if you looked under the covers, somewhat hidden, was a Python interpreter, some extension DLLs, and a bunch of .pyc files. Microsoft certainly didn't advertise that fact, but it was there if you knew were to look.

Early Language Design and Development

by Guido van Rossum
Tuesday, February 3, 2009

From ABC to Python

Python’s first and foremost influence was ABC, a language designed in the early 1980s by Lambert Meertens, Leo Geurts and others at CWI. ABC was meant to be a teaching language, a replacement for BASIC, and a language and environment for personal computing. It was designed by first doing a task analysis of the programming task and then doing several iterations that included serious user testing. My own role in the ABC group was mainly that of implementing the language and its integrated editing environment.

Python’s use of indentation comes directly from ABC, but this idea didn’t originate with ABC--it had already been promoted by Donald Knuth and was a well-known concept of programming style. (The occam programming language also used it.) However, ABC’s authors did invent the use of the colon that separates the lead-in clause from the indented block. After early user testing without the colon, it was discovered that the meaning of the indentation was unclear to beginners being taught the first steps of programming. The addition of the colon clarified it significantly: the colon somehow draws attention to what follows and ties the phrases before and after it together in just the right way.

Python’s major data types also derive from ABC, though with some modifications. ABC’s lists were really bags or multisets, which were always kept sorted using a modified B-tree implementation. Its tables were associative arrays that were similarly kept sorted by key. I found that neither data type was suitable to represent, for example, the sequence of lines read from a file, which I anticipated would be a common use case. (In ABC you'd have to use a table with the line numbers as keys, but that complicates insertions and deletions.) So I changed the list type into a flexible array with insert and delete operations, giving users complete control over the ordering of items in a list. A sort method supported the occasional need for sorted results.

I also replaced the sorted tables with a hash table implementation. I chose a hash table because I believed this to be faster and easier to implement than ABC’s B-tree implementation. The latter was theoretically proven to be asymptotically optimal in space and time consumption for a wide variety of operations, but in practice it had turned out to be difficult to implement correctly due to the complexity of the B-tree algorithms. For the same reason, the performance was also sub-optimal for small table sizes.

I kept ABC’s immutable tuple data type--Python’s tuple packing and unpacking operations are taken directly from ABC. Since tuples are internally represented by arrays, I decided to add array-like indexing and slicing.

One consequence of adding an array-like interface to tuples is that I had to figure out some way to resolve the edge cases of tuples with length 0 or 1. One of the rules I took from ABC was that every data type, when printed or converted to a string, should be represented by an expression that was a valid input to the language’s parser. So, it followed that I needed to have notations for 0- and 1-length tuples. At the same time I didn’t want to lose the distinction between a one-tuple and a bare parenthesized expression, so I settled for an ugly but pragmatic approach where a trailing comma would turn an expression into a one-tuple and "()" would represent a zero-tuple. It's worth nothing that parentheses aren't normally required by Python's tuple syntax, except here--I felt representing the empty tuple by "nothing" could too easily mask genuine typos.

Python's strings started with very similar (immutable) semantics as ABC's strings, but with a different notation, and 0-based indexing. Since I now had three indexable types, list, tuples, and strings, I decided to generalize these to a common concept, the sequence. This generalization made it so certain core operations such as getting the length (`len(s)`), indexing (`s[i]`), slicing (`s[i:j]`), and iteration (`for i in s`) worked the same on any sequence type.

Numbers are one of the places where I strayed most from ABC. ABC had two types of numbers at run time; *exact* numbers which were represented as arbitrary precision rational numbers and *approximate* numbers which were represented as binary floating point with extended exponent range. The rational numbers didn't pan out in my view. (Anecdote: I tried to compute my taxes once using ABC. The program, which seemed fairly straightforward, was taking way too long to compute a few simple numbers. Upon investigation it turned out that it was doing arithmetic on numers with thousands of digits of precision, which were to be rounded to guilders and cents for printing.) For Python I therefore chose a more traditional model with machine integers and machine binary floating point. In Python's implementation, these numbers are simply represented by the C datatypes of long and double respectively.

Feeling that there was still an important use case for unbounded exact numbers, I added a *bignum* data type, which I called *long*. I already had a bignum implementation that was the result of an unfinished attempt at improving ABC's implementation a few years earlier (ABC's original implementation, one of my first contributions to ABC, used a decimal representation internally). Thus, it made sense to me to use this code in Python.

Although I added bignums to Python, it's important to emphasize that I didn't want to use bignums for all integer operations. From profiling Python programs written by myself and colleagues at CWI, I knew that integer operations represent a significant fraction of the total program running time of most programs. By far, the most common use of integers is to index sequences that fit in memory. Thus, I envisioned machine integers being used for all of the most-common cases and the extra range of bignums only coming into play when doing "serious math" or calculating the national debt of the US in pennies.

The Problem With Numbers

The implementation of numbers, especially integers, is one area where I made several serious design mistakes, but also learned important lessons concerning Python's design.

Since Python had two different integer types, I needed to figure out some way to distinguish between the two types in a program. My solution was to require users to explicitly state when they wanted to use longs by writing numbers with a trailing `L` (e.g., `1234L`). This is one area where Python violated the ABC-inspired philosophy of not requiring users to care about a uninteresting implementation details.

Sadly, this was only a minor detail of much larger problem. A more egregious error was that my implementation of integers and longs had slightly different semantics in certain cases! Since the int type was represented as a machine integer, operations that overflowed would silently clip the result to 32 bits or whatever the precision of the C long type happened to be. In addition, the int type, while normally considered signed, was treated as an unsigned number by bitwise and shift operations and by conversions to/from octal and hexadecimal representations. Longs, on the other hand, were always considered signed. Therefore, some operations would produce a different result depending on whether an argument was represented as an int or a long. For example, given 32-bit arithmetic, 1 (1 shifted left by 31 bits) would produce the largest negative 32-bit integer, and 1 would produce zero, whereas 1L (1 represented as long shifted left by 31 bits) would produce a long integer equal to 2**31, and 1L would produce 2**32.

To resolve some of these issues I made a simple fix. Rather than having integer operations silently clip the result, I changed most arithmetic operations to raise an OverflowError exception when the result didn't fit. (The only exception to this checking were the "bit-wise" operations mentioned above, where I assumed that users would expect the behavior of these operations in C.) Had I not added this check, Python users would have undoubtedly started writing code that relied on the semantics of signed binary arithmetic modulo 2**32 (like C users do), and fixing the mistake would have been a much more painful transition to the community.

Although the inclusion of overflow checking might seem like a minor implementation detail, a painful debugging experience made me realize that this was a useful feature. As one of my early programming experiments in Python, I tried to implement a simple mathematical algorithm, the computation of “Meertens numbers”, a bit of recreational mathematics invented by Richard Bird for the occasion of ABC’s primary author’s 25ths anniversary at CWI. The first few Meertens numbers are small, but when translating the algorithm into code I hadn’t realized that the intermediate results of the computation are much larger than 32 bits. It took a long and painful debugging session before I discovered this, and I decided there and then to address the issue by checking all integer operations for overflow, and raising an exception whenever the result could not be represented as a C long. The extra cost of the overflow check would be dwarfed by the overhead I was already incurring due to the implementation choice of allocating a new object for the result.

Sadly, I'm sorry to say that raising an overflow exception was not really the right solution either! At the time, I was stuck on C’s rule “operations on numeric type T return a result of type T”. This rule was also the reason for my other big mistake in integer semantics: truncating the result of integer division, which I will discuss in a later blog post. In hindsight, I should have made integer operations that overflow promote their result to longs. This is the way that Python works today, but it took a long time to make this transition.

Despite the problem with numbers, one very positive thing came out of this experience. I decided that there should be no undefined result values in Python – instead, exceptions should always be raised when no correct return value can be computed. Thus, Python programs would never fail due to undefined values being silently passed around behind the scenes. This is still an important principle of the language, both in the language proper and in the standard library.

Python's Use of Dynamic Typing

by Guido van Rossum
Tuesday, February 10, 2009

An important difference between ABC and Python is the general flavor of the type system. ABC is statically typed which meant that the ABC compiler analyzes the use of types in a program and decides whether they are used consistently. If not, the program is rejected and its execution cannot be started. Unlike most statically typed languages of its days, ABC used type inference (not unlike Haskell) instead of explicit type declarations as you find in languages such as in C. In contrast, Python is dynamically typed. The Python compiler is blissfully unaware of the types used in a program, and all type checking is done at run time.

Although this might seem like a large departure from ABC, it is not as different as you might imagine. Unlike other statically typed languages, ABC doesn't (didn't? it's pretty much purely historic now :-) exclusively rely on static type checking to keep the program from crashing, but has a run-time library that checks the argument types for all operations again each time they are executed. This was done in part as a sanity check for the compiler’s type-checking algorithms, which were not fully implemented in the initial prototype implementation of the language. The runtime library was also useful as a debugging aid since explicit run time type checking could produce nice error messages (aimed at the implementation team), instead of the core dumps that would ensue if the interpreter were to blindly go ahead with an operation without checking if the arguments make sense.

However, the most important reason why ABC has run time type checking in addition to static type checking is that it is interactive. In an interactive session, the user types ABC statements and definitions which are executed as soon as they are completed. In an interactive session, it is possible to create a variable as a number, delete it, and then to recreate it (in other words, create another variable with the same name) as a string. Inside a single procedure, it would be a static typing error to use the same variable name first as a number and then as a string, but it would not be reasonable to enforce such type checking across different statements entered in an interactive session, as the accidental creation of a variable named x as a number would forever forbid the creation of a variable x with a different type! So as a compromise, ABC uses dynamic type checking for global variables, but static type checking for local variables. To simplify the implementation, local variables are dynamically type checked as well.

Thus, it is only a small step from ABC’s implementation approach to type checking to Python’s approach--Python simply drops the compile-time type checking completely. This is entirely in line with Python’s “corner-cutting” philosophy as it’s a simplification of the implementation that does not affect the eventual safety, since all type errors are caught at run time before they can cause the Python interpreter to malfunction.

