

Visionäre der Programmierung

Die Sprachen und ihre Schöpfer



O'REILLY®

Federico Biancuzzi
Shane Warden

Deutsche Übersetzung Thomas Demmig

INHALT

	VORWORT	VII
	EINFÜHRUNG	IX
1	C++	1
	<i>Bjarne Stroustrup</i>	
	Designentscheidungen	2
	Anwenden der Sprache	6
	OOP und Nebenläufigkeit	10
	Zukunft	14
	Lehren	18
2	PYTHON	21
	<i>Guido von Rossum</i>	
	Der pythonische Weg	22
	Der gute Programmierer	29
	Viele Pythons	35
	Hilfen und Erfahrungen	39
3	APL	45
	<i>Adin D. Falkoff</i>	
	Papier und Stift	46
	Grundlegende Prinzipien	49
	Parallelität	55
	Vermächtnis	58
4	FORTH	61
	<i>Charles H. Moore</i>	
	Die Sprache Forth und das Sprachdesign	62
	Hardware	70
	Anwendungsdesign	73
5	BASIC	81
	<i>Thomas E. Kurtz</i>	
	Die Ziele von BASIC	82
	Compilerdesign	89
	Sprach- und Programmierpraktiken	92
	Sprachdesign	94
	Arbeitsziele	99

6	AWK	103
	<i>Alfred Aho, Peter Weinberger und Brian Kernighan</i>	
	Das Leben von Algorithmen	104
	Sprachdesign	106
	Unix und seine Kultur	108
	Die Rolle der Dokumentation	113
	Informatik	117
	Aufzucht kleiner Sprachen	119
	Entwerfen einer neuen Sprache	124
	Legacy-Kultur	131
	Transformative Technologien	134
	Bits, die das Universum ändern	139
	Theorie und Praxis	145
	Warten auf den Durchbruch	152
	Programming by Example	157
7	LUA	163
	<i>Luiz Henrique de Figueiredo und Roberto Ierusalimsky</i>	
	Die Macht der Skripten	164
	Erfahrung	167
	Sprachdesign	172
8	HASKELL	181
	<i>Simon Peyton Jones, Paul Hudak, Philip Wadler und John Hughes</i>	
	Ein funktionales Team	182
	Trajektorien der funktionalen Programmierung	184
	Die Sprache Haskell	191
	(Funktionales) Wissen verbreiten	198
	Formalisten und Evolution	200
9	ML	207
	<i>Robin Milner</i>	
	Die Solidität von Theoremen	208
	Die Theorie der Bedeutung	216
	Über Informatik hinaus	222
10	SQL	229
	<i>Don Chamberlin</i>	
	Ein bahnbrechender Artikel	230
	Die Sprache	233
	Feedback und Weiterentwicklung	237
	XQuery und XML	242

11	OBJECTIVE-C	247
	<i>Brad Cox und Tom Love</i>	
	Die Entwicklung von Objective-C	248
	Das Wachsen einer Sprache	250
	Ausbildung und Training	255
	Projektmanagement und alte Software	257
	Objective-C und andere Sprachen	264
	Komponenten, Sand und Steine	269
	Qualität als ökonomisches Phänomen	276
	Ausbildung	278
12	JAVA	283
	<i>James Gosling</i>	
	Stärke oder Einfachheit	284
	Eine Frage des Geschmacks	287
	Nebenläufigkeit	290
	Entwerfen einer Sprache	292
	Feedbackschleife	297
13	C#	301
	<i>Anders Hejlsberg</i>	
	Sprache und Design	302
	Wachsen einer Sprache	308
	C#	312
	Die Zukunft der Informatik	317
14	UML	323
	<i>Ivar Jacobson, James Rumbaugh und Grady Booch</i>	
	Lernen und Lehren	324
	Die Rolle der Leute	329
	UML	333
	Wissen	337
	Bereit für Änderungen	340
	Die Verwendung von UML	345
	Schichten und Sprachen	350
	Ein bisschen Wiederverwendbarkeit	354
	Symmetrische Relationen	359
	UML	362
	Sprachdesign	365
	Entwickler ausbilden	371
	Kreativität, Verbesserung und Muster	373

15	PERL	381
	<i>Larry Wall</i>	
	Die Sprache von Revolutionen	382
	Sprache	386
	Community	392
	Evolution und Revolution	396
16	POSTSCRIPT	401
	<i>Charles Geschke und John Warnock</i>	
	Entworfen für die Ewigkeit	402
	Forschung und Bildung	412
	Schnittstellen zur Langlebigkeit	416
	Standardwünsche	420
17	EIFFEL	423
	<i>Bertrand Meyer</i>	
	Ein inspirierender Nachmittag	424
	Wiederverwendbarkeit und Generik	431
	Korrigieren von Sprachen	435
	Wachstum und Evolution	442
	NACHWORT	447
	INTERVIEWPARTNER	449
	INDEX	465

Python

Guido van Rossum

Python ist eine moderne, universell einsetzbare High-Level-Sprache, die von Guido van Rossum als ein Ergebnis seiner Arbeit mit der Programmiersprache ABC entwickelt wurde. Die Philosophie von Python ist pragmatisch – ihre Benutzer sprechen häufig über das »Zen of Python« und bevorzugen einen einzigen, offensichtlichen Weg, um eine beliebige Aufgabe zu erledigen. Es gibt Portierungen für VMs wie die CLR von Microsoft und die JVM, aber die wichtigste Implementierung ist CPython, die immer noch von van Rossum und anderen Freiwilligen entwickelt wird. Gerade wurde Python 3.0 veröffentlicht, eine nicht abwärtskompatible Überarbeitung von Teilen der Sprache und ihrer Basisbibliotheken.

Der pythonische Weg

Was für Unterschiede gibt es zwischen dem Entwickeln einer Programmiersprache und dem eines »normalen« Softwareprojekts?

Guido van Rossum: Mehr als bei den meisten normalen Softwareprojekten sind hier die Programmierer selber die wichtigsten Anwender. Damit erhält ein Sprachprojekt viele »Meta«-Inhalte. Im Abhängigkeitsbaum von Softwareprojekten befinden sich Programmiersprachen ziemlich am untersten Ende – alles andere hängt von einer oder mehreren Sprachen ab. Damit wird es auch schwierig, eine Sprache zu ändern – eine inkompatible Änderung betrifft so viele abhängige Objekte, dass so etwas häufig gar nicht durchführbar ist. Mit anderen Worten: Alle Fehler, die einmal veröffentlicht werden, sind wie in Stein gemeißelt. Das ultimative Beispiel dafür ist vermutlich C++, das von Kompatibilitätsanforderungen belastet ist, durch die Code, der vor vielleicht 20 Jahren geschrieben wurde, immer noch gültig sein muss.

Wie debuggen Sie eine Sprache?

Guido: Gar nicht. Sprachdesign ist ein Bereich, in dem agile Entwicklungsmethoden einfach keinen Sinn haben – bis die Sprache stabil ist, wollen sie nur wenige Leute nutzen, aber Sie finden die Fehler in der Sprachdefinition erst, wenn Sie so viele Anwender haben, dass es zu spät ist, um etwas zu ändern.

Natürlich lässt sich in der *Implementierung* Vieles wie jedes normale Programm debuggen, aber das Sprachdesign selbst muss von vornherein sorgfältig entworfen sein, da seine Fehlerkosten so exorbitant hoch sind.

Wie entscheiden Sie, ob ein Feature als Erweiterung in eine Bibliothek wandern soll oder durch die eigentliche Sprache zu unterstützen ist?

Guido: Historisch gesehen hatte ich eine ziemlich gute Antwort darauf. Mir fiel sehr schnell auf, dass jeder sein Lieblingsfeature in der Sprache wiederfinden wollte und die meisten in Bezug auf Sprachdesign recht unerfahren sind. Jeder schlägt immer vor: »Lass uns dies zur Sprache hinzufügen!« oder »Lass uns eine Anweisung bauen, die X tut.« In vielen Fällen ist die Antwort: »Tja, du kannst X oder etwas Ähnliches schon umsetzen, indem du diese zwei oder drei Zeilen Code schreibst. Das ist gar nicht so schwer.« Sie können ein Dictionary verwenden oder eine Liste, ein Tupel und einen regulären Ausdruck verbinden oder eine kleine Metaklasse schreiben – all diese Sachen. Es kann sogar sein, dass ich die ursprüngliche Version dieser Antwort von Linus habe, der eine ähnliche Philosophie zu vertreten scheint.

Den Leuten zu erklären, dass sie dies oder das schon längst machen können, ist die erste Verteidigungslinie. Die zweite ist: »Nun, das ist schon ganz praktisch, und wir oder du sind sicherlich dazu in der Lage, ein eigenes Modul oder eine eigene Klasse dafür zu schreiben und diese Abstraktionseinheit damit zu kapseln.« Dann kommt: »Okay, das sieht so interessant und nützlich aus, dass wir es tatsächlich als Ergänzung zur Standardbibliothek übernehmen werden und es echtes Python sein wird.« Und dann gibt es schließlich Dinge, die sich im reinen Python gar nicht so einfach umsetzen lassen, und wir schlagen vor, sie in eine C-Extension umzuwandeln. Die C-Extensions sind die letzte Verteidigungslinie, bevor wir zugeben müssen: »Ja, okay, das ist so nützlich und man kann es wirklich nicht umsetzen, also müssen wir die Sprache ändern.«

Dies ist ein Auszug aus dem Buch "Visionen der Programmierung: Die Sprachen und ihre Schöpfer", ISBN 978-3-89721-934-2
<http://www.oreilly.de/catalog/hallowegel/>
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2009

Es gibt noch andere Kriterien, die entscheiden, ob es sinnvoller ist, etwas der Sprache oder eher der Bibliothek hinzuzufügen, denn wenn es um die Semantiken von Namensräumen oder Ähnliches geht, kann man eigentlich nicht viel anderes machen, als die Sprache anzupassen. Andererseits ist der Extension-Mechanismus mächtig genug, dass es eine erstaunliche Menge Dinge in C-Code gibt, die die Bibliothek erweitern und eventuell sogar neue, eingebaute Funktionalität bereitstellen, ohne die Sprache selbst zu verändern. Der Parser ändert sich nicht. Der Parse Tree ändert sich nicht. Die Dokumentation für die Sprache ändert sich nicht. Alle Ihre Tools funktionieren immer noch, und trotzdem hat Ihr System neue Funktionalität erhalten.

Ich vermute, es gibt Features, die Sie sich angeschaut haben und die sich nicht in Python implementieren lassen, ohne die Sprache zu verändern, und die Sie trotzdem abgelehnt haben. Welche Kriterien legen Sie an, um zu sagen, das hier ist pythonisch, das da aber nicht?

Guido: Das ist viel schwieriger. In vielen Fällen ist das eher eine Frage des Bauchgefühls. Die Leute nutzen die Wendungen »pythonisch« und »das ist pythonisch« häufig, aber niemand kann Ihnen eine wasserdichte Definition dessen geben, was pythonisch oder unpythonisch ist.

Sie haben das »Zen of Python«, was gibt es noch?

Guido: Da braucht man eine Menge Interpretation, wie bei jeder guten heiligen Schrift. Wenn ich ein gutes oder ein schlechtes Proposal sehe, kann ich sagen, ob es gut oder schlecht ist, aber es ist wirklich schwer, das ganze in eine Reihe von Regeln zu gießen, die jemand anderem dabei helfen, zwischen guten und schlechten Proposals zum Ändern der Sprache zu unterscheiden.

Das klingt, als ob es mehr eine Frage des Geschmacks ist als irgendetwas anderes.

