第二章

Python

Python 是個現代、具通用目的性之語言，由 Guido van Rossum 運用高階語言 ABC 開發。Python 的哲學為實用性，使用者通常談論 Python 的禪哲學；對於實現任何任務的單一清楚方式有強烈的偏好。它也提供虛擬機器端口，包括 Microsoft 的 CLR 與 JVM，但主要的執行是 CPython，目前仍由 van Rossuma 和其他的自願者持續開發；最新釋出的版本為 Python 3.0，對於改變語言的某些部分及核心程式庫的向後不相容性（backward-incompatible）進行重新考慮。

Python 的途徑

開發程式語言與開發通用軟體專案之間的差異為何？

Guido van Rossum：程式語言比多數軟體專案擁有更多的使用者，而且最重要的使用者是程式設計師本身；這賦予程式語言專案內容的高層次。軟體專案的依存關係階層中，程式語言是在最底層，而且可能採用一種或更多種程式語言；這使得改變語言非常困難，畢竟不相容的改變會影響許多相依的部份，使改變不可行。換言之，所有的錯誤，版本一旦釋出則完全確定。C++ 大概是終極的例子，因為相容性需求的包袱，而要求那些二十年前所寫的程式必須仍然有效。

您如何在語言中除錯？

Guido：我通常不除錯。在語言設計的領域，靈活的開發方法有時是不合常理的；直到語言變得穩定，少數人開始使用它，而且逐漸累積夠多的使用者，才容易在語言定義中發現錯誤，但等到那時再改變已經太遲了。

當然，就像任何舊的程式，執行時有許多機會能除錯，但是程式語言本身需要預先謹慎的設計，因為錯誤的成本是非常高的。

您如何決定語言的功能應該放在程式庫作為一種擴充，或是當使用者需要時，應該能從核心語言中提供功能上的支援呢？

Guido：從歷史的角度而言，我有非常好的答案。早期我注意到的一件事是，每個人都希望他們喜歡的功能被加入語言中，而且這些人多半缺乏程式設計的經驗。每個人總是提議：「讓我們在語言中增加這個功能」或「讓我們加個陳述式來做 X 這件事」。然而，許多情況下，我們的答案是：「如果加寫這兩、三行程式碼，你早就能做 X 這件事了，事情沒有那麼困難」。實際上，你能使用字典，或是結合列表、多元組（tuple），與一般的表達式，或甚至透過撰寫元類別（metaclass）等方式去實現。我或許有從 Linus 取得的答案原始版本，也有類似的哲學。

告訴人們你已經能做到某些功能，第一道防線是顯示你如何能做到，接著告訴人們：「那是件有用的事，我們或許能各自撰寫模組或類別，並且封裝特別抽象的部份」。然後，下一道防線是「這看起來非常有趣、實用，我們將會接受它成為新的功能，並附加至標準的程式庫，成為純正的 Python」。最後，有些需求以純粹的 Python 來做並不容易，我們會建議或推薦將它們轉為 C 語言的擴充。在我們承認並接受：「是的，這非常實用，但你實際上無法做到，因此我們必須改變語言」的說法之前，C 語言的擴充是最後一道防線。

我們也有其他標準是用來判斷是否在語言中增加新功能是合理的，或是在程式庫增加新功能更合理；因為如果那與名稱空間的語意有關，除了改變語言之外，別無它法。另一方面來說，擴充機制足夠強大，有許多部分能從擴充程式庫的 C 程式碼著手，甚至在不需改變語言的情況下，增加新的內建功能。解析器並未改變；解析樹未改變；語言的文件也未改變；所有的工具仍然能運作，然而你已達到增加新功能至系統的目標。

假設有些功能，除了改變語言之外，無法透過 Python 執行，但是您或許否決了應該改變語言的決定。面對這樣的情境，您如何決定這應該 Python 化，那不應該 Python 化，您所依據的標準是什麼？

Guido：依據標準反而更難決定。許多個案的情況，主要是靠直覺。人們很常使用使用 Pythonic 這個字，但是沒有人能提供無懈可擊的定義來解釋什麼是 Python 化（Pythonic），什麼是非 Python 化（un-Pythonic）。

您有「Python 的禪哲學」，但是除此之外呢？

Guido：如同每本神聖的書籍，哲學觀念需要許多的詮釋。當看到好的或不好的提議計畫，我能立即判斷它的好壞；但是要寫出一套規則，而能協助其他人判別好的與不好的語言改善提議，確實非常困難。

聽起來像是鑒賞力主導一切。

Guido：第一件事總是嘗試拒絕不必要的需求，然後看看他們是否能找到其他方式，在不需要改變語言的前提下，滿足需求。值得注意的是，這樣的方式通常行得通。這更像是個「沒有必要改變語言」的操作定義。

如果你堅持讓語言保持不變，人們仍會試著找出方法，做他們需要做的事。除此之外，來自四面八方的使用案例需求，通常沒有一個是應用特定的功能。將那些對網站來說很酷的功能納入語言中並不會讓它成為很好的功能。適合加入語言的功能應該是那些簡潔的功能或更容易維護的類別。程式語言必須超越應用領域的層次，並且讓它變得更簡單或更優雅。

當你改變語言，影響遍及每個人。沒有任何功能是可以被隱藏的很好，因而認為其他人不需要知道。遲早，總有人會發現其他人使用某部分功能所寫的程式碼，或者有人會遭遇某些陰暗角落的特殊案例，而必須從中學習如何處理。

通常優雅性也是見仁見智。我們最近討論 Python 列表功能之一，有人強烈地爭辯使用 `dollar` 比使用 `self-dot` 更優雅。我認為他們對優雅的定義只是幾個按鍵數而已。

語言的簡潔性有個惜墨如金的爭議，但是大多仍取決於個人鑒賞力的差異。

Guido：優雅性、簡潔性，和一般性皆與個人鑒賞力有關，因為對我而言，看起來涵蓋大範圍的事物，對其他人而言可能涵蓋的範圍不夠大，反之亦然。

Python 增強提案（Python Enhancement Proposal, PEP）的建議是如何產生的呢？

Guido：那是非常有趣的歷史檔案。我認為那主要是從其中一位核心開發者 Barry Warsaw 開始支持的。他和我從 1995 年開始共事，大約在 2000 年時，他想出關於改變語言的正式流程提議。

我傾向慢慢的處理這些議題。意思是說，我並不是當初提議我們需要郵件訂閱群組，後來發現郵件訂閱很難處理，而建議我們需要新聞群組的人；我也不是那位提議我們需要討論與發明語言改革流程，並且確認避免不經意錯誤發生的人。

在 1995 與 2000 年間，Barry、我，以及其他幾位核心的開發者，如：Fred Drake 和參與一段時間的 Ken Manheimer，全都在 CNRI，當時，CNRI 負責的其中一件事是籌辦 IETF 會議。CNRI 擁有一個小分支機構，雖然最終分裂收場，但它當時是研討會籌辦單位，而且唯一的顧客是 IETF。它們有段期間曾籌辦 Python 研討會，因此，參加 IETF 研討會是相當容易的事。我和 Barry 相當熟悉 IETF 與其 RFC，以及聚會團隊的審核過程偏好。當他