However, once you decide on dynamic typing, there is no way to go back. ABC’s built-in operations were carefully designed so that the type of the arguments could be deduced from the form of the operation. For example, from the expression “`x^y`” the compiler would deduce that variables x and y were strings, as well as the expression result. In Python, such deductions cannot generally be made. For example, the expression “`x+y`” could be a string concatenation, a numeric addition, or an overloaded operation on user-defined types.

Adding Support for User-defined Classes

by Guido van Rossum
Wednesday, February 18, 2009

Believe it or not, classes were added late during Python's first year of development at CWI, though well before the first public release. However, to understand how classes were added, it first helps to know a little bit about how Python is implemented.

Python is implemented in C as a classic stack-based byte code interpreter or "virtual machine" along with a collection of primitive types also implemented in C. The underlying architecture uses "objects" throughout, but since C has no direct support for objects, they are implemented using structures and function pointers. The Python virtual machine defines several dozen standard operations that every object type may or must implement (for example, "get attribute", "add" and "call"). An object type is then represented by a statically allocated structure containing a series of function pointers, one for each standard operation. These function pointers are typically initialized with references to static functions. However, some operations are optional, and the object type may leave the function pointer NULL if it chooses not to implement that operation. In this case, the virtual machine either generates a run-time error or, in some cases, provides a default implementation of the operation. The type structure also contains various data fields, one of which is a reference to a list of additional methods that are unique to this type, represented as an array of structures containing a string (the method name) and a function pointer (its implementation) each. Python's unique approach to introspection comes from its ability to make the type structure itself available at run-time as an object like all others.

An important aspect of this implementation is that it is completely C-centric. In fact, all of the standard operations and methods are implemented by C functions. Originally the byte code interpreter only supported calling pure Python functions and functions or methods implemented in C. I believe my colleague Siebren van der Zee was the first to suggest that Python should allow class definitions similar to those in C++ so that objects could also be implemented in Python.

To implement user-defined objects, I settled on the simplest possible design; a scheme where objects were represented by a new kind of built-in object that stored a class reference pointing to a "class object" shared by all instances of the same class, and a dictionary, dubbed the "instance dictionary", that contained the instance variables.

In this implementation, the instance dictionary would contain the instance variables of each individual object whereas the class object would contain stuff shared between all instances of the same class--in particular, methods. In implementing class objects, I again chose the simplest possible design; the set of methods of a class were stored in a dictionary whose keys are the method names. This, I dubbed the class dictionary. To support inheritance, class objects would additionally store a reference to the class objects corresponding to the base classes. At the time, I was fairly naïve about classes, but I knew about multiple inheritance, which had recently been added to C++. I decided that as long as I was going to support inheritance, I might as well support a simple-minded version of multiple inheritance. Thus, every class object could have one or more base classes.

In this implementation, the underlying mechanics of working with objects are actually very simple. Whenever changes are made to instance or class variables, those changes are simply reflected in the underlying dictionary object. For example, setting an instance variable on an instance updates its local instance dictionary. Likewise, when looking up the value of a instance variable of an object, one merely checks its instance dictionary for the existence of that variable. If the variable is not found there, things become a little more interesting. In that case, lookups are performed in the class dictionary and then in the class dictionaries of each of the base classes.

The process of looking up attributes in the class object and base classes is most commonly associated with locating methods. As previously mentioned, methods are stored in the dictionary of a class object which is shared by all instances of the same class. Thus, when a method is requested, you naturally won't find it in the instance dictionary of each individual object. Instead, you have to look it up in the class dictionary, and then ask each of the base classes in turn, stopping when a hit is found. Each of the base classes then implements the same algorithm recursively. This is commonly referred to as the depth-first, left-to-right rule, and has been the default method resolution order (MRO) used in most versions of Python. More modern releases have adapted a more sophisticated MRO, but that will be discussed in a later blog.

In implementing classes, one of my goals was to keep things simple. Thus, Python performs no advanced error checking or conformance checking when locating methods. For example, if a class overrides a method defined in a base class, no checks are performed to make sure that the redefined method has the same number of arguments or that it can be called in the same way as the original base-class method. The above method resolution algorithm merely returns the first method found and calls it with whatever arguments the user has supplied.

A number of other features also fall out of this design. For instance, even though the class dictionary was initially envisioned as a place to put methods, there was no inherent reason why other kinds of objects couldn't be placed there as well. Thus, if objects such as integers or strings are stored in the class dictionary, they become what are known as class variables---variables shared by all instances of a given class instead of being stored inside each instance.

Although the implementation of classes is simple, it also provides a surprisingly degree of flexibility. For instance, the implementation not only makes classes “first-class objects”, which are easily introspected at run time, it also makes it possible to modify a class dynamically. For example, methods can be added or modified by simply updating the class dictionary after a class object has already been created! (*) The dynamic nature of Python means that these changes have an immediate effect on all instances of that class or of any of its subclasses. Likewise, individual objects can be modified dynamically by adding, modifying, and deleting instance variables (a feature that I later learned made Python's implementation of objects more permissive than that found in Smalltalk which restricts the set of attributes to those specified at the time of object creation).

Development of the class Syntax

Having designed the run-time representations for user-defined classes and instances, my next task was to design the syntax for class definitions, and in particular for the method definitions contained therein. A major design constraint was that I didn’t want to add syntax for methods that differed from the syntax for functions. Refactoring the grammar and the byte code generator to handle such similar cases differently felt like a huge task. However, even if I was successful in keeping the grammar the same, I still had to figure out some way to deal with instance variables. Initially, I had hoped to emulate implicit instance variables as seen in C++. For example, in C++, classes are defined by code like this:

```
class A {
public:
  int x;
  void spam(int y) {
       printf("%d %d\n", x, y);
  }
};
```

In this class, an instance variable x has been declared. In methods, references to x implicitly refer to the corresponding instance variable. For example, in the method spam(), the variable x is not declared as either function parameter or as local variable However, since the class has declared an instance variable with that name, references to x simply refer to that variable. Although I had hoped to provide something similar in Python, it quickly became clear that such an approach would be impossible because there was no way to elegantly distinguish instance variables from local variables in a language without variable declarations.

In theory, getting the value of instance variables would be easy enough. Python already had a search order for unqualified variable names: locals, globals, and built-ins. Each of these were represented as a dictionary mapping variable names to values. Thus, for each variable reference, a series of dictionaries would be searched until a hit was found. For example, when executing a function with a local variable p, and a global variable q, a statement like "print p, q" would look up p in the first dictionary in the search order, the dictionary containing local variables, and find a match. Next it would look up q in the first dictionary, find no match, then look it up in the second dictionary, the global variables, and find a match.

It would have been easy to add the current object's instance dictionary in front of this search list when executing a method. Then, in a method of an object with an instance variable x and local variable y, a statement like "print x, y" would find x in the instance dictionary (the first dictionary on the search path) and y in the local variable dictionary (the second dictionary).

The problem with this strategy is that it falls apart for setting instance variables. Python's assignment doesn't search for the variable name in the dictionaries, but simply adds or replaces the variable in the first dictionary in the search order, normally the local variables. This has the effect that variables are always created in the local scope by default (although it should be noted that there is a "global declaration" to override this on a per-function, per-variable basis.)

Without changing this simple-minded approach to assignment, making the instance dictionary the first item in the search order would make it impossible to assign to local variables in a method. For example, if you had a method like this

```
def spam(y):
   x = 1
   y = 2
```

The assignments to x and y would overwrite the instance variable x and create a new instance variable y that shadowed the local variable y. Swapping instance variables and local variables in the search order would merely reverse the problem, making it impossible to assign to instance variables.

Changing the semantics of assignment to assign to an instance variable if one already existed and to a local otherwise wouldn't work either, since this would create a bootstrap problem: how does an instance variable get created initially? A possible solution might have been to require explicit declaration of instance variables as was the case for global variables, but I really didn't want to add a feature like that given that that I had gotten this far without any variable declarations at all. Plus, the extra specification required for indicating a global variable was more of a special case that was used sparingly in most code. Requiring a special specification for instance variables, on the other hand, would have to be used almost everywhere in a class. Another possible solution would have been to make instance variables lexically distinct. For example, having instance variables start with a special character such as @ (an approach taken by Ruby) or by having some kind of special naming convention involving prefixes or capitalization. Neither of these appealed to me (and they still don't).

Instead, I decided to give up on the idea of implicit references to instance variables. Languages like C++ let you write this->foo to explicitly reference the instance variable foo (in case there's a separate local variable foo). Thus, I decided to make such explicit references the only way to reference instance variables. In addition, I decided that rather than making the current object ("this") a special keyword, I would simply make "this" (or its equivalent) the first named argument to a method. Instance variables would just always be referenced as attributes of that argument.

With explicit references, there is no need to have a special syntax for method definitions nor do you have to worry about complicated semantics concerning variable lookup. Instead, one simply defines a function whose first argument corresponds to the instance, which by convention is named "self." For example:

```
def spam(self,y):
   print self.x, y
```

This approach resembles something I had seen in Modula-3, which had already provided me with the syntax for import and exception handling. Modula-3 doesn't have classes, but it lets you create record types containing fully typed function pointer members that are initialized by default to functions defined nearby, and adds syntactic sugar so that if x is such a record variable, and m is a function pointer member of that record, initialized to function f, then calling x.m(args) is equivalent to calling f(x, args). This matches the typical implementation of objects and methods, and makes it possible to equate instance variables with attributes of the first argument.

The remaining details of Python's class syntax follow from this design or from the constraints imposed by the rest of the implementation. Keeping with my desire for simplicity, I envisioned a class statement as a series of method definitions, which are syntactically identical to function definitions even though by convention, they are required to have a first argument named "self". In addition, rather than devising a new syntax for special kinds of class methods (such as initializers and destructors), I decided that these features could be handled by simply requiring the user to implement methods with special names such as __init__, __del__, and so forth. This naming convention was taken from C where identifiers starting with underscores are reserved by the compiler and often have special meaning (e.g., macros such as __FILE__ in the C preprocessor).