Guido: Naja, zunächst versucht man, immer »Nein« zu sagen und zu schauen, ob die Leute weggehen und einen Weg finden, ihr Problem zu lösen, ohne die Sprache zu ändern. Es ist erstaunlich, wie oft das funktioniert. Dabei handelt es sich mehr um eine operative Definition von »Es ist nicht notwendig, die Sprache zu ändern.«

Wenn Sie die Sprache stabil halten, werden die Leute trotzdem einen Weg finden, das zu tun, was sie tun müssen. Darüber hinaus ist es häufig eine Frage der Use Cases, die aus verschiedenen Bereichen kommen und bei denen es nichts Spezifisches für die Anwendung gibt. Wenn zum Beispiel etwas für das Web richtig cool wäre, würde das kein gutes Feature für eine Sprachergänzung sein. Wenn etwas richtig hilfreich für das Schreiben kürzerer Funktionen oder besser wartbarer Klassen wäre, könnte es eventuell eine gute Ergänzung für die Sprache sein. Es muss wirklich über einzelne Anwendungsdomänen hinausgehen und die Angelegenheit einfacher oder eleganter machen.

Wenn Sie die Sprache verändern, ist jeder betroffen. Es gibt kein Feature, das Sie so gut verstecken können, dass die meisten Leute nichts darüber wissen müssen. Früher oder später werden die Leute über Code stolpern, der von jemand anderem geschrieben wurde und dieses Feature nutzt, oder sie haben es mit einem obskuren Grenzfall zu tun, bei dem sie sich mit dem Feature auseinandersetzen müssen, weil das Ganze nicht so funktioniert, wie sie es erwartet haben.

Häufig liegt die Eleganz auch im Auge des Betrachters. Wir hatten kürzlich eine Diskussion auf einer der Python-Listen, bei der die Leute überzeugend argumentierten, dass die Verwendung von `dollar` statt von `self.dot` viel eleganter wäre. Ich denke, ihre Definition von Eleganz war die Anzahl der einzugebenden Zeichen.

Es gibt eine Debatte darüber, ob man hier sehr sparsam vorgehen sollte, aber das liegt stark im Rahmen des persönlichen Geschmacks.

Guido: Eleganz, Einfachheit und Allgemeingültigkeit sind alles Dinge, die weitgehend vom persönlichen Geschmack abhängen, denn was für mich eine große Rolle spielt, mag für jemand anderen überhaupt nicht wichtig sein – und umgekehrt.

Wie läuft es mit dem Python Enhancement Proposal (PEP)?

Guido: Das ist ein sehr interessantes Thema. Ich denke, es wurde vor allem von Barry Warsaw angestoßen und beworben, einem der Basisentwickler. Er und ich begannen unsere Zusammenarbeit 1995, und ich glaube um das Jahr 2000 herum kam er mit dem Vorschlag, dass wir für die Sprachänderungen einen formelleren Prozess benötigen würden.

Ich bin bei solchen Sachen immer etwas langsam. Ich meine, ich war nicht derjenige, der herausgefunden hat, dass wir eine Mailingliste benötigen würden. Ich war nicht derjenige, der bemerkte, dass die Mailingliste unpraktisch wurde und wir eine Newsgroup brauchen würden. Ich war auch nicht derjenige, der vorschlug, dass wir eine Website haben müssten. Und ich war nicht derjenige, der vorschlug, dass wir einen Prozess benötigen würden, um Sprachänderungen zu besprechen und auszuarbeiten und um sicherzustellen, dass die üblichen Fehler vermieden werden, die entstehen, wenn schnell etwas vorgeschlagen und umgesetzt wird, ohne alle Konsequenzen zu durchdenken.

In der Zeit zwischen 1995 und 2000 waren Barry, ich selber und zwei andere Basisentwickler, Fred Drake und Ken Manheimer, alle an der CNRI, und etwas von dem, was die CNRI organisierte, waren die IETF-Meetings. Die CNRI hatte diese kleine Abteilung zur Organisation von Konferenzen, die sich schließlich selbstständig machte und deren einziger Kunde die IETF war. Sie organisierten später auch eine zeitlang die Python-Konferenzen. Daher war es ziemlich einfach, an den IETF-Meetings teilzunehmen, auch wenn sie nicht direkt dort abgehalten wurden. Ich bekam einen Eindruck vom IETF-Prozess mit seinen RFCs, den Meeting-Gruppen und den Phasen, und Barry bekam das auch. Als er vorschlug, etwas Ähnliches für Python umzusetzen, war ich leicht zu überzeugen. Wir entschieden uns absichtlich dazu, es bei Weitem nicht so aufwendig und kompliziert zu machen, wie es sich zu dem Zeitpunkt bei der IETF schon entwickelt hatte, da Internetstandards (zumindest manche von ihnen) weitaus mehr Branchen und Personen beeinflussen als eine Änderung an Python, aber wir nutzen die IETF durchaus als Vorbild. Barry ist beim Erfinden von Namen schon immer gut gewesen, daher bin ich ziemlich sicher, dass PEP seine Idee war.

Wir waren zu der Zeit eines der ersten Open Source-Projekte, die so etwas hatten, und es wurde recht häufig kopiert. Die Tcl/Tk-Community änderte im Prinzip nur den Namen, verwendete aber ansonsten genau die gleichen Dokumente und Prozesse, und andere Projekte sind ähnlich vorgegangen.

Finden Sie, dass sich durch das Hinzufügen von ein wenig Formalismus die Designentscheidungen rund um Python-Verbesserungen besser herauskristallisiert haben?

Guido: Ich denke, es wurde notwendig, als die Community wuchs und ich nicht mehr in der Lage war, jeden einzelnen Vorschlag zu bewerten. Es war sehr hilfreich für mich, dass andere Leute über die verschiedenen Details diskutierten und dann mit recht klaren Schlussfolgerungen ankamen.

Führen sie zu einem Konsens, bei dem jemand Sie bitten kann, sich bei bestimmten Erwartungen und Vorschlägen einzumischen?

Guido: Ja, es funktioniert häufig so, dass ich einem PEP zu Beginn die Zustimmung gebe, indem ich sage: »Sieht so aus, als hätten wir hier ein Problem. Vielleicht findet ja jemand die richtige Lösung dafür.« Oft kommen die Leute dann mit einer Reihe klarer Schlussfolgerungen, wie das Problem gelöst werden sollte, aber auch mit einer Reihe offener Fragen. Manchmal kann mir mein Bauchgefühl dabei helfen, diese offenen Fragen zu beantworten. Ich bin im PEP-Prozess sehr aktiv, wenn es um einen Bereich geht, den ich spannend finde – wenn wir eine neue Schleifenkontrollanweisung einfügen müssen, möchte ich nicht, dass sie von anderen Leuten entworfen wird. Manchmal halte ich mich aber auch fern davon, wenn es zum Beispiel um Datenbank-APIs geht.

Wodurch wird eine neue große Version notwendig?

Guido: Das hängt davon ab, was Sie als »groß« definieren. In Python sehen wir im Allgemeinen Releases wie 2.4, 2.5 und 2.6 als »groß« an, was nur alle 18–24 Monate vorkommt. Das sind die einzigen Gelegenheiten, zu denen wir neue Features einführen können. Viel früher wurden Releases aufgrund des Gejammers von Entwicklern (insbesondere von mir) veröffentlicht. Zu Beginn dieses Jahrzehnts haben die Anwender allerdings ein wenig Voraussagbarkeit eingefordert – sie wandten sich gegen Features, die in »kleineren« Revisionen hinzugefügt oder geändert wurden (zum Beispiel erhielt 1.5.2 zusätzliche »große« Features im Vergleich zu 1.5.1), und sie wünschten sich, dass die großen Releases eine gewisse Mindestzeit lang unterstützt werden (18 Monate). Jetzt haben wir also mehr oder weniger zeitabhängige große Releases: Wir planen die Zeitpunkte bis zu einem großen Release (zum Beispiel, wann Alpha- und Betaversionen sowie Release-Kandidaten abgeschlossen werden) lange im Voraus, basierend auf verschiedenen Dingen, wie zum Beispiel der Verfügbarkeit des Release-Managers, und wir nötigen die Entwickler, ihre Änderungen rechtzeitig vor dem abschließenden Release-Datum einzubringen.

Features, die in einem Release neu dazukommen, werden im Allgemeinen von den Basisentwicklern abgestimmt, nach (manchmal sehr langen) Diskussionen über die Vorteile des Features und seine exakte Spezifikation. Das ist der PEP-Prozess: Python Enhancement Proposal, ein Dokumenten-basierter Prozess, der dem RFC-Prozess der IETF oder dem JSR-Prozess in der Java-Welt nicht unähnlich ist, nur dass wir nicht so formell sind, da wir eine viel kleinere Entwicklercommunity haben. Falls es langwierigere Meinungsverschiedenheiten gibt (sei es in Bezug auf die Vorteile eines Features oder ein bestimmtes Detail), treffe ich eventuell eine abschließende Entscheidung – mein entsprechender Algorithmus ist da vor allem intuitiv, da zum Zeitpunkt meines Eingreifens jegliche rationalen Argumente sowieso schon ausgetauscht sind.

Die kontroversesten Diskussionen drehen sich im Allgemeinen um Sprachfeatures, die für die Anwender sichtbar sind – Ergänzungen bei den Bibliotheken sind meist einfach besprochen (da sie Anwender, die es nicht interessiert, auch nicht betreffen) und interne Verbesserungen werden eigentlich nicht als Features betrachtet, auch wenn sie durch die Rückwärtskompatibilität auf C-API-Ebene recht beschränkt sind.

Da die Entwickler im Allgemeinen die lautesten Anwender sind, kann ich nicht genau sagen, ob Features von Anwendern oder Entwicklern vorgeschlagen werden – meist schlagen Entwickler Features vor, die auf den Anforderungen basieren, die sie von den ihnen bekannten Anwendern

erhalten. Wenn ein Anwender ein neues Feature vorschlägt, wird es selten umgesetzt, da es ohne ein umfassendes Verständnis der Implementierung (und von Sprachdesign und Implementierung im Allgemeinen) nahezu unmöglich ist, ein neues Feature ordentlich vorzuschlagen. Wir bitten die Anwender gern, uns ihre Probleme ohne eine bestimmte Lösung im Hinterkopf vorzutragen, damit die Entwickler dann Lösungen vorschlagen und die Vorteile der verschiedenen Alternativen mit den Anwendern besprechen können.

Dann gibt es noch das Konzept einer richtig großen oder Breakthrough-Version, wie zum Beispiel 3.0. Historisch gesehen war 1.0 sehr nah an 0.9, und 2.0 war nur einen recht kleinen Schritt von 1.6 entfernt. Heutzutage, mit der deutlich größeren Anwenderbasis, sind solche Versionen allerdings recht selten und bieten die einzige Möglichkeit, wirklich inkompatibel zu früheren Versionen zu werden. Große Versionen werden durch einen bestimmten Mechanismus rückwärtskompatibel gemacht, indem veraltete Features zum Entfernen gekennzeichnet werden.

Wie haben Sie sich dazu entschieden, Zahlen als beliebig genaue Integer zu behandeln (mit all den coolen Vorteilen), statt den alten und üblichen Ansatz zu verfolgen, sie an die Hardware weiterzureichen?

Guido: Ich habe diese Idee ursprünglich von Pythons Vorgänger ABC übernommen. ABC verwendet beliebig genaue rationale Zahlen, aber ich mochte die rationalen Zahlen nicht so sehr, daher wechselte ich zu Integer-Werten. Bei realen Zahlen nutzt Python die Standardrepräsentation als Gleitkommazahlen, die von der Hardware unterstützt werden (genau wie ABC, nur mit ein wenig mehr Unterstützung).