提議進行類似 Python 的東西，我們下意識的決定，我們不會創造出像 IETF RFC 那麼不靈活的東西，因為某些網路標準的改變，影響遍及更多產業、人員，與軟體，遠超過 Python 的改變所造成的影響，但是我們絕對會在那些改變的標準釋出後才進行模擬。Barry 對於命名有獨特的天份，因此我很確定 PEP 是他的傑作。

我們是當時最早的開放原始碼專案中有此創舉的團隊，因此我們的作法被廣泛地複製。Tcl/Tk 社群基本上只是改變名稱，但是使用和我們完全一樣的定義文件與流程，其他專案也陸續開始模仿。

您認為加入一點形式主義的概念對於 Python 增強設計決策的具體化有任何幫助嗎？

Guido：我認為隨著社群成長，那是必要的。但是我無法僅從每個提議企劃書去判斷它本身的價值。對我而言，讓其他人有機會針對各式各樣的細節進行辯論是有幫助的，也有助於產生相對明確的結論。

當某人請您們評鑑一組具體化的期望與提議時，那有助於導向共識嗎？

Guido：是的。評鑑運作的方式通常是，我剛開始會給予 PEP 一些激勵的評論：「看起來我們這裡有個問題。讓我們看看是否有人能想出正確的解決方案」。通常他們能想出許多關於問題應該如何解決的清楚結論，也同時發現不少懸而未決的議題；通常我的直覺能協助解決那些議題。當討論的主題是我感興趣的領域時，我在 PEP 討論過程中會非常主動積極，例如，如果我們必須增加新的迴圈控制陳述式，我會希望那是由我們來設計；但有些時候我會刻意與某些主題保持距離，例如：資料庫 API。

您覺得什麼創造了新的主要版本需求呢？

Guido：那取決於你對於主要版本的定義。在 Python，每隔 18 至 24 個月，我們通常考慮釋出像是 2.4、2.5，與 2.6「主要的」版本；那是我們能介紹新功能的唯一機會場合。很久以前，新版本的釋出通常是開發者（特別是像我這樣的人）心血來潮的行為。然而，使用者要求可預期性 — 他們反對在次要的改版中（例如：版本 1.5.2 比版本 1.5.1 增加新的主要功能）增加或改變功能；而且，他們希望主要的釋出能提供至少一段時間的支援（約 18 個月）。因此現在我們或多或少朝週期性發佈主要版本的目標邁進：在達成主要的釋出

之前，我們事先計畫重要的時程任務（例如：alpha 與 beta 版本何時推出、新釋出的功能內容為何等），而且，我們要求開發者在最後釋出的期限來臨前，預先將他們計畫完成的改變交付給我們。

選擇在新版本中加入的功能，通常是經過功能優點與其精確規格的長時間討論後，取得核心開發者的共識。所謂的 PEP 流程：Python 增強提議，是以文件為基礎的過程，像是 IETF 的 RFC 流程或是 Java 世界的 JSR 流程，除了沒有那麼正式之外。因為我們的開發者社群規模相對地小，如果（因為產品的優點或特殊的細節）無法達成共識而延宕，最後我可能會打破僵局；我的演算法大部分是直觀的。

然而，最具爭議性的討論通常與使用者可見的語言特色有關，例如：程式庫的附加通常很簡單；雖然在 C 語言的 API 層次上，受限於相當迫切的向後相容性，內部的改善並未被視為功能。

因為典型的開發者都是最暢所欲言的使用者，我無法判斷功能的建議是來自於使用者或開發者。一般而言，開發者根據所接收到來自身邊使用者的需求而提議功能。然而，如果是由使用者提議新功能，極少會成功，因為他們缺乏對於執行（以及語言設計）的徹底瞭解，幾乎是不可能適當地提議新功能。我們喜歡要求使用者解釋他們的問題，而不要在心中預設特定的解決方案;然後，開發者將會提議解決方案，與使用者討論不同作法的優點。

有個關於主要或突破的版本概念，像是 Python3.0 版。從歷史的角度而言，1.0 相當接近 0.9，而且 2.0 距離 1.6 也只是相當小的一步。從現在起，在擁有更大的使用者基礎上，Python3.0 這類型版本的確很稀少，因為它提供與先前版本完全不相容的唯一情況。主要版本的創造通常都維持與先前主要版本的向後相容性，而且提供特別的機制用來防止功能的移除。

您如何決定選擇處理數字作為任意精度的整數（以所有您能取得的優勢），以取代舊有的（而且非常普遍的）方式來傳遞給硬體？

Guido：基本上，我從 Python 的前身 ABC 承襲這個想法。ABC 使用任意精度的有理數，但我不太喜歡有理數，因此我轉而使用整數；Python 使用硬體支援的標準浮點數表達方式（ABC 也是，只是多加了一些東西）。

最初，Python 有兩種整數：一種是平常的 32-bit（int）類型，另一種是單獨的任意精度（long）類型。許多語言也如此分類，但精度的種類被歸類至程式庫，像是 Java 和 Perl 的 Bignum，或是供 C 語言使用的 GNU MP。在 Python 中，兩種整數類型幾乎總是在核心語言中並列，但是使用者必須透過附加“L”到數字上，來選擇長整數的使用。逐漸地，這被視為困擾。在 Python2.2 中，我們介紹長整數的自動化轉換，當一般整數的數學運算正確結果無法以 int 呈現時（例如：2**100），會進行自動轉化處理。

在過去，這會引起 `OverflowError` 的例外。有一段時間，結果會在沒有提醒使用者的情況下被截短，但是我後來又將它改為顯示錯誤提醒的例外狀況。在 1990 年早期，我浪費整個下午為一個短小的程式樣本進行除錯，其實能透過撰寫演算法來執行，讓非常大的整數也能適用。這類型的除錯經驗在語言的萌芽階段是很重要的。

然而，仍然有些情況，兩種整數類型的行為會有些微的差異；例如，在十六進制或八進制列印輸出一個整數會產生無符號整數的結果（如：-1 會被列印輸出為 FFFFFFFF），然而對於數學上等長的數字進行同樣的運作，結果有時卻產生有符號的結果（如：-1）。在 Python3.0 中，我們採取激進的步驟，改為只支援單一整數類型；我們雖然稱它為 `int`，但在執行上卻大部分沿用舊的長整數類型。

為什麼您稱它為激進的步驟？

Guido：大部分是因為從目前的 Python 實務，有許多關於這方面的討論，而且人們提議各種不同的替代方案，讓兩種（或更多）表達方式能在內部使用，但是從使用者介面完全地或是大部分隱藏（對於 C 語言擴充的撰寫者則例外）。這也許有助於效能改善，但是最終那需要大量的工作才能達成；而且，在內部擁有兩種呈現方式增加正確執行的負擔，也使它與 C 程式碼的連結更麻煩。我們現在希望那對效能的影響並不嚴重，也希望能因此改善其他技術（如：快取）的效能。

您如何採用「應該有個方法 — 而且最好是唯一明顯的方法來進行」的哲學？

Guido：那也許是一開始的潛意識想法。當 Tim Peters 撰寫「Python 的禪哲學」時，他明確的制定許多規則，我們已經習慣應用；這麼說來，這些特定的規則（在我知情的情況下，經常被違背）直接來自對於數學與電腦科學優雅性的期望。ABC 的作者也應用它，期望能有少數正交的類型和概念。正交性的概念源自數學，指的是擁有一種方法（或是一種真