Thus, I envisioned that a class would be defined by code that looked like this:

```
class A:
    def __init__(self,x):
        self.x = x
    def spam(self,y):
       print self.x, y
```

Again, I wanted to reuse as much of my earlier code as possible. Normally, a function definition is an executable statement that simply sets a variable in the current namespace referencing the function object (the variable name is the function name). Thus, rather than coming up with an entirely different approach for handling classes, it made sense to me to simply interpret the class body as a series of statements that were executed in a new namespace. The dictionary of this namespace would then be captured and used to initialize the class dictionary and create a class object. Underneath the covers, the specific approach taken is to turn the class body into an anonymous function that executes all of the statements in the class body and then returns the resulting dictionary of local variables. This dictionary is then passed to a helper function that creates a class object. Finally, this class object is then stored in a variable in the surrounding scope, whose name is the class name. Users are often surprised to learn that any sequence of valid Python statements can appear in a class body. This capability was really just a straightforward extension of my desire to keep the syntax simple as well as not artificially limiting what might possibly be useful.

A final detail is the class instantiation (instance creation) syntax. Many languages, like C++ and Java, use a special operator, "new", to create new class instances. In C++ this may be defensible since class names have a rather special status in the parser, but in Python this is not the case. I quickly realized that, since Python's parser doesn't care what kind of object you call, making the class object itself callable was the right, "minimal" solution, requiring no new syntax. I may have been ahead of my time here---today, factory functions are often the preferred pattern for instance creation, and what I had done was simply to turn each class into its own factory.

Special Methods

As briefly mentioned in the last section, one of my main goals was to keep the implementation of classes simple. In most object oriented languages, there are a variety of special operators and methods that only apply to classes. For example, in C++, there is a special syntax for defining constructors and destructors that is different than the normal syntax used to define ordinary function and methods.

I really didn't want to introduce additional syntax to handle special operations for objects. So instead, I handled this by simply mapping special operators to a predefined set of "special method" names such as __init__ and __del__. By defining methods with these names, users could supply code related to the construction and destruction of objects.

I also used this technique to allow user classes to redefine the behavior of Python's operators. As previously noted, Python is implemented in C and uses tables of function pointers to implement various capabilities of built-in objects (e.g., “get attribute”, “add” and “call”). To allow these capabilities to be defined in user-defined classes, I mapped the various function pointers to special method names such as __getattr__, __add__, and __call__. There is a direct correspondence between these names and the tables of function pointers one has to define when implementing new Python objects in C.

____

(*) Eventually, new-style classes made it necessary to control changes to the class __dict__; you can still dynamically modify a class, but you must use attribute assignment rather than using the class __dict__ directly.

First-class Everything

by Guido van Rossum
Friday, February 27, 2009

[Folks, please don't use the comments section of this blog to ask questions. If you want to suggest a topic for a future blog entry, send me email. (Use Google to find my home page, which has my email address.) If you want to propose a change or discuss the merits of alternative designs, use the python-ideas mailing list at python.org.]

One of my goals for Python was to make it so that all objects were "first class." By this, I meant that I wanted all objects that could be named in the language (e.g., integers, strings, functions, classes, modules, methods, etc.) to have equal status. That is, they can be assigned to variables, placed in lists, stored in dictionaries, passed as arguments, and so forth.

The internal implementation of Python made this simple to do. All of Python's objects were based on a common C data structure that was used everywhere in the interpreter. Variables, lists, functions, and everything else just used variations of this one data structure---it just didn't matter if the structure happened to represent a simple object such as an integer or something more complicated such as a class.

Although the idea of having "first-class everything" is conceptually simple, there was still one subtle aspect of classes that I still needed to address---namely, the problem of making methods first class objects.

Consider this simple Python class (copied from last week's blog post):

```
class A:
    def __init__(self,x):
        self.x = x
    def spam(self,y):
       print self.x, y
```

If methods are going to be first-class objects, then they can be assigned to other variables and used just like other objects in Python. For example, someone could write a Python statement such as "s = A.spam". In this case, the variable "s" refers to a method of a class, which is really just a function. However, a method is not quite the same as ordinary function. Specifically, the first argument of a method is supposed to be an instance of the class in which a method was defined.

To deal with this, I created a type of callable object known as an "unbound method." An unbound method was really just a thin wrapper around the function object that implemented a method, but it enforced a restriction that the first argument had to be an instance of the class in which the method was defined. Thus, if someone wanted to call an unbound method "s" as a function, they would have to pass an instance of class "A" as the first argument. For example, "a = A(); s(a)". (*)

A related problem occurs if someone writes a Python statement that refers to a method on a specific instance of an object. For example, someone might create an instance using "a = A()" and then later write a statement such as "s = a.spam". Here, the variable "s" again refers to a method of a class, but the reference to that method was obtained through an instance "a" . To handle this situation, a different callable object known as a "bound method." is used. This object is also a thin wrapper around the function object for the method. However, this wrapper implicitly stores the original instance that was used to obtain the method. Thus, a later statement such as "s()" will call the method with the instance "a" implicitly set as the first argument.

In reality, the same internal object type is used to represent bound and unbound methods. One of the attributes of this object contains a reference to an instance. If set to None, the method is unbound. Otherwise, the method is bound.

Although bound and unbound methods might seem like an unimportant detail, they a critical part of how classes work underneath the covers. Whenever a statement such as "a.spam()" appears in a program, the execution of that statement actually occurs in two steps. First, a lookup of "a.spam" occurs. This returns a bound method--a callable object. Next, a function call operation "()" is applied to that object to invoke the method with user supplied arguments.

(*) In Python 3000, the concept of unbound methods has been removed, and the expression "A.spam" returns a plain function object. It turned out that the restriction that the first argument had to be an instance of A was rarely helpful in diagnosing problems, and frequently an obstacle to advanced usages --- some have called it "duck typing self" which seems an appropriate name.

How Everything Became an Executable Statement

by Guido van Rossum
Tuesday, March 3, 2009

New users to Python are sometimes surprised to find out that every part of the language is an executable statement, including function and class definitions. That means that any statement can appear anywhere in a program. For instance, a function definition could appear inside an "if" statement if you wanted.

In a very early version of Python’s grammar, this was not the case: grammar elements that had a “declarative flavor”, like import statements and function definitions, were allowed only at the top level in a module or script (where they were being executed in order to become effective). However, at the time I was adding support for classes, I decided that this was too restrictive.

My reasoning went roughly as follows. Rather than defining the class body as a series of function declarations only, it seemed to make sense to also allow regular variable assignments there. However, if I was going to allow that, why not go one step further and allow arbitrary executable code? Or, taking this even further, why not allow function declarations inside an “if” statement, for example? It quickly became clear that this enabled a simplification of the grammar, since now all uses of statements (whether indented or not) could share the same grammar rule, and hence the compiler could use the same byte code generation function for all of them.

Although this reasoning allowed me to simplify the grammar and allowed users to place Python statements anywhere, this feature did not necessarily enable certain styles of programming. For example, the Python grammar technically allowed users to write things such as nested functions even though the underlying semantics of Python didn't support nested scopes. Therefore, code such as that would often operate in ways that were unexpected or "broken" compared to languages that were actually designed with such features in mind. Over time, many of these "broken" features have been fixed. For example, nested function definitions only began to work more sanely in Python 2.1.

How Exceptions Came to be Classes

by Guido van Rossum
Friday, March 6, 2009

Early on, I knew I wanted Python to use exceptions for error handling. However, a critical part of making exceptions work is to come up with some kind of scheme for identifying different kinds of exceptions. In modern languages (including modern Python :-), exceptions are defined in terms of user-defined classes. In early Python however, I chose to identify exceptions by strings. This was unfortunate, but I had two reasons for taking this approach. First, I learned about exceptions from Modula-3, where exceptions are unique tokens. Second, I introduced exceptions before I introduced user-defined classes.

In theory, I suppose I could have created a new type of built-in object to be used for exceptions, but as every built-in object type required a considerable coding effort in C, I decided to reuse an existing built-in type. And, since exceptions are associated with error messages, it seemed natural to use strings to represent exceptions.

Unfortunately I chose semantics where different string objects would represent different exceptions, even if they had the same value (i.e. contained the same sequence of characters). I chose these semantics because I wanted exceptions defined in different modules to be independent, even if they happened to have the same value. The idea was that exceptions would always be referenced by their name, which would imply object identity, never by their value, which would require string equality.

This approach was influenced by Modula-3's exceptions, where each exception declaration creates a unique "exception token" that can't be confused with any other exception token. I think I also wanted to optimize testing for exceptions by using pointer comparisons instead of string value comparisons in a misguided attempt to prematurely optimize execution time (a rare one – I usually optimized for my own coding time!). The main reason however is that I worried about name clashes between unrelated exceptions defined in different modules. I intended the usage pattern to strictly adhere to the convention of defining an exception as a global constant in some module, and then using it by name in all code raising or catching it. (This was also long before certain string literals would be automatically be "interned".)

Alas, in practice things never quite work out as you expect. Early Python users discovered that within the same module, the byte code compiler would unify string literals (i.e., create a single shared object for all occurrences of string literals with the same value). Thus, by accident, users found that exceptions could either be caught be specifying the exception name or the string literal containing the error message. Well, at least this seemed to work most of the time. In reality, it only worked for code defined in the same module---if one tried to catch exceptions using the exception error message in a different module, it broke mysteriously. Needless to say, this is the sort of thing that causes widespread confusion.

In 1997, with Python 1.5, I introduced class exceptions into the language. Although class exceptions have been the recommended approach ever since, string exceptions were still supported for use by certain legacy applications through Python 2.5. They were finally removed in Python 2.6.

The Problem with Integer Division

by Guido van Rossum
Tuesday, March 10, 2009

Python's handling of integer division is an example of early mistake with huge consequences. As mentioned earlier, when Python was created, I abandoned the approach to numbers that had been used in ABC. For example, in ABC, when you divided two integers, the result was an exact rational number representing the result. In Python however, integer division truncated the result to an integer.

In my experience, rational numbers didn't pan out as ABC's designers had hoped. A typical experience would be to write a simple program for some business application (say, doing one’s taxes), and find that it was running much slower than expected. After some debugging, the cause would be that internally the program was using rational numbers with thousands of digits of precision to represent values that would be truncated to two or three digits of precision upon printing. This could be easily fixed by starting an addition with an inexact zero, but this was often non-intuitive and hard to debug for beginners.

