Oral History of Guido van Rossum

Interviewed by:
Hansen Hsu

Recorded February 1, 2018
Mountain View, CA

CHM Reference number: X8483.2018

Hsu: So, we'll start at the beginning. Where and when were you born?

Van Rossum: I was born in The Hague in the Netherlands in 1956.

Hsu: And tell us about your family, your parents. What were their occupations? Did you have any siblings?

Van Rossum: I was the eldest of three. My sister was born two and a half years after me, my brother ten years later. My parents—my mom was trained as a schoolteacher. Once she got married, she quit her job, or maybe she was asked to quit her job. My dad was an architect. I would say—they were both born in the Netherlands in 1931. The most intense experience in their lives was World War II. And all through my childhood and young adulthood, I just remember always hearing anecdotes, and stories, and details, and sometimes just attitudes about what happened then, and Germany, and Germans.

Hsu: So, did they live through the occupation?

Van Rossum: They both lived through the German occupation from 1940 to 1945.

Hsu: Wow. Hmm. And I assume your family is Dutch—goes back generations.

Van Rossum: Yeah, we don't have—we just came from nowhere basically. I think my mother's family lived in the north of North Holland. Her dad was a small dairy farmer. My dad grew up in a suburb north of Amsterdam where his dad had an office job.

Hsu: What sorts of values were you raised with? Did politics or religion or ethics play an important part in your family?

Van Rossum: Politically, my parents leaned left, my mother a bit more than my dad. My mother was a pacifist, and that influenced me a lot. My dad was more like middle of the road Labor I think, or the Dutch version of that at the time. Religiously, they were unreligious. I would say they were atheists. But they raised us with respect for other people's religion. Ethically, we—yeah, that being moral, doing the right thing, was very important to them.

Hsu: Undoubtedly, that—some of those values were shaped by their experiences in the war, correct?

Van Rossum: Absolutely, yeah. Yeah, I don't know how much to add there. I think in the Netherlands from the '50s until the '70s, that was just a very common story. I mean the war was this big shadow over every aspect of society. People were divided in who was good during the war and who was bad. It was that simple.

Hsu: Meaning who had collaborated?

Van Rossum: Who had collaborated versus who was in the resistance. And of course, the majority of people did neither. But it was—it was considered as a very black and white. You were either on this side or on that side. And it drove Dutch politics.

Hsu: Can you describe your childhood?

Van Rossum: It was pretty uneventful, very middle class. My parents started out relatively poor but with relatively good education. They had both finished high school and done some follow-on education. So, my mother had trained to be a teacher. My dad—oh, he—there's one detail I forgot. My dad refused to join the army. And as a result, he spent some time in prison. And he spent some time in various work camps making up for his opinion on that—in that area.

Hsu: So, your dad was also a pacifist. You said that your mother was, and—

Van Rossum: He—yes. Yes, he—I guess, at the time, the Dutch political spectrum was divided in all these different political parties where there was the Labor in the middle and then to the left of it were ever smaller and more left-leaning parties, one of which was stated to be a pacifist party. And so—but yeah, in the general sense, my parents both were very much sick and tired of the violence of the war and took a stance.

Hsu: Yeah. So, how old were you when he went to prison?

Van Rossum: I wasn't born yet. That all happened before I was born. Then when I was born, I think he was doing—taking evening classes to become a licensed architect. And he had a day job as a draftsperson at an architectural firm. And so, gradually, we moved to a different city. We moved from—first, we moved to a different house in The Hague, with—when I was born, it was this—they rented two rooms in a kind of a shabby neighborhood. And then they moved to a flat in a newer neighborhood. Then they moved to a different city[1] where my dad got a better job at an architectural firm. By then, he had got all his papers. And he was quite good at his work. And things just gradually got better. Finally, they bought a house. Then they bought another house. And I went to university. My sister went to music school—conservatory. My little brother went to art school.

Hsu: So, you guys did pretty well eventually?

Van Rossum: We—yeah, not like extravagantly well, but we were good middle class people with no real connections or that there was like—it felt like one generation before, we came from the mud.

Hsu: What was your neighborhood like—or at least the various neighborhoods that you grew up in?

Van Rossum: Yeah, it's that—well it was pretty consistent. It was all like either row houses or four-story apartment buildings. That was the—what the neighborhood where I grew up. I think when I was in high

[1][Interviewee's note] The city was Haarlem.

school, at some point, we moved much closer downtown. And we lived in an older neighborhood. But it was still like all row houses. That’s how most neighborhoods in the Netherlands are organized.

Hsu: And could you talk about your hobbies and interests when you were a kid?

Van Rossum: Well, the first thing I remember was when I was around ten, I got an electronics kit I think for a birthday. And I was super excited about that. And all through high school, I was an electronics hobbyist. And I was not very good at soldering actually. But I was good at designing circuits. So, I went in the direction of ever more complicated digital circuits. That was my main hobby. And I had a somewhat related hobby, which was building mechanical models. But the electronics I think I enjoyed more. I didn’t really do sports, didn’t do a whole lot of outdoor stuff. Although our parents always dragged us out to various parks for walks and hikes on weekends and bike rides.

Hsu: What sorts of literature or media did you consume?

Van Rossum: We watched a moderate amount of TV. I think my parents were—initially, when we were small, they were pretty against it. And there were like two channels in the Netherlands. Eventually, I think, at some point, my dad got a really old TV that one of his colleagues was throwing out. But it still—it still worked, but it only received one channel. We read a lot of books, Dutch literature, foreign literature always in Dutch translation. That and well, we listened to music on the radio. Yeah, I developed my musical taste listening to—I don’t know. These days you would call it alternative rock or something on Dutch radio. But again, there were only a few radio channels. And a lot of the music was like horrible popular music, as I thought about it at the time. A lot of the interesting music came from the United States.

Hsu: Were you into science fiction at all?

Van Rossum: Not really. I remember reading, very occasionally reading some science fiction stories. But at the time, it was not really a thing.

Hsu: Describe your experiences at school.

Van Rossum: I was always—I was good at learning. All the material, whether it was languages, or arts, or science, or math, all was—it came easy. I did my homework very quickly. There was—I always aced all the tests. I didn’t have a lot of friends. I mean I was very nerdy myself. The only people who wanted to hang out with me were the other nerdy kids, not the other kids who were also at the top of the class somehow.

Hsu: Oh so, meaning other kids that were into nerdy interests like you but not necessarily the high achieving.

Van Rossum: There were—I remember hanging out with a few other kids who were electronics hobbyists. And but I—socially, I didn’t have a lot of friends.

Hsu: Did you have any favorite topics or teachers?

Van Rossum: I really was—I think I loved my science teachers most and especially my physics teacher. My math teacher was also very good. I also loved Dutch literature.

Hsu: Any topics that you didn't like?

Van Rossum: Sadly, I couldn't get along with history very well. So, I dropped that as a subject. And I think of all the sciences, chemistry interested me least because the way it was being taught was like memorizing large tables of apparently randomly chosen numbers. And I—that was not my skill. My skills were always like given two basic rules, I could derive everything else I needed to solve a problem from just from basics. But the basics—the way that it works in chemistry didn't work for me.

Hsu: So, then you started at university in—University of Amsterdam?

Van Rossum: That's right.

Hsu: In what year?

Van Rossum: In '74.

Hsu: And given what you just told me about being able to derive things, it sounds like you were a natural to go into mathematics.

Van Rossum: Yeah, I think for a few seconds, I played with the idea of doing something completely different like studying Dutch literature. I'm very glad I didn't do that. I would probably have been terrible at that. I didn't know much about what it was to study any particular subject at university. In the Netherlands, they make you choose a major when you enter. So, it was like you could choose to go study chemistry, or physics, or math, or there were five other scientific branches. So, I chose math because I enjoyed the topic. I think my physics teacher was a little disappointed. In the end, it turned out that the first year was mostly a repetition of the math I had already mostly understood during high school. And during high school, I had been working ahead. So, I had done like every possible math topic that was taught in my high school. But after that first year, it turned out that the real math I wasn't particularly good at. And I think there were some great teachers at that university and some super cool topics being taught. And I couldn't keep up. I remember something about a particular form of group theory. And I knew—and a few other students who were like, "Oh, you've got to go do graph theory, or group theory." And I was like—it went way too fast. And I suddenly realized I didn't have the skills to keep up with those topics. But in the meantime, starting almost from my first month I entered the math department, I had been learning to program because they had I think one of the first-year undergraduate courses was programming in Pascal.

Hsu: That was a math course?

Van Rossum: At the—it was taught by one of the professors in the math department I believe. But it was taught—I think there were like three hundred people or so.

Hsu: Oh, so it wasn't exclusively for math majors. It was for—

Van Rossum: I don't think so. Yeah, this is like really long ago and—I've only got a few memories. But I do remember that there was a programming course. And Pascal was the language. But I also remember that there were other languages that were also accessible on the mainframe. And I learned ALGOL 60 basically just by going to the library and borrowing the reference manual or just studying it there.

Hsu: And so, you mentioned that you only had sporadic access to the mainframe prior to taking that first job. So, it was only in that initial class and maybe in some other classes?

Van Rossum: The way it was done was each student was given an account. And the account had a certain value. And you would use a card punch to write your program. And you would combine that with a special card that identified your account. And then your job—you gave your deck of punch cards to the computer operator. And they put it in the computer. And it entered into some kind of queue. At some point, your job was run. And then the printout with the errors came out. And there was like a system of cubbyholes and numbers where you would get your output. And the Pascal compiler took like a fraction of a second to compile my little program and tell me what was wrong with it. So, that student account was worth probably hundreds of little batch jobs like that. So, during a semester, I could experiment quite a bit and learn a lot and—it was just completely up to me, do I try to learn Pascal, ALGOL, Fortran? And I hung out in the basement where the mainframe access was enough that I met other students who had similar interests, including more senior students who had more experience and some people who had part-time jobs there. And I think that prepared me for that job I got myself in my third year.

Hsu: Oh, okay. So, you were already hanging out in the basement of the computer center even before you got the job.

Van Rossum: That's right.

Hsu: You were hanging out with those people.

Van Rossum: Yeah.

Hsu: Like that nerdy crowd. Okay.

Van Rossum: And we were exchanging programming tips and—we were always debating, "Well, ALGOL versus Pascal," or "ALGOL or Fortran, which is better?" And all the physicists said, "Oh, Fortran is much easier." And the mathematicians amongst the programmers said, "No, ALGOL 60, it's designed with block structure in it," or simple language debates like that.

Hsu: Right. And so, you were part of the ALGOL side?

Van Rossum: I—yeah, I—

Hsu: Because you were from the math side?

Van Rossum: I naturally sided with the ALGOL people.

Hsu: Oh, that's interesting. So, I don't know exactly where Dijkstra was, but did you ever meet him around this period?

Van Rossum: I did not. Yeah, I think, by then, he had moved to Eindhoven where he worked for a different university. And I didn't actually know the names of any of the luminaries. I had—I read the like the ALGOL 60 reference manual, which was actually the—it was the original language definition actually. It was very formal, very dry. It had like, I don't know, twelve or twenty authors on the cover. But I didn't recognize any of those names and also didn't—none of them stood out. The only language where I remember the name because it was like one or two people was the Pascal manual where the manual was a much thicker book because it was actually an introduction to programming and an intr—more explanatory rather than just the language definition. And so, from that I remember Kathleen Jensen and Niklaus Wirth. And I assume that Kathleen Jensen was like the educational person, and Wirth was the language design genius. Eventually, I think in '89 I think, I attended a talk given by Wirth in Palo Alto.