Ursprünglich besaß Python zwei Arten von Integer-Werten: die normale 32-Bit-Variante (»int«) und die mit beliebiger Genauigkeit (»long«). Das machen viele Sprachen, aber die Variante mit der beliebigen Genauigkeit ist mit einer Bibliothek verbunden, so wie Bignum in Java und Perl oder GNU MP für C. In Python lebten diese beiden schon (nahezu) immer friedlich nebeneinander in der Kern-Sprache, und die Anwender mussten sich entscheiden, welche sie verwenden wollten, indem sie ein »L« an eine Zahl anhängten, um die long-Variante zu wählen. Das stellte sich nach und nach als zu nervig heraus – in Python 2.2 führten wir automatische Konvertierungen nach long ein, wenn das mathematisch korrekte Ergebnis einer Operation mit ints nicht als int dargestellt werden konnte (zum Beispiel `2**100`).

Früher hätte das eine `OverflowError`-Exception ausgelöst. Es gab eine Zeit, in der das Ergebnis still und leise abgeschnitten worden wäre, aber ich änderte das Verhalten und warf eine Exception, bevor irgendjemand anderes die Sprache nutzte. Anfang 1990 verschwendete ich einen Nachmittag damit, ein kleines Demoprogramm zu debuggen, das ich geschrieben hatte und das einen Algorithmus nutzte, der eine nicht so offensichtliche Anwendung sehr großer Integer-Werte enthielt. Solche Debugging-Sitzungen sind einschneidende Erfahrungen.

Wie auch immer, es gab immer noch bestimmte Fälle, in denen sich die beiden Zahlentypen etwas unterschiedlich verhielten. So führte zum Beispiel die Ausgabe eines `int` im hexadezimalen oder oktalen Format zu einem Ergebnis ohne Berücksichtigung des Vorzeichen (`-1` wäre also als `FFFFFFF` ausgegeben worden), während das Gleiche für das mathematisch identische long ein Ergebnis mit Vorzeichen geliefert hat (in diesem Fall `-1`). In Python 3.0 wagen wir den radikalen Schritt und unterstützen nur noch genau einen Integer-Typ – wir bezeichnen ihn als `int`, aber die Implementierung ist größtenteils die des alten long-Typs.

Warum bezeichnen Sie das als »radikalen Schritt«?

Guido: Vor allem, weil es ein großer Unterschied zur bisherigen Praxis in Python ist. Es gab dazu eine Reihe von Diskussionen, und die Leute haben verschiedene Alternativen vorgeschlagen, in denen zwei (oder mehr) interne Repräsentationen genutzt, aber vollständig oder größtenteils vor den Anwendern verborgen worden wären (aber nicht vor den Schreibern von C-Extensions). Das hätte eine etwas bessere Performance gebracht, aber letztendlich war es sehr viel Arbeit, und durch zwei interne Repräsentationen wäre der Aufwand gewachsen, es richtig zu machen, und für Schnittstellen vom C-Code wäre es noch haariger geworden. Wir hoffen nun, dass der Performanceverlust nur geringfügig ist und wir die Performance durch andere Techniken wie Caching verbessern können.

Wie gehen Sie mit der Philosophie »Es sollte einen – und möglichst nur einen – offensichtlichen Weg geben« um?

Guido: Das war zu Beginn wahrscheinlich eher unterbewusst. Als Tim Peters das »Zen of Python« schrieb (aus dem Sie zitieren), hat er eine Reihe von Regeln explizit geäußert, die ich schon angewandt hatte, ohne mir dessen bewusst zu sein. Damit ist klar, dass diese spezielle Regel (die häufig verletzt wird, auch mit meiner Zustimmung) direkt aus dem allgemeinen Wunsch nach Eleganz in der Mathematik und Informatik kommt. Die Autoren von ABC haben sie ebenfalls angewandt, um damit nur wenige orthogonale Typen oder Konzepte haben zu müssen. Die Idee der Orthogonalität kommt direkt aus der Mathematik, wo sie sich auf die *Definition* bezieht, nur einen Weg zu haben (oder zumindest nur einen »echten« Weg), um etwas auszudrücken. So lassen sich zum Beispiel die XYZ-Koordinaten jedes Punktes im 3-D-Raum eindeutig bestimmen, wenn Sie erst einmal einen Ursprung und drei Basisvektoren ausgewählt haben.

Ich mag auch den Gedanken, dass ich den meisten Anwendern einen Gefallen damit tue, dass sie nicht zwischen verschiedenen Alternativen wählen müssen. Sie können das im Gegensatz zu Java sehen, wo die Standardbibliothek viele verschiedene Versionen anbietet, eine listenähnliche Datenstruktur umzusetzen (ein verkettete Liste, eine Array-Liste oder andere), oder C, wo Sie sich entscheiden müssen, wie Sie Ihren eigenen Listendatentyp implementieren.

Wie ist Ihre Meinung zu statischer versus dynamischer Typisierung?

Guido: Ich wünschte, ich könnte etwas Einfaches sagen wie »Statische Typisierung ist böse, dynamische ist gut.«, aber es ist nicht immer so einfach. Es gibt verschiedene Ansätze für die dynamische Typisierung, von Lisp bis Python, und ebenso verschiedene für die statische Typisierung, von C++ bis Haskell. Sprachen wie C++ und Java bringen die statische Typisierung vermutlich in Misskredit, weil Sie dem Compiler immer und immer wieder das Gleiche erzählen müssen. Sprachen wie Haskell und ML dagegen nutzen die Typinferenz, die etwas anderes ist und ein paar der Vorteile der dynamischen Typisierung besitzen, zum Beispiel einen exakteren Ausdruck von Ideen im Code. Aber das funktionale Paradigma scheint schwierig für sich selbst nutzbar zu sein – Dinge wie I/O oder GUI-Interaktionen passen nicht so gut dazu und werden oft mithilfe einer Brücke zu einer klassischeren Sprache gelöst, zum Beispiel C.

In manchen Situationen wird die Geschwätzigkeit von Java als Vorteil gesehen, denn sie ermöglicht das Erstellen mächtiger Tools zum Durchsuchen von Code, die Fragen wie »Wo wird diese Variable geändert?« oder »Wer ruft diese Methode auf?« beantworten können. Bei dynamischen Spra-

chen ist die Antwort auf solche Fragen schwieriger zu finden, da es häufig schwer ist, den Typ eines Methodenarguments herauszufinden, ohne jeden Pfad durch die gesamte Codebasis zu verfolgen. Ich bin mir nicht sicher, wie funktionale Sprachen wie Haskell solche Tools unterstützen – es könnte gut sein, dass Sie für dynamische Sprachen ähnliche Techniken nutzen müssen, da das genau das ist, was die Typinferenz tut – meiner Meinung nach!

Bewegen wir uns in Richtung hybride Typisierung?

Guido: Ich gehe davon aus, dass über hybride Typisierung noch eine Menge zu sagen sein wird. Ich habe beobachtet, dass die meisten großen Systeme, die in einer statisch typisierten Sprache geschrieben wurden, tatsächlich eine signifikante Untermenge enthalten, die prinzipiell dynamisch typisiert ist. So hat man zum Beispiel bei GUI-Widgetsets und Datenbank-APIs für Java oft das Gefühl, als ob sie die statische Typisierung überall bekämpfen und die meisten Korrektheitsprüfungen zur Laufzeit ausführen.

Eine hybride Sprache mit funktionalen und dynamischen Aspekten könnte ziemlich interessant sein. Ich sollte hinzufügen, dass Python trotz seiner Unterstützung für manche funktionalen Tools, wie `map()` und `lambda`, *keine* funktionale Untermenge besitzt – es gibt keine Typinferenz und keine Möglichkeit zur Parallelisierung.

Warum haben Sie sich dafür entschieden, mehrere Paradigmen zu unterstützen?

Guido: Das habe ich gar nicht. Python unterstützt die prozedurale Programmierung – bis zu einem gewissen Grad – und OO. Die beiden sind gar nicht so verschieden, und Pythons prozeduraler Stil ist immer noch stark von Objekten beeinflusst (da die grundlegenden Datentypen alle Objekte sind). Python unterstützt ein bisschen funktionale Programmierung – aber es ähnelt keiner realen funktionalen Sprache und wird es auch nie tun. Funktionale Sprachen versuchen, so viel wie möglich zum Zeitpunkt der Kompilierung zu machen – der »funktionale« Aspekt bedeutet, dass der Compiler Dinge unter der Annahme optimieren kann, dass es keine Nebenwirkungen geben wird, sofern sie nicht explizit deklariert wurden. Python hat den einfachsten und dümmsten vorstellbaren Compiler, und die offiziellen Laufzeitsemantiken verhindern kluges Vorgehen im Compiler aktiv, zum Beispiel das Parallelisieren von Schleifen oder das Umwandeln von Rekursionen in Schleifen.

Python hat vermutlich den Ruf, funktionale Programmierung zu unterstützen, weil es `lambda`, `map`, `filter` und `reduce` enthält, aber in meinen Augen ist das nur syntaktischer Zuckerguss und kein grundlegender Block wie in funktionalen Sprachen. Eine eher grundlegende Eigenschaft von Python, die es mit Lisp teilt (auch keine funktionale Sprache!), ist, dass Funktionen richtige Objekte sind und wie alle anderen auch übergeben werden können. Das zusammen mit verschachtelten Gültigkeitsbereichen und einem allgemein Lisp-ähnlichen Vorgehen bei Funktionsstatus ermöglicht es, Konzepte einfach zu implementieren, die oberflächlich gesehen Konzepte aus funktionalen Sprachen nachbilden, wie zum Beispiel Currying, Map und Reduce. Die primitiven Operationen, die notwendig sind, diese Konzepte zu *implementieren*, sind in Python gebaut, während diese Konzepte in funktionalen Sprachen die primitiven Operationen *sind*. Sie können `reduce()` in Python in ein paar Zeilen schreiben, aber nicht in einer funktionalen Sprache.

Haben Sie sich den Typ Programmierer vorgestellt, der diese Sprache interessant finden könnte, als Sie sie entworfen haben?

Guido: Ja, aber ich hatte vermutlich nicht genug Vorstellungskraft. Ich dachte an professionelle Programmierer in einer Unix- oder Unix-ähnlichen Umgebung. Frühe Versionen des Python-Tutorials nutzten einen Slogan wie »Python füllt die Lücke zwischen C und der Shell-Programmierung.«, weil ich mich und die Leute um mich herum genau dort wiederfand. Es kam mir nie in den Sinn, dass Python eine gute Sprache für das Einbetten in Anwendungen sein würde, bis die Leute begannen, danach zu fragen.

Die Tatsache, das es nützlich war, die ersten Grundlagen der Programmierung in der Schule oder am College zu unterrichten oder sich selbst anzueignen, war bloß ein Zufall, ermöglicht durch die vielen ABC-Features, die ich übernommen habe – ABC war dafür gedacht, Programmierung für Nichtprogrammierer zu lehren.

Wie finden Sie die Balance zwischen den verschiedenen Anforderungen an eine Sprache, die für Einsteiger leicht zu lernen, aber auch mächtig genug sein soll, damit erfahrene Programmierer nützliche Dinge umsetzen können? Ist das eine falsche Unterteilung?