So with Python, I relied on the other numerical model with which I was familiar, C. C has integers and floating point numbers of various sizes. So, I chose to represent Python integers by a C long (guaranteeing at least 32 bits of precision) and floating point numbers by a C double. I then added an arbitrary precision integer type which I called "long."

The major mistake was that I also borrowed a rule that makes sense in C but not so much in a very-high-level language. For the standard arithmetic operations, including division, the result would always be the same type as the operands. To make matters worse, I initially used another misguided rule that forbade mixed-mode arithmetic, with the aim of making the type implementations independent from each other. So, originally you couldn’t add an int to a float, or even an int to a long. After Python was released publicly, Tim Peters quickly convinced me that this was a really bad idea, and I introduced mixed-mode arithmetic with the usual coercion rules. For example, mixing an int and a long operand would convert the argument of type int to long and return a long result and mixing either with float would convert the int or long argument to float and return a float result..

Unfortunately, the damage was done--integer division returned an integer result. You might be wondering "Why was this so bad?" Was this really just an ado about nothing? Historically, the proposal to change this has had some tough opposition from folks who believed that learning about integer division was one of the more useful "rites of passage" for all programmers. So let me explain the reasons for considering this a design bug.

When you write a function implementing a numeric algorithm (for example, calculating the phase of the moon) you typically expect the arguments to be specified as floating point numbers. However, since Python doesn’t have type declarations, nothing is there to stop a caller from providing you with integer arguments. In a statically typed language, like C, the compiler will coerce the arguments to floats, but Python does no such thing – the algorithm is run with integer values until the wonders of mixed-mode arithmetic produce intermediate results that are floats.

For everything except division, integers behave the same as the corresponding floating point numbers. For example, 1+1 equals 2 just as 1.0+1.0 equals 2.0, and so on. Therefore one can easily be misled to expect that numeric algorithms will behave regardless of whether they execute with integer or floating point arguments. However, when division is involved, and the possibility exists that both operands are integers, the numeric result is silently truncated, essentially inserting a potentially large error into the computation. Although one can write defensive code that coerces all arguments to floats upon entry, this is tedious, and it doesn’t enhance the readability or maintainability of the code. Plus, it prevents the same algorithm from being used with complex arguments (although that may be highly special cases).

Again, all of this is an issue because Python doesn’t coerce arguments automatically to some declared type. Passing an invalid argument, for example a string, is generally caught quickly because few operations accept mixed string/numeric operands (the exception being multiplication). However, passing an integer can cause an answer that is close to the correct answer but has a much larger error – which is difficult to debug or even notice. (This recently happened to me in a program that draws an analog clock – the positions of the hands were calculated incorrectly due to truncation, but the error was barely detectable except at certain times of day.)

Fixing integer division was no easy task due to programs that rely on the behavior of integer truncation. A truncating division operator (//) was added to the language to provide the same functionality. In addition, a mechanism ("from __future__ import division") was introduced by which the new integer division semantics could be enabled.Finally , a command line flag (-Qxxx) was added both to change the behavior and to aid in program conversion. Fortunately, the correct behavior has become the default behavior in Python 3000.

Dynamically Loaded Modules

by Guido van Rossum
Tuesday, March 17, 2009

Python’s implementation architecture made it easy to write extension modules written in C right from the start. However, in the early days, dynamic loading technology was obscure enough that such extensions had to be statically linked into Python interpreter at build time. To do this, C extension modules had to be added to a shell script that was used to generate the Makefile for Python and all of its extension modules.

Although this approach worked for small projects, the Python community started producing new extension modules at an unanticipated rate, and demanded that extension modules could be compiled and loaded separately. Shortly thereafter, code interfacing to the platform-specific dynamic linking API was contributed which allowed the import statement to go out to disk looking for a shared library as well as a “.py” file. The first mention of dynamic loading in the CVS logs stems from January 1992 and most major platforms were supported by the end of 1994.

The dynamic linking support proved to be very useful, but also introduced a maintenance nightmare. Each platform used a different API and some platforms had additional constraints. In January 1995, the dynamic linking support was restructured so that all the dynamic linking code was concentrated in a single source file. However, the approach resulted in a large file cluttered with conditional compilation directives (#ifdef). In December 1999, it was restructured again with the help of Greg Stein so that the platform-specific code for each platform was placed in a file specific to that platform (or family of platforms).

Even though Python supported dynamically loadable modules, the procedure for building such modules often remained a mystery to many users. An increasingly large number of users were building modules--especially with the introduction of extension building tools such as SWIG. However, a user wishing to distribute an extension module faced major hurdles getting the module to compile on all possible combinations of platforms, compilers, and linkers. In a worst-case scenario, a user would have to write their own Makefile and configuration script for setting the right compiler and linker flags. Alternatively, a user could also add their extension module to Python's own Makefile and perform a partial Python rebuild to have the module compiled with the right options. However, this required end users to have a Python source distribution on-hand.

Eventually, a Python extension building tool called distutils was invented that allowed building and installing extension modules from anywhere. The necessary compiler and linker options were written by Python’s makefile to a data file, which was then consulted by distutils when building extension modules. Largely written by Greg Ward, the first versions of distutils were distributed separately, to support older Python versions. Starting with Python 1.6 it was integrated into Python distributions as a standard library module.

It is worth noting that distutils does far more than simply building extension modules from C source code. It can also install pure Python modules and packages, create Windows installer executables, and run third party tools such as SWIG. Alas, its complexity has caused many folks to curse it, and it has not received the maintenance attention it deserved. As a result, in recent times, 3rd party alternatives (especially ez_install, a.k.a. "eggs") have become popular, unfortunately causing fragmentation in the development community, as well as complaints whenever it doesn't work. It seems that the problem in its full generality is just inherently difficult.

The Great (or Grand) Renaming

by Guido van Rossum
Tuesday, March 31, 2009

When Python was first created, I always envisioned it as a stand-alone program, occasionally linking in third-party libraries. The source code therefore freely defined global names (in the C/linker sense) like ‘object’, ‘getlistitem’, ‘INCREF’ and so on. As Python’s popularity grew, people started to ask for an “embedded” version of Python, which would itself be a library that could be linked into other applications – not unlike the way that Emacs incorporates a Lisp interpreter.

Unfortunately, this embedding was complicated by name clashes between Python’s global names and those defined by the embedding application – the name 'object’ was especially popular. To deal with this problem, a naming convention was chosen, whereby all Python globals would have a name starting with “Py” or “_Py” (for internal names that had to be global for technical reasons) or “PY” (for macros).

For backwards compatibility reasons (there were already many third party extension modules) and to ease the transition for core developers (who had the old names engrained in their brain) there were two phases. In phase one the linker saw the old names, but the source code used the new names, which were translated to the old names using a large number of C preprocessor macros. In phase two the linker saw the new names, but for the benefit of some laggard extension modules that hadn’t been ported yet, another set of macros now translated the old names to the new names. In both phases, the code could mix old and new names and work correctly.

I researched the history of these renamings a bit in our Subversion logs. I found r4583 from January 12, 1995, which signalled phase two of the great renaming was started by introducing the new names to all header files. But in December 1996 the renaming of .c source files was still going on. Around this time the renaming seems to have been renamed, and checkin comments often refer to the "Grand Renaming". The backwards compatibility macros were finally removed in May 2000, as part of the Python 1.6 release effort. The check-in comment for r15313 celebrates this event.

Much credit goes to Barry Warsaw and Roger Masse, who participated in the unthankful task of renaming the contentes of file after file (albeit with the help of a script). They also helped with the equally tedious task of adding unit tests for much of the standard library.

Wikipedia has a reference to an earlier Great Renaming event, which apparently involved renaming USENET groups. I probably unconsciously remembered that event when I named Python's Great Renaming. I also found some references to a later Grand Renaming in Sphinx, the package used for generating Python's documentation. Zope also seems to have had a Grand Renaming, and some recent Py3k discussions also used the term for the PyString -> PyBytes renaming (although this is a minor one compared to the others).

Great or Grand Renamings are often traumatic events for software developer communities, since they requires the programmers' brains to be rewired, documentation to be rewritten, and complicate the integration of patches created before the renaming but applied after. (This is especially problematic when unrenamed branches exist.)

Origins of Python's "Functional" Features

by Guido van Rossum
Tuesday, April 21, 2009

I have never considered Python to be heavily influenced by functional languages, no matter what people say or think. I was much more familiar with imperative languages such as C and Algol 68 and although I had made functions first-class objects, I didn't view Python as a functional programming language. However, earlier on, it was clear that users wanted to do much more with lists and functions.

A common operation on lists was that of mapping a function to each of the elements of a list and creating a new list. For example:

```
def square(x):
   return x*x

vals = [1, 2, 3, 4]
newvals = []
for v in vals:
   newvals.append(square(v))
```

In functional languages such as Lisp and Scheme, operations such as this were provided as built-in functions of the language. Thus, early users familiar with such languages found themselves implementing comparable functionality in Python. For example:

```
def map(f, s):
   result = []
   for x in s:
           result.append(f(x))
   return result

def square(x):
   return x*x

vals = [1, 2, 3, 4]
newvals = map(square,vals)
```

A subtle aspect of the above code is that many people didn't like the fact that you to define the operation that you were applying to the list elements as a completely separate function. Languages such as Lisp allowed functions to simply be defined "on-the-fly" when making the map function call. For example, in Scheme you can create anonymous functions and perform mapping operations in a single expression using lambda, like this:

```
(map (lambda (x) (* x x)) '(1 2 3 4))
```

Although Python made functions first-class objects, it didn't have any similar mechanism for creating anonymous functions.