Hsu: At the time, what attracted you to ALGOL instead of Pascal, which was what you had been taught first?

Van Rossum: Actually, I think I have to revise what I said before a little bit. I think maybe my first year I learned ALGOL. And my second year, somehow suddenly the departmental staff were recommending students learn Pascal instead. So, I can't say I preferred ALGOL over Pascal. I think it was more the other way around. I had never really liked the Fortran side. I mean I'd had tried it a little bit. But I—ALGOL 60 was much more the language I initially chose. But then once I discovered Pascal, I—there were a number of little details in Pascal. And maybe it had to do with how the language was expressed on punched cards. It was still mostly punched cards. That just felt like oh, this is much easier because in ALGOL 60, you have to type BEGIN END. And every time you type BEGIN, at least in the version of the compiler we had, you had to type double quotes around it. And I think on an IBM card punch, that's even a somewhat awkward position. So, in Pascal, they just did away with the quote, or strops I think they were called, around the keywords of the language. And it was a compromise because it meant you couldn't have a variable named BEGIN. And the ALGOL people were always very proud that they could have—that keywords were not impinging on the variable names, and function names, and so on. It turns out that is absolutely not a problem. And Pascal was just the first language where I encountered that little language design detail.

Hsu: So, describe your university life.

Van Rossum: Well, I think the first two years I actually lived at home. And I commuted to university by train. It was like a fifteen-minute train ride plus a, I don't know, ten-minute bus ride on one side and a fifteen-minute bike ride on the other or something. I didn't have much of a university life except that, during lunch break, I often hung out in the basement where the mainframe was or the access to the mainframe was. The actual mainframe was across town. It was a infamously ugly, I don't know, fifteen story building in a fairly historic part of Amsterdam. I think the staff of the math department didn't like their location very much. But for me, it was all I knew. And Amsterdam was a big city. And after two years, my dad said, "Oh, you're old enough to live on your own." And I moved to a dorm. And after a year, I moved to a dorm that was in the center of town. And that's where I stayed until I graduated. So, my—I don't know. I had friends, not a lot. I had some friends outside university. I had joined a club that popularizes the study of nature like ornithology, bird-watching, identifying plants, environmentalism. I had a lot of friends in that community. And every Sunday, we would go on a bike ride and spot the birds and look at the plants. And then in summer, there were—I think there were some summer camps. What I remember from university? Well, classrooms—the best part was actually, after I took that job, I realized that there was second university in Amsterdam where the computer science program was much better. I think the department was led by Andrew S. Tanenbaum. And he had a bunch of other good teachers. And he was raising his own crop of PhD students. And it turns out that the computer classes, the programming classes I got at University of Amsterdam in the math department, were a pretty haphazard collection of topics that didn't interest me. I mean, sometimes the topics interested me. I mean I remember one semester we were all learning ALGOL 68. And the teacher was super excited, but other times, it was like numerical programming, calculating what the error is after a certain set of matrix operations. And I was not interested in anything involving floating-point numbers, basically. But Tanenbaum, or his group, taught topics like operating systems, databases, networks, and languages, I believe. Yeah, Tanenbaum himself did a class where he taught like seven non-mainstream programming languages. And that was all I just soaked that up. And somehow, the two universities had an arrangement where students could just take classes at the other university for points.

Hsu: And Tanenbaum is the person who created MINIX, right?

Van Rossum: MINIX, yes. That's what he's most—

Hsu: That Linux was eventually based on.

Van Rossum: Eh.

Hsu: Or kind of—

Van Rossum: MINIX is known from a major spat between Tanenbaum and Linus Torvalds. The two were actually independent and very differently oriented descendants from UNIX. And Tanenbaum had had a particular philosophy about making everything small and elegant. And Linus was just hacking around without much particular sense for what is the right way of doing things. And that, yeah, I think there's a very, very famous, or infamous, debate that went on in the very early days of Linux. But yeah, Tanenbaum developed MINIX. That was actually—I think he did that after I had graduated.

Hsu: Oh, okay.

Van Rossum: But I ended up eventually working on a group that was collaborating with Tanenbaum's team.

Hsu: Oh, okay. So, Amoeba was—

Van Rossum: Yeah.

Hsu: Okay, was part of that. So, we'll get to that later. Did you have any important mentors and professors at university?

Van Rossum: I'd say that the—not that much. I was pretty—I was basically just studying whatever I wanted. There was one professor, van Emde Boas, who occasionally encouraged me, pushed me in certain directions. I think that's the only name I really remember.

Hsu: And was there actually a formal computer science degree? Or did you—

Van Rossum: There was not.

Hsu: So, you still had a mathematics degree?

Van Rossum: Yeah, I ended up—my degree, and I think they customized it a little bit for my situation, was in mathematics and computers science. There was essentially a math degree. Yeah, van Emde Boas helped me make sure that I graduated because at some point, that job and other programming things were much more fun than studying.

Hsu: Right. Let me go skip around a little bit. So, then let's talk a little bit more about that first job that you mentioned. So, that was in the university computer lab?

Van Rossum: It was not really a computer lab. It was actually a service organization.

Hsu: Oh, okay right.

Van Rossum: The—I think in the past, computers had been very rare. And I think there was some trauma in the attempt to develop their own computer. And at some point, the university decided that they needed a mainframe. And it turned out that mainframes were so expensive that they actually—the two universities in Amsterdam, plus a place that—a research lab that was called the Mathematical Center, which did a lot of specific computer science and related math research, the three of them got together and bought a Control Data mainframe. I'm sure it took them years to actually decide which brand to buy, especially to decide to not buy IBM. I think the Control Data mainframe excelled in numerical computations. It had superior floating-point performance. Anyway so, the—there was a separate entity that ran the mainframe and was supposed to give everyone equal access and keep track of bookkeeping, making sure that everybody paid for the time they used on the computer. And that was the entity that employed me.

Hsu: Oh, okay. Yeah, the service center.

Van Rossum: Yeah, it was—at the time, it was called SARA, Stichting Academisch Rekencentrum Amsterdam. It's like the foundation for the academical compute center in Amsterdam. And their physical location, initially, was with the Free University of Amsterdam, which was like where Tanenbaum was also teaching.

Hsu: So, it says that you worked on the operating systems that were on those mainframes. You helped to develop them?

Van Rossum: No, no, the—it was pretty lowly—there was things like installing new versions.

Hsu: Oh, so like sysadmin type stuff.

Van Rossum: Well, yeah. So, the operating systems group I think was responsible for most of the sysadmin work. The actual—the operating system was produced by Control Data. And you paid for the hardware, and all the software was free. So, the compilers were also produced by Control Data, CDC. What we did do was write additional software. For example, there was accounting software that was customized. And there were integrations, and we were responsible for installing third-party software and keeping the compilers and the operating system up to date. And I remember there was like this row of shelves, it was from here to the back wall, of computer printout that was I think refreshed like weekly or monthly that was the complete source code for the operating system. So, that if something went wrong, you could actually look up what had happened. My role was to write small utility tools and help with all tasks. And eventually, that included implementing a simplified programming language that supported control flow in the very arcane job control language that CDC used.

Hsu: And throughout this period, you were exposed to hacker culture?

Van Rossum: A bit. The—not at all to the extent that it existed in places like MIT at the time. I was completely unaware of any of that. But there was—there were a few other people, including a few other part-time employed students at that place who were not just doing it for work, but also passionate about the topic. We were exchanging stories and learning together, and there was a group of passionate users and sometimes there was an adversarial relationship, they were like users who were trying to break through the accounting system so they could get free or unlimited access of the software and the system people then had to plug the holes. But at some point, the group of people was small enough and the computer wasn't really on any kind of network. There was no truly anonymous access, so at some point, I remember the head of the accounting department just talking to one of the persistent little hackers, who was one of the very, very bright students, who was just thought it was great fun to break into that system and crash it. And the head of the accounting department was tired of that and just had the—I think he threatened with physical violence. He didn't apply any, but he said, "If you keep doing this, if you do that particular trick again, I will hurt you." It's that—I wasn't there, but that's the story I heard at the time.

Hsu: So then did you—you decided to continue with graduate studies?

Van Rossum: Not really. I—at least at the time, university in the Netherlands was organized a little different than it is these days in US. I didn't know how it was then in US or how it is now in Holland, but you picked your topic and after a few years you had some kind of bachelor's degree, and as long as you showed up for classes, that's what you've got. Then the natural next step was something called "doctorandus" which means "He who is to become a doctor," which is very much not a doctor, but it's a degree that's equivalent to a master's degree. And you have to write a small thesis and do half a year of research, and I eventually did that. And at that point, I was truly done with academic learning and I had no interest in a PhD position.

Hsu: And that was in 1982 that you finished?

Van Rossum: That was in '82, yeah.

Hsu: So then you took your first permanent job was at CWI?

Van Rossum: Correct, yeah. And this had previously been the Mathematical Center. I don't remember. It might still have been called the Mathematical Center when I first joined, or they might have been in the middle of changing the name to CWI, and I think that nobody at the time really understood why they were changing the name. Because it was a huge problem, because the Mathematical Center, at least in European computer science circles, had a very good reputation and nobody knew what CWI was. Plus it's a Dutch abbreviation and it doesn't—the English equivalent doesn't have the same letters.

Hsu: Right. So CWI stands for?

Van Rossum: It's Center for Mathematics and Computer Science. And so that is Centrum voor Wiskunde en Informatica. So computer sciences is called informatics [in the Netherlands].

Hsu: Right. So the letters are different when you translate it.

Van Rossum: Yeah.

Hsu: And this job was after you finished that master's degree, or was it concurrent?

Van Rossum: It was after.

Hsu: That was after?

Van Rossum: Yeah. I'd—in fact, I think I had to—there was one final exam I had to take in a subject of physics that I wasn't very good at, and I failed the test the first time, and I was almost in tears because I couldn't start my job that month. I had to call them and say, "Oh, I'm going to start another month later or so." And it turned out that nobody cared, it was fine. But for me, it was a big drama.

Hsu: Well, what was it about this job that attracted you?

Van Rossum: I think it was primarily Lambert Meertens because I had met him in previously in the—I didn't know exactly. I probably knew him two or three years before he employed me, and the way we met was, oh, yes, here's this pacifism coming in again. I was a member of the Dutch Pacifist Society. Sorry, political party, and at some point, I volunteered to help the party headquarters to move from the Stone Age to the Middle Ages in terms of information technology. They were buying their first computer and they had to transfer all their membership database to that computer, and they were—they wanted to do everything themselves because they were super-afraid that their information about who was a member of the party would fall in the wrong hands and in the '70s, that was a serious concern. Anyway, so the party formed a committee that was to decide on the hardware and the software and then they sought volunteers to help them with the programming, and I thought, "Oh, I know programming, and even if I have to learn a different language—" it turned out the computer they bought could only be programmed in COBOL and I had never seen a line of COBOL in my life, but again I could learn any programming language from the reference manual, so I learned how to do that, and that helped out. And it turned out that Lambert was one of the party bigshots at the time, besides his career as a computer scientist at the Mathematical Center, he was also—I think he ran for a seat in the Dutch Parliament once and spent a summer holding political stump speeches throughout the country. I guess he wasn't elected. But so that's where we met, and he recognized my skills and somehow we got to talk and at some point, after that project had already started becoming a minor disaster, and I think it was eventually finished without my further help. They changed directions a bit, but by then Lambert and I knew each other and at some point he said, "Oh, you're graduating soon, right? Hmm, I think I might have a job for you." And that was very welcome.