Guido: Balance ist das entscheidende Wort. Es gibt ein paar wohlbekanntes Fallen, die zu vermeiden sind, wie zum Beispiel Sachen, die dafür gedacht sind, Einsteigern zu helfen, die Experten aber nerven, und Sachen, die Experten brauchen, die aber Anfänger verwirren. Es gibt genug Platz dazwischen, um beide Seiten glücklich zu machen. Eine andere Strategie ist, Wege für Experten zu haben, um fortgeschrittenere Dinge umzusetzen, denen Anfänger nie begegnen – so zum Beispiel, dass die Sprache Metaklassen unterstützt, es aber keinen Grund dafür gibt, dass Einsteiger davon wissen müssten.

Der gute Programmierer

Wie erkennen Sie einen guten Programmierer?

Guido: Man braucht Zeit, um einen guten Programmierer zu erkennen. So ist es zum Beispiel sehr schwer, gute von schlechten Entwicklern in einem einstündigen Bewerbungsgespräch zu unterscheiden. Wenn Sie aber mit jemandem zusammenarbeiten, wird es normalerweise ziemlich klar, wer gut ist. Ich zögere, bestimmte Kriterien anzugeben – vermutlich zeigen sie Kreativität, lernen schnell und schreiben bald Code, der funktioniert und nicht mehr stark geändert werden muss, bevor er sich einchecken lässt. Beachten Sie auch, dass manche Leute gut in Algorithmen und Datenstrukturen sind, andere bei der Integration von vielen Komponenten oder im Protokoll-design, beim Testen, beim Erstellen von APIs oder Benutzerschnittstellen oder was für Aspekte der Programmierung noch gibt.

Welche Methode würden Sie anwenden, um Programmierer einzustellen?

Guido: Basierend auf meinen Erfahrungen mit Bewerbungsgesprächen denke ich nicht, dass ich auf dem klassischen Weg viel Erfolg haben würde – ich habe nur minimale Fähigkeiten in Bewerbungsgesprächen –, und zwar auf keiner der Seiten des Tisches! Ich würde vielleicht eine Art Ausbildungssystem verwenden, bei dem ich für einige Zeit mit den Leuten zusammenarbeiten und schließlich ein Gefühl für ihre Stärken und Schwächen bekommen könnte. So ähnlich, wie ein Open Source-Projekt abläuft.

Gibt es einen Charakterzug, auf den man besonders achten sollte, wenn man nach guten Python-Programmierern sucht?

Guido: Ich fürchte, Sie fragen das aus Sicht des typischen Managers, der einfach ein paar Python-Programmierer einstellen will. Ich glaube nicht, dass es eine einfache Antwort darauf gibt, und tatsächlich denke ich sogar, dass das vermutlich die falsche Frage ist. Sie wollen keine Python-Programmierer einstellen. Sie wollen kluge, kreative, motivierte Leute einstellen.

Wenn man sich Stellenangebote für Programmierer anschaut, enthalten so gut wie alle die Anforderung, in einem Team arbeiten zu können. Wie sehen Sie die Rolle des Teams in der Programmierung? Sehen Sie immer noch Platz für den brillanten Programmierer, der nicht mit anderen zusammenarbeiten kann?

Guido: Unter diesem Aspekt kann ich die Stellenausschreibungen schon nachvollziehen. Brillante Programmierer, die nicht im Team arbeiten können, sollten vermeiden, in einem normalen Programmiererjob zu arbeiten – es wird eine Katastrophe für alle Beteiligten, und der Code ein Alptraum für denjenigen, der ihn übernehmen muss. Heutzutage gibt es Möglichkeiten, zu lernen, wie man mit anderen Menschen zusammenarbeitet, und wenn Sie wirklich so brillant sind, sollte es kein Problem sein, sich Teamworkfähigkeiten anzueignen – es ist nicht mal so schwer, wie zu lernen, eine effiziente Fast-Fourier-Transformation zu implementieren, wenn Sie sich nur darauf einlassen.

Was für Vorteile sehen Sie – als Designer von Python –, wenn Sie mit Ihrer Sprache entwickeln, im Vergleich zu einem anderen erfahrenen Entwickler, der Python nutzt?

Guido: Ich weiß es nicht – mittlerweile wurden die Sprache und die VM von so vielen Leuten angefasst, dass ich manchmal selbst überrascht bin, wie bestimmte Dinge im Detail funktionieren! Wenn ich einen Vorteil gegenüber anderen Entwicklern habe, dann hat das vermutlich mehr damit zu tun, dass ich die Sprache länger eingesetzt habe, als dass ich sie geschrieben habe. Über die lange Zeit hatte ich die Gelegenheit, herauszubekommen, welche Operationen schneller und welche langsamer sind – so bin ich mir vermutlich mehr als andere Anwender dessen bewusst, dass lokale Variablen schneller als globale Variablen sind (obwohl es *andere* damit übertrieben haben, nicht ich!) oder dass Funktionen- und Methodenaufrufe teuer sind (noch mehr als in C oder Java) oder dass der schnellste Datentyp ein Tupel ist.

Wenn es um die Anwendung der Standardbibliothek oder anderer Sachen geht, habe ich häufig das Gefühl, andere haben einen Vorsprung. So schreibe ich zum Beispiel alle paar Jahre an einer Webanwendung, und die verfügbare Technologie hat sich jedes Mal geändert, daher läuft es darauf hinaus, dass ich bei jedem Versuch eine »erste« Webanwendung entwickle, die ein neues Framework oder ein neues Vorgehen nutzt. Und ich hatte immer noch nicht die Gelegenheit, in Python ernsthaft mit XML zu jonglieren.

Es scheint, dass eines der Features von Python seine Prägnanz ist. Wie beeinflusst das die Wartbarkeit des Codes?

Guido: Ich habe von Forschungen wie auch von anekdotischen Berichten gehört, die darauf hindeuten, dass die Fehlerrate pro Anzahl an Codezeile unabhängig von der verwendeten Programmiersprache ziemlich konstant ist. Daher würde eine Sprache wie Python, in der eine typische Anwendung deutlich kleiner ist als zum Beispiel die gleiche Menge an Funktionalität in C++ oder Java, die Anwendung deutlich leichter wartbar machen. Natürlich wird das vermutlich bedeuten,

dass ein einzelner Programmierer auch für mehr Funktionalität verantwortlich ist. Das ist ein anderes Thema, aber es gereicht Python auch zum Vorteil: Mehr Produktivität pro Programmierer bedeutet wahrscheinlich weniger Programmierer in einem Team, was wiederum weniger Kommunikations-Overhead bedeutet. Nach *Vom Mythos des Mann-Monats* (Frederick P. Brooks; mitp) steigt dieser mit der Teamgröße im Quadrat, wenn ich mich richtig erinnere.

Welche Verbindung sehen Sie zwischen der Einfachheit des von Python angebotenen Prototyping und dem Aufwand, der notwendig ist, um eine vollständige Anwendung zu bauen?

Guido: Ich habe Python nie als Prototyping-Sprache gesehen. Ich glaube nicht, dass es eine klare Trennung zwischen Prototyping- und »Produktions«-Sprachen geben sollte. Es gibt Situationen, in denen der beste Weg, einen Prototyp zu schreiben, ein kleiner C-Hack ist, den man danach wegwirft. In anderen Situationen kann ein Prototyp ganz ohne »Programmieren« erstellt werden – zum Beispiel mit einem Spreadsheet oder eine Reihe von find- und grep-Anweisungen.

Als erste Absicht hatte ich bei Python eine Sprache im Sinn, die genutzt werden konnte, wenn C übertrieben und Shell-Skripten zu sperrig wären. Dazu gehört eine Menge Prototyping, aber auch viel »Businesslogik« (wie es heutzutage genannt wird), die keine besonders hohen Anforderungen an die Computerressourcen stellt, aber recht viel Code enthält. Ich würde sagen, dass der meiste Python-Code nicht als Prototyp geschrieben wird, sondern einfach, um eine Aufgabe zu erledigen. In den meisten Fällen ist Python dafür wunderbar geeignet, und es gibt keinen Grund, viel zu ändern, um nachher die eigentliche Anwendung fertigzustellen.

Häufig kommt es vor, dass eine einfache Anwendung nach und nach mehr Funktionalität erhalten soll, sie schließlich viel komplexer geworden ist als zu Anfang, und es keinen definierten Moment gab, wo sie vom Prototyp- in den Final-Status gewechselt hat. So hat zum Beispiel die Code-Review-Anwendung Mondrian, die ich bei Google angefangen habe, vermutlich mittlerweile das Zehnfache an Codeumfang angenommen, seit ich sie das erste Mal bereitgestellt habe, und sie ist immer noch in Python geschrieben. Natürlich gibt es auch Beispiele, in denen Python schließlich durch eine schnellere Sprache ersetzt wurde – zum Beispiel waren die allerersten Crawler/Indexer bei Google in Python geschrieben – aber das sind Ausnahmen, nicht die Regel.

Wie beeinflusst die Unmittelbarkeit von Python den Designprozess?

Guido: Das ist meist meine Arbeitsweise, und zumindest für mich funktioniert das oft sehr gut! Ich schreibe viel Code, den ich wegwerfe, aber es ist immer noch viel weniger Code, als ich in einer anderen Sprache geschrieben hätte. Das Schreiben von Code (ohne ihn jemals auszuführen) hilft mir häufig sehr dabei, die Details des Problems zu verstehen. Indem ich darüber nachdenke, wie der Code umzubauen wäre, damit er das Problem optimal löst, komme ich der Problemlösung selbst näher. Natürlich sollte das keine Ausrede sein, auf ein Whiteboard zu verzichten, um ein Design, eine Architektur oder eine Benutzerschnittstelle zu skizzieren oder um andere Designtechniken zu ignorieren. Der Trick ist, das richtige Tool für die entsprechende Aufgabe zu nutzen. Manchmal sind es ein Stift und eine Serviette – ein anderes Mal dagegen ein Emacs-Fenster und ein Shell-Prompt.

Denken Sie, dass Bottom-up-Entwicklung besser in Python aufgehoben ist?

Guido: Ich sehe bottom-up versus top-down nicht so als religiöse Gegensätze wie vi versus Emacs. In jedem Softwareentwicklungsprozess gibt es Momente, in denen Sie bottom-up arbeiten, und andere, in denen Sie top-down vorgehen. Top-down bedeutet vermutlich, dass Sie mit etwas arbeiten, das sorgfältig begutachtet und entworfen werden muss, bevor Sie mit dem Schreiben von Code beginnen können, während beim Bottom-up-Vorgehen neue Abstraktionsebenen auf schon bestehenden Ebenen aufgebaut werden, wie zum Beispiel beim Erstellen neuer APIs. Ich sage damit nicht, dass Sie APIs schreiben sollten, ohne irgendein Design im Hinterkopf zu haben, aber häufig ergeben sich neue APIs logisch aus den verfügbaren APIs auf niedrigerer Ebene, und die Designarbeit geschieht erst, während Sie den eigentlichen Code schreiben.

Wann schätzen Python-Programmierer Ihrer Meinung nach die dynamische Natur der Sprache am meisten?

Guido: Die dynamischen Features der Sprache sind oft dann am nützlichsten, wenn Sie ein großes Problem oder einen Lösungsraum angehen und nicht wissen, wie Sie vorgehen sollen – Sie können eine Reihe von Experimenten durchführen, wobei Sie bei jedem Durchlauf von seinem Vorgänger lernen, ohne allzu viel Code zu haben, der Sie in einem bestimmten Bereich festhält. Hier ist es wirklich hilfreich, dass Sie in Python sehr kompakten Code schreiben können – 100 Zeilen Python-Code, um einmalig ein Experiment durchzuführen und dann von vorne zu beginnen, ist eben deutlich effizienter als ein Framework aus 1.000 Zeilen in Java schreiben zu müssen und danach herauszufinden, dass damit das falsche Problem gelöst wurde!