In late 1993, users had been throwing around various ideas for creating anonymous functions as well as various list manipulation functions such as map(), filter(), and reduce(). For example, Mark Lutz (author of "Programming Python") posted some code for a function that created functions using exec:

```
def genfunc(args, expr):
   exec('def f(' + args + '): return ' + expr)
   return eval('f')

# Sample usage
vals = [1, 2, 3, 4]
newvals = map(genfunc('x', 'x*x'), vals)
```

Tim Peters then followed up with a solution that simplified the syntax somewhat, allowing users to type the following:

```
vals = [1, 2, 3, 4]
newvals = map(func('x: x*x'), vals)
```

It was clear that there was a demand for such functionality. However, at the same time, it seemed pretty "hacky" to be specifying anonymous functions as code strings that you had to manually process through exec. Thus, in January, 1994, the map(), filter(), and reduce() functions were added to the standard library. In addition, the lambda operator was introduced for creating anonymous functions (as expressions) in a more straightforward syntax. For example:

```
vals = [1, 2, 3, 4]
newvals = map(lambda x:x*x, vals)
```

These additions represented a significant, early chunk of contributed code. Unfortunately I don't recall the author, and the SVN logs don't record this. If it's yours, leave a comment!

I was never all that happy with the use of the "lambda" terminology, but for lack of a better and obvious alternative, it was adopted for Python. After all, it was the choice of the now anonymous contributor, and at the time big changes required much less discussion than nowadays, for better and for worse.

Lambda was really only intended to be a syntactic tool for defining anonymous functions. However, the choice of terminology had many unintended consequences. For instance, users familiar with functional languages expected the semantics of lambda to match that of other languages. As a result, they found Python's implementation to be sorely lacking in advanced features. For example, a subtle problem with lambda is that the expression supplied couldn't refer to variables in the surrounding scope. For example, if you had this code, the map() function would break because the lambda function would run with an undefined reference to the variable 'a'.

```
def spam(s):
   a = 4
   r = map(lambda x: a*x, s)
```

There were workarounds to this problem, but they counter-intuitively involved setting default arguments and passing hidden arguments into the lambda expression. For example:

```
def spam(s):
   a = 4
   r = map(lambda x, a=a: a*x, s)
```

The "correct" solution to this problem was for inner functions to implicitly carry references to all of the local variables in the surrounding scope that are referenced by the function. This is known as a "closure" and is an essential aspect of functional languages. However, this capability was not introduced in Python until the release of version 2.2 (though it could be imported "from the future" in Python 2.1).

Curiously, the map, filter, and reduce functions that originally motivated the introduction of lambda and other functional features have to a large extent been superseded by list comprehensions and generator expressions. In fact, the reduce function was removed from list of builtin functions in Python 3.0. (However, it's not necessary to send in complaints about the removal of lambda, map or filter: they are staying. :-)

It is also worth nothing that even though I didn't envision Python as a functional language, the introduction of closures has been useful in the development of many other advanced programming features. For example, certain aspects of new-style classes, decorators, and other modern features rely upon this capability.

Lastly, even though a number of functional programming features have been introduced over the years, Python still lacks certain features found in “real” functional programming languages. For instance, Python does not perform certain kinds of optimizations (e.g., tail recursion). In general, because Python's extremely dynamic nature, it is impossible to do the kind of compile-time optimization known from functional languages like Haskell or ML. And that's fine.

And the Snake Attacks

by Greg Stein
Wednesday, April 22, 2009

Alright. Fine. In 1995, when I was first exposed to Python, any reference to "snake" was verboten. Python was named after Monty Python, not the reptile. If anybody was attacking, it was Knights who say Ni or possibly the Rabbit of Caerbannog.

In any case... back in 1994, I was battling fictitious baddies in the LPMUD scene. The Web was barely present, and broadband was unheard of. Low-bandwidth entertainment was the order of the day.

Wait. One more step back. 1979. My first computer was an Apple][, and one of my favorite games was the Colossal Cave. Soon after that, I learned about and played Zork. I fell in love with the concept of interaction fiction and how a computer could lead you through these stories. These games hooked me, and led me into a life of computers. *(and yes, you can imagine my ecstasy at meeting Don Woods some 25+ years later!)*

So the MUD scene was quite interesting to me. But I wanted to help *build* these games. I met John Viega, a fellow LPMUD game author, coder, and designer. At the time, he was working at the University of Virginia in their Computer Graphics Lab, working on a system called Alice. Their system was intended for less computer-savvy and they wanted an easy-to-learn language for people to create animations. They chose Python for its clarity, power, and simplicity.

John was a huge fan, and pointed me at Python. "You have to learn this!" "Fine. Fine.", I said. The language was easy, yet powerful. It could do everything, without a lot of hassle.

That was February, 1995, and I haven't turned back.

Little did I know, at the time, just how pivotal Python would be to my career and my life. Thank you, Guido, for your creation.

New! Now in Japanese!

by Guido van Rossum
Thursday, April 23, 2009

There's now a Japanese translation of this blog. Yay! There is also a Spanish version, and a French version (sorry, I don't know the URL yet -- let me know if you do).

I don't read those languages (or not well enough) and am not in a position to "approve" them, but I'm fine with their existence. If you want to start another translation, be my guest -- send me a link to the blog so I can update this page. If you see a mistake in a translation, just contact the translator directly. (However, I do *not* want to see *copies* of my blogs appearing. There's no point for those, and typically sites that copy popular blogs are just trying to attract eyeballs to their ads by reproducing popular material without putting any creative work into it.)

Metaclasses and Extension Classes (a.k.a. “The Killer Joke”)

by Guido van Rossum
Friday, April 24, 2009

In Python’s original implementation, classes were first class objects that could be manipulated just like any other object. However, the process of creating a class object was something that was set in stone. Specifically, when you defined a class such as this

```
class ClassName(BaseClass, ...):
    ...method definitions...
```

The body of the class would be executed in its own local dictionary. The name of the class, a tuple of base classes, and this local dictionary would then be passed to an internal class creation function that was responsible for creating the class object. Since all of this machinery was hidden away behind the scenes, it was an implementation detail that users didn’t need to worry about.

Don Beaudry was the first to point out that this was a missed opportunity for expert users. Specifically, if classes were simply special kinds of objects, why couldn’t you create new kinds of classes that could be customized to behave in different ways? He then suggested a very slight modification to the interpreter that would allow new kinds of class objects to be created by C code extension modules. This modification, first introduced in 1995, has long been known as the “Don Beaudry hook” or “Don Beaudry hack”, where the ambiguity about the name was an intentional joke. Jim Fulton later generalized the modification where it remained a part of the language (albeit under-documented) until it was replaced by true support for metaclasses through the introduction of new-style classes in Python 2.2 (see below).

The basic idea behind the Don Beaudry hook is that expert users would be able to create custom class objects if there was some way to supply a user-supplied function in the final step of class creation. Specifically, if the class name, base classes, and local dictionary could be passed to a different construction function, then that function could do whatever it wanted with the information to create a class object. The only catch was that I didn't want to make any changes to the class syntax which had already been well-established.

To do this, the hook required you to create a new type object in C that was callable. Then, when an instance of such a callable type was used as a base class in a class statement, the class creation code would magically call that type object instead of creating a standard class object. The behavior of classes created this way (and their instances) was completely up to the extension module providing the callable type object.

To modern Python users it may sound strange that this was considered such a hack. But at the time, type objects were not callable -- e.g. 'int' was not a built-in type but a built-in function that returned an instance of the int object, and the int type was neither easily accessible nor callable. User-defined classes were of course callable, but this was originally special-cased by the CALL instruction, as these were implemented in a completely different way than built-in types. Don Beaudry is eventually responsible for planting the insight in my head that led first to metaclasses and later to new-style classes and the eventual death of classic classes.

Originally, Don Beaudry’s own set of Python extensions named MESS was the only user of this feature. However, by the end of 1996, Jim Fulton had developed a very popular third-party package called Extension Classes, which used the Don Beaudry hook. The Extension Classes package eventually became unnecessary after Python 2.2 introduced metaclasses as a standard part of the object machinery.

In Python 1.5, I removed the requirement to write a C extension in order to use the Don Beaudry hook. In addition to check for a callable base class type, the class creation code now also checked for an attribute named "__class__" on the base class, and would call this if present. I wrote an essay about this feature that was the first introduction to the idea of metaclasses for many Python users. Due to the head-exploding nature of the ideas presented therein, the essay was soon nicknamed "The Killer Joke" (a Monty Python reference).

Perhaps the most lasting contribution of the Don Beaudry hook is the API for the class creation function, which was carried over to the new metaclass machinery in Python 2.2. As described earlier, the class creation function is called with three arguments: a string giving the class name, a tuple giving the base classes (possibly empty or a singleton), and a dictionary giving the contents of the namespace in which the indented block with method definitions (and other class-level code) has been executed. The return value of the class creation function is assigned to a variable whose name is the class name.

Originally, this was simply the internal API for creating classes. The Don Beaudry hook used the same call signature and hence it became a public API. The most important aspect of this API is that the block containing the method definitions is executed before the class creation function is called. This places certain restrictions on the effectiveness of metaclasses, since a metaclass cannot influence the initial contents of the namespace in which the method definitions are executed.

This was changed in Python 3000, so that now a metaclass can provide an alternative mapping object in which the class body is executed. In order to support this, the syntax for specifying an explicit metaclass is also changed: it uses keyword argument syntax in the list of base classes, which was introduced for this purpose.

In the next episode I'll write more about how the idea of metaclasses led to the introduction of new-style classes in 2.2 (and the eventual demise of classic classes in 3.0).

New-style Classes

by Guido van Rossum
Monday, June 21, 2010

[After a long hiatus, this blog series is back! I will continue where I left off last year. I'll try to keep the frequency up.]

Earlier, I described how the addition of classes to Python was essentially an afterthought. The implementation chosen was definitely an example of Python's "cut corners" philosophy. However, as Python evolved, various problems with the class implementation became a popular topic of griping from expert Python users.

One problem with the class implementation was that there was no way to subclass built-in types. For example, lists, dictionaries, strings, and other objects were somehow "special" and could not be specialized via subclassing. This limitation seemed rather odd for a language that claimed to be "object oriented."

Another problem was that whole type system just seemed to be "wrong" with user defined classes. For example, if you created two objects a and b, statements such as type(a) == type(b) would evaluate as True even if a and b were instances of completely unrelated classes. Needless to say, developers who were familiar with languages such as C++ and Java found this be rather odd since in those languages, classes were tightly integrated with the underlying type system.