Hsu: Let me take a slight step back. You mentioned being a member of the pacifist party. So what was—how affected were you by the Cold War and other things that were going on at the time politically?

Van Rossum: Well, I remembered that politics was a big thing, and there was—I mean, in the Netherlands, I felt pretty safe from actual Russian bombs, so I was never really concerned about my safety, but on the other hand, it felt like we were a lot closer to Russia than Americans are. I think during the '80s at some point there was a big movement in the Netherlands to kick out American weapons that were posted in the Netherlands and throughout all Western European countries as long as they were part of NATO. Because that—it wasn't always completely crystal clear that the Russians were the bad guys. It sometimes felt more like, "Well, there are two sides, and they're clearly disagreeing with each other, but it's not clear who's right." And so I never felt comfortable with the communist system, but I did feel that a socialist system was better than capitalism. And that, yeah, that probably colored some of my views on life in general.

Hsu: So then you started this job and was your first position there to work on the ABC project?

Van Rossum: Yes. So Lambert had basically just spun up the ABC project. I think he had pulled some strings with management and gotten funding so he could hire or requisition programmers. I think some of the team members had already been programmers at CWI before, and other people were hired fresh. The interesting thing was that ABC, the language, was basically designed when I joined the team. There was also a project lead who was hired fresh, a guy named Steven Pemberton. He was British. I think he had graduated on some kind of Pascal compiler and somehow that had impressed [Lambert]. I mean, I think he had sought out the job. He didn't get the job through connections. He was a very interesting guy. He was the project lead. I was one of the younger programmers. There were two other programmers, and we had, like, Lambert and another guy named Leo Geurts. I think Lambert and Leo had designed the language together, but I think Lambert was the major push. Later, I found out that Lambert had also spent a lot of time with ALGOL 68, because he's the—I think he ended up being on the cover page of the ALGOL 68 revised report as the editor, even though they—I think he was a very junior person when that language was actually being designed by all the big shots in the European language design community. One of the biggest big shots was van Wijngaarden, who I think until maybe a year before I joined, had actually been the director of the Mathematical Center[2].

Hsu: Okay, wow.

Van Rossum: That's how it—I don't remember the exact chronology. I know that he still occasionally taught a class at the University of Amsterdam, and I caught a semester of that, and I was all in awe because here was this guy who had invented the grammar description mechanism used by ALGOL 68.

Hsu: And so yeah, you joined CWI in '83. That's correct, right?

Van Rossum: I joined in December '82, I believe.

Hsu: Oh, December '82. Pretty close.

Van Rossum: Yeah. Yeah. And so yeah, here we were with a freshly-recruited group of programmers, and we were to implement ABC, and it turned out that Lambert, on his own, had already written what was called a prototype implementation, which was like this huge, big C program that took a few shortcuts and didn't always follow the exact language specification, but it could run, basically, almost any ABC program, but there were details of the language, it was supposed to have arbitrary precision arithmetic, and that wasn't implemented. But we—I think Steven settled on a strategy of re-writing the prototype one component at a time, so we would always have a working interpreter. But he rebuilt all the fundamental data types. We rebuilt the parser. We rebuilt the interpreter. Every part was eventually completely different from the original prototype. The prototype was definitely more than just inspiration. It was the starting point for the implementation. Then, we started adding things. I remember working on a

[2] At the time Adriaan van Wijngaarden had been director of the Mathematical Center for a decade [interviewee's note].

programming environment for ABC, and I think we were not very inspired for that, or we didn't have the right ideas. We were trying to build a syntax-directed editor, which, and this was, like, the early '80s. There was some very advanced research where people were building meta-compilers for editors where you would somehow feed the grammar of the language into a complicated piece of machinery, and it would generate an editor for you or maybe it would just be an editor or that knew about the syntax of the language you were trying to edit. But those things had not been tried out on real users very much. And some of the ideas didn't work, or maybe we didn't understand some of the ideas. We didn't have great, great success with doing that, but we did—we built something, and I remember that building editors was one of my passions in addition to languages. And so I had—I was familiar with a number of different editors. There was like vi and Emacs and a few other older ones, and computers—there was just enough control over where the text would appear on the screen that we could make it a visual editor, and I had some very specific opinions on how that should integrate with the rest of the world. I was probably wrong, but there was one of the things I tried to design myself.

Hsu: What was the purpose of ABC? Why design this new language?

Van Rossum: You should really interview Lambert. He will have a much more detailed story. The way I understood it was that Lambert and Leo had spent many years standing in front of a classroom teaching scientists how to program using ALGOL 60. And through those years of teaching, they had collected large lists of gripes about how arbitrary the design of programming languages in general were and arbitrary limitations like the hardware that—ALGOL 60 was designed in the time where every computer had its own unique hardware, almost. Or there were very few similar computers. And so there were all these vague abstractions like, "Well, there's an integer, but we can't tell how many bits are in the integer." But if you run it on a particular computer, sure enough, if the number you're trying to represent doesn't fit in the 27 bits that that particular computer uses as its word size, then you just get an arbitrary programming error, because the compiler doesn't generate any range checks, and if you do X plus Y and the result doesn't fit, it just chops off some bits on the top and you get out a very large negative number, maybe. And things like running out of space, recursion or just function call depth limits, everything that was a limit was a problem for their programmers because their programmers were scientists. They had—they weren't professional programmers. They had scientific data processing problems that were suitable to process in a computer. And so Lambert and Leo set out to design a language that would be useful for researchers, lab assistants, professional users of computers who were not also professional programmers. Because if you're trained to be a programmer, then you know all those limitations and you know how to write code that doesn't suffer from overflow or whatever it is. But if you're a scientist and you're quickly once a week or once a month you have a programming project, it can be very frustrating to have to learn all that arcane stuff, and you forget it, and then the next time you have to read the manual again, or you have to debug the same program again because you sent it different data and it crashes in a different way. And so they wanted a language that was easy to learn, easy to use, didn't have any limitations. It would always give you a nice error message when the data didn't fit or when the numbers didn't fit. And they, I don't know, I'm sure they were inspired by other languages[3] that I'd never seen, and some of their design was interactive, where they had discussions and conversations and blackboard

[3] [Interviewee's note] Also probably by the Algol-68's debacle.

sessions with other well-known language designers and they came up with certain ideas for data types that would cover most of the needs of data types in traditional programs.

Hsu: So the intention was to make a language easy for scientists who are nonprogrammers?

Van Rossum: Not necessarily just scientists, but scientists were definitely a big part of the intended audience. I don't necessarily know that they got a lot of scientists to use ABC because after we had built all of ABC, we were fairly naïve in how do we get people to use it? And we basically failed. There were like—I remember there was one mathematician who worked at CWI who was super into ABC because he did things with primality tests and early crypto, and he—ABC had this almost hidden feature that you could use it to do calculations with arbitrarily large numbers, including fractions, and it would all be precise. And so for numerical algebra, that's great, because if you have a number that's 95 digits that's computed in a certain way, you can actually just represent that without any work. In any other programming language, you would have to use a “bignum” library, which means that suddenly it's a tedious task of using APIs. In ABC, that was just the built-in one number data type supported all that. But we didn't get a lot of people write a lot of real coding in ABC.

Hsu: How was the group trying to get people to use it?

Van Rossum: Well, so there was no Internet, and there was a little bit of email. I think there was a bit of USENET, but I remember, and I think this was in '86, traveling to the United States on my personal vacation with a nine track tape in my luggage, and the addresses and phone numbers of, like, three or four people I had met through email who were interested in trying out ABC on their UNIX system, and it was pretty limited because you also had to—you had to have a certain type of UNIX computer, and it had to be a certain type of hardware and a certain version of the compiler and the operating system. Otherwise, you had no chance of running it. I mean, in theory it was all written in portable C, but in practice, I don't think it was all that portable, or at least it would require—we didn't have a variety of hardware to try it out ourselves. So for every new C compiler, again hardware vendors built their own C compilers. Every new C compiler probably had some bug that—this was—a very large C program would probably tickle one compiler bug or a runtime issue or just some unanticipated hardware feature where things wouldn't work.

Hsu: Right. So portability was a big issue?

Van Rossum: Portability was an issue, and also just getting the word out, because the way to get the word out for software was still, yeah, unclear. You can publish a paper, but it takes many months to publish and then many more months for people to maybe see it. So I don't know, I wasn't in charge of that, but I know it was a real struggle for the team, and at some point there was—oh yeah, there was, I think Lambert and Leo had written a book, and they had found a publisher who wanted to publish it, and all the hopes were on that, because in the back of the book, there would be some instructions for how to get the implementation, how to get the source code. And it was all free because it was just research work. But the publisher was dragging their feet and the editor was taking a very long time to respond to our letters or emails. I don't know. And then suddenly word came that the publisher had been acquired by a different publisher and all the contracts that hadn't been signed yet would never be signed. Or at least everything would be revisited and the ABC book wasn't published. So that was a big setback. And at some point, I don't know, somewhere between '87 and '88, I believe CWI management decided to kill the project.

Hsu: And you mentioned UNIX. This ABC was all being done on UNIX machines?

Van Rossum: Yeah. I think CWI had a small fleet of large UNIX machines. I remember we had a VAX and then we had maybe another smaller VAX. And then we had, I think, a Harris Computer, but everything ran UNIX. Now CWI was also one of the first places that would eventually be connected to the Internet. But that hadn't quite happened yet. So we were connected to USENET and that was like a lifeline because there were, like, things like newsgroups and so once a day we would get a fresh batch of messages and responses, and there was some email, too, but it was all very primitive.

Hsu: So then after that, you moved onto the Amoeba project?

Van Rossum: Yeah. After ABC was cancelled, of course everyone who worked for that project had to find a new home. I think one of the programmers found a job outside CWI and I joined the Amoeba project.

Hsu: Which was still part of CWI?

Van Rossum: Well, so Amoeba was Tanenbaum's research project, so that was at the Free University. But his first successful PhD student moved to CWI to set up his own research group. That was Sape Mullender. And so he basically at that point, the research for Amoeba was divided in two halves, and some of the more pragmatic work, I think, was being done at CWI.

Hsu: And what was your part of that project?