Was bietet Python dem Programmierer in Bezug auf Sicherheitsaspekte?

Guido: Das hängt von den Angriffen ab, um die Sie sich sorgen. Python hat eine automatische Speicherallokation, daher sind Python-Programme nicht für bestimmte Arten von Fehlern anfällig, die in C oder C++ recht verbreitet sind, wie zum Beispiel Pufferüberläufe oder die Verwendung von nicht mehr reserviertem Speicher, die die Grundlagen für viele Angriffe auf Microsoft-Software bilden. Natürlich ist die Python-Runtime selbst in C geschrieben, und es gab auch im Laufe der Jahre Schwachstellen, die gefunden wurden; und es gibt bewusste Ausnahmemöglichkeiten, wie zum Beispiel das Modul ctypes, mit dem man beliebigen C-Code aufrufen kann.

Ist die dynamische Natur da hilfreich oder eher hinderlich?

Guido: Ich glaube nicht, dass die dynamische Natur hilft oder stört. Jemand kann leicht eine dynamische Sprache entwerfen, die viele Schwachstellen hat, oder eine statische, die keine hat. Aber eine Runtime – oder *virtuelle Maschine*, wie der hippe Begriff heutzutage lautet – hilft hier, weil sie den Zugriff auf die eigentliche, zugrunde liegende Maschine begrenzt. Das ist zufällig auch einer der Gründe dafür, dass Python die erste Sprache ist, die von der Google App Engine unterstützt wird – dem Projekt, an dem ich momentan beteiligt bin.

Wie kann ein Python-Programmierer Codesicherheit prüfen und verbessern?

Guido: Ich denke, Python-Programmierer sollten sich nicht allzu viele Gedanken über Sicherheit machen, insbesondere nicht, ohne eine bestimmte Angriffsart im Hinterkopf zu haben. Das Wichtigste ist, dass man genau auf das achtet, worauf man auch bei allen anderen Sprachen achtet: Sei kritisch mit Daten, die von jemandem kommen, dem du nicht vertraust (bei einem Webserver ist

das jedes Byte des eintreffenden Request, selbst bei den Headern). Besonders sollte man noch auf reguläre Ausdrücke achten – es ist leicht, einen regulären Ausdruck zu schreiben, der exponentiell lange läuft. Daher sollten Webanwendungen, die Suchen nach vom Anwender eingegebenen regulären Ausdrücken starten, einen Mechanismus haben, um die Laufzeit zu begrenzen.

Gibt es irgendwelche grundlegenden Konzepte (allgemeine Regeln, Sichtweisen, Denkweisen, Prinzipien), die Sie vorschlagen würden, um bei der Entwicklung mit Python kompetent vorzugehen?

Guido: Ich würde sagen, Pragmatismus. Wenn Sie zu sehr an theoretischen Konzepten festhalten, wie dem Verbergen von Daten, der Zugriffskontrolle, Abstraktionen oder Spezifikationen, sind Sie kein echter Python-Programmierer, und Sie verschwenden letztendlich zu viel Zeit damit, gegen die Sprache zu kämpfen, statt sie zu nutzen (und Spaß mit ihr zu haben). Zudem steigt die Wahrscheinlichkeit, sie ineffizient zu verwenden. Python ist sehr gut, wenn Sie sofortige Bestätigung brauchen, so wie ich. Sie funktioniert ganz gut, wenn Sie Ansätze wie Extreme Programming oder andere agile Entwicklungsmethoden mögen, aber auch dort würde ich empfehlen, nur gemäßigt vorzugehen.

Was meinen Sie mit »gegen die Sprache kämpfen«?

Guido: Das bedeutet im Allgemeinen, dass man versucht, seine Gewohnheiten weiterzuführen, die in einer anderen Sprache gut funktioniert haben.

Viele der Vorschläge, irgendwie vom expliziten `self` loszukommen, stammen von Leuten, die gerade erst zu Python gewechselt sind und noch nicht mit der Sprache vertraut sind. Es wird eine fixe Idee von ihnen. Manchmal wird daraus ein Proposal, um die Sprache zu ändern, oder sie schlagen hochkomplizierte Metaklassen vor, die `self` irgendwie implizit machen. Normalerweise ist so etwas total ineffizient oder funktioniert nicht in einer Umgebung mit mehreren Threads oder in einem anderen Grenzfall; oder sie sind so besessen davon, diese vier Buchstaben nicht tippen zu wollen, dass sie die Konventionen von `self` nach `s` oder `S` ändern. Andere wandeln alles in eine Klasse und jeden Zugriff in eine Accessor-Methode um, wo das doch in Python nicht sehr weise ist – Sie haben nur umfangreicheren Code, der schlechter zu debuggen ist und viel langsamer läuft. Sie kennen den Satz »Sie können FORTRAN in jeder Sprache schreiben«? Nun, Sie können auch Java in jeder Sprache schreiben.

Sie haben so viel Zeit mit dem Versuch verbracht, den einen, offensichtlichen (zu bevorzugenden) Weg zu erstellen, um etwas zu erledigen. Es scheint, als ob Sie der Meinung seien, nur wenn man etwas auf diese Weise erledige – auf dem pythonischen Weg – würde man wirkliche Vorteile aus Python ziehen.

Guido: Ich bin mir nicht so sicher, dass ich wirklich viel Zeit damit verbracht habe, sicherzustellen, dass es nur einen Weg gibt. Das »Zen of Python« ist viel jünger als die Sprache Python, und die meisten definierenden Eigenschaften der Sprache gab es schon lange, bevor Tim Peters sie in Form eines Gedichts aufgeschrieben hat. Ich denke nicht, dass er davon ausging, diese Sätze würden sich so weit verbreiten und so erfolgreich sein, als er sie formulierte.

Es ist eine griffige Phrase.

Guido: Tim kann gut mit Worten umgehen. »Es gibt nur einen Weg, es zu tun«, ist tatsächlich in den meisten Fällen eine glatte Lüge. Es gibt viele Möglichkeiten, Datenstrukturen umzusetzen. Sie können Tupel und Listen verwenden. In vielen Fällen ist es sogar völlig unwichtig, ob Sie ein Tupel oder eine Liste oder manchmal ein Dictionary nutzen. Es stellt sich meist heraus, dass bei sorgfältiger Betrachtung eine Lösung objektiv besser ist, weil sie in vielen Situationen gut funktioniert, und es nur ein oder zwei Fälle gibt, in denen Listen deutlich besser als Tupel sind, wenn sie größer werden.

Das kommt eigentlich mehr aus der ursprünglichen ABC-Philosophie, die versucht hat, mit den Komponenten sehr sparsam umzugehen. ABC teilte seine Philosophie eigentlich mit ALGOL-68, das zwar mittlerweile so was von tot ist, aber sehr viel Einfluss hatte. Auf jeden Fall hatte es in den 80er-Jahren dort, wo ich damals gearbeitet habe, sehr viel Einfluss, weil Adriaan van Wijngaarden der große ALGOL 68-Guru war. Er unterrichtete immer noch, als ich ans College kam. Ich hatte ihn ein oder zwei Semester, in denen er, wenn er in der Stimmung dazu war, nur Anekdoten aus der Geschichte von ALGOL 68 zum Besten gab. Er war der Direktor des CWI gewesen, allerdings hatte jemand anderes dann diesen Posten, als ich dazustieß.

Es gab viele Leute, die sehr intensiv mit ALGOL 68 gearbeitet haben. Ich denke, Lambert Meertens, der ursprüngliche Autor von ABC, war auch einer der ersten Bearbeiter des ALGOL 68-Reports. Er hat also vermutlich einen Großteil des Satzes erschaffen, aber auch ab und zu viel nachgedacht und überprüft. Er war eindeutig von der Philosophie von ALGOL 68 beeinflusst, Konstrukte bereitzustellen, die auf vielen verschiedenen Wegen kombiniert werden können, um alle möglichen verschiedenen Datenstrukturen zu produzieren oder Wege zu finden, ein Programm zu strukturieren.

Es war definitiv sein Einfluss, wenn es hieß: »Wir haben Listen oder Arrays und sie können beliebige andere Dinge enthalten. Es kann sich um Zahlen oder Strings handeln, aber auch um andere Arrays und Tupel mit anderen Dingen. Sie können all das kombinieren.« Plötzlich brauchen Sie kein eigenes Konzept multidimensionaler Arrays, weil ein Array aus Arrays das schon für jede Dimensionalität löst. Diese Philosophie, ein paar Schlüsselemente zu haben, die verschiedene Richtungen durch Flexibilität abdecken und miteinander kombiniert werden können, war ein wichtiger Teil von ABC. Ich habe mir all das ausgeliehen, ohne groß darüber nachzudenken.

Während Python versucht, so zu erscheinen, als ob Sie alles sehr flexibel kombinieren können, solange Sie nicht versuchen, Anweisungen innerhalb von Ausdrücken einzubetten, gibt es tatsächlich eine gewisse Anzahl von Spezialfällen in der Syntax, bei denen in manchen Fällen ein Komma eine Trennung zwischen Parametern bedeutet, in anderen aber die Elemente einer Liste unterteilt und in wieder anderen Fällen für ein implizites Tupel steht.

Es gibt eine ganze Reihe von Variationen in der Syntax, bei denen bestimmte Operatoren nicht erlaubt sind, weil sie mit der umgebenden Syntax in Konflikt geraten würden. Das ist nie ein echtes Problem, weil Sie immer ein Paar zusätzliche Klammern um etwas einfügen können, wenn es nicht funktioniert. Deshalb ist der Umfang der Syntax, zumindest aus Sicht des Parser-Autoren, ein bisschen gewachsen. Dinge wie List Comprehensions und Generator-Ausdrücke sind syntaktisch immer noch nicht vollständig vereinheitlicht. Ich gehe davon aus, dass das in Python 3000 abgeschlossen sein wird. Es gibt immer noch ein paar subtile Semantikunterschiede, aber zumindest die Syntax wird dieselbe sein.

Viele Pythons

Wird der Parser in Python 3000 einfacher werden?

Guido: Eher nicht. Er wird nicht komplexer, aber eigentlich auch nicht einfacher.

Keine zusätzliche Komplexität ist meiner Meinung nach ein Gewinn.

Guido: Ja.

Warum der einfachste, dümmste vorstellbare Compiler?

Guido: Das war ursprünglich ein sehr zweckmäßiges Ziel, da ich beim Generieren von Code nicht so bewandert bin. Es gab nur mich, und ich musste den Bytecode-Generator fertigstellen, bevor ich irgendetwas Interessantes an der Sprache machen konnte.

Ich glaube immer noch, dass es gut ist, einen sehr einfachen Parser zu haben – schließlich dient er nur dazu, den Text in einen Baum umzuwandeln, der die Struktur des Programms repräsentiert. Wenn die Syntax so mehrdeutig ist, dass man größeren Aufwand betreiben muss, um das Gewünschte zu ermitteln, werden die menschlichen Leser vermutlich die Hälfte der Zeit ebenso verwirrt sein. Zudem ist es ziemlich schwierig, so einen anderen Parser zu schreiben.

Python ist unglaublich einfach zu parsen, zumindest auf der syntaktischen Ebene. Auf lexikalischer Ebene ist die Analyse recht subtil, da Sie die Einrückung mit einem kleinen Stack lesen müssen, der im lexikalischen Analyzer eingebettet ist – was ein Gegenbeispiel für die Theorie der Trennung zwischen lexikalischer und grammatikalischer Analyse ist. Trotzdem ist es die richtige Lösung. Lustig ist, dass ich automatisch generierte Parser liebe, aber nicht so sehr an automatisch generierte lexikalische Analyse glaube. Python hatte schon immer einen manuell erzeugten Scanner und einen automatisch generierten Parser.