實的方法）來表達事物的**定義**。例如，一旦你選擇了原點和三個基本的向量，XYZ 在 3D 空間的座標是獨一無二地明確。

我也認為我實際上幫助大多數的使用者，讓他們不需要在兩個類似的選項之間抉擇。你可以與 Java 對照，如果需要列表狀的資料結構，標準程式提供許多版本（如：連結的列表、陣列的列表，及其他），或是透過 C 語言，你必須決定如何執行自己的列表資料型態。

關於靜態與動態類型，您的立場支持哪一種類型？

Guido：我希望我能簡單的說，像是「靜態類型不好，動態類型好」，但事實並不總是那麼單純。有幾種不同處理方式的動態類型，從 Lisp 到 Pyhton，也有不同方式的靜態類型，從 C++ 到 Haskell。語言像是 C++ 和 Java 也許給予靜態類型不好的名字，因為它們需要你告訴編譯器同樣的事物好幾次。然而，語言像是 Haskell 與 ML 使用類型推論（type inferencing），是相當不同的方式，與動態類型具備同樣的效益，例如：更簡潔的程式碼表達方式。不過，功能的樣板、規範似乎很難使用 ── 事物像是輸入 / 輸出，或是圖形使用者介面的互動與模型並非十分契合，而常常透過另一種較傳統語言（如：C 語言）的橋樑來協助解決。

在某些情況下，Java 的冗長被視為優點；它使程式碼瀏覽工具的創造成為可能，而且能回答問題像是：「這個變數哪個地方被改變了？」或是「誰呼叫這個方法？」。然而，動態的語言使回答類似的問題更難，因為如果沒有透過程式碼庫來分析每個路徑，通常很難找出方法的類型。

我不確定像 Haskell 這種函式語言如何支援類似的工具；就我的理解，你可能必須使用與動態語言同樣的技術，因為那是類型推論的任務。

我們正朝著混合類型發展嗎？

Guido：我預期未來將有很多關於混合類型的議題可供討論。我注意到多數以靜態類型語言所撰寫的大型系統，實際上包含顯著數量的子集程式實質上是動態類型。例如，Java 語言的圖形使用者介面工具集與資料庫 API 常常看似與靜態類型對抗，但是卻將多數的正確性檢查移至運行期。

擁有函式與動態層面的混合性語言是相當有趣的。我應該將它加入語言，儘管 Python 已支援某些函式工具，像是 `map()` 與 `lambda`。然而，Python 並未擁有函式語言的子集：沒有類型推論，也沒有平行化的機會。

為何您選擇支持多元典範呢？

Guido：我並沒有支持多元典範。Python 支持程序式的程式設計，就某種程度而言，還有物件導向程式設計。這兩種典範並沒有太大的差異，而且 Python 的程序式風格仍然受到物件強烈地影響（因為基礎的資料類型都是物件）。Python 支援一小部分的函式語言 ── 但是它不像任何真正的函式語言。函式語言主要是在編譯期儘可能地處理能處理的工作 ── 函式的層面指的是除非明確地宣告，編譯器能強力保證完全無副作用，而能完善任務。Python 擁有最簡單、最被動的編譯器，而且官方的運行期語義阻撓了編譯器的巧妙設計，像是平行式的迴圈或是將循環轉變成迴圈。

Python 也許擁有支援函式程式設計的聲譽，因為它將 `lambda`、`map`、`filter` 與 `reduce` 納入語言中，但是，在我眼中，這些只是語法上的甜頭，並非函式語言的基礎根基。Python 與 Lisp（也不是函式語言！）分享基礎的屬性在於，函數是一級的物件，能像其他物件一般任意的傳遞。這與嵌套範圍（nested scopes）及類似 Lisp 的方法結合，使執行函式語言概念的過程變得容易。在函式語言中，這些概念本來是原始運算，但是必要的原始運算執行已經內建於 Python。因此，只需幾行 Python 程式，你就能撰寫 reduce()，而不需透過函式語言。

當您創造語言時，您曾考慮語言會吸引的程式設計師類型嗎？

Guido：是的，但是我也許沒有足夠的想像力。我心目中的潛在使用者是使用 Unix 或類似 Unix 環境的專業程式設計師。Python 早期版本的手冊採用的標語是「Python 彌補了 C 與 Shell 程式設計的鴻溝」。

因為那是我對自己工作的定位，而且很快地人們開始圍繞在我身旁。我從來沒想過 Python 會成為內嵌於應用的優秀程式語言，直到人們開始探詢我們的作品。

對於在中等學校、大學，或甚至針對自學程式設計者教導程式設計的優先準則是很實用的事實，只是個幸運的巧合；我所保留的許多功能其實來自於 ABC ── 它的目標特別是教導非專業程式設計師如何寫程式。

您如何平衡語言不同需求的差異呢？語言應該讓新手容易學習，還是應該讓資深程式設計師擁有足夠強大的功能，而能從事實用的應用。這算是錯誤的二分法嗎？

Guido：平衡是重點。有一些著名的陷阱應該避免，像是設計初衷是為了幫助新手，卻對專家造成困擾，或是專家需要，但很容易讓新手混淆的功能；因此，讓雙方滿意的空間還很寬廣。另一個策略則是讓專家能運用進階的功能，例如，語言雖然支援元類（metaclasses）的使用，但是沒有理由讓新手瞭解那些功能。

優秀的程式設計師

您如何辨識優秀的程式設計師？

Guido：辨識優秀的程式設計師需要時間。例如，在一小時的面談時間內，實在很難判斷一個人的好壞。當你與某個人一起工作，處理各式各樣的問題，是好是壞就會變得相當清楚。我對於提供特定的標準有點猶豫，一般而言，優秀的人會顯露出創意，學習速度快，而且很快就寫出能運作的程式碼，在準備交付成果之前，也不需要進行過多的修改。值得注意的是，大家擅長程式設計的不同層面 ── 有些人擅長演算法與資料結構，其他人則擅長大規模的整合、協定的設計，測試、API 設計、使用者介面，或是其他程式設計的面向。

您會運用什麼方法來雇用程式設計師？

Guido：根據過去的面談經驗，我不認為我擅長於傳統的員工聘顧方式 ── 我的面談技巧幾乎是不存在的。我所採用的方式像是學徒制，與人們近距離共事一段期間，最終能體會他們的優點與缺點；有點類似開放程式碼專案運作的方式。

是否有任何特質，已經成為您基本的評估標準，而能判定是否找到優秀的 Python 程式設計師？

Guido：我擔心你的問題是從一般經理人的角度，只是想要雇用幾位撰寫 Python 的程式設計師。我並不認為這個問題有任何簡單的答案，而且事實上，我認為這也許是個錯誤的問題；因為你想雇用的不僅是 Python 程式設計師，而是聰明、有創意、自動自發的員工。

如果查看程式設計師的徵才廣告，幾乎每個都有一行關於能與團隊合作的要求。您對於團隊在程式設計中所扮演的角色看法如何？對於那些無法與他人合作的聰明程式設計師，您認為他們仍有發展的空間嗎？