In Python 2.2, I finally took the time to reimplement classes and "do it right." This change was, by far, the most ambitious rewrite of a major Python subsystem to date and one could certainly accuse me of a certain amount of "second-system syndrome" in this effort. Not only did I address the immediate problem of allowing built-in types to be subclassed, I also added support for true metaclasses, attempted to fix the naïve method resolution order for multiple inheritance, and added a variety of other features. A major influence on this work was the book "Putting Metaclasses to Work" by Ira Forman and Scott Danforth, which provided me with a specific notion of what a metaclass is, different from a similar concept in Smalltalk.

An interesting aspect of the class rewrite was that new-style classes were introduced as a new language feature, not as a strict replacement for old-style classes. In fact, for backwards compatibility, the old class implementation remains the default class creation protocol in Python 2. To create a new-style class, you simply have to subclass an existing new-style class such as object (which is the root for the new-style class hierarchy). For example:

```
class A(object):
statements
...
```

The change to new-style classes has been a major success. The new metaclasses have become popular amongst framework writers and explaining classes has actually become easier since there were fewer exceptions. The fact that backwards compatibility was maintained meant that old code has continued to work as new-style classes have evolved. Finally, although the old-style classes will eventually be removed from the languages, users are getting used to writing "class MyClass(object)" to declare a class, which isn't so bad.

The Inside Story on New-Style Classes

by Guido van Rossum
Monday, June 21, 2010

[Warning, this post is long and gets very technical.]

On the surface, new-style classes appear very similar to the original class implementation. However, new-style classes also introduced a number of new concepts:

- low-level constructors named __new__()
- descriptors, a generalized way to customize attribute access
- static methods and class methods
- properties (computed attributes)
- decorators (introduced in Python 2.4)

- slots
- a new Method Resolution Order (MRO)

In the next few sections, I will try to shine some light on these concepts.

Low-level constructors and __new__()

Traditionally, classes defined an __init__() method which defined how new instances are initialized after their creation. However, in some cases, the author of a class may want to customize how instances are created—for example, if an object was being restored from a persistent database. Old-style classes never really provided a hook for customizing the creation of objects although there were library modules allowed certain kinds of objects to be created in non-standard ways (e.g., the "new" module).

New-style classes introduced a new class method __new__() that lets the class author customize how new class instances are created. By overriding __new__() a class author can implement patterns like the Singleton Pattern, return a previously created instance (e.g., from a free list), or to return an instance of a different class (e.g., a subclass). However, the use of __new__ has other important applications. For example, in the pickle module, __new__ is used to create instances when unserializing objects. In this case, instances are created, but the __init__ method is not invoked.

Another use of __new__ is to help with the subclassing of immutable types. By the nature of their immutability, these kinds of objects can not be initialized through a standard __init__() method. Instead, any kind of special initialization must be performed as the object is created; for instance, if the class wanted to modify the value being stored in the immutable object, the __new__ method can do this by passing the modified value to the base class __new__ method.

Descriptors

Descriptors are a generalization of the concept of bound methods, which was central to the implementation of classic classes. In classic classes, when an instance attribute is not found in the instance dictionary, the search continues with the class dictionary, then the dictionaries of its base classes, and so on recursively. When the attribute is found in a class dictionary (as opposed to in the instance dictionary), the interpreter checks if the object found is a Python function object. If so, the returned value is not the object found, but a wrapper object that acts as a currying function. When the wrapper is called, it calls the original function object after inserting the instance in front of the argument list.

For example, consider an instance x of a class C. Now, suppose that one makes a method call x.f(0). This operation looks up the attribute named "f" on x and calls it with an argument of 0. If "f" corresponds to a method defined in the class, the attribute request returns a wrapper function that behaves approximately like the function in this Python pseudocode:

```
def bound_f(arg):
    return f(x, arg)
```

When the wrapper is called with an argument of 0, it calls "f" with two arguments: x and 0. This is the fundamental mechanism by which methods in classes obtain their "self" argument.

Another way to access the function object f (without wrapping it) is to ask the class C for an attribute named “f”. This kind of search does not return a wrapper but simply returns the function f. In other words, x.f(0) is equivalent to C.f(x, 0). This is a pretty fundamental equivalence in Python.

For classic classes, if the attribute search finds any other kind of object, no wrapper is created and the value found in the class dictionary is returned unchanged. This makes it possible to use class attributes as “default” values for instance variables. For example, in the above example, if class C has an attribute named “a” whose value is 1, and there is no key “a” in x’s instance dictionary, then x.a equals 1. Assignment to x.a will create an key “a” in x’s instance dictionary whose value will shadow the class attribute (by virtue of the attribute search order). Deleting x.a will reveal the shadowed value (1) again.

Unfortunately, some Python developers were discovering the limitation of this scheme. One limitation was that it was prevented the creation of “hybrid” classes that had some methods implemented in Python and others in C, because only Python functions were being wrapped in such a way as to provide the method with access to the instance, and this behavior was hard-coded in the language. There was also no obvious way to define different kinds of methods such as a static member functions familiar to C++ and Java programmers.

To address this issue, Python 2.2 introduced a straightforward generalization of the above wrapping behavior. Instead of hard-coding the behavior that Python function objects are wrapped and other objects aren’t, the wrapping is now left up to the object found by the attribute search (the function f in the above example). If the object found by an attribute search happens to have a special method named __get__, it is considered to be a "descriptor" object. The __get__ method is then called and whatever is returned is used to produce the result of the attribute search. If the object has no __get__ method, it is returned unchanged. To obtain the original behavior (wrapping function objects) without special-casing function objects in the instance attribute lookup code, function objects now have a __get__ method that returns a wrapper as before. However, users are free to define other classes with methods named __get__, and their instances, when found in a class dictionary during an instance attribute lookup, can also wrap themselves in any way they like.

In addition to generalizing the concept of attribute lookup, it also made sense to extend this idea for the operations of setting or deleting an attribute. Thus, a similar scheme is used for assignment operations such as x.a = 1 or del x.a. In these cases, if the attribute "a" is found in the instance's class dictionary (not in the instance dictionary), the object stored in the class dictionary is checked to see if it has a __set__ and __delete__ special method respectively. (Remember that __del__ has a completely different meaning already.) Thus, by redefining these methods, a descriptor object can have complete control over what it means to get, set, and delete an attribute. However, it's important emphasize that this customization only applies when a descriptor instance appears in a class dictionary—not the instance dictionary of an object.

staticmethod, classmethod, and property

Python 2.2 added three predefined classes: classmethod, staticmethod, and property, that utilized the new descriptor machinery. classmethod and staticmethod were simply wrappers for function objects, implementing different __get__ methods to return different kinds of wrappers for calling the underlying function. For instance, the staticmethod wrapper calls the function without modifying the argument list at all. The classmethod wrapper calls the function with the instance's class object set as the first argument instead of the instance itself. Both can be called via an instance or via the class and the arguments will be the same.

The property class is a wrapper that turned a pair of methods for getting and setting a value into an "attribute." For example, if you have a class like this,

```
class C(object):
   def set_x(self, value):
       self.__x = value
   def get_x(self):
       return self.__x
```

a property wrapper could be used to make an attribute "x" that when accessed, would implicitly call the get_x and set_x methods.

When first introduced, there was no special syntax for using the classmethod, staticmethod, and property descriptors. At the time, it was deemed too controversial to simultaneously introduce a major new feature along with new syntax (which always leads to a heated debate). Thus, to use these features, you would define your class and methods normally, but add extra statements that would "wrap" the methods. For example:

```
class C:
   def foo(cls, arg):
       ...
   foo = classmethod(foo)
   def bar(arg):
       ...
   bar = staticmethod(bar)
```

For properties, a similar scheme was used:

```
class C:
def set_x(self, value):
    self.__x = value
def get_x(self):
    return self.__x
x = property(get_x, set_x)
```

Decorators

A downside of this approach is that the reader of a class had to read all the way till the end of a method declaration before finding out whether it was a class or static method (or some user-defined variation). In Python 2.4, new syntax was finally introduced, allowing one to write the following instead:

```
class C:
 @classmethod
 def foo(cls, arg):
    ...
 @staticmethod
 def bar(arg):
    ...
```

The construct @expression, on a line by itself before a function declaration, is called a decorator. (Not to be confused with descriptor, which refers to a wrapper implementing __get__; see above.) The particular choice of decorator syntax (derived from Java's annotations) was debated endlessly before it was decided by "BDFL pronouncement". (David Beazley wrote a piece about the history of the term BDFL that I'll publish separately.)

The decorator feature has become one of the more successful language features, and the use of custom decorators has exceeded my widest expectations. Especially web frameworks have found lots of uses for them. Based on this success, in Python 2.6, the decorator syntax was extended from function definitions to include class definitions.

Slots

Another enhancement made possible with descriptors was the introduction of the __slots__ attribute on classes. For example, a class could be defined like this:

```
class C:
 __slots__ = ['x','y']
 ...
```

The presence of __slots__ does several things. First, it restricts the valid set of attribute names on an object to exactly those names listed. Second, since the attributes are now fixed, it is no longer necessary to store attributes in an instance dictionary, so the __dict__ attribute is removed (unless a base class already has it; it can also be added back by a subclass that doesn't use __slots__). Instead, the attributes can be stored in predetermined locations within an array. Thus, every slot attribute is actually a descriptor object that knows how to set/get each attribute using an array index. Underneath the covers, the implementation of this feature is done entirely in C and is highly efficient.

Some people mistakenly assume that the intended purpose of __slots__ is to increase code safety (by restricting the attribute names). In reality, my ultimate goal was performance. Not only was __slots__ an interesting application of descriptors, I feared that all of the changes in the class system were going to have a negative impact on performance. In particular, in order to make data descriptors work properly, any manipulation of an object's attributes first involved a check of the class dictionary to see if that attribute was, in fact, a data descriptor. If so, the descriptor was used to handle the attribute access instead of manually manipulating the instance dictionary as is normally the case. However, this extra check also meant that an extra lookup would be performed prior to inspecting the dictionary of each instance. Thus the use of __slots__ was a way to optimize the lookup of data attributes—a fallback, if you will, in case people were disappointed with the performance impact of the new class system. This turned out unnecessary, but by that time it was of course too late to remove __slots__. Of course, used properly, slots really can increase performance, especially by reducing memory footprint when many small objects are created.