Van Rossum: I remembered that we tried to productionize it in a sense. And not in a very commercial sense, but whereas in Tannenbaum's group, they were, like, building a microkernel and there was one student and they were building a file system, that was another student, and they were porting compilers. The Free University had its own set of compilers. That was maybe another student. And at CWI, we were doing things like—oh, we had, like, 10 or 12 MicroVaxes and we hooked them all together with ethernet. Oh yeah, I think some of the work we did was also porting Amoeba to ethernet because the original design was based on a different networking technology. So I think we had to write ethernet—oh, no, I think it was—it's not so important. I think it was all based on ethernet, but we ported it to a different set of networking protocols on top of ethernet. I think that we made it compatible with modern Internet style networking protocols, whereas amoeba originally had its own—if you just plug a bunch of amoeba machines together, they can talk to each other, but they couldn't talk to anything else. CWI had a whole ethernet system full of UNIX systems. Well, I don't know, there were small workstations in some places and central computers in other places, but there was ethernet in the building and we were trying to make it so that the Amoeba systems could talk to the UNIX system, and then it turned out we needed a suite of applications because you can't just connect the networks together and say, "Well, let's do some networking." You have to have, like, applications. Well, yeah, let's write an email application. That's the kind of thing that I worked on. I remember at some point we had to write a login program, because there had not been—I mean, there was all this theory about authentication, but there was not a system that actually managed usernames and passwords. So we had fun with a lot of programming tasks like that. And that's where the idea for Python came up where I thought, "Well, I'm writing all of this code in C and it's getting kind of tedious, like writing a tape backup program or some other thing that traverses the file system and looks for all your files and checks whether they've changed, and I felt that that was a lot of work to write in C. And I was very good at C at that point. I had spent probably 10, 15 years coding in C. But it still felt like there were lots of bugs and it just was slow going, and I was thinking, "If we had an ABC implementation here, I would just write that whole login program in 15 minutes and then I would move on to the account management program or something and in C it takes me a week each." And so I thought, "Well—" I somehow started thinking about coming up with a way to use some of ABC's features in the Amoeba environment. Yeah, and I'm sure I'm skipping a whole bunch of steps. I had experiment—I mean, I was one of the people who was very sad that ABC was canceled, that it was not successful, also, and I had a released some other pieces, some small pieces of software through the USENET channels that, by '88, '89, were a little better than in '86 when we were struggling with ABC. And yeah, the story as I always tell it is that at some point I think Sape Mullender went for sabbatical to the US West Coast to learn things. I think he spent a year at Digital Equipment's System Research Center under, I think, Bob Taylor, who had just split off from Xerox PARC. So Sape was, I mean, we were still in touch through email, but he wasn't keeping a very close watch on what the team was doing on a day-to-day basis. And somehow I thought that—I guess it was better to ask forgiveness than to ask for permission, and there was no one to ask permission anyway, and I was doing all the scheduled work sufficiently fast, yet I had extra time and I also still didn't have a whole very complicated social life. So all my spare time, I would also work on this new language that I was designing. But my goal was really to not make this another project the size of ABC, but to spend a few months doing this on my own and then make it part of the Amoeba system programming toolkit. And from then on, everything would be better. That was my idea. I do remember that, I forget when, probably December of 1990 when Python was a few months old[4], really, we did a project where we ported Python to Amoeba and we made we made a Python extension module that supported all the Amoeba system calls and—or important library calls. So you could run a portable Python program on your Amoeba system but you could also write an Amoeba-specific program that was interacting directly with the Amoeba file system, which is—has a very different philosophy than the Unix file system and various things like that, and you could talk directly to the networking APIs on Amoeba and as far as I recall we did make all that work.

Hsu: So you first started on this project on your own in ’89, over the Christmas break?

Van Rossum: Yeah, Christmas break ’89. I think the first thing I wrote was actually the lexical analyzer. I must have had a design in my head because I just—one day I said, “Okay, now I’m ready to start coding,” and the first thing—I was coding it bottom up—the first thing I needed was a lexical analyzer because that’s like the front end of the parser and so I had some—I had seen a whole bunch of lexical analyzers

[4] [Interviewee’s note] At that point Python was already a year old.

and I was aware of tools like Lex and Yacc and I had mixed feelings about Yacc. In the end, I decided not to use it, but I knew for sure that Lex was a piece of crap...

Hsu: <laughs>

Van Rossum: ...because if your input was too long it would always just crash on you because the—Lex was a code generator but the generated C code was very naïve and it wouldn't check for invalid input very well. So you would write a nice Lex rule for oh, this is what a comment looks like and if your comment was 500 characters long it would just bomb. So I wrote my—I decided oh, I know how to write lexical analyzers. I'll write my own and I think the next thing I did was I had my own idea about how to do a compiler generator or parser generator and so I built that out and then I designed the simple Python grammar and I had the lexer already. So I hooked it all up, and somehow I connected it to an interpreter and library functions and that's—I think the first three or four months of 1990 were spent with that.

Hsu: And so the name Python famously comes from "Monty Python's Flying Circus," of which you are a fan?

Van Rossum: That's correct.

Hsu: <laughs>

Van Rossum: Yeah, and still am.

Hsu: How did you first get into Monty Python?

Van Rossum: I think one of the smaller Dutch TV channels aired a number of episodes, and it was just insanely fun.

Hsu: Was it translated, or was it in English?

Van Rossum: Subtitles. Yeah, so you always—Dutch TV is all subtitled except when it's for little children. So yeah, we would always watch the shows in English and—by then, maybe my English was okay, but I still needed the subtitles.

Hsu: And you're also known as a fan of "The Hitchhiker's Guide to the Galaxy"?

Van Rossum: Yeah, that's correct. Yeah, that's—I think there was one of the other guys on the Amoeba project who first introduced me to that. Yeah, and then we were just like—he had one copy of the book and he would read passages from it and then once he was finished we <chuckles> were all fighting over who got to read it next.

Hsu: <laughs> So how did you arrive at Python being the name of your language?

Van Rossum: I didn't think much about it. I knew I needed a name. I—oh yeah, I remember ABC had originally been named B and choosing the proper name for ABC was—had been a very traumatic experience because they're—like people do with branding exercises. There was a list of a thousand potential names and they were selected and everybody got to vote and nobody could agree and there was always a good reason why a name couldn't be used and then eventually they settled on ABC and I thought it was just the blandest name ever and then I traveled to the United States and I noticed that every no-name business calls itself ABC, the American Business Consulting Group or whatever. <laughs> So I felt vindicated <chuckles> in my opinion that ABC was a bad name, and so I don't know. I was looking for something that was maybe attention-getting, also not too long, not too short. Naming of languages has always been a fun game. I kind of liked the idea of naming languages after heroes, like Pascal was named after a French philosopher, but—and I was aware of a language named Eiffel, named after the inventor of the Eiffel Tower, and I thought that I wanted to be more modern and I thought I wanted it to be a cultural reference but not in the somewhat dry academic culture where everything is always named after Greek—minor Greek gods or other mythological figures. I mean, at DEC SRC, everything was named after mythological figures, and it was all Olympus and Hera and Hippocrates and—I don't know. I'm making those up but that's—that was another thing that I had seen enough of and I thought, well, popular culture has not been a great source of software names yet[5] and we—there were a few of us at the Amoeba—on the Amoeba team who had similar ideas but I mean similar sense of humor. We watched the same crazy shows and we—I think we had named some other piece of software or system component after a TV show that we liked at the time. I don't even remember <laughs> what that was but—so suddenly it occurred to me that Python—it checked all the boxes. It was six letters long. It was a familiar word but not so—not a common word. It presents a certain attitude and—that I thought okay, let's go with that, and I never looked back.

Hsu: But why Python rather than Monty?

Van Rossum: That's a good question. I don't think I knew what Monty meant. I still don't, so—and Monty Python in full was too long and I guess I wanted it to be somewhat unobvious where the name came from although I've always resisted the—the snake thing.

Hsu: <laughs> Right. So then you've said that you <clears throat> developed it partly because you were writing a lot of this code for Amoeba. So you said that it was supposed to be a bridge between Unix shell scripting and C was part of the way...

Van Rossum: That was my original positioning, more of a real language than shell scripts. Shell scripting developed from these sort of Job Control Languages where originally it wasn't a language. It was just a sequence of single commands that were executed sequentially just one after another and it was all just the name of the program and then the name of the files you wanted to send to the program or something like that and that was how I invoked the Pascal compiler when I was still using punch cards, Pascal input

[5] [Interviewee's note] Around these times there was some database language named *Linda* and when I met someone who was involved with its naming they explained that is was named after "the other Lovelace," i.e. a famous porn actress.

or something and then you didn't even have to say input. And Job Control Languages and so Unix shell also evolved from the specific Job Control Language for the Unix system where most of the time you just type the command and if you can do something by typing three commands, you can also put those three commands in a file and say, "Now I have a script," and then the next thing you want to do is you want to have a script but you want to write a slightly more complicated script that, say, checks if the output already exists and then does something else than when the output doesn't exist yet or when a certain input doesn't exist it skips a certain step or just prints an error message and you want—so you want to have some logic and so I imagine that the original Unix shell started with an if-statement. And [an] if statement had to somehow fit into the existing structure of "it's just a sequence of commands" and you can still see that in the syntax of the modern shell, the Bourne shell—sorry, the—sorry, Bourne-again shell, Bash, which came from the Bourne shell which was a major improvement over the original Unix V6 shell and who knows what they had before. So there was programming in shell scripts but there was—the syntax was sort of ugly because it had to fit in the—it always had to be backward compatible and so that's why you end up with the—I'm sure that's how they invented $ for variables because $ was not a valid filename originally or maybe it was a very uncommon filename. So [for] certain tasks, it's great to be able to do it in a shell script, but then suddenly you want to do something 10 times. How do you do something 10 times? Well, if you're writing a real programming language then you do "for i from 1 to 10" or something. There is always a nice notation for that. In shell scripting, there's no way to do something 10 times. You can do it for every line in a file or for every argument on your argument list, but you can't do it 10 times. At least, just adding two numbers in shell is very cumbersome and it's a little bit better in Bash but you still end up invoking a lot of funky characters on the keyboard just because all the regular characters were already taken and then on the other side you have C, which is a programming language. You can do anything. It can control the hardware very closely. You can—at least in the '80s you could sort of—you had to decide whether to put a particular variable in a register or not. It was very close control over memory and everything in the machine. That was great, if you needed that, you could make things super-efficient, but there was also—it could be a lot of work if you had to read all the lines of a file and then sort them. You had to invent your own linked list data structure to hold onto all those lines and oh, the—I knew how to do that but I thought it was too much work and there wasn't a lot of standard libraries. So I wanted Python to be a compromise between the super-convenience of shell scripting, which becomes harder the more programming you need, and the super-control of C, which becomes tedious the less control you actually need, and in Python you have more control than in shell script because every line is not a new process, but you have less control than in C because you don't have to worry about where the bytes go in memory and I had some ideas about what is good syntax versus bad syntax and it goes all the way back to the debate between ALGOL 60 versus Fortran at the start of my career <clears throat> and so the—I think Lambert had helped me think about what is elegant use of mechanisms in programming language design and other languages I had learned and I had read about. As I mentioned, I was in a habit of just reading, learning a new language from the language specification and usually you get a fair bit of the philosophy of the language designers or the design committee or however it's done and so I—I had developed a fair amount of opinion on that and I thought, well, there's a whole bunch of stuff that I could borrow from C. There's a whole bunch of stuff I can borrow from ABC, and then there's things where I just have to invent my own syntax or mechanisms.

Hsu: Right. So how many—what sort of things were borrowed from ABC and—but also how did Python differ from ABC?