Guido：我同意徵才廣告強調團隊合作的層面。聰明的程式設計師如果不能與團隊合作，就不應該讓自己處於被雇用為傳統程式設計師的情境 — 那對於所有相關的人都是個災難，而且他們的程式碼會變成承接者的夢魘。我認為不能與團隊合作是明顯缺乏才華的象徵。在現代，有許多管道能學習團隊合作的技巧，而且如果你真的很聰明，應該更能輕易的學會團隊合作的技巧 — 如同學習如何執行有效的快速 Fourier 變換（Fast Fourier Transform），只要你下定決心學習，它並沒有想像中的那麼難。

身為 Python 的設計者，與其他技藝高超的 Python 開發者比較，您所擁有的語言優勢為何？

Guido：我不知道 — 從發展初期到現在，語言和虛擬機器已被許多人使用過，以至於有時我對於目前累積的成果大為訝異。如果說到擁有其他開發者所沒有的優勢，那或許是我使用語言的時間比任何人都長。經歷這麼長的時間，我有機會仔細思考何種運算比較快，哪些比較慢 — 例如，我或許更容易注意到局部（locals）比全局（globals）快速（雖然其他人可能過於激進的使用它）、函數和方法呼叫的代價高昂（比在 C 或 Java 還高），或最快速的資料型態是多元組（tuple）。

然而，當談及標準程式庫的使用，我常常覺得其他開發者擁有優勢。例如，我每隔幾年會撰寫一項網路應用，但我發現可用的科技隨著時間而改變；因此，每次都變成使用新的架構或方法撰寫「最初的」網路應用。而且，我仍然沒有機會著手處理 Python 重要的 XML 問題。

Python 的特色之一是簡潔。您覺得這如何影響程式碼的可維護性呢？

Guido：我聽過研究及坊間證據顯示，不論使用何種程式語言，每行程式的平均錯誤率是相當一致的。因此，透過像 Python 這類的程式語言進行開發，同樣功能的應用比起以 C++ 或 Java 開發的應用會小很多，當然也會讓應用更容易維護。這同時也可能顯示單一的程式設計師負責更多功能。

這雖然是個獨立的議題，但是仍然對 Python 有利：因為根據人月神話（*The Mythical Man-Month*）[Frederick P. Brooks; Addison-Wesley Professional] 書中提到的原則，每個程式設計師具備更高的生產力，可能代表團隊擁有較少的程式設計師，但相對地，需要的溝通成本也較低。如果我沒記錯的話，書中提到所增加的溝通成本是團隊規模的平方。

介於 Python 提供的原型與建立完整應用所需投入的努力之間，您看到任何關連嗎？

Guido：我從不認為 Python 是原型設計的語言。而且，我不相信在語言的原型與產品之間有清楚的界線。有些情況下，撰寫原型最好的方法是寫個用後即丟的 C 語言駭客程式，也有些狀況是使用非程式設計的方式創造原型 — 例如，使用試算表或一組 `find` 和 `grep` 指令。

對於 Python 最早的意圖，只是希望成為當 C 語言及 shell 程式腳本太笨重時，一種可以替代 C 的語言。那包含許多原型的設計，但也同時包含許多商業邏輯，並不特別耗費運算資源，但是需要撰寫許多程式碼。多數 Python 程式碼不是作為原型，而只是將它的任務完成；因此，對於最後應用的版本應該不需要進行太多改變。

一般的流程是從簡單的應用開始，逐漸增加更多功能，而且，最後複雜的程度成長了十倍。從原型到最終的應用之間，從來沒有一個精確的分界點。例如，我在 Google 開始的一個應用 Mondrian，從我首次釋出第一個版本到現在，程式碼的規模可能已經成長了大約十倍，而且全部都用 Python 撰寫而成。當然，也有些例子，Python 最後被更快的語言所取代 — 例如，最早的 Google 網路蜘蛛 / 索引器大部分都是 Python 撰寫成的 — 但是這些例子是例外，不是規則。

Python 的直接性如何影響設計過程？

Guido：直接性常是我工作的方式，至少對我而言，Python 的直接性運作良好。當然，我寫過許多程式碼，後來丟棄，但是撰寫程式碼（甚至不需執行）通常幫助我理解問題的細節。考慮如何重新安排程式碼，而能以更完善的方式解決問題，也對於我看待問題有所助益。當然，這不應該是避免使用白板來草擬設計、架構、互動，或其他早期設計技術的藉口。技巧在於使用正確的工具來工作；有時，需要的只是鉛筆和紙巾 — 其他時候需要的是 Emacs 視窗與 shell 指令提示。

您認為由下而上的程式開發更適合 Python 嗎？

Guido：我不覺得由下而上和由上而下的開發方式像 vi 和 Emacs 那樣被視為互相對立。在任何軟體開發的過程中，有些時候會選擇由下而上的方式開發，其他時候則會採用由上而下。由上而下也許指的是在開始寫程式之前，需要詳細地檢視與設計；而由下而上的方式也許適合在現有層次上建立新的抽象概念，例如：建立新的 API。我並不是暗示應該直接開始為 API 撰寫程式，而不需要在心中存有設計的藍圖，但是通常新的 API 邏輯上依循現有可取得的低階 API，因此設計與撰寫程式可以同時進行。

在何種情況下，您覺得 Python 程式設計師會更欣賞它的動態本質？

Guido：當你探索大問題或解決空間，而沒什麼頭緒時，語言的動態功能通常最有用 ── 你能進行實驗，並從每次實驗的結果中學習，而不需要產生太多程式碼，將自己侷限在特定的方法。使用 Python，你能撰寫非常簡潔的程式碼 ── 撰寫百行左右的 Python 程式碼來進行一次實驗，然後再正式開始，會比使用 Java 撰寫千行程式的實驗架構更有效率。

從安全的觀點而言，Python 提供什麼關於安全的支援給程式設計師呢？

Guido：那取決於你所顧慮的攻擊種類。Python 擁有自動化的記憶配置，因此 Python 程式不易遭遇特定錯誤類型。當然 Python 的運行期是以 C 語言撰寫，這些年來，漏洞的確已被發現，因此有些刻意逃避 Python 運行期範圍的設計，如 `ctypes` 模組可供呼叫任意的 C 程式碼。

Python 的動態本質對它本身有助益，還是產生相反的效果呢？

Guido：我不認為動態的本質有助益或是有害。你能很容易地設計出充滿安全漏洞的動態語言，或是沒有弱點的靜態語言。然而，擁有運行期或是**虛擬機器**（*virtual machine*）來限制原始底層機器的存取是有助益的。同時，Python 是第一個被 Google App 引擎專案所支援的程式語言，也是原因之一；我目前仍參與這個專案。

Python 的程式設計師如何檢查並改善程式碼的安全性？

Guido：我認為 Python 程式設計師不應該過度擔心安全性的問題，當然還是得考慮特定的攻擊模式。應該留意的重點和語言一樣：對某些你不信任的資料來源（對於網路服務器而

言，指的是每個傳入的網路請求，即使是網頁標題）抱持懷疑態度。特別需要注意的是正規表達式 — 撰寫執行指數時間的正規表達式是很容易的，因此，執行檢索的網路應用，其中使用者類型的正規表達式，應該提供機制去限制運行時間。