Man hat viele verschiedene Parser für Python geschrieben. Jede Portierung von Python auf einer anderen virtuellen Maschine, sei es Jython oder IronPython oder PyPy, hat ihren eigenen Parser, und es ist auch nicht so schwierig, weil der Parser niemals ein wirklich komplexes Element im Projekt ist. Denn die Struktur der Sprache sieht so aus, dass Sie sie ganz einfach mit dem simpelsten one-token-lookahead rekursiv absteigenden Parser parsen können.

Langsam wird ein Parser durch tatsächliche Mehrdeutigkeiten, die nur aufgelöst werden können, indem man vorausschaut bis zum Ende des Programms. In natürlichen Sprachen gibt es viele Beispiele, bei denen es unmöglich ist, einen Satz zu parsen, bevor Sie nicht das letzte Wort und die Verschachtelungen im Satz gelesen haben. Oder es gibt Sätze, die nur geparkt werden können, wenn Sie die Person kennen, über die gesprochen wird, aber das ist eine ganz andere Situation. Beim Parsen von Programmiersprachen mag ich meinen One-Token-Lookahead.

Damit sieht es für mich so aus, als ob es niemals Makros in Python geben wird, weil Sie dann eine weitere Parsingphase durchlaufen müssen!

Guido: Es gibt Wege, die Makros in den Parser einzubetten, die eventuell funktionieren könnten. Ich bin allerdings auch nicht so überzeugt, dass Makros ein Problem lösen können, das in Python besonders drängend ist. Da die Sprache andererseits leicht zu parsen ist, könnte es sehr einfach

sein, eine Mikroevaluation als Parse-Tree-Manipulation zu implementieren, wenn Sie eine saubere Menge von Makros haben, die gut in die Sprachsyntax passen. Das ist nur kein Bereich, an dem ich sonderlich interessiert bin.

Warum haben Sie sich dazu entschieden, eine strenge Formatierung im Quellcode zu nutzen?

Guido: Die Wahl der Einrückung zum Gruppieren ist kein neues Konzept von Python. Ich habe es von ABC übernommen, aber es kommt auch in occam vor, einer älteren Sprache. Ich weiß nicht, ob die ABC-Autoren die Idee von occam übernommen oder sie unabhängig davon entwickelt haben, oder ob es einen gemeinsamen Vorgänger gab. Die Idee mag auf Don Knuth zurückzuführen sein, der das schon 1974 vorschlug.

Natürlich hätte ich mich auch dazu entschließen können, ABC hier nicht zu folgen, wie ich es in anderen Bereichen tat (zum Beispiel verwendet ABC Großbuchstaben für die Schlüsselwörter der Sprache und für Prozedurnamen – eine Idee, die ich nicht übernahm), aber ich mochte das Feature ganz gerne, und es schien auch die nutzlosen Debatten zu umgehen, die es damals unter C-Anwendern gab, wo denn bitteschön die geschweiften Klammern zu stehen hätten. Zudem war mir bewusst, dass gut lesbarer Code sowieso freiwillig Einrückungen verwendet, um eine Gruppierung deutlich zu machen, und ich sah mich auch subtilen Bugs im Code gegenüber, bei denen die Einrückung nicht mit der syntaktischen Gruppierung durch geschweifte Klammern übereinstimmte – die Programmierer und alle Reviewer waren davon ausgegangen, dass die Einrückung stimmt, und übersahen den Fehler. Auch hier brachte eine lange Debugging-Sitzung einen echten Erkenntnisgewinn.

Eine strikte Formatierung sollte für einen saubereren Code sorgen und vermutlich die Unterschiede im »Layout« verschiedener Programmierer verringern, aber klingt das nicht ein bisschen danach, einen Menschen dazu zu zwingen, sich an die Maschine anzupassen statt umgekehrt?

Guido: Genau das Gegenteil: Es hilft dem menschlichen Leser dabei mehr als der Maschine, wie das vorige Beispiel gezeigt hat. Die Vorteile dieses Ansatzes sind vermutlich besser zu erkennen, wenn man Code warten muss, der von einem anderen Programmierer geschrieben wurde.

Neue Anwender werden davon zunächst oft abgeschreckt, aber ich höre davon mittlerweile nicht mehr so viel. Vielleicht haben die Leute, die Python unterrichten, gelernt, mit diesem Effekt zu rechnen und ihm direkt entgegenzutreten.

Ich möchte Sie zu den verschiedenen Implementierungen von Python befragen. Es gibt vier oder fünf große Implementierungen, einschließlich Stackless und PyPy.

Guido: Stackless ist technisch gesehen keine eigene Implementierung. Es wird häufig als eigene Python-Implementierung aufgeführt, weil es ein Zweig von Python ist, der einen ziemlich kleinen Teil der virtuellen Maschine durch einen anderen Ansatz ersetzt.

Im Prinzip den Bytecode-Dispatch, oder?

Guido: Ein Großteil des Bytecode-Dispatch ist sehr ähnlich. Ich denke, die Bytecodes sind gleich, und mit Sicherheit sind alle Objekte gleich. Der Unterschied findet sich beim Aufruf einer Python-Prozedur von einer anderen Prozedur: Das geschieht dort durch die Manipulation von Objekten, wobei nur ein Stack mit Stackframes abgelegt wird und der gleiche C-Code weiterarbeitet. In C-

Python wird hier eine C-Funktion aufgerufen, die schließlich eine neue Instanz der virtuellen Maschine nutzt. Es ist eigentlich nicht die ganze virtuelle Maschine, sondern die Schleife, die den Bytecode interpretiert. Es gibt in Stackless nur eine solche Schleife auf dem C-Stack. Im klassischen C-Python können Sie die gleiche Schleife mehrmals auf Ihrem C-Stack haben. Das ist der einzige Unterschied.

PyPy, IronPython, Jython sind eigene Implementierungen. Ich weiß nichts über eine Übersetzung in JavaScript, aber ich wäre nicht überrascht, wenn jemand damit schon ziemlich weit gekommen wäre. Ich habe von experimentellen Projekten gehört, die nach OCaml, Lisp und wer weiß was noch übersetzen. Es gab einmal ein Projekt, bei dem Python in C-Code umgewandelt wurde. Mark Hammond und Greg Stein arbeiteten daran in den späten 90er Jahren, aber sie stellten fest, dass der Geschwindigkeitsgewinn sehr, sehr gering war. In den besten Fällen lief der Code doppelt so schnell, zudem war der generierte Code so umfangreich, dass Sie riesige Binaries hatten, was dann zu einem Problem wurde.

Die Anlaufzeit schmerzt.

Guido: Ich denke, die PyPy-Leute sind auf dem richtigen Weg.

Es klingt, als ob Sie diese Implementierungen im Allgemeinen unterstützen.

Guido: Ich habe alternative Implementierungen immer gut gefunden. Ab dem Tag, als Jim Hugunin mit einer mehr oder weniger vollständigen JPython-Implementierung zur Tür hereinkam, war ich davon begeistert. Denn ich sehe das als Validierung des Sprachdesigns an. Zudem bedeutet es, dass Leute ihre bevorzugte Sprache auf der Plattform nutzen können, auf die sie ansonsten keinen Zugriff hätten. Wir haben hier immer noch einiges zu tun, aber es hat mir auf jeden Fall dabei geholfen, herauszufinden, welche Features wirklich Features der Sprache waren, um die ich mich kümmerte, und welche Features einer bestimmten Implementierung waren, bei denen ich es in Ordnung fand, wenn andere Implementierungen hier anders vorgehen. So begaben wir uns schließlich leider auf das Glatteis der Garbage Collection.

Das ist doch immer ein heikles Thema.

Guido: Aber es ist auch notwendig. Ich kann nicht glauben, wie lange wir mit dem reinen Zählen von Referenzen leben konnten und keine Möglichkeit hatten, Ringschlüsse aufzubrechen. Ich habe das Zählen von Referenzen immer als eine Möglichkeit angesehen, eine Garbage Collection durchzuführen, und zwar nicht die schlechteste. Es gab diesen heiligen Krieg zwischen dem Referenzzählen und der Garbage Collection, der mir immer so überflüssig erschien.

Nochmal zurück zu diesen Implementierungen: Ich denke, Python ist ein interessantes Gebiet, weil es eine ziemlich gute Spezifikation hat, besonders im Vergleich zu anderen Sprachen wie Tcl, Ruby und Perl 5. Entstand das, weil Sie die Sprache standardisieren wollten, weil Sie auf mehrere Implementierungen hofften oder warum?

Guido: Es war vermutlich eher eine Nebenwirkung des Community-Prozesses um die PEPs und die vielen Implementierungen herum. Als ich ursprünglich die erste Dokumentation schrieb, begann ich sehr enthusiastisch mit einer Referenzanleitung für die Sprache, die so präzise Spezifikationen enthalten sollte, dass jemand vom Mars oder Jupiter die Sprache implementieren und die Semantiken korrekt erhalten könnte. Ich bin diesem Ziel nicht einmal ansatzweise nahegekommen.

ALGOL 68 ist vermutlich die Sprache, die mit ihrer stark mathematischen Spezifikation noch am nächsten herankommt. Andere Sprachen wie C++ und JavaScript haben es durch die pure Willensanstrengung des Standardisierungskomitees geschafft, insbesondere im Fall von C++. Das ist offensichtlich ein erstaunlich beeindruckender Aufwand. Gleichzeitig kostet es so viel Arbeitskraft, eine so präzise Spezifikation zu schreiben, dass ich nie ernsthaft die Hoffnung hatte, so etwas für Python zu bekommen.

Wir haben aber genug Verständnis davon, wie die Sprache arbeiten soll, und genug Unit Tests und genug Leute, die bei der Implementierung anderer Versionen in endlicher Zeit Antworten geben können. Ich weiß zum Beispiel, dass die Jungs von IronPython sehr gewissenhaft die gesamte Python-Testsuite ausgeführt und bei jedem Fehler diskutiert haben, ob die Testsuite eigentlich das bestimmte Verhalten der C-Python-Implementierung testete oder ob sie tatsächlich noch an ihrer Implementierung etwas korrigieren mussten.

Die PyPy-Leute machten das Gleiche und gingen noch einen Schritt weiter. Unter ihnen gibt es mehrere Leute, die deutlich klüger sind als ich und die mit Grenzfällen ankamen, die vermutlich ihren eigenen Überlegungen über das Generieren von Code und das Analysieren in einer JIT-Umgebung entstammten. Sie haben tatsächlich eine Reihe von Tests, Mehrdeutigkeiten und Fragen eingebracht, wenn Sie bemerkten, dass es eine bestimmte Kombination von Dingen gab, über die noch niemand nachgedacht hatte. Das war sehr hilfreich. Durch die vielen Implementierungen der Sprache wurden viele Mehrdeutigkeiten in der Spezifikation der Sprache ausgemerzt.

Sehen Sie einen Zeitpunkt, an dem C-Python nicht mehr die wichtigste Implementierung sein könnte?

Guido: Das ist schwer vorauszusagen. Ich meine, manche Leute sagen einen Zeitpunkt voraus, an dem .NET die Welt beherrschen wird, andere sehen das bei den JVM. Für mich scheint das alles Wunschdenken zu sein. Andererseits weiß ich nicht, was geschehen wird. Es könnte einen Quantensprung geben, bei dem ein bestimmter Typ von Plattform plötzlich deutlich vorherrschender wird, obwohl sich die Computer gar nicht so sehr geändert haben. Dann sind auch die Regeln anders.