Van Rossum: So yeah, the interesting thing was that over the few years when I worked for the Amoeba project, I had been processing the ABC's failure and developed my own opinion about that and so, in my view, there were a number of things in ABC that were great and I wanted to keep those. Examples are actually use of indentation for statement grouping, data structures like lists and dictionaries, the idea of immutable strings and numbers, the idea that you don't have to declare the types of your variables because the language can figure it out from how you use them. Then again, there were a whole bunch of things in ABC that I didn't think were very successful. In ABC, all the keywords are spelled in uppercase. This is still a relic from that original ALGOL-versus-Pascal idea where in ALGOL the language-reserved words, the language-special words have to be spelled in a different way to—so that the parser is—can be stupid and can always tell this is a keyword of the language and this is just a variable or function name and so in ABC that's—that debate was still reflected and they had chosen what I thought was the wrong outcome which was that statements and language keywords were all uppercase and functions and expressions and variables were all lowercase and I hated the whole look of all these uppercase keywords in the language. It felt like shouting, and it also felt old-fashioned because the typical representation of what you see on punch cards is all uppercase and Unix—in Unix, by default, everything is lowercase. So there was some Unix-versus-mainframes that I also probably put into that. I didn't like some details of the—especially the list data structure in ABC. For some reason, ABC keeps its list in sorted order always, which is very handy when you're talking about, say, a set of colors, not so great when you're talking about the lines in a file when you're writing, say, an editor. So I wanted—I was interested in slightly different things than ABC's target audience. I was not a scientist. I was a programmer, and I wanted to write tools for other programmers in Python. So the—another big thing where I thought ABC had failed was that ABC had this philosophy that there is only one system. It's ABC. It's the language. It's the environment. It's your persistent storage, your file system, so to speak. It's your editor. It's everything, and again, Unix counters that with a philosophy of small tools that do one thing well and work very well with other small tools that do different things well. My big frustration with ABC was, for example, that in practice, a computer that was running ABC was also maybe the—maybe there was a word processor or a spreadsheet, and it was impossible to open a file in ABC and just read the data from the spreadsheet and do something with it. So you needed a separate conversion tool between spreadsheet data and ABC's internal data structures and I thought in Python if that spreadsheet is stored on a file there is nothing that should stop you from just opening that file and reading it in and maybe you have to decode the spreadsheet's internal data format but that's a separate problem. You can code that up but if you can't even open a file you have no access to the spreadsheet and so I—ABC was this single monolithic implementation that did everything and if ABC couldn't do it for you, then you were stuck. You were hosed and in Python I wanted the language to be extensible and I wanted [it] to interact well with an environment and not necessarily only a Unix environment.

Hsu: Yeah, so we're talking about extensibility, the—how you wanted to make Python more extensible than ABC.

Van Rossum: Yeah, so another very frustrating experience we had one summer with ABC was we had a couple of interns on the project who were interested in building a graphing backend for ABC. Basically, you would process numerical data in ABC and then you would—you would want to, say, turn that into a plot or a chart or something visual and originally ABC had been designed completely as everything was text because that was the only thing that computers could do until, <clears throat> I guess, the late '70s or so and so they—again, they found that ABC actually had a big limitation, which was that you couldn't easily add a library with new functionality to ABC, and there were a couple of things that got in the way. One was that there was no—the language implementation didn't have an import mechanism. You would have to basically add new built-in functions or add new built-in statements or commands in order to add new library functionality. There was a concept of namespaces but namespaces always—I think you could only use one namespace at a time and a namespace was just a collection of ABC code. So you could have a bunch of functions, but there was no further structuring. So these poor students really—I think they came up with some kind of hack but it was a really pretty sad experience and it turned out even if you did want to add a new statement or a new built-in function to the language—the implementation also wasn't structured to make it easy to find, oh, here is the piece of code you have to change to add the new built-in function. It was all—it was spread over the parser and none of that made much sense to me because I was familiar with C itself, which has a great way of linking separately compiled pieces of code together, and most other languages I was aware of had some mechanism for that, too, and I suppose we could have added something to ABC but it was certainly not part of the original design and so ABC's implementation recognized a number of data types, like there were—let's see, what was there? There were numbers. There were strings. There were tuples. There were lists, and there were dictionaries[6]. That was the data, and all the code would just—if it had to do something with data, it would just check, well, if it's a number we can do it. If it's a string we don't do it. Yeah, you—everything would only apply to one or two data types, like there were some operations that could be applied both to lists and to dictionaries or tables, I think they were called, but it was like everything was hardcoded in the implementation and there was some code reuse inside the implementation because lists and dictionaries were actually sharing much of their implementation but it would've been impossible to add a third data type that would also share that implementation. The code base just wasn't prepared for extensions along those lines and so I wanted to do things very differently in Python. I wanted to have a module system. I—my summer—I think it was the summer of '89[7] that I spent as an intern at DEC SRC or a visitor. Yeah, I was—technically, I was an intern. I spoke a lot to the people who were designing or had just finished a design of Modula-3 and the—a similar group who were working on the Modula-2+, which was like Modula-3's more practical predecessor, and that language obviously had a module system and I didn't know where else I saw these things but it was very clear to me that Python needed to have extensibility and that modules were a way to extend it both with Python code but also with things that were not implemented in Python but could still be used from Python and that's where the idea of built-in or extension modules—I think originally there were [what were] termed built-in modules because the original implementation had all the concepts right but it wasn't that easy to hook it up to third-party code. You had—you basically still had to recompile, but you—yeah, there was one piece of source code that just linked the name of a built-in module with a function that initialized that module and returned all the

[6] [Interviewee's note] Which were called *tables*.
[7] [Interviewee's note] It could have been Summer 1988.

necessary data structures. So if you had—if you wanted to add a whole new piece of functionality to Python even in that very early version, you would write the new functionality and you would hook it up to the rest of Python with a single line in that config file, I think that the grandchild of that config file still exists in the current implementation. So I also somehow picked up the idea that extensibility is important. It's even better to have multiple levels of extensibility, and that's where on the one hand, writing extension modules in C and on the other hand, adding new code to the standard library or adding new code to a user's own workspace was—were all things I cared about.

Hsu: And so there's a number of other critical design choices that you made. First of all, one of the big things is you designed Python for programmer productivity, and so I guess a lot of other features fall into that; so interactivity, being an interpreted language, more—terser, more expressive code, readability, what you mentioned earlier, use of indentation for statement grouping. What was your overall philosophy in these—designing Python, these choices that you made?

Van Rossum: It's interesting. Pretty much everything in the list you just mentioned came from ABC. ABC was interactive. In fact, it was only interactive. ABC barely had a batch or scripting mode. ABC had—I don't know if I would use the term "terse," but it had this compact way of expressing quite complicated concepts. Python's interactive command line prompt is even the same as ABC's because I figured there were—ABC was not sufficiently popular that I had to distinguish myself by changing the prompt. There was no worry about confusion. Obviously, I couldn't make the prompt be similar to the prompt in some other language like the shell prompt. I was careful to make it look different. I forget. I think everything on your list, actually the philosophy came from ABC.

Hsu: One of the things that I think is—Python has—one of the big uses of Python has been in education and a lot of introductory classes in programming are taught in Python and you said before that that came from ABC, as well. That wasn't—

Van Rossum: That's correct.

Hsu: —your design, but…

Van Rossum: My original positioning of Python was actually not as an educational language. For ABC, the authors always objected when it was called an educational language because they wanted to have these dual properties almost of easy to learn and easy to use. I had neither of those directly in my goals, but one of my goals was [to] borrow all the good stuff from ABC and with all the other things I stole from ABC it turned out that it was quite good to teach in. I hadn't anticipated that, and I remember when I moved to northern Virginia I got to meet a whole bunch of new people and my colleagues were eager to introduce me to other people they knew and so I met this guy and he said, "Yeah, I taught my kid Python," and the kid was like, I don't know, 9 or 11 or so and I was wondering <laughs> if that was a good idea but it—clearly the guy was very proud and it was also pretty—had been very easy for the kid to pick up Python and then the story he told me was, "Oh yeah, but we're using Windows," and he couldn't figure out how to exit the Python interpreter but he had discovered some crazy thing that would crash the interpreter. There was a bug. He had basically discovered a bug, and whenever he wanted to leave the interpreter, he would just type the line of code that would trigger the bug <laughs> but that was the—I think, the first time I realized that Python was also going to be effective for education and over the years people have started using it more and I've always said, "Oh, that's great," that was one of ABC's original goals, to be easy to learn.

Hsu: <clears throat> I want to talk about the indentation feature again because it's kind of a widely known thing. You feel very strongly about this feature, that it's very beneficial. You stated this in a podcast.

Van Rossum: That's probably a little more strongly than I would say it now. It's—I don't think it's a bad idea. It's an idea, however, that no other successful language has copied with that—I think the only other language that I'm aware of nowadays that uses indentation is Haskell, and they do it slightly different. At the time, I had spent afternoons debugging code myself where, I think, a year or so ago there was an infamous Apple security bug...

Hsu: Ah, yes.

Van Rossum: ...caused by two statements that were both indented the way the programmer wanted the code to be grouped, but of course, there were no curly braces around the group. So only the first of the two was actually being executed conditionally...

Hsu: Right.

Van Rossum: ...and that <clears throat>—I had run into versions of exactly that bug myself and so I was very sensitive to the argument that ABC's authors made for why indentation is good for us. BEGIN and END in languages that support BEGIN and END as keywords, those are more common than any other language keyword because you're constantly doing this and I don't know. Curly braces, I suppose, are okay but there is endless style arguments about where do the curly braces go and you—you waste space. You waste vertical space very—which is very precious on a computer screen, still after all those years, for the closing curly brace. So I'm still very happy with the decision. I still don't know if I were to design a new language today with all the same design goals as Python if I would use indentation just because it's different from every other language. Every other language has—that is currently in use, at least in the crowd where I hang out, whether it's JavaScript or Go or Rust or C or C++ or C#, it's all curly-braces based, and so tooling and people's brains are just pre-greased for dealing with that.

Hsu: So in some ways, you're a pragmatist. You would design something that, in some ways, conforms to the syntax that people are familiar with in some ways. What—I guess Python itself is C-like syntax in many ways and so...

Van Rossum: Apart from the curly braces, I borrowed a lot <clears throat> I borrowed a lot of other things from C, like choice of keywords, expression operators and priorities, format strings, at least the first few generations of those, definition of what constitutes an identifier, calling notations and conventions. Oh yeah, I hope this isn't too much of a detour, but in ALGOL 60 and ALGOL 68[8], to call a function that has no parameters, you don't write an empty parameter list in parentheses. You just write the function name, and so that gives visually—a zero-parameter function is very different from a one- or more parameter function. It turns out that syntactically this is also a stumbling block, and one of the things I remember van Wijngaarden telling his students that one semester that he taught a class that I attended. He said, I think one of his classes he was just reminiscing about ALGOL 68, which had been quite a bit in the past by then. And he said "all the complications we added to the grammar to get zero parameter function calls working syntactically correctly, and the hours of debate it took to push that through the language design committee, you know, all the cost for compilers and users, and then, god damnit—" I'm sure he didn't use that strong language, "but then after we had finalized everything, we encountered C, which was then an upstart language, and we saw that they just said, 'You just write empty parentheses.'" And he said, "We never believed that the programmers would accept that. If we had thought that the programmers would be happy to write that empty pair of parentheses, we would have happily done it that way, because it was easier all around. And yeah, so that's another thing I just borrowed from C; it's also the pragmatic approach to design.

Hsu: One of the things that you have said about the indentation feature is that it forces a consistency, because there's only one way to do things.

Van Rossum: Absolutely.

Hsu: And that allows for readability and a maintainability of code.

Van Rossum: Yeah. I mean, I still see that in my day job at Dropbox, that people can read each other's code because it all looks roughly the same. And yeah, there is a lot more to coding style than indentation style, obviously, but it does help that there is never any debate possible about where the curly braces go or, I mean, people can even argue about whether to put curly braces around a single statement or not. A whole bunch of the code becomes more readable because it's more familiar. Once you know Python, you can read someone else's Python, and I guess the language that is most different perhaps is Perl, where everybody seems to have their own unique style and every package has its own conventions, and the language proudly declares its philosophy as there's more than one way to do it, which is great when you want—when you're an extreme individualist and you want everything to do exactly the way you want to do it and you don't want anyone else to try and control that. On the other hand, it's bad when you develop software in a group; as soon as you have a team of more than one person, I think the uniformity of indentation is helpful.