對於精通使用 Python 進行開發工作，您是否建議任何關於基礎的觀念（如：通用規則、觀點、心態，與原則）的理解？

Guido：我的建議是實用主義。如果你太執著於理論上的概念像是資料隱藏、存取控制、抽象化，或規格，你不是真正的 Python 程式設計師，而且最後導致浪費時間與語言對抗，而不是使用（以及享受）語言；你也可能會很沒有效率地使用它。如果你像我一樣是個即時滿足迷，Python 是很好的選擇。

但是如果你享受極限程式設計或是靈活的開發方式，我仍會建議採取中庸之道。

您提到的「與語言對抗」指的是什麼？

Guido：那通常指的是他們試著以過去的習慣來使用不同的語言。許多關於移除明顯表達式的提議，多數來自於最近才開始使用 Python，但尚未習慣的使用者。有時候他們提出改變語言的建議，有時候又想出超級複雜的元類別，反而更不容易理解。這同時會導致超級無效率或無法在多執行緒環境中運作的後果。任意改變大小寫的作法將會把所有的事物轉變為類別，將每個存取轉變為存取方法，在 Python 中不是個明智的作法，只是擁有更多冗贅的程式碼、很難除錯，而且拖慢運作速度。你如果聽過「你可以用任何語言來撰寫 FORTRAN」的說法，你也能使用任何語言來撰寫 Java。

您花費許多時間試著創造做事的明確方法。您的意見似乎暗示透過 Python 的方式來做事，才能善加利用 Python 的優勢。

Guido：我不確定我真的花費許多時間確認只有一種可行的方法。與 Python 語言相比，「Python 的禪哲學」是很新的概念；而且，在 Tim Peters 以這種詩歌形式來描述 Python 的設計哲學之前，多數定義清楚的語言特色已經存在許久。我不認為他完成著作時，曾預期他的書籍會如此廣泛的散佈以及成功。

那是個容易琅琅上口的標題。

Guido：Tim 對於字詞有他獨特的見解。實際上，在多數情況下，「只有一種可行的方式」是個善意的謊言。有許多可供處理資料結構的方式，你可以使用多元組與列表。在大多數個案中，使用多元組、列表，或是字典，其實沒有太大的差異。如果你仔細觀察，一種解決方案運作的更好，因為它在許多情況下也運作的一樣好；只有在一或二個個案中，如果資料量持續增加，列表會比多元組運作的更好。

這實際上來自於原始 ABC 語言的哲學，試著維持元件的稀疏性。ABC 與 ALGOL-68 實際上有共通的設計哲學。ALGOL-68 是目前最沉寂的語言之一，但是當時非常具影響力。當然那時大約是 1980 年代，它非常有影響力是因為 Adriaan van Wijngaarden 這位大人物來自 ALGOL-68 開發團隊；當我進入大學時，他仍然在學校教書。我曾經修過他的課，當時他告訴我們許多關於 ALGOL-68 的歷史趣聞。他也曾經是 CWI 的主任。

當時許多人深入研究 ALGOL 68。Lambert Meertens 是 ABC 的主要作者，也是 ALGOL 68 研究報告的主要編輯之一。他深受 ALGOL 68 設計哲學的影響，設計提供可供組合的變數，而能產生各種不同的資料結構或建構程式的方法。

他提出非常具有影響力的說法：「我們擁有列表或陣列，而且它們能包含任何種類的事物。它們能包含許多字串，但是也包含其他陣列和多元組。你能將所有的事物結合在一起。」突然地，你不需要關於多維陣列的獨立概念，因為陣列的陣列解決了維度的問題；這種將幾個關鍵事物，讓它們包含不同面向的彈性，以及允許這些面向相互結合的哲學，是 ABC 語言的主軸。我不加思索的借用這方面的設計哲學。

當 Python 試著營造一種形象，只要不將陳述式嵌入表達式中，就能透過很有彈性的方式結合物件。實際上語法規則有相當多的特殊個案，像是逗號指的是參數之間的分隔；在其他情況下，逗號指的卻是列表中的項目；還有另一種情況，指的是隱性的多元組。

在語法中有許多變化，因此特定的運算子不被允許，因為它們會與相鄰的語法衝突。但是那從來不是個問題，因為當無法運作時，你總能額外加上一組括號來解套。至少從解析器的作者觀點而言，這種語法的使用成長了不少。某些事物，如列表理解和產生的表達式在語法上仍無法完全地結合。在 Python 3000 中，我相信它們已經能完全的結合。雖然仍有些細微的語意差異，但至少語法上是相同的。

多元化的 Python

解析器在 Python 3000 中變得更簡單嗎？

Guido：幾乎沒有。它沒有變得更加複雜，也沒有變得更簡單。

沒有變得更複雜，我認為這是它成功的地方。

Guido：是的。

為何最簡單與最被動的編譯器是可以想像的呢？

Guido：這原本是個非常實用的目標，因為我並沒有程式背景與相關學位。而且，在能對語言進行任何有趣的工作之前，我必須擁有位元碼產生器在背後支持。

我仍然相信擁有非常簡單的解析器是件好事；畢竟，那只是將文字轉成樹狀結構，呈現程式結構的工具。如果語法是如此的模稜兩可，以至於需要透過科技最先進部分來瞭解它，可想而知人類讀者半數的時間都感到困惑的狀態。這個事實也讓它很難撰寫另一個解析器。

使用 Python 讓解析工作變得非常簡單，至少從語法的層次上而言。而在詞彙的層次上，分析是相對的細微，因為必須使用內嵌於詞彙分析器的堆疊來讀取首行縮排的空白，對於詞彙與文法分析隔離的理論剛好是個反例。儘管如此，那是正確的解決方案。有趣的是，我喜歡自動化產生的解析器，但是我並不十分相信自動化產生的詞彙分析。Python 總是支援手動產生的掃瞄器，與自動化的解析器。

人們曾為 Python 寫過許多不同的解析器。即使是 Python 連結至不同虛擬機器（無論是 Jython, IronPython, 或 PyPy）的端口，也有自己的解析器，那沒什麼大不了的，因為解析器從來都不是專案中非常複雜的工作；語言的結構很容易透過最基本的預讀標記原理，使用遞歸下降解析器（recursive descent parser）來處理。

使解析器變慢的原因實際上是字義的模稜兩可，而且只能透過預讀來解決，直到程式結束。在自然語言中，有諸多例子是關於在讀到最後一個字之前，是不可能解析一個句子

的。或者如果你碰巧認識句中談論到的某個人名，那些句子能被解析，但那是完全不同的情況。對於解析程式語言，我還是喜歡我的預讀標記（one-token lookahead）。

那麼說來，在 Python 中似乎永遠不會有巨集的產生，因為你必須執行另一個解析階段的工作！

Guido：雖然將巨集內嵌於解析器的方法也許是可行的，我一點也不相信巨集能解決任何 Python 中特別急迫的問題。從另一個角度看，因為語言很容易解析，如果你想出某種乾淨的巨集組，相容且適合語言的語法，執行微評估作為解析樹的運作可能會非常簡單。但這不是我特別有興趣的領域。