I'll leave the history of Python's Method Resolution Order (MRO) to the next post.

import this and The Zen of Python

by Guido van Rossum
Monday, June 21, 2010

Barry Warsaw posted an interesting blog that tells an obscure part of Python history (the kind I like to keep alive): http://www.wefearchange.org/2010/06/import-this-and-zen-of-python.html

import antigravity

by Guido van Rossum
Monday, June 21, 2010

The antigravity module, referencing the XKCD comic mentioning Python, was added to Python 3 by Skip Montanaro. You can read more about it here, one of the first spottings that I know of: http://sciyoshi.com/blog/2008/dec/30/import-antigravity/).

But it really originated in Google App Engine! It was a last-minute addition when we launched App Engine on April 7, 2008. A few weeks before launch, when most code was already frozen, the App Engine team at Google decided we wanted an easter egg. After a great many suggestions, some too complex, some too obscure, some too risky, we chose the "antigravity" module. The App Engine version is a little more elaborate than the Python 3 version: it defines a fly() function while can randomly do one of two things: with a probability of 10%, it redirects to the XKCD comic; otherwise, it simply renders the text of the comic in HTML (with a link to the comic on the last line). To invoke it in your App Engine app, you'd have to write a little main program, like this:

```
import antigravity

def main():
   antigravity.fly()

if __name__ == '__main__':
   main()
```

Update: The Python 3 stdlib version has an easter egg inside the easter egg, if you inspect the source code: it defines a function that purports to implement the XKCD geohashing algorithm.

Method Resolution Order

by Guido van Rossum
Wednesday, June 23, 2010

In languages that use multiple inheritance, the order in which base classes are searched when looking for a method is often called the Method Resolution Order, or MRO. (In Python this also applies to other attributes.) For languages that support single inheritance only, the MRO is uninteresting; but when multiple inheritance comes into play, the choice of an MRO algorithm can be remarkably subtle. Python has known at least three different MRO algorithms: classic, Python 2.2 new-style, and Python 2.3 new-style (a.k.a. C3). Only the latter survives in Python 3.

Classic classes used a simple MRO scheme: when looking up a method, base classes were searched using a simple depth-first left-to-right scheme. The first matching object found during this search would be returned. For example, consider these classes:

```
class A:
 def save(self): pass

class B(A): pass

class C:
 def save(self): pass

class D(B, C): pass
```

If we created an instance x of class D, the classic method resolution order would order the classes as D, B, A, C. Thus, a search for the method x.save() would produce A.save() (and not C.save()). This scheme works fine for simple cases, but has problems that become apparent when one considers more complicated uses of multiple inheritance. One problem concerns method lookup under "diamond inheritance." For example:

```
class A:
 def save(self): pass

class B(A): pass

class C(A):
 def save(self): pass

class D(B, C): pass
```

Here, class D inherits from B and C, both of which inherit from class A. Using the classic MRO, methods would be found by searching the classes in the order D, B, A, C, A. Thus, a reference to x.save() will call A.save() as before. However, this is unlikely what you want in this case! Since both B and C inherit from A, one can argue that the redefined method C.save() is actually the method that you want to call, since it can be viewed as being "more specialized" than the method in A (in fact, it probably calls A.save() anyways). For instance, if the save() method is being used to save the state of an object, not calling C.save() would break the program since the state of C would be ignored.

Although this kind of multiple inheritance was rare in existing code, new-style classes would make it commonplace. This is because all new-style classes were defined by inheriting from a base class object. Thus, any use of multiple inheritance in new-style classes would always create the diamond relationship described above. For example:

```
class B(object): pass

class C(object):
 def __setattr__(self, name, value): pass

class D(B, C): pass
```

Moreover, since object defined a number of methods that are sometimes extended by subtypes (e.g., __setattr__()), the resolution order becomes critical. For example, in the above code, the method C.__setattr__ should apply to instances of class D.

To fix the method resolution order for new-style classes in Python 2.2, I adopted a scheme where the MRO would be pre-computed when a class was defined and stored as an attribute of each class object. The computation of the MRO was officially documented as using a depth-first left-to-right traversal of the classes as before. If any class was duplicated in this search, all but the last occurrence would be deleted from the MRO list. So, for our earlier example, the search order would be D, B, C, A (as opposed to D, B, A, C, A with classic classes).

In reality, the computation of the MRO was more complex than this. I discovered a few cases where this new MRO algorithm didn't seem to work. Thus, there was a special case to deal with a situation when two bases classes occurred in a different order in the inheritance list of two different derived classes, and both of those classes are inherited by yet another class. For example:

```
class A(object): pass
class B(object): pass
class X(A, B): pass
class Y(B, A): pass
class Z(X, Y): pass
```

Using the tentative new MRO algorithm, the MRO for these classes would be Z, X, Y, B, A, object. (Here 'object' is the universal base class.) However, I didn't like the fact that B and A were in reversed order. Thus, the real MRO would interchange their order to produce Z, X, Y, A, B, object. Intuitively, this algorithm tried to preserve the order of classes for bases that appeared first in the search process. For instance, on class Z, the base class X would be checked first because it was first in the inheritance list. Since X inherited from A and B, the MRO algorithm would try to preserve that ordering. This is what I implemented for Python 2.2, but I documented the earlier algorithm (naïvely thinking it didn't matter much).

However, shortly after the introduction of new-style classes in Python 2.2, Samuele Pedroni discovered an inconsistency between the documented MRO algorithm and the results that were actually observed in real-code. Moreover, inconsistencies were occurring even in code that did not fall under the special case observed above. After much discussion, it was decided that the MRO adopted for Python 2.2 was broken and that Python should adopt the C3 Linearization algorithm described in the paper "A Monotonic Superclass Linearization for Dylan" (K. Barrett, et al, presented at OOPSLA'96).

Essentially, the main problem in the Python 2.2 MRO algorithm concerned the issue of monotonicity. In a complex inheritance hierarchy, each inheritance relationship defines a simple set of rules concerning the order in which classes should be checked. Specifically, if a class A inherits from class B, then the MRO should obviously check A before B. Likewise, if a class B uses multiple inheritance to inherit from C and D, then B should be checked before C and C should be checked before D.

Within a complex inheritance hierarchy, you want to be able to satisfy all of these possible rules in a way that is monotonic. That is, if you have already determined that class A should be checked before class B, then you should never encounter a situation that requires class B to be checked before class A (otherwise, the result is undefined and the inheritance hierarchy should be rejected). This is where the original MRO got it wrong and where the C3 algorithm comes into play. Basically, the idea behind C3 is that if you write down all of the ordering rules imposed by inheritance relationships in a complex class hierarchy, the algorithm will determine a monotonic ordering of the classes that satisfies all of them. If such an ordering can not be determined, the algorithm will fail.

Thus, in Python 2.3, we abandoned my home-grown 2.2 MRO algorithm in favor of the academically vetted C3 algorithm. One outcome of this is that Python will now reject any inheritance hierarchy that has an inconsistent ordering of base classes. For instance, in the previous example, there is an ordering conflict between class X and Y. For class X, there is a rule that says class A should be checked before class B. However, for class Y, the rule says that class B should be checked before A. In isolation, this discrepancy is fine, but if X and Y are ever combined together in the same inheritance hierarchy for another class (such as in the definition of class Z), that class will be rejected by the C3 algorithm. This, of course, matches the Zen of Python's "errors should never pass silently" rule.

From List Comprehensions to Generator Expressions

by Guido van Rossum
Tuesday, June 29, 2010

List comprehensions were added in Python 2.0. This feature originated as a set of patches by Greg Ewing with contributions by Skip Montanaro and Thomas Wouters. (IIRC Tim Peters also strongly endorsed the idea.) Essentially, they are a Pythonic interpretation of a well-known notation for sets used by mathematicians. For example, it is commonly understood that this:

```
{x | x > 10}
```

refers to the set of all x such that x > 10. In math, this form implies a universal set that is understood by the reader (for example, the set of all reals, or the set of all integers, depending on the context). In Python, there is no concept of a universal set, and in Python 2.0, there were no sets. (Sets are an interesting story, of which more in a future blog post.)

This and other considerations led to the following notation in Python:

```
[f(x) for x in S if P(x)]
```

This produces a list containing the values of the sequence S selected by the predicate P and mapped by the function f. The if-clause is optional, and multiple for-clauses may be present, each with their own optional if-clause, to represent nested loops (the latter feature is rarely used though, since it typically maps a multi-dimensional entity to a one-dimensional list).

List comprehensions provide an alternative to using the built-in map() and filter() functions. map(f, S) is equivalent to [f(x) for x in S] while filter(P, S) is equivalent to [x for x in S if P(x)]. One would think that list comprehensions have little to recommend themselves over the seemingly more compact map() and filter() notations. However, the picture changes if one looks at a more realistic example. Suppose we want to add 1 to the elements of a list, producing a new list. The list comprehension solution is [x+1 for x in S]. The solution using map() is map(lambda x: x+1, S). The part "lambda x: x+1" is Python's notation for an anonymous function defined in-line.

It has been argued that the real problem here is that Python's lambda notation is too verbose, and that a more concise notation for anonymous functions would make map() more attractive. Personally, I disagree—I find the list comprehension notation much easier to read than the functional notation, especially as the complexity of the expression to be mapped increases. In addition, the list comprehension executes much faster than the solution using map and lambda. This is because calling a lambda function creates a new stack frame while the expression in the list comprehension is evaluated without creating a new stack frame.

Given the success of list comprehensions, and enabled by the invention of generators (of which more in a future episode), Python 2.4 added a similar notation that represents a sequence of results without turning it into a concrete list. The new feature is called a "generator expression". For example:

```
sum(x**2 for x in range(1, 11))
```

This calls the built-in function sum() with as its argument a generator expression that yields the squares of the numbers from 1 through 10 inclusive. The sum() function adds up the values in its argument resulting in an answer of 385. The advantage over sum([x**2 for x in range(1, 11)]) should be obvious. The latter creates a list containing all the squares, which is then iterated over once before it is thrown away. For large collections these savings in memory usage are an important consideration.