Hsu: So what features make Python good for rapid prototyping and development?

Van Rossum: I would call out there the standard library and the lack of typing and the powerful data structures which are also part of the standard library, but there is so much more in the standard library where, you need to do some networking? Where is the standard library module for that? You need to do

[8] [Interviewee's note] In Algol-68 the feature is called "deproceduring".

some web crawling. There's a standard library module on top of the networking code to do that. You need to read a spreadsheet, oh, there's a module for that, too. And so prototyping is often an exercise in integration where you take existing components and you try to quickly put them together to get something working. And because there are so many ways to work with other software from Python, if it's easier to just fork off a subprocess that does the rest of the work written in a different language, you can do it that way. There is the large collection of things that's the language, and nowadays also the collection of third-party modules lets you do—there is also that you don't have to write a lot of code to do any of that. You—if what you want to do is three operations, it's probably three or four lines of code. There's no boiler plate or overhead. You don't have to prepare, set up, declare, check. Exceptions are another example. In a prototype, often you'll want to play loose and fast with exception handling or error handling and you'll say, "Okay, we'll just open the socket, then we'll bind it to a port, and then we'll write a few bytes to it and then we'll read some bytes from it and then we'll close it again." Well, we don't even have to close it, actually. Implicit is that if any of the operations fail, you don't have to write error checking code. This particular example actually comes from things I remember doing in Amoeba, where I was doing things with sockets in C code, and because I was just writing a quick prototype, I hadn't put in very careful error checking, and so the right operation was silently failing because I hadn't put in the right argument or something. And so the whole program didn't work, but it didn't give me a very clear error message, because the right operation just returned an error code, and I wasn't checking for that error code. Python defaults to doing things the other way around, you, if you expect an error, you can catch it. It's not like everything is a panic, but if you expect things to go right and somehow you make a mistake, the mistake will be brought to your attention. I remember a very early version of Python did not have thorough integer overflow checking, and then I was writing a little program that was copying from a book and I couldn't get it to work. The reason was that, unbeknown to me, the program was manipulating very large numbers, like on the order of hundreds of digits. And that very early version of Python didn't have protection against that. And so it was essentially having the same problem you would have if I had sort of written it down in C. And so I think that afternoon, once I had debugged my problem, I decided Python will check on overflow on all parenthetical operations, because we don't want errors to fail silently. This eventually became part of the Zen of Python.

Hsu: You mentioned the weak typing and dynamic typing. So Python is an object-oriented language with a dynamic runtime, dynamic typing and binding. What inspired this feature? Were you familiar with Lisp or Smalltalk?

Van Rossum: In fact, I was not. Well, I had spent probably a few afternoons playing with Lisp. I had probably heard of Smalltalk, but never got my hands on a Smalltalk system at that point.[9] And I had forgotten everything about Lisp by the time I was on Python's design. It actually came from the implementation of ABC. So ABC was technically—it was strongly or statically typed, but it, like Python, ABC also did not have type declarations. But instead, it had a type checker somewhat similar to what you find in Haskell these days, but much more constrained. But basically, it would say, "If you have an

[9] [Interviewee's note] That's not entirely correct. I recall briefly playing with a Smalltalk port to an early Macintosh or Sun workstation but not liking it much. I also recall reading up on Smalltalk-80's bytecode, and it inspired the design of Python's bytecode.

argument, ‘a’, and we see an operation where you say ‘a + 1’ anywhere in your function, then we conclude that ‘a’ must be a number." And ABC had cleverly spelled all the operations it differently, so you could always tell from the operation whether something was a string or a number or some other datatype. And actually one of the weakest parts of ABC's implementation was the type checker, because Lambert's prototype did not have a type checker. And the runtime was actually safe, in a sense, without relying on the type checker. So in C++, if you manage to smuggle something past the compiler’s type checking, you can probably also use that to crash your program. In ABC, at runtime, every object was tagged with its type. So there were two notions of type. The compiler during the type checking phase would eventually discover, "Oh, this variable is a number," but quite separately, in the interpreter, the implementation, it would essentially have objects, and it would say, "This object is a number, and that object is a tuple," and the interpreter, once you go past the type-checker, the interpreter would not actually care about the types in the program, but it would care about the types of the objects. And if you try to concatenate—if you tried to use the concatenate operation on an object that was not actually a string, it would issue a runtime error. And so I was quite familiar with that implementation technique. There was also—there was a backdoor in ABC where if you had separate functions— yeah, if you used it interactively, sometimes the type checker wouldn't know about the types of a different function, and then the dynamic type checking would—the runtime type checking would always take over. So the runtime checking in ABC was the safety measure and the static checker in the compiler was more for looks or to find bugs early, if you could. So I didn't quite understand the algorithm that the static checker in ABC was using. I don’t know if that had been implemented by someone else or if I had just taken the description and implemented it without really understanding why it was the way it was. It was probably someone else who did it. So I never had much love for that particular aspect of ABC, so I just left it out from Python. Again, just to save myself time, because it was like a skunkworks project that I had to do in my spare time and I wasn't planning to spend more than a few months on it, and so every shortcut I could take, including, "Well, just leave out portions of the language that I didn't care much about anyway," would save me time. But the runtime checking was always important, because I was really tired of core dumps.

Hsu: Right. But that puts Python in this family of languages like Smalltalk, Lisp, Ruby.

Van Rossum: And JavaScript.

Hsu: Yeah, and JavaScript, that are known for runtime type checking, and there are arguments going both ways about whether dynamic typing makes a programmer more productive in certain ways versus static typing makes certain bugs easier to find.

Van Rossum: That's one of those arguments that keeps going around, and where probably studying history carefully will be useful for future generations. I know that when I started with Python, dynamic checking was in the doghouse. I remember encountering many people who had had a computer science professor who was very influential and who had categorically stated, "Static type checking is good; dynamic type checking or weak type checking—" it was originally couched in this strong versus weak typing, which is a slightly different dimension than static versus dynamic. But the “static or strong typing is good and dynamic type checking is bad,” that attitude was still very prevalent when I came up with Python. And I noticed that often people who were arguing for that couldn't actually express very clearly why that was better. It felt like they were just repeating arguments they had heard somewhere else, or they were using arguments that were based in previous compiler culture, because I'm sure that when Fortran first introduced the distinction between integers and floating-point numbers, it was incredibly important because it could map the difference to different hardware instructions, and all languages since have used the type system as a crutch to help the compiler generate type-specific code and C probably represented a local maximum in that attitude because in terms of type safety, C is absolutely crap. I mean, don't get me wrong. It is actually one of my favorite languages, but it's not very type-safe, even with later standards' additions. The C compiler primarily needs the types so that it knows what code to generate. But for a higher-level language like Python, that's not so important. And it comes with the territory, maybe, of an interpreted language. Interpreted languages have always had their niche, and often there's been a race towards making faster interpreters. And for an interpreted language, this—If you have a type system, it's used for very different purposes, and the type system should focus on what helps the user of the language. And so things like the C type system, which is full of declarations of how big the words are that are needed to store your variables. Like is it an int or a long, or a 64-bit long, or is it a float or a double. That is all nonsense. That is, from the user's perspective, that is most of the time just a distraction. I mean, it's a rare program where you have so much data that you really want to store your data in 16-bit words instead of 32-bit words. Usually, you have bigger fish to fry. So that's where I went with Python and it's with ABC's implementation's example, I found my way to the dynamic typing. And yeah, it was later that in '95 or '96 or so, Bob Kahn, the guy I was working for at the time in Northern Virginia, took me apart aside and he said, "This is a great language you've made, but did you know that you have—how much your language has in common with Lisp?" And I was offended, because all I knew about Lisp was the "Lots of Irritating Silly Parentheses" part. But the dynamic type checking part, the whole runtime system, I wasn't really aware that that was really where Lisp was big, because there was no one around when I learned to program who was ever using Lisp. I think maybe it was one of those languages that Tanenbaum used in his class of seven esoteric languages.

Hsu: It seems like dynamic languages have had a—they were pretty popular maybe five years ago or so, but in the meantime, a lot of the newer languages that have been designed, that have come up, like Go and Kotlin and Swift and things have been statically-typed languages.

Van Rossum: Yeah.

Hsu: What do you think is the cause of this?

Van Rossum: It's an interesting development that dynamically-typed languages were effective, but not very much respected in the '90s, but the Internet was built by a variety of dynamic languages. Between Perl, Python and JavaScript, so much Internet stuff was built using dynamic languages because of the programmer productivity and being able to quickly throw together prototypes or things that do a bunch of functionality without necessarily having to be super-fast has been incredibly valuable. And slowly, dynamic languages gained more respect and static—in a sense, there was a backlash against static languages where people started saying, "Oh, well, I can do it in Java, and I have to type every variable, I have to type the type of the variable three times before I can use it, or I can write it in Python where I have none of that." And so eventually designers of newer static languages started thinking, "Well, what can we do about this argument that dynamic languages are better or more productive than static languages, and what can we do to keep the type safety of the static language but combine it with the productivity or maybe perceived productivity or readability or conciseness of dynamic languages, and you get languages like Go that do a lot of type inferencing, so you don't have to be explicit so much about the types of variables and arguments. Or even C++, the ‘auto’ keyword in the latest version of the language. And so that's been a very interesting discussion and it's been very exciting to me to see that the dynamic languages have actually caused a response amongst designers of more traditionally-compiled languages. And every language is always scratching some particular itch. I mean, C# scratched the itch of having a Microsoft corporate language, I guess, that was not [Visual] Basic. Swift scratched the itch of having something that was a little better than Objective-C, especially for iOS. Go, I think the itch that it originally tried to scratch was just compiling faster than C++, and I think Rust is all about memory safety, which has been a challenge for many other languages. So yeah, I'm happy this is happening. In the meantime, there is also a recent development in the Python world where we’re now experimenting with optional static typing for Python, where [we're] actually using a similar model like the static type checking [in] ABC, but more extreme. The type checker is actually a separate program that you don't have to run. It's voluntary, like a linter, but we're experimenting with that. And so what I've eventually learned is that the advantages of dynamic languages are more pronounced for small to mid-sized scales. And for really large code bases, static languages, despite all their visual overhead and compilation cycles and build tools have their uses, too. I mean, when very large teams build very large pieces of software, the static types actually are not redundant.

Hsu: That brings up a really interesting point that I wanted to get into, which is about how Python is able to scale. You know, you mentioned Python's extensive use on the web. You know, Python is used for anything from small strips of glue code all the way to large production web servers, web applications, so what makes it so versatile? And what makes it able to be robust enough to produce good, well-engineered production code?