為何您在原始程式碼中選擇使用嚴格的格式？

Guido：對於分群時，首行縮排的選擇並不是 Python 的新觀點；我從 ABC 直接繼承這個觀點，但它同時也出現在 occam 這個舊語言中。我不知道 ABC 作者群是從 occam 取經，或是獨立發明這個概念，或是兩者都有共同的祖先？這可歸功於早在 1974 年提出這個觀念的 Don Knuth。

當然，我能選擇不跟進 ABC 的原則（如：ABC 使用大寫字母作為語言的關鍵字與程序名稱，這是唯一我沒有沿用的原則），但是當使用 ABC 時，我竟然有點喜歡這個功能，因為那似乎能擺脫在 C 語言使用者之間，關於放置大括號位置這種沒有意義的爭論。我也注意到可閱讀的程式碼自願使用首行縮排，來顯示分群；但是我在程式碼中看到微小的錯誤，首行縮排與使用大括號來處理語法分群不一致 — 程式設計師和評論者假設首行縮排與分群相配合，因此沒有注意到這個錯誤。然而，長時間的除錯階段能帶來很寶貴的經驗。

嚴格的格式應該能產生更乾淨的程式碼，而且也許能減少程式設計師對於程式排列格式的差異；但是，這聽起來不是強迫人類去適應機器嗎？

Guido：剛好相反 — 格式提供給予人類讀者的幫助超過於機器；看看先前的例子，這種方式的優點在於，當維護由其他程式設計師撰寫的程式時，也許更容易讓人看見。

新使用者常常一開始會產生排拒，雖然現在已經比較少聽到類似的現象了；也許教導 Python 的人們已學會預先考慮這類的效果，而能進行有效的處理。

我想請教您關於 Python 處理多重執行的問題。Python 大約有四個或五個大型的執行，包括 Stackless 與 PyPy。

Guido：技術上，Stackless 並不是獨立的執行。Stackless 常常被列為獨立的 Python 執行，是因為身為 Python 的分支，它以一種不同的方式，取代虛擬機器中很小的部分。

基本上是位元碼傳送分派，對嗎？

Guido：多數位元碼的傳送分派非常類似。我認為位元碼是一樣的，當然所有的物件也是相同的。它們運作的相異之處在於從 Python 的一個程序到另一個程序的呼叫：它們透過物件的操作來進行，在其中推動堆疊架構的堆疊，而且同樣的 C 程式仍然維持領導的角色。它在 CPython 中的作法是，使用一個 C 函式，最終將會使用虛擬機器的新實體；那並不是整個虛擬機器，卻是用來解釋位元碼的迴圈。在 Stackless 中，只有其中一個迴圈會出現在 C 堆疊上。然而，在傳統 CPython 的 C 堆疊上，你能擁有許多次同樣的迴圈。這是唯一的不同。

PyPy, IronPython, 與 Jython 皆是獨立的執行。我不清楚那些關於轉譯成 JavaScript 的事物，但是如果某些人已經考慮到相關的問題，我不會感到驚訝。我聽過轉譯成 OCaml 與 Lisp 的實驗性作法，也曾出現轉譯成 C 程式碼的個案。

Mark Hammond 與 Greg Stein 在 1990 年代晚期從事轉譯的工作，但是發現能獲得的加速非常有限。最好的情況時，它能以約兩倍快的速度執行；但是產生的程式碼也非常龐大，導致管理大型程式成為問題。

啓動時間造成傷害。

Guido：我認為 PyPy 的使用者群朝向正確的開發方向。

您似乎傾向支持那些執行。

Guido：我總是支持那些在不同時期交替出現的執行。從 Jim Hugunin 走進門，帶著完成的 JPython 執行出現時，我興奮不已。就某種意義來說，它可算是語言設計的驗證；它也同時傳達，在平台上，人們能使用他們喜愛的語言。我們雖然有方法達到類似目標，但是

它確實能幫助將其他功能與我所關心的語言功能隔離；以及隔離那些我認為能與其他不同方式處理的執行並存的特殊執行功能；那導致後來我們陷入垃圾收集的更糟境地。

那是可能變得更糟的情況。

Guido：但是它也是必要的。我無法相信我們花費多久的時間設法在純粹的參考計算法（reference counting）生存，但是仍無法打破循環。我總是將參考計算法視為進行垃圾收集的方法，而且它不是個特別差的方法。過去曾有參考計算法對垃圾收集的聖戰，在我眼中是很愚蠢的對抗。

回到關於那些執行，我認為 Python 是個很有趣的空間，因為和其他語言比較，如：Tcl、Ruby 與 Perl 5，它擁有相當好的具體說明規範。規範的出現是因為您想要將語言及其行為標準化，還是因為您考慮多重執行的情況，或是其他因素呢？

Guido：可能是提出 PEP 的社群流程與多重執行的副作用。當我最初撰寫第一版文件時，我滿腔熱情的開始撰寫語言參考手冊，那應該能提供足夠精確的說明，讓幾乎一無所知的使用者能正確的執行語言與使用語義。但是我從來沒有確實的實現那個目標。

擁有高度數學化的具體說明，ALGOL 68 或許是目前最接近那個目標的語言。其他語言，例如 C++ 與 JavaScript，僅依賴標準化委員會的純粹意志力來管理，特別是 C++ 的個案。很明顯地，那是非常令人欽佩的努力。同時，撰寫精確的說明文件也需要投入高度的勞動力。然而，我對 Python 說明文件的相同期望也從未真正的實現。

不過，我們擁有的是關於語言應該如何運作的足夠理解，以及現有足夠人力能回答其他版本執行者的問題。例如，我知道 IronPython 團隊成員認真的嘗試運作整個 Python 的測試組，針於失敗的部份判定測試組是否測試到 C Python 執行的特殊行為，或是否在執行上仍有更多工作待處理。

PyPy 團隊也進行同樣的工作，但是他們更進一步。他們擁有幾個比我更聰明的設計師，想出邊緣個案，像是關於如何產生程式碼，以及如何在 JIT 的環境下分析程式碼。當發現特殊情況及沒有人真正想過的問題，他們會實際上貢獻一些測試、消除歧異的方法與問題；這些都非常有幫助。擁有多重執行流程的程式語言對於釐清語言的規範歧異非常有幫助。

您是否曾預見 C Python 未來可能不會成為主要的執行呢？

Guido：這很難預測。我是說有些人預見 .NET 會主導世界一段時間；其他人則預測 JVM 會主導世界一段時間。對我而言，那看起來都像是如意算盤。同時，我不知道未來將會發生什麼事。可能會有些巨大突破，即使我們所知的電腦不會改變，不同種類的平台會突然變得更加普遍流行，導致完全不同的規則產生。

也許從 von Neumann 的架構改變？

Guido：我甚至沒想過這個問題，但是，那當然也是一種可能性。我曾經考慮的情況是如果行動電話變成無所不在的運算設備。行動電話只落後筆記型電腦幾年，這暗示著，幾年後，行動電話，除了鍵盤與螢幕，將會擁有足夠的運算能力，因此你再也不需要筆記型電腦。也有可能行動電話的平台政治角力會以擁有 JVM 告終、或是其他 C Python 並非最佳方法的標準環境勝出，但是一些其他 Python 的執行在標準環境下運作的更好。