Vielleicht eine Verschiebung weg von der von-Neumann-Architektur?

Guido: Daran habe ich gar nicht gedacht, aber das ist sicherlich eine Möglichkeit. Ich dachte eher daran, was passieren würde, wenn Mobiltelefone der universelle Computer werden würden. Mobiltelefone liegen gegenüber den normalen Laptops nur ein paar Jahre zurück, was heißt, dass sie in ein paar Jahren abgesehen von der mickrigen Tastatur und dem Bildschirm genug Rechenleistung haben werden, dass Sie auf einen Laptop verzichten können. Es kann gut sein, dass Mobiltelefone egal welcher Plattform alle eine JVM oder irgendeine andere Standardumgebung haben werden, auf der C-Python nicht der beste Ansatz ist und eine andere Python-Implementierung besser funktioniert.

Es gibt sicherlich auch die Frage, was wir tun, wenn wir 64 Kerne auf einem Chip haben, selbst in einem Laptop oder einem Mobiltelefon. Ich weiß tatsächlich nicht, ob sich dadurch das Programmierparadigma für die meisten Dinge ändern sollte, die wir so tun. Es mag eine Anwendungsmöglichkeit für eine Sprache geben, mit der Sie unglaublich subtil konkurrierende Prozesse definieren

können, aber in den meisten Fällen kann der durchschnittliche Programmierer sowieso keinen korrekten, Thread-sicheren Code schreiben. Davon auszugehen, dass die vielen Kerne ihn dazu zwingen, ist ziemlich unrealistisch. Ich gehe davon aus, dass mehrere Kerne sicherlich hilfreich sein können, aber nur für eine grob unterteilte Parallelität genutzt werden. Das ist sowieso besser, denn durch die enormen Kostenunterschiede zwischen Cache Hits und Cache Misses dient der Hauptspeicher nicht mehr als Shared Memory. Sie werden Ihre Prozesse so isoliert wie möglich haben wollen.

Wie sollten wir mit Nebenläufigkeit umgehen? Auf welcher Ebene sollte mit diesem Problem umgegangen oder besser noch, es gelöst werden?

Guido: Ich habe das Gefühl, dass das Schreiben von Code für einen einzelnen Thread schon schwer genug ist, und das Schreiben für mehrere Threads noch viel schwerer – so schwer, dass die meisten Leute gar nicht die Hoffnung haben, es richtig zu machen, einschließlich meiner selbst. Daher glaube ich nicht, dass feingranulare Synchronisations-Primitive und Shared Memory die Lösung sind – stattdessen sehe ich viel mehr, dass Lösungen wieder attraktiv werden, bei denen Nachrichten herumgeschickt werden. Ich bin mir ziemlich sicher, dass das Anpassen aller Programmiersprachen, um Synchronisationskonstrukte hinzuzufügen, eine schlechte Idee ist.

Ich glaube auch nicht, dass der Versuch, den GIL aus CPython zu entfernen, gelingen wird. Ich denke, dass eine gewisse Unterstützung für die Verwaltung mehrerer Prozesse (im Gegensatz zu Threads) ein Teil des Puzzles sein wird. Daher werden Python 2.6 und 3.0 ein neues Standardbibliotheksmodul `multiprocessing` besitzen, das eine dem `Threadening`-Modul ähnliche API anbieten wird, die genau das tut. Als Bonus wird sogar das Ausführen von Prozessen auf unterschiedlichen Hosts unterstützt werden!

Hilfen und Erfahrungen

Gibt es ein Tool oder Feature, das Sie vermissen, wenn Sie Software schreiben?

Guido: Wenn ich auf einem Computer genauso einfach Skizzen machen könnte wie mit Stift und Papier, würde ich vielleicht mehr Skizzen machen, während ich mich mit dem schwierigen Nachdenken über ein Design beschäftige. Ich fürchte, dass ich warten muss, bis die Maus allgemein durch einen Stift (oder einen Finger) ersetzt wird, mit dem auf dem Bildschirm gezeichnet werden kann. Persönlich fühle ich mich furchtbar eingeschränkt, wenn ich irgendein Zeichenprogramm auf dem Computer nutzen muss, obwohl ich eigentlich mit Stift und Papier ganz gut bin – vielleicht habe ich das von meinem Vater geerbt, der Architekt war und andauernd Skizzen machte, daher habe ich als Teenager auch immer gezeichnet.

Am anderen Ende der Skala vermute ich, dass ich nicht einmal weiß, was ich vermisste, um in einer großen Codebasis zu stöbern. Java-Programmierer haben heutzutage IDEs, die schnelle Antworten auf Fragen wie »Wer ruft diese Methode auf?« oder »Wo ist diese Variable zugewiesen?« geben können. Bei großen Python-Programmen wäre das auch hilfreich, aber die notwendigen statischen Analysen sind aufgrund der dynamischen Natur von Python viel schwieriger durchzuführen.

Wie testen und debuggen Sie Ihren Code?

Guido: Wie immer es gerade nützlich ist. Ich teste viel, wenn ich Code schreibe, aber die Testmethode variiert von Projekt zu Projekt. Wenn Sie einen grundlegenden, rein algorithmischen Code schreiben, sind Unit Tests im Allgemeinen großartig, aber bei Code, der stark interaktiv ist oder als Schnittstelle zu alten APIs dient, läuft es bei mir doch häufig auf manuelles Testen heraus, unterstützt von der Befehlszeilen-History in der Shell oder einem Neuladen einer Seite im Browser. Als (extremes) Beispiel können Sie schlecht einen Unit Test für ein Skript schreiben, dessen einziger Zweck darin liegt, den aktuellen Rechner herunterzufahren. Sicherlich können Sie den Teil simulieren, der das tatsächliche Herunterfahren betrifft, aber Sie müssen ihn trotzdem testen, denn wie sollen Sie sonst erfahren, ob Ihr Skript tatsächlich funktioniert?

Auch das Testen in verschiedenen Umgebungen lässt sich oft nur schwer automatisieren. Buildbot ist bei großen Systemen eine wunderbare Hilfe, aber der Overhead beim Einrichten ist doch spürbar, daher läuft es bei kleineren Systemen doch meist darauf hinaus, viel selber zu prüfen. Ich habe bei der Qualitätskontrolle eine ziemlich gute Intuition, aber sie lässt sich leider nur schwer erklären.

Wann sollte das Debuggen unterrichtet werden? Und wie?

Guido: Andauernd. Sie debuggen Ihr ganzes Leben lang. Ich habe gerade mit meinem sechs Jahre alten Sohn ein Problem »debuggt«, bei dem es darum ging, dass seine Holzeisenbahn immer an einer bestimmten Stelle aus den Gleisen sprang. Debuggen bedeutet meist, eine oder zwei Abstraktionsebenen nach unten zu wandern. Hilfreich ist dabei, sorgfältig zu beobachten, nachzudenken und (manchmal) die richtigen Tools zu verwenden.

Ich glaube nicht, dass es einen einzigen »richtigen« Weg des Debuggens gibt, der zu einem bestimmten Zeitpunkt unterrichtet werden kann, selbst wenn es um ein sehr spezifisches Ziel wie das Debuggen von Programmfehlern geht. Es gibt ein unglaublich großes Spektrum an möglichen Gründen für Programmfehler. Dazu gehören einfache Tippfehler, falsche Gedankengänge, verborgene Einschränkungen von zugrunde liegenden Abstraktionen und schließlich Fehler in den Abstraktionen selbst oder ihren Implementierungen. Das richtige Vorgehen variiert von Fall zu Fall. Tools kommen meist dann ins Spiel, wenn die erforderliche Analyse (»sorgfältig beobachten«) langwierig und sich wiederholend ist. Ich beobachte, dass Python-Programmierer oft nur wenige Tools brauchen, weil der Suchraum (das zu debuggende Programm) so viel kleiner ist.

Wie gehen Sie vor, wenn Sie nach einer Pause mit dem Programmieren weitermachen?

Guido: Das ist eine ziemlich interessante Frage. Ich kann mich nicht daran erinnern, bewusst darüber nachgedacht zu haben, obwohl ich es doch immer wieder tue. Das Tool, das ich dafür vermutlich am ehesten benutze, ist die Versionskontrolle: Wenn ich zu einem Projekt zurückkomme, mache ich einen Diff zwischen meinem Workspace und dem Repository und weiß damit, in welchem Zustand sich mein Code befindet.

Wenn ich die Möglichkeit habe, hinterlasse ich in unfertigem Code XXX-Marker, wenn ich weiß, dass ich meine Programmierung unterbrechen muss. Damit weiß ich später, wo noch etwas zu tun ist. Manchmal verwende ich auch etwas, was ich vor 25 Jahren von Lambert Meertens übernommen habe: Ich hinterlasse eine spezielle Markierung in der aktuellen Quelldatei direkt an der Cursorposition. Die Markierung ist zu seinen Ehren »HIRO«. Das ist umgangssprachliches Niederländisch für »hier« und wurde ausgewählt, weil es in fertigem Code recht selten vorkommt. :-)

Bei Google haben wir auch Tools, die in Perforce integriert sind und mir in einem früheren Stadium helfen. Wenn ich zur Arbeit komme, kann ich einen Befehl ausführen, der mir alle noch nicht abgeschlossenen Projekte in meinem Workspace aufführt, sodass ich daran erinnert werde, woran ich am Tag zuvor gearbeitet habe. Ich führe auch ein Tagebuch, in dem ich dann und wann bestimmte, schwer zu merkende Strings notiere (wie zum Beispiel Shell-Befehle oder URLs), die mir dann dabei helfen, bestimmte Aufgaben für das aktuelle Projekt durchzuführen – zum Beispiel die vollständige URL zu einer Statistikseite eines Servers oder der Shell-Befehl, mit dem die Komponenten neu gebaut werden, an denen ich gerade arbeite.

Was sind Ihre Vorschläge für das Design einer Schnittstelle oder einer API?

Guido: Das ist ein weiterer Bereich, in dem ich mir noch nicht viele bewusste Gedanken über den besten Prozess gemacht habe, obwohl ich schon Tonnen von Schnittstellen (oder APIs) entworfen habe. Ich wünschte, ich könnte hier einfach einen Vortrag von Josh Bloch zum Thema einfügen – er hat über den Entwurf von Java-APIs gesprochen, aber das meiste von dem, was er sagte, gilt für alle Sprachen. Es gibt viele grundlegende Ratschläge, wie das Wählen klarer Namen (Nomen für Klassen, Verben für Methoden), das Vermeiden von Abkürzungen, konsistente Namenswahl, das Anbieten einer kleinen Menge einfacher Methoden, die durch Kombination maximale Flexibilität bieten, und so weiter. Es ist ihm wichtig, die Liste mit Argumenten klein zu halten: Zwei oder drei Argumente sind normalerweise das Maximum, das Sie nutzen sollten, damit Sie nicht mit der Reihenfolge durcheinanderkommen. Das Schlimmste sind viele aufeinanderfolgende Argumente, die alle den gleichen Typ besitzen – vertauscht man sie unabsichtlich, kann das lange Zeit gar nicht auffallen.

Ich habe ein paar persönliche Lieblingsärgernisse: Zuallererst, und das ist spezifisch für dynamische Sprachen, sollten Sie den Rückgabetypp einer Methode nicht vom Wert eines der Argumente abhängig machen, ansonsten kann es schwer sein, zu verstehen, was zurückgegeben wird, wenn Sie den Zusammenhang nicht kennen – vielleicht wird das typbestimmende Argument über eine Variable übergeben, deren Inhalt Sie beim Lesen des Codes nicht einfach erraten können.