I should add that the differences between list comprehensions and generator expressions are fairly subtle. For example, in Python 2, this is a valid list comprehension:

```
[x**2 for x in 1, 2, 3]
```

However this is not a valid generator expression:

```
(x**2 for x in 1, 2, 3)
```

We can fix it by adding parentheses around the "1, 2, 3" part:

```
(x**2 for x in (1, 2, 3))
```

In Python 3, you also have to use these parentheses for the list comprehension:

```
[x**2 for x in (1, 2, 3)]
```

However, in a "regular" or "explicit" for-loop, you can still omit them:

```
for x in 1, 2, 3: print(x**2)
```

Why the differences, and why the changes to a more restrictive list comprehension in Python 3? The factors affecting the design were backwards compatibility, avoiding ambiguity, the desire for equivalence, and evolution of the language. Originally, Python (before it even had a version :-) only had the explicit for-loop. There is no ambiguity here for the part that comes after 'in': it is always followed by a colon. Therefore, I figured that if you wanted to loop over a bunch of known values, you shouldn't be bothered with having to put parentheses around them. This also reminded me of Algol-60, where you can write:

for i := 1, 2, 3 **do** *Statement*

except that in Algol-60 you can also replace each expression with step-until clause, like this:

for i := 1 **step** 1 **until** 10, 12 **step** 2 **until** 50, 55 **step** 5 **until** 100 **do** *Statement*

(In retrospect it would have been cool if Python for-loops had the ability to iterate over multiple sequences as well. Alas...)

When we added list comprehensions in Python 2.0, the same reasoning applied: the sequence expression could only be followed by a close bracket ']' or by a 'for' or 'if' keyword. And it was good.

But when we added generator expressions in Python 2.4, we ran into a problem with ambiguity: the parentheses around a generator expression are not technically part of the generator expression syntax. For example, in this example:

```
sum(x**2 for x in range(10))
```

the outer parentheses are part of the call to sum(), and a "bare" generator expression occurs as the first argument. So in theory there would be two interpretations for something like this:

```
sum(x**2 for x in a, b)
```

This could either be intended as:

```
sum(x**2 for x in (a, b))
```

or as:

```
sum((x**2 for x in a), b)
```

After a lot of hemming and hawing (IIRC) we decided not to guess in this case, and the generator comprehension was required to have a single expression (evaluating to an iterable, of course) after its 'in' keyword. But at the time we didn't want to break existing code using the (already hugely popular) list comprehensions.

Then when we were designing Python 3, we decided that we wanted the list comprehension:

```
[f(x) for x in S if P(x)]
```

to be fully equivalent to the following expansion using the built-in list() function applied to a generator expression:

```
list(f(x) for x in S if P(x))
```

Thus we decided to use the slightly more restrictive syntax of generator expressions for list comprehensions as well.

We also made another change in Python 3, to improve equivalence between list comprehensions and generator expressions. In Python 2, the list comprehension "leaks" the loop control variable into the surrounding scope:

```
x = 'before'
a = [x for x in 1, 2, 3]
print x # this prints '3', not 'before'
```

This was an artifact of the original implementation of list comprehensions; it was one of Python's "dirty little secrets" for years. It started out as an intentional compromise to make list comprehensions blindingly fast, and while it was not a common pitfall for beginners, it definitely stung people occasionally. For generator expressions we could not do this. Generator expressions are implemented using generators, whose execution requires a separate execution frame. Thus, generator expressions (especially if they iterate over a short sequence) were less efficient than list comprehensions.

However, in Python 3, we decided to fix the "dirty little secret" of list comprehensions by using the same implementation strategy as for generator expressions. Thus, in Python 3, the above example (after modification to use print(x) :-) will print 'before', proving that the 'x' in the list comprehension temporarily shadows but does not override the 'x' in the surrounding scope.

And before you start worrying about list comprehensions becoming slow in Python 3: thanks to the enormous implementation effort that went into Python 3 to speed things up in general, both list comprehensions and generator expressions in Python 3 are actually *faster* than they were in Python 2! (And there is no longer a speed difference between the two.)

UPDATE: Of course, I forgot to mention that Python 3 also supports set comprehensions and dictionary comprehensions. These are straightforward extensions of the list comprehension idea.

Why Python's Integer Division Floors

by Guido van Rossum
Tuesday, August 24, 2010

I was asked (again) today to explain why integer division in Python returns the floor of the result instead of truncating towards zero like C.

For positive numbers, there's no surprise:

```
>>> 5//2
2
```

But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity):

```
>>> -5//2
-3
>>> 5//-2
-3
```

This disturbs some people, but there is a good mathematical reason. The integer division operation (//) and its sibling, the modulo operation (%), go together and satisfy a nice mathematical relationship (all variables are integers):

a/b = q with remainder r

such that

b*q + r = a and 0 <= r < b

(assuming a and b are >= 0).

If you want the relationship to extend for negative a (keeping b positive), you have two choices: if you truncate q towards zero, r will become negative, so that the invariant changes to 0 <=

abs(r) < otherwise, you can floor q towards negative infinity, and the invariant remains 0 <= r < b. **[update: fixed this para]**

In mathematical number theory, mathematicians always prefer the latter choice (see e.g. Wikipedia). For Python, I made the same choice because there are some interesting applications of the modulo operation where the sign of a is uninteresting. Consider taking a POSIX timestamp (seconds since the start of 1970) and turning it into the time of day. Since there are 24*3600 = 86400 seconds in a day, this calculation is simply t % 86400. But if we were to express times before 1970 using negative numbers, the "truncate towards zero" rule would give a meaningless result! Using the floor rule it all works out fine.

Other applications I've though of are computations of pixel positions in computer graphics. I'm sure there are more.

For negative b, by the way, everything just flips, and the invariant becomes:

0 >= r > b.

So why doesn't C do it this way? Probably the hardware didn't do this at the time C was designed. And the hardware probably didn't do it this way because in the oldest hardware, negative numbers were represented as "sign + magnitude" rather than the two's complement representation used these days (at least for integers). My first computer was a Control Data mainframe and it used one's complement for integers as well as floats. A pattern of 60 ones meant negative zero!

Tim Peters, who knows where all Python's floating point skeletons are buried, has expressed some worry about my desire to extend these rules to floating point modulo. He's probably right; the truncate-towards-negative-infinity rule can cause precision loss for x%1.0 when x is a very small negative number. But that's not enough for me to break integer modulo, and // is tightly coupled to that.

PS. Note that I am using // instead of / -- this is Python 3 syntax, and also allowed in Python 2 to emphasize that you know you are invoking integer division. The / operator in Python 2 is ambiguous, since it returns a different result for two integer operands than for an int and a float or two floats. But that's a totally separate story; see PEP 238.

Karin Dewar, Indentation and the Colon

by Guido van Rossum
Friday, July 8, 2011

In a recent post on my other blog I mentioned a second-hand story about how Python's indentation was invented by the wife of Robert Dewar. I added that I wasn't very sure of the details, and I'm glad I did, because the truth was quite different. I received a long email from Lambert Meertens with the real story. I am going to quote it almost completely here, except for some part which he requested not to be quoted. In summary: Karin Dewar provided the inspiration for the use of the colon in ABC (and hence in Python) leading up to the indentation, not for indentation itself. Here is Lambert's email:

> The Dewar story is not about indentation, but about the invention of the colon.
>
> First about indentation in *B*. Already *B*0, the first iteration in the *B*0, *B*1, *B*2, ... sequence of designs leading to ABC, had non-optional indentation for grouping, supplemented by enclosing the group between `BEGIN` and `END` delimiters. This can be seen in [GM76], section 4.1 (*Layout*). The indentation was supposed to be added, like pretty printing, by a dedicated *B* editor, and the user had no control over this; they were not supposed to be able to turn this off or otherwise modify the indentation regime.
>
> Given the mandatory indentation, separate `BEGIN` and `END` delimiters are of course superfluous; in *B*1 we had no `BEGIN`, but only `END IF`, `END FOR`, and so on, and then the latter delimiters were also dropped in *B*2, leaving pure indentation as the sole indicator of grouping. See [ME81], Section 4 (STATEMENT SYNTAX).
>
> The origin of indentation in ABC is thus simply the desire to have the program text look neat and be suggestive of the meaning, codifying what was already common practice but not enforced. The decision to remove the noise of `BEGIN` ... `END` may have been influenced by [PL75], which actually advocated using pure indentation for grouping. Since occam came later (1983), the feature can't have been copied from that language. Same for Miranda (1985). As far as I am aware, *B* was the first actually published (and implemented) language to use indentation for grouping.
>
> Now the Dewar story, which is how I got the idea of the colon, as I wrote it down in a memoir of the ABC design rationale:

And here I will paraphrase, at Lambert's request.

In 1978, in a design session in a mansion in Jabłonna (Poland), Robert Dewar, Peter King, Jack Schwartz and Lambert were comparing various alternative proposed syntaxes for B, by comparing (buggy) bubble sort implementations written down in each alternative. Since they couldn't agree, Robert Dewar's wife was called from her room and asked for her opinion, like a modern-day Paris asked to compare the beauty of Hera, Athena, and Aphrodite. But after the first version was explained to her, she remarked: "You mean, in the line where it says: 'FOR i ... ', that it has to be done for the lines that follow; not just for that line?!" And here the scientists realized that the misunderstanding would have been avoided if there had been a colon at the end of that line.

Lambert also included the following useful references:

> [PL75] P. J. Plauger. Signal and noise in programming language. In J. D. White, editor, *Proc. ACM Annual Conference 1975*, page 216. ACM, 1975.
>
> [GM76] Leo Geurts and Lambert Meertens. Designing a beginners' programming language. In S.A. Schuman, editor, *New Directions in Algorithmic Languages 1975*, pages 1–18. IRIA, Rocquencourt, 1976.
>
> [ME81] Lambert Meertens. Issues in the design of a beginners' programming language. In J.W. de Bakker and J.C. van Vliet, editors, *Algorithmic Languages*, pages 167–184. North-Holland Publishing Company, Amsterdam, 1981.