Van Rossum: Well, production code is usually not all that well-engineered. <laughter> Well, that's also not true, but production code is not necessarily well engineered and sometimes there is the engineering that goes into making production code production-ready is a very different kind of engineering that goes into writing code well. There's a whole bunch of tooling around that. You should interview Googlers about that topic, though. But I think Python's extensibility has been the key to making it work well for a large dynamic range of application areas, like from the very small to pretty large, and across web and data science and other forms of science. Extensions really have been key to that, because if your problem involves looping over a matrix of 1000 by 1000 numbers or 1,000,000 by 1,000,000 numbers, Python's for loop is not the fastest for loop in the universe. But you don't actually have to write that as a Python for loop, because you can use a NumPy extension that has the for loop, the whole algorithm that you want to run, whether it's a matrix transformer or a filter or a dimensional reduction, whatever. There are libraries written in C, C++, Fortran sometimes, that have interfaces that make it very easy to use them from Python code. And that is really the secret for this much of the large scale use of Python. Sort of the—like in data science, and this started, I think, in the late '90s, there were people at Lawrence Livermore National Labs across the Bay who were using Python to replace a homegrown scripting language that was basically a glue language that glued together all their dusty Fortran and C++ decks. But previously, they had been

either writing some dead-end homegrown scripting language or C++ as the main driving language. And they switched to linking everything to Python, but using Python to drive the dusty decks and Python just tells the dusty decks, "Here's the file and this is the format and this is the operation I want you to do." And it gave the scientists just the right toolkit and it—the scientists who were previously doing that in C++ would constantly struggle with the C++ compiler and syntax and one missed comma or semicolon would cause everything to go mysteriously wrong or to [cause you to] bang your head against the wall. So that trend has 20 years or so of gluing numerical
code together with Python, and ironically that's—the numerical stuff is the stuff I never liked in college. I still don't understand it, but there's a whole community of people who maintain and create libraries for use in Python so that scientists can feed their data into these libraries without having to know C++ or Fortran. But all of the performance critical code is actually written in C++ or Fortran. They just don't have to write any new C++ or Fortran. And on the web, it's a somewhat different story. But again, at some point, a web application probably is just an interface to a database. That database is MySQL or some other SQL variation. That's all written super-optimized and it does query optimization and you can have teams of database administrators sometimes who can put the right indexes in. The web application just takes input from the web, turns it into a query that then runs against the database and then turns the output—does light processing on the output of the query and runs it through a templating engine, the templating engines are these days usually also written in C++, as Python extensions to generate HTML, or the modern way of doing that of course is you just send JSON data to the browser and there is JavaScript in the browser that renders it. But again, the Python code does not do the heavy lifting, but it controls how all the heavy lifting components are composed together. That is, in some sense, also that UNIX philosophy of making tools that work together.

Hsu: Right. That gets around this perceived performance disadvantage that Python has of having a dynamic runtime that—I've heard a similar argument, about Objective-C, actually, where like—

Van Rossum: Totally.

Hsu: Where all the performance-critical code in Objective-C is actually straight C.

Van Rossum: Exactly, but it's not so straight. Yeah.

Hsu: So yeah, it sounds very similar. Could you talk a bit more about why Python has been so successful on the web?

Van Rossum: It's still hard for me to get my head around that, to be honest. I've never particularly pushed that. I remember in '94 or '95, someone observed that there was this thing called CGI on the web. Previously, web servers had mostly been serving static HTML and it was still a novelty to see images embedded in that HTML. And then people came up with dynamic web applications, and the first generation of that was you had this special part of your filesystem namespace where the web browser would say, "Oh, if it starts with /cgi-bin/, then give it to a script.” And the name of the script is the rest of the URL or something like that. And someone said, "Well, there are all these people writing shell scripts and Perl scripts for their web servers. I want to write a Python script. And here is 35 lines of Python code.

Will you please put this in the standard library, because then everyone can write their own CGI scripts in Python easily, because there is, like, some detail you have to pull out, the form data out of your environmental variables. Well, nobody does it that way anymore. But that was—

Hsu: So it began because it was a scripting language?

Van Rossum: Python was one of the many scripting languages, and among scripting languages, Python was known as having the most pleasant syntax, if you cared about syntax. I think Perl had cornered the market for productivity in pure terms of if you want to get a thing—a one-off job done quickly, you write it in Perl. Tcl/Tk had cornered the market for embeddability and for actually graphics UIs. And Python had cornered the market for somewhat embeddability but if you actually were doing more programming in your scripting language than your mother thought was good for you. So Python just became a natural choice for a small group of people who are here in Silicon Valley and all over the world pushing web stuff, and it worked. Because the startups were all competing on who is first to market. I remember that an early core dev in the Python community worked for a very early e-commerce platform. It was this tiny startup, and because they did their whole implementation in Python, they had launched a product with sufficient features to be viable before any of their competitors, at least in that particular subsection of the market, and what they did was they realized they couldn't actually compete beyond that, but they sold the platform to Microsoft and they were all—but if they had chosen to do it in Java, they [would] have been too late and nobody would have been interested in acquiring their startup. And that story, at some point, Python was, like, people wouldn't tell you that they were using Python because it was their secret weapon. But eventually, obviously the word came out, but that is where Python's productivity and all the things that went into this from the large standard library to the active community to my sense of keeping the design coherent all came together and contributed to the success.

Hsu: Earlier, you mentioned the Zen of Python. So that's this list of 19 aphorisms.

Van Rossum: Technically, it's a poem.

Hsu: Oh, that's true.

Van Rossum: The author says that it's a poem. But yeah, they're aphorisms.

Hsu: That layout of the philosophy of Python, so I'll just name off the first couple, like "Beautiful is better than ugly." "Explicit is better than implicit." "Simple is better than complex." So how did this come about and how does this reflect your view of Python and also the values of the Python community?

Van Rossum: So it was written by a guy named Tim Peters, who I think the first showed up in the Python community in the early '90s. He was actually a compiler expert working for some unknown mainframe company and he somehow decided that he enjoyed working on Python and working with Python in his spare time, and he started giving me advice. He was the first to observe that I had done something stupid with integers and floating-point numbers and he told me, in a very friendly but full of authority way that I should change that. And that became the big—I think that was one of the starting points of a very longlasting relationship where he's been just a great collaborator and I think in 2000, we finally started working together in a startup that—that was a terrible thing, that startup, but it got Tim Peters out to Northern Virginia to work with me, and that was good. And I think during those days, I mean, he had been guiding many newbies and others on the Python mailing lists, giving excellent advice, always answering questions. He had this thing "channeling Guido" where he knew how I thought about language design and he could articulate it well and explain it well it to people who were asking questions. And he kept them from bugging me. And so he knew Python very well. At the time, I think he probably knew it better than anyone besides me, and so somehow he—the way his brain works, he came up with I don't know how long he worked on it, but he came up with all these short slogans that sort of, together, describe Python's philosophy and it became an underground hit in the community somehow. Yeah, because the first one, "Beautiful Is Better Than Ugly," nobody can argue with that. But "Explicit is better than implicit" is already positioning itself against Perl, for example. And I've never been a fan of the type of language debate where you say that this language is better than that. I guess I'd had enough of that in the Fortran versus ALGOL 60 days. So I never wanted to come out myself and say, "I think Python is better than Perl." I would always couch it in, "Well, for this particular application, Python has more tools. Or if you like to share your code, it's more readable in Python." Other people turned this into Python versus Perl wars, and I was trying to tell people "Don't be so direct about it. There's room enough for all of us." But yeah, the Zen of Python somehow is still often quoted. Actually it's probably only five or six of the 19 that are referenced a lot. "Explicit is better than implicit" is one. "There's only one way to do it," which is an incredible exaggeration. That one is the the most directly opposing Perl, probably. "Flat is better than nested." That's something he just observed in how I structured Python's standard library.

Hsu: So let's talk about the community, and also open source. So how did you get involved in the open source community and what led you to open source Python?

Van Rossum: Well, so first chronologically, Python was open source before the term open source had even been invented. The reason that I made Python available was a not quite an accident. Somehow, the Amoeba team at CWI also did a bunch of stuff with X11, an early windowing system. X11 was created by this collaboration between industry and academia, and to save everyone trouble, it was licensed under something that was essentially what is still known as the MIT license. It's now a famous open-source license. At the time, I think it was just the X11 license, or that was the only one that I knew. When I released Python in, I think, February '91, I realized that there had to be a license on there. I mean, smaller bits of source code or bits of code that were written by individuals in their spare time were often released as freeware and various other things, including shareware, where you were asked to pay but not required to pay. There's a whole bunch of variations. I had a day job—I mean, CWI paid me a salary, so I didn't want to make money off Python. I was also not very commercially interested. But I felt there had to be a license and I had seen that X11 license as a good example of a software license that was short enough that I could understand it and that looked like it was entirely unoffensive to people who would possibly want to use it. And so I don't know, the week before we were planning the release, I went to some assistant, I think, of CWI's management who went over legal business and I said, "Hey, we're going to release this piece of source code and we don't want to make money off of it. It wasn't made for money. It's a byproduct of a bit of research that were doing. Would you be okay if I used this particular license?" And she said, "Okay, I'll get back to you on that." And, I don't know, I think within a day or a few days she said, "Sure, yeah, management has signed off on that." I have no idea if she just signed off on it herself or if she had to go to the director or how that worked, but that was that and we released it with, and I always try to make this Monty Python joke, it was the MIT license with MIT crossed out and CWI written in in crayon. And so all through the '90s, Python went out with that license, and when I moved to the US and started working for CNRI, their lawyers were not so keen on that license, but they realized that it had been released under that license for a long time, and they couldn't really change it, at least not immediately. They asked me to add one innocent paragraph recognizing that CNRI also had something to do with it. And then I think maybe through Tim O'Reilly, who was probably already publishing a Python book. This was, like, the second half of the '90s. I got in touch with a guy named Eric Raymond, and I think that Eric Raymond had just been pushing the idea of open source and the specific term “open source” had been very recently invented. And I think that Tim O'Reilly invited me to come to the West Coast for a few days for the very first open source summit where a bunch of people who had created or worked on big open source projects or free software projects were all getting together and agreeing that they should use this term, because it was—there was an issue of terminology. There was also the Free Software Foundation, Richard Stallman's organization, and the GNU project who were in my view—their attitude is a more communist view of the world. "We want everything to be free." And they were in contrast with Eric Raymond’s side of the party, which was more, "Make it free so that everyone can benefit" without enforcing things on others. And because Python was essentially open source because the license was just a copy of the MIT license and because somebody recognized that Python was an influential thing, they invited me to that meeting. I also remember that around the same time or probably just before that time someone else who was a little more tuned into that world, another Python core developer was asking me, "So, from now on, are we going to call Python ‘open source’?” instead of— I don't know what we used before, maybe we just said, "Python is free software." And at first I thought, "Well, why should we use this different terminology?" But I think after I had met Eric Raymond, I was sold on the idea.

Hsu: Can you talk a bit about how the community around Python grew?