Zweitens mag ich keine »Flag«-Argumente, die dafür gedacht sind, das Verhalten einer Methode komplett zu ändern. Bei solchen APIs ist das Flag immer eine Konstante in den tatsächlichen Parameterlisten, und der Aufruf wäre besser lesbar, wenn die API getrennte Methoden hätte: eine für jeden Flag-Wert.

Ein weiterer Punkt ist, APIs zu vermeiden, bei denen unklar ist, ob sie ein neues Objekt zurückgeben oder ein bestehendes verändern. Das ist der Grund dafür, dass die Listenmethode `sort()` in Python keinen Wert zurückgibt – damit wird hervorgehoben, dass sie die aktuelle Liste verändert. Alternativ gibt es die eingebaute Funktion `sorted()`, die eine neue, sortierte Liste zurückgibt.

Sollten Anwendungsprogrammierer die Philosophie »weniger ist mehr« übernehmen? Wie sollten sie die Benutzerschnittstelle vereinfachen, um eine kürzere Einarbeitungsphase zu ermöglichen?

Guido: Wenn es um grafische Benutzerschnittstellen geht, scheint es mittlerweile eine wachsende Unterstützung meiner Position »weniger ist mehr« zu geben. Die Mozilla Foundation hat Aza Raskin als UI-Designer eingestellt, den Sohn von Jef Raskin (dem Mitdesigner des ursprünglichen Macintosh-UIs). Firefox 3 hat mindestens ein Beispiel für eine UI, die sehr mächtig ist, dabei aber keine

Buttons, Konfigurationen, Voreinstellungen oder Ähnliches braucht: die Smart Location Bar beobachtet, was ich eingebe, vergleicht es mit Seiten, die ich zuvor angesurft habe, und macht sinnvolle Vorschläge. Wenn ich die Vorschläge ignoriere, versucht sie, meine Eingabe als URL zu interpretieren oder, falls das nicht geht, als Eingabe für Google. Das ist piffig! Und es ersetzt drei oder vier Funktionsblöcke, die ansonsten eigene Buttons oder Menüeinträge erforderlich machen würden.

Das spiegelt wider, was Jef und Aza seit vielen Jahren predigen: Die Tastatur ist ein sehr mächtiges Eingabegerät, daher sollten wir sie auf neuartigen Wegen nutzen, statt den Anwender dazu zu zwingen, alles mit der Maus machen zu müssen, dem langsamsten aller Eingabegeräte. Das Schöne daran ist, dass Sie keine neue Hardware dafür brauchen, anders als bei Science-Fiction-Lösungen, bei denen Virtual-Reality-Helme oder Augenbewegungssensoren gebraucht werden, gar nicht erst zu reden von Hirnstromdetektoren.

Es gibt natürlich noch viel zu tun – so hat zum Beispiel der Einstellungsdialog von Firefox das schreckliche Look and Feel eines Microsoft-Produkts, mit mindestens zwei Tabulatorebenen und vielen modalen Dialogen, die sich an seltsamen Plätzen verstecken. Wie soll ich mir merken können, dass ich zum Abschalten von JavaScript zur Inhaltsregisterkarte wechseln muss? Finden sich Cookies unter Datenschutz oder unter Sicherheit? Vielleicht kann Firefox 4 den Einstellungsdialog durch ein »smarteres« Feature ersetzen, bei dem Sie Schlüsselwörter tippen, sodass ich bei der Eingabe von »Pass« zum Abschnitt über das Konfigurieren von Passwörtern gelange.

Was bringen die Lektionen über den Aufbau, die weitere Entwicklung und Anpassung Ihrer Sprache für Leute, die heutzutage und in naher Zukunft Computersysteme entwickeln?

Guido: Ich habe dazu ein oder zwei kleine Anmerkungen. Ich bin nicht so der philosophische Typ, daher ist das nicht die Art von Fragen, die ich mag oder auf die ich eine vorbereitete Antwort habe, aber eine Sache ist mir bei Python recht früh als »ich habe es richtig gemacht« aufgefallen (was ABC als Vorgänger von Python zu seinem Schaden nicht hatte): Ein System sollte durch seine Anwender erweiterbar sein. Mehr noch, ein großes System sollte auf zwei (oder mehr) Ebenen erweiterbar sein.

Seit ich Python das erste Mal öffentlich gemacht hatte, erhielt ich Anfragen, die Sprache zu ändern, um bestimmte Arten von Anwendungsfällen zu unterstützen. Meine erste Reaktion auf solche Anfragen ist immer, vorzuschlagen, ein bisschen Python-Code dafür zu schreiben und in ein eigenes Modul zu stecken. Das ist die erste Ebene der Erweiterbarkeit – wenn die Funktionalität nützlich genug ist, landet sie vielleicht schließlich in der Standardbibliothek.

Die zweite Ebene ist, ein Extension-Modul in C (oder C++ oder einer anderen Sprache) zu schreiben. Extension-Module können Vieles, was in reinem Python nicht möglich wäre (auch wenn die Fähigkeiten von reinem Python im Lauf der Jahre gewachsen sind). Ich würde viel eher eine API auf C-Ebene hinzufügen, sodass Extension-Module in den internen Datenstrukturen von Python herumfuhrwerken können, als die Sprache selbst zu ändern. Denn Sprachänderungen müssen immer eine möglichst hohe Kompatibilität, Qualität, semantische Klarheit und so weiter aufweisen. Zudem können Verzweigungen in der Sprache auftreten, wenn sich die Leute »selber helfen« und die Implementierung in ihrer eigenen Version des Interpreters anpassen und dann an andere weitergeben. Solche Verzweigungen verursachen alle möglichen Probleme, zum Beispiel das Warten der eigenen Änderungen, während sich die Basissprache ebenfalls weiterentwickelt, oder das

Zusammenführen mehrerer unabhängiger Zweige, die andere Anwender vielleicht kombinieren müssen. Extension-Module haben diese Probleme nicht: In der Praxis ist die meiste Funktionalität, die von Extensions benötigt wird, schon in der C-API vorhanden, sodass Änderungen daran nur sehr selten notwendig sind, um eine bestimmte Erweiterung zu ermöglichen.

Ein anderer Gedanke ist, zu akzeptieren, dass Sie nicht alles beim ersten Mal richtig machen können. Wenn Sie zu Beginn der Entwicklung nur wenige »Early Adopters« als Anwender haben, ist das Beheben von Problemen schnell erledigt, da Sie sich keine Gedanken um Rückwärtskompatibilität machen müssen. Eine schöne Anekdote, die ich gerne erzähle und die von jemandem als wahr bestätigt wird, der damals dabei war, ist folgende: Stuart Feldman, der ursprüngliche Autor von »Make« in Unix v7, wurde gebeten, die Abhängigkeit der Makefile-Syntax von echten Tab-Zeichen zu ändern. Seine Antwort lautete sinngemäß, dass er ja einsehen würde, dass Tab-Zeichen ein Problem seien, es aber für einen Fix zu spät wäre, weil es schon ungefähr ein Dutzend Anwender gäbe.

Wenn die Benutzerbasis wächst, müssen Sie konservativer werden, und irgendwann ist absolute Rückwärtskompatibilität eine Notwendigkeit. Es gibt einen Zeitpunkt, zu dem Sie so viele Unschönheiten angesammelt haben, dass es nicht mehr tragbar ist. Eine gute Strategie, um damit umzugehen, ist das, was ich mit Python 3.0 mache – für eine bestimmte Version auf die Rückwärtskompatibilität verzichten, die Gelegenheit nutzen, so viele Probleme wie möglich zu beheben, und der Anwender-Community viel Zeit geben, mit dem Übergang klarzukommen.

In Pythons Fall planen wir, Python 2.6 und 3.0 für eine lange Zeit parallel zu unterstützen – viel länger als die üblichen Supportzeiten älterer Releases. Wir bieten zudem verschiedene Übergangsstrategien an: ein automatisches Quellcode-Konvertierungstool, das bei Weitem noch nicht perfekt ist, kombiniert mit optionalen Warnmeldungen in Version 2.6 über die Verwendung von Funktionalität, die sich in 3.0 ändern wird (insbesondere, wenn das Konvertierungstool die Situation nicht ordentlich erkennen kann), sowie eine ausgewählte Rückportierung von bestimmten 3.0-Features nach 2.6. Gleichzeitig wird 3.0 keine komplett neue Sprache und kein vollständiges Redesign sein (anders als Perl 6 oder in der Python-Welt Zope 3), daher werden die Chancen recht klein sein, unbeabsichtigt wichtige Funktionalität zu verlieren.

Ein Trend, den ich in den letzten vier oder fünf Jahren bemerkt habe, ist die verstärkte Vereinnahmung von dynamischen Sprachen durch Firmen. Erst PHP, Ruby in gewissem Maße, definitiv Python in anderen Kontexten, insbesondere Google. Das finde ich interessant. Ich frage mich, wo diese Leute vor 20 Jahren waren, als Sprachen wie Tcl und Perl und ein bisschen später auch Python angingen, all diese nützlichen Dinge zu ermöglichen. Haben Sie den Drang verspürt, diese Sprachen firmenfreundlicher zu machen, was immer das auch bedeutet?

Guido: »Firmenfreundlich« heißt im Allgemeinen, dass die wirklich klugen Köpfe das Interesse verlieren und die eher durchschnittlich Begabten für sich selbst sorgen müssen. Ich weiß nicht, ob Python für durchschnittlich begabte Leute schwieriger zu nutzen ist. In gewissem Sinne kann man sich vorstellen, dass man in Python weniger Schaden anrichten kann, weil es interpretierend ist. Wenn Sie andererseits ein wirklich großes System schreiben und nicht genug Unit Tests durchführen, wissen Sie eventuell gar nicht, was das System tut.

Sie haben gesagt, dass eine Zeile Python, eine Zeile Ruby, eine Zeile Perl, eine Zeile PHP zehn Zeilen Java-Code entsprechen.

Guido: Häufig ist das der Fall. Ich denke, dass das Level in Firmenumgebungen nur der Angst sehr konservativer Manager entspricht, auch wenn es bestimmte Funktionalitätspakete gibt, die hilfreich sind. Stellen Sie sich die für IT-Ressourcen verantwortlichen Personen in einer Firma mit 100.000 Mitarbeitern vor, bei denen die IT gar nicht das Hauptprodukt ist – vielleicht bauen sie Autos, verkaufen Versicherungen oder etwas anderes, aber alles, was sie tun, betrifft irgendwie etwas am Computer. Die für die Infrastruktur verantwortlichen Personen müssen notwendigerweise sehr konservativ sein. Sie greifen lieber auf Dinge zurück, an denen große Namen stehen, wie zum Beispiel Sun oder Microsoft, weil sie wissen, dass Sun und Microsoft zwar auch dauernd etwas vermasseln, diese Firmen aber gezwungen sind, es wieder geradezurücken, auch wenn es fünf Jahre dauert.

Open Source-Projekte haben solche Beruhigungsspillen für den normalen CIO bisher traditionell nicht angeboten. Ich weiß nicht genau, ob, wie und wann sich das ändern wird. Falls Microsoft oder Sun plötzlich Python für ihre jeweiligen VMs unterstützen würden, könnte es passieren, dass die Programmierer in den Firmen tatsächlich bemerken würden, dass sie durch die Verwendung fortgeschrittenerer Sprachen eine höhere Produktivität ohne irgendwelche Nachteile erreichen würden.