當然，也會有當我們在晶片上，甚至在筆記型電腦或是行動電話上，擁有六十四個核心時，應該如何做的問題產生。我不知道那是否應該改變程式設計的典範。可能會使用某些語言讓你能明確說明微妙的並行過程，但是在多數情況下，一般的程式設計師無法撰寫正確的執行緒安全的程式碼。假設多元核心的攀升強迫它們那麼做，有點不切實際，我仍期望多元核心將會非常實用，但它們將會被用作粗粒度的並行性，那是更好的作法，因為高速緩存命中與遺漏之間有龐大的成本差距，讓主要記憶體不再提供共享記憶體的功能。你會想要讓流程愈獨立愈好。

我們如何處理並行的問題？這個問題應該在什麼層次上被處理，或者更好的是，被解決？

Guido：我的感覺是撰寫單一執行緒程式已經很難了，撰寫多元執行緒程式更難，困難的程度讓人們沒有將它做對的希望，當然也包括我在內。因此，我不相信細微粒度的同步原始與共享記憶體會是解決之道 ── 我寧可看到訊息傳遞解決方案回歸它的風格。我非常確定透過增加同步建構（synchronization constructs）來改變所有的程式語言是個壞主意。

我仍然不相信試著從 CPython 移除 GIL 有效。我相信某些管理多重流程（相對於執行緒）的支援仍是一個難題，因此，Python 2.6 與 3.0 將會有個新標準程式庫模組，進行多重處理，提供類似執行緒模組功能的 API。此外，它還支援運作在不同主機上的流程。

權宜之計與經驗

當撰寫軟體時，您是否覺得遺漏了任何工具或功能呢？

Guido：如果能像使用鉛筆和紙般容易的在電腦上描繪，我也許會在設計的思考過程中繪製更多的略圖。我擔心這必須等到滑鼠普遍地被筆（或手指）取代，才能直接在螢幕上描繪。以我個人的觀點，當使用任何電腦化的繪圖工具，我感覺受到高度障礙，即使我非常擅長使用紙筆 — 也許因為我繼承建築師父親的天份，他的工作必須繪製許多草圖，因此我也耳濡目染的在青少年時期養成繪草圖的習慣。

我假設我可能不知道在探查大型程式庫的過程中遺漏了什麼。Java 程式設計師現在擁有 IDE，能提供問題像是「這個方法的呼叫者在哪裡？」或「這個變數被指定到哪裡？」的快速答案。對於大型的 Python 程式，這也是很實用的工具，但因為 Python 的動態本質，必要的靜態分析反而變得更困難。

您如何測試程式碼與除錯？

Guido：當我寫程式時，我進行許多測試工作，但是每個專案的測試方法都不同。當撰寫基本純粹的演算法程式，初步的單元測試通常效果不錯，但是當撰寫高度互動的程式碼或是連結舊 API 的介面時，藉由 shell 中的指令歷史搜尋與瀏覽器頁面重新載入的協助，我最後常常進行許多手動測試。舉一個（極端）的例子，當程式腳本的單一目標是關閉目前的機器時，你無法為它撰寫一個很好的單元測試；當然，你可以模擬真正關閉的情況，但是仍然必須測試那部分的程式，否則如何知道你的程式實際能運作？

在不同的環境測試也常常很難自動化。Buildbot 適用於大型系統，但是設定的成本非常高，因此對於小型系統，你最後只是進行許多手動的測試。對於品管測試工作，我已培養出很好的直覺，可惜的是，那實在很難解釋。

您覺得應該在何時教導程式除錯呢？應該如何教導呢？

Guido：持續不斷地進行。基本上設計師整個職業生涯都在除錯。我剛剛對我六歲兒子的木製火車玩具組問題進行除錯，他的火車在某個軌道的轉彎點持續出軌。除錯的工作主要是向下移動一、兩個抽象層，而且藉由暫停來觀察清楚問題與思考，以及使用正確的工具來解決問題。

我認為除錯不是只有單一「正確」的方式，而且不可能在某個時點上被教導，即使是非常明確的目標，例如針對程式錯誤進行除錯。因為，導致程式錯誤的可能原因範圍很廣，包括簡單的打字錯誤、“thinkos”、隱藏於抽象層之下的限制，以及在抽象層中或執行的明顯錯誤。正確的除錯方式也依個案而有不同的答案。工具能發揮作用的場合多數是當必要的分析（仔細觀察）非常繁瑣，而且重複性高。我注意到 Python 的程式設計師需要的工具不多，因為被除錯程式的搜尋空間更小。

您如何重啓程式設計？

Guido：這是個非常有趣的問題。我不記得曾經刻意觀察如何進行，然而我確實常常處理它。也許我使用最多的工具是版本控制：當我回到一個專案，我會比較工作空間和儲存空間的差異，得知目前所處的狀態。

如果有機會的話，當知道即將被暫時打擾時，我會留下 XXX 的標記在未完成的程式碼中，以提醒自己特別的子任務。有時我也使用 25 年前從 Lambert Meertens 那裡學來的方式：在目前原始檔中游標的位置留下特別的標記。我使用的標記是“HIRO”，向他表示敬意；那個字是荷蘭口語表示“here”，而且這個選擇也是因為它不可能出現在未完成的程式碼中。:-)

在 Google 我們也有與 Perforce（版本控制軟體）整合的工具，在更早期的階段提供協助：當每天開始工作時，我會執行一個指令列出工作空間中每個尚未完成的專案，以提醒前一天我所進行的工作。我也保持一本日記偶爾會記錄特別不容易記得的字串（像是 shell 指令或 URL），協助我執行專案的特定任務 ── 例如，連接至主機的完整 URL，或是我正在進行的重組元件的 shell 指令。

關於設計介面或 API，您的建議為何？

Guido：即使我設計許多介面（或 API），那卻是另一個我並沒有投入許多思考時間的領域。我希望我能納入 Josh Bloch 談論設計 Java API 的演講，他所談到的多數內容適用於任何語言。

許多基本的建議，像是選擇清楚的名稱（類別使用名詞，方法使用動詞）、避免縮寫、命名的一致性，與提供一小組簡單的方法，當結合在一起時，能提供最大的彈性等。他能讓

引數列表很短：二到三個引數通常是可接受的最大值，而不致於造成順序的混淆。最糟的情況是，有幾個連續的引數，都擁有同樣的類型；偶然的互換可能會被忽略很長一段時間。

我有些個人覺得懊惱的經驗：首先，特別是動態語言，不讓方法的回傳類型依賴引數之一的值；否則，如果不知道它們之間的關聯可能會很難理解回傳了什麼 ── 也許類型決定的引數會被傳遞進來，但它來自於當閱讀程式碼時，內容不易猜測的變數。

第二，我不喜歡標記那些用來改變方法行為的引數。透過這種 API，標記總是在實際觀察的參數列表中被視為常數，而且如果 API 有獨立的方法，呼叫會更具可讀性：每個標記值都有對應的一種方法。

另外一種經常抱怨的問題，是去避免會產生混淆的 API，這攸關它們會回傳新物件或是修改在適當地方的物件；這是 Python 列表方法 `sort()` 不回傳值的原因：強調它修改在適當位置的列表。作為一個替代方案，內建的 `sorted()` 函式能回傳一個新的、分類排序的列表。