Van Rossum: Yeah, so I started by—I always wanted my software to be shared. Anything I'd written, I always wanted other people to use, and even when I was down in the basement of the math department. So releasing Python was never a question. I always wanted to release it. It was traditional, I think, on USENET to have newsgroups specific to various bits of technology. We started, actually, right after the release, we started with a mailing list, and it was a very primitive thing because there was no mailing list management software. There was just a file containing email addresses that lifted my home directory, and if you sent email to python-list@cwi.nl, the mail server at CWI just peeked in my home directory and expanded the list. Terrible security problems. But that's how it was done, and so when we had 700 people on that list in the summer of '94 or so, the spring of '94, some people said, "Well, there's really too many of us to do this as a mailing list. We should have a USENET newsgroup." And there was already a standard naming scheme, so people started saying, "We've got to have comp.lang.python." And I was an avid reader of comp.lang. “everything”, but I didn't know how to create new ones, and I knew that there was some kind of bureaucracy, so I told my followers, "Well, if you want a newsgroup, go figure out how to do it and then just do it and I'll start using it." So people did it. And then people said, "Well, we want to have a physical get-together of Python users," and someone said, "Okay, I'll try to organize that." And we had the first Python workshop. That's where I also got this job offer in the US, because the workshop was in northern Virginia. Actually, no, the workshop was in Maryland, at NIST. So the community just grew itself. There were more and more people who thought, "This is a cool language, Guido is a nice person and he's responsive," because the one thing I cared about was a keeping people happy, so if people sent me a cool idea, then if they send me code, it was likely that I would just merge that code into Python. If they sent me an idea, sometimes I would implement the idea if I liked it. I mean, if they sent me something I didn't like, I was direct enough to say "Oh I don't like that," or "You can solve that problem much simpler without changing the language." But there was a lot of interactivity and I guess people liked that style of online community and like so many other online communities, started growing around the same time, of course. And so that first workshop, everybody who went to that workshop was super happy and proud, and so someone volunteered to organize the second Python workshop, and then someone organized the third Python workshop, and before we knew it we had hundreds of people showing up, and it was like an annual business instead of every six months. And so it's always been—the community has grown itself and self-organized and whenever there were things that weren't working right, somehow people started asking for help or for direction or proposing changes. And so we got our release versioning sorted out and we got a process for proposing changes to the language, so that debates don't go on forever or get restarted every six months. And we're still learning. We're still growing and we always look for good ideas in other communities, other languages, other big software systems. But sometimes, we also lead the way, a Python Enhancement Proposal system which we actually borrowed from the IETF's RFC system, but we toned it down. The RFC's are very strict. So we tweaked it a bit and then other languages started doing the same thing.

Hsu: It seems like in the beginning, when you just had a mailing list, then people would email you their contributions directly and you would integrate them directly. At what point was there a more formal contribution process where there was like source control and everything and a repository and things like that?

Van Rossum: Well, so originally there was—I had source control, but it wasn't shared. There was no server. I have commits going back to, I think, August of 1990, so the first eight months or so are lost, I mean it's just one initial commit that has Python as it was in 1990, but after that, everything was done through source control, and we've used every single source control system that was invented since then, I think. But originally, I did all the review and I merged everything into the repository, and it wasn't shared. The way that things came out was just, like, regular releases. So I did, like, I don't know, the first releases, initially, probably did the release every month or every few months or so. And sometimes, there would be that—if there was a screw-up, there would be another release a week later, but not too often, because I learned from that to double-check the releases. I think in the late '90s, when I worked at CNRI, CNRI hosted an internal CVS server, and some of my colleagues, who were also working on Python also had write access to that repository, so that lightened the load a bit. Then, when we left CNRI, we switched to one of the first public source control servers, which was—what was it? I think it was Subversion hosted by SourceForge. Yeah, and at the time, SourceForge was a really cool thing. Now, it's like this, I don't know, it's the bad neighborhood of source control. But when it started, it was really cool. And we, in 2000, when the whole Python Labs team left the CNRI, and that's when we also—Tim Peters joined us as a team

member and we all worked for this horrible startup that collapsed within six months, and after that we were rescued by a different startup that was a much better home for Python.
But we innovated tremendously that year, and so that's also when the PEP [Python Enhancement Proposal] system came up. Suddenly, we had five people, the whole Python Labs team, where we were doing nothing but Python. So we were improving the interpreter and every aspect of the language, but we were also working on the process and community management and websites and fighting the lawyers sometimes and behind the scenes discussions with Stallman to make sure that it was both open source and GPL compatible. But yeah, the process improved quite a bit. The community also yet again reinvented the conference cycle into a completely self-organized group called PyCon. Oh yeah, we created the Python Software Foundation, which is technically the modern organization of PyCon and also all the intellectual property is owned by the PSF. That was necessary so that we had clear legal status to keep Python open source for everyone.

Hsu: And that existed separately from your actual employer?

Van Rossum: Yeah, the PSF was a nonprofit with me as the president and a few Python luminaries as board members. And there was one guy who knew a bit about legal stuff who had made sure that it was properly registered and all that. But yeah, that was fortunately completely separate from employment, and from then on I've always managed to keep that clearly separated.

Hsu: How did you become the "Benevolent Dictator for Life?" Or the designated BDFL?

Van Rossum: Yeah, the "Benevolent Dictator for Life," that was a joke title that was assigned during some meeting of a bunch of Python luminaries in the late '90s. An email[10] that reports from that meeting I think is online. I forget what all the other titles were. They were, like, they were all funny and all had very little to do with what people were actually doing. There wasn't really an organization, it was just a bunch of people interested in Python. So I was given the title "Benevolent Dictator for Life," reflecting the tension between being the lead designer and having to balance my sense of what's right with what people actually want and deciding what to do, if not everybody wants the same thing. And still, I'm still stuck with that role.

Hsu: Yeah, and that's also another Monty Python reference, correct?

Van Rossum: It is not actually literally from Monty Python. I think that none of the terms we assigned at that time were direct Monty Python references, but each of them was jocular in a way that was aligned with Monty Python's sense of humor.

Hsu: You just spoke about that tension. Do open source projects need something like a benevolent dictator or at least a core group of maintainers who control the process, and how does that—you know, how do you manage that tension between control and democratic participation, which is in open source.

[10] [Interviewee's note] https://www.artima.com/weblogs/viewpost.jsp?thread=235725 .

Van Rossum: There is no one answer for that. I think every open source project needs some kind of structure, but depending on the nature of the project and the nature of the people involved, many different structures work. LINUX, I think, is organized somewhat similar to Python at a larger scale, where there's also one strong guy at the top and there are lieutenants and there's not really a democracy but the guy at the top tries to listen to everyone to the extent that there is time. That's what I try to do, too. I know the original Apache web server was created by a collective of people, none of which really wanted to step forward and be the leader. And so they created an explicitly sort of not necessarily democratic structure, but they created a structure where there was, like, a group of leadership where everyone had the same power. I don't know if they still do it that way. There are many different ways of organizing a project. A lot of small projects just start with one guy or one woman at the top, and often, until the project is super-successful, that one person is really the only developer. And some projects are incredibly successful with one developer. And like everyone else is just a user. Some products quickly acquire other developers who are interested in contributing. If a project is healthy, then they find a structure where it's clear where the buck stops, what the ultimate responsibility is, who gets credit. It's sometimes possible for projects to accidentally fall into success without having figured that out. And that's when you get all kind of unhealthy things where there could be infighting or just disagreement on who gets credit for the project. I just read up about a project named ZeroMQ where apparently, at some point, a number of the original developers just decided that they were tired of the project, and they started a clone of ZeroMQ under a different name. Usually, those things don't end so well, but yeah, there are all sorts of different structures possible. I mean, there's Perl, where for the longest time, Larry Wall was where the buck stops, and then at some point I think in the early 2000s, Larry decided that he wanted to focus on developing Perl 6, and Perl 5 was left in—for a while I think there was a leadership vacuum where that the people who had previously been Larry's lieutenants suddenly had to figure out how to decide without being able to fall back on Larry. I think they mostly figured it out, although I heard that recently there are still—there have been some continued arguments, I think. Maybe Perl 6 versus Perl 5 is now, again, an issue now that Perl 6 is finally been released. But it's important to have clarity about your structure so that, I mean, when everything is clear sailing, you don't have to worry about anything. But when things are—when some unexpected thing strikes, you need to know what to do. You need to have some kind of fallback mechanism. You need to have a—well, where does the buck stop? What do you do? Actually one of the early threads that started the Python community was also, "What if Guido were hit by a bus?" Where people realized that there had to be some kind of contingency planning in case I would no longer personally be able to run the community.

Hsu: You've also been described as Python's chief evangelist. So what do you do in that informal role?

Van Rossum: I don't know. That's interesting. I mean, an evangelist is typically someone who goes out and spreads the word and I do actually very little of that. I probably did more of that in the past, but these days almost all my speaking engagements, and I have very few of those, they are mostly internally focused. They're in the Python community. I give a talk at a conference for Python users and developers, so I don't go out. I don't go to general computer conferences and speak. I don't write about Python, at least not much. I mean, I have a blog, but I hardly write there anymore. So the evangelism, I do maybe more implicit just through being a good person and encouraging the community to behave in a certain way and make their whole community be the evangelists to the outside world, if there is such a thing.

Hsu: How has the language, change in the language, reflected the growth of the community or change in the community?

Van Rossum: Well, that growth of the community and growth of the Python user base. The main effect of that has been to slow down development. That is not completely true, because sometimes it feels like we're changing the language more these days than maybe we did five years ago. But most of the time, I think most of the effect of growth has been to become more conservative in language design. Like if in '92, you told me that a certain thing that Python had always done was completely ridiculous and should absolutely be changed, and maybe objectively it was the case that Python was doing the wrong thing, like not allowing integers to be added to floating point numbers. In '92 it was easy to change. I wasn't all that worried about what would happen to the users' code that was already written, because I was aware that most of the world's Python code hadn't been written yet, and the users were early adopters, and they were very supportive of change as long as the change was for the better. And now, everybody always wants one new feature, and they don't want anything else to change, but that was a very forgiving group of users. And like in the early 2000's, I remember being surprised at a Python conference, where there was, like, a delegation of users [who] came to me and I think they had even selected another senior person to represent them so that they were sure that I would hear them. They came to me and they said "Python is developed—is releasing new versions too fast. We can't keep up with the upgrades. You've got to pace the releases." So ever since the early 2000's, Python has had a release schedule where with a few exceptions, but the general rule is releases are 18 to 24 months apart. Farther than that, it becomes awkward, because you have to wait too long for a new feature to actually land. But shorter than a year and a half means that users are constantly struggling to keep up. And so that is a real, real change. And I expect that in the next decade, we'll have to revisit that release cycle and what we do with features versus bug fixes versus compatibility. We'll have to change that again. Yeah, one of the things we also have is there is no—not only is the release cycle really slow, but we're also incredibly conservative in compatibility, with the exception of Python 3 versus Python 2, in general new versions of Python must run older programs that were written with older versions in mind. Now, there are definitely languages that are even stricter than that. Java is stricter. Well, Java also has way more users than Python. C++ is even stricter. Again, C++ probably has even more users. C is probably—for C, it doesn't matter, because that language doesn't change anymore, so they don't have compatibility issues in that sense. But it's every decade we have to become more conservative, and that also means that the filter I have to apply or that the core devs apply for deciding whether a new feature is worth including in the language when it's proposed becomes more and more strict. Again, in the early 90s, someone sent me a mega patch that introduced [the] functions map, filter, reduce and lambda notation. And I hardly knew what the plan was with that and I had only a very rudimentary understanding of functional programming at the time, but I could see some of the cool things you could do with that and the person who gave it to me sounded like they knew what they were talking [about]—they were obviously a very good programmer, they had studied Python source code very well to be able to construct that complicated patch. And so I accepted it. I tweaked the layout of the C code a bit, maybe, but essentially a new feature of that magnitude would require a huge foray into the PEP process these days that would be treated completely differently and it probably wouldn't make it. You'd have to argue that a large number Python users would benefit from this language change. And so the standard library is also very different. I mean, the old standard library had a lot of things that were really more demos than useful utility modules or just things that I thought were fun to have. And now the standard library is like—it's already huge but it's not growing at the same pace as the collection of third-party code that we have on the package archives, because the standard library is beholden to this gold standard of compatibility.

END OF THE INTERVIEW