應用的程式設計師應該採用「少即是多」的哲學嗎？他們應該如何簡化使用者介面，來提供更短的學習路徑？

Guido：談到圖形化使用者介面，終於大家對於我所提出「少即是多」的觀點，有逐漸支持的趨勢。Mozilla 基金會雇用 Aza Raskin，他是 UI 設計師 Jef Raskin（原始 Macintosh UI 共同設計師）的兒子。Firefox 3 至少有一個 UI 設計的例子是關於不需要按鈕、配置、或偏好，仍能維持程式能力：智慧型的網路位址輸入列會觀察使用者輸入的內容、並與他們之前曾經瀏覽的網頁內容互相比較，而且提供有用的建議。如果使用者對於建議不予理會，它會試著詮釋所輸入的 URL 或 Google 查詢。現在的它變得很聰明，而且取代了那些可能會需要獨立按鈕或選單項目中三、四個選項來處理的功能。

這反映出 Jef 和 Aza 多年來一直談論的：鍵盤是個功能非常強大的輸入設備，讓我們透過新的方式使用它，而不是強迫使用者做每件事都需要滑鼠，而且滑鼠是所有輸入設備最慢的。鍵盤的優點在於不需要新的硬體，不像其他人提出的 Sci-Fi 解決方案，包括虛擬實境鋼盔或是眼睛移動感應器，更不用說腦波偵測儀。

當然他們還有很多工作待完成，例如，Firefox 的 Preferences dialog，擁有最糟的外觀，而且看起來感覺像是來自於 Microsoft；需要透過至少兩個層次的 tab 按鍵執行，以及許多模式對話隱藏在偏遠的地方；此外，使用者如何能記得為了將 JavaScript 關閉，必須使用 Content tab 鍵？ Cookies 是在 Privacy tab 之下，或是在安全之下？

也許 Firefox 4 能設計更聰明的功能來取代 Preferences dialog，而能讓你輸入關鍵字，像是如果我開始輸入“pass”這個字，它就會把我帶到設置密碼的地方。

關於您的語言的發明、進展，以及採用，有什麼經驗是您覺得能與目前及未來開發電腦系統的人們分享的呢？

Guido：我有一個或兩個小想法。我不是那種哲學派，因此這不是我喜歡的問題類型，也不是說我擁有已經準備好的回應；但是我在開發早期體會到的一件事是，我堅持為 Python 所做的是正確的事（但 Python 的前身 ABC 並沒有這麼做而造成損害）。一個系統應該對於使用者有擴展性。而且，一個大型的系統應該能擴充至兩層（或是更高）的層次。

自從首次釋出 Python，我接到許多關於修改語言的要求，以達到支援特別類型的使用個案目標。對於這類型的要求，我的第一個反應總是，建議他們透過撰寫 Python 程式碼的方式，達成需求，並且將它放到模組中，提供他們本身使用。這是擴充性的第一個層次 ── 如果功能的實用性足夠，最後可能會被考慮納入標準程式庫。

擴充性的第二個層次是以 C 語言（或 C++ 與其他語言）撰寫擴充模組。擴充模組能處理一些在純 Python 環境中不可行的任務（雖然純 Python 環境的能力已經逐漸提高）。我寧願增加 C 層次的 API，讓那些擴充模組能在 Python 內部資料結構中模擬，而不是改變語言本身，畢竟語言改變牽涉到維繫語言相容性、品質、語義清晰度等各方面的標準。而且，當人們藉由運用他們的解譯器，而自助改變語言的執行時，可能會產生語言的分支。這些分支會導致各種問題的產生，例如：當核心語言持續演進時，這些私自改變的維護，或是當使用者需要時，如何合併多種獨立語言分支的版本等議題。擴充模組並沒有這方面的問題；實務上，許多擴充所需的功能，在 C 語言的 API 上皆能取得，因此幾乎不需要進行任何改變，就能使特別的擴充成為可能。

另一個想法是接受你不可能第一次就把所有事情做對的事實。早期的開發階段，當你擁有少數的早期採用者，那是及早發現問題與修復錯誤的最好時機，不用顧慮向後相容性的問

題。我常常喜歡引述的一個歷史軼事，後來被證實是事實的例子是關於 Stuart Feldman 的故事；他是 Unix v7 中“Make”的原作者，當時被要求改變 Makefile 語法對於 hard tab 字符的相依性；他的回應是他同意 tab 是個問題，但是因為已經擁有一打左右的使用者，修復問題的時機已經太遲了。

隨著使用者群的成長，你需要變得更加保守，而且在某種程度上，絕對的向後相容性是必要的；然而，在某個時間點上，你已經累積許多不好的特色，而且不再適用。

處理這個問題的可行策略是 Python3.0 正在進行的：公開宣佈暫停一個特殊版本的向後相容性，藉由這個機會儘可能修復既存的問題，而且給予使用者社群許多時間來適應這個轉變。

以 Python 的個案來說，我們計畫長期支援 Python 2.6 和 3.0 兩個版本；我們也提供幾個轉變中的策略：一個尚未完善的自動 source-to-source 轉換工具，與版本 2.6 的可選擇警示相結合，提示關於 3.0 版功能使用的改變（特別是當轉換工具無法適當辨識的情況），以及選擇性的將 3.0 版特定功能移植至 2.6 版。同時（不像 Perl 6，或是在 Python 世界的 Zope 3），我們也避免讓 3.0 版成為完全的重新設計，因而能減少意外排除重要功能的風險。

過去四、五年來，我所注意到的一個趨勢是，愈來愈多的公司採用動態語言。特別是 Google，一開始使用 PHP、Ruby，到後來採用 Python；這對我來說是非常有趣的現象。我很好奇當 Tcl 和 Perl，以及後來的 Python 早期正在開發這些實用的工具時，這些人二十年前身在何處或是在做什麼。您曾看過讓這些語言更利於企業經營的期望嗎？

Guido：利於企業經營使用的選擇通常是當絕頂聰明的人們失去興趣，而擁有平庸技能的人們必須在沒有幫助的情況下設法應付。對於平庸的人們，我不確定 Python 是否更難使用。在某種意義上，你會認為因為 Python 是一種解釋性語言，所以很難造成什麼損害。從另一個角度來看，如果撰寫大型程式，卻不使用足夠的單元測試，最後你可能不知道程式寫的如何。

您曾經提到，一行 Python 程式、一行 Ruby 程式，或一行 PHP 程式所能達成的功能，使用 Java 撰寫，也許需要十行的程式。

Guido：常常是如此。我認為即使特定的功能組件是有助益的，企業的採用率仍取決於那些過度保守的經理人。想像那些負責管理十萬人公司使用的 IT 資源的經理人，那種規模的公司，也許生產汽車，或是提供保險服務等，IT 並不是主要的產品，但是他們所從事的業務都與電腦有關。而且，負責必要電腦架構的人們必須非常保守，因此，他們傾向選擇那些大品牌，像是昇陽或微軟，因為他們知道昇陽和微軟常常搞砸，但是這些公司有義務挽回他們的損失，把問題解決，即使那需要五年的時間。

傳統上，開放原始碼專案並未提供一般 CIO 同樣程度的心如止水。我並不知道這是否會改變，而且如何、何時改變。如果微軟或昇陽對於各自虛擬機器的開發突然支持 Python，企業中的程式設計師會發現他們能透過使用更先進的語言，而獲得更高的生產力。