LUCIANO RAMALHO

LANGUAGE-CREATORS.MP4

START (LANGUAGE-CREATORS.MP4)

[BEGIN PART I]

[0:48:29]

MC: Okay, are you guys ready for some language creators?

[APPLAUSE/CHEERING]

MC: Okay. Without further ado I will introduce Carol Willing. She is a Python steering council and Project Jupyter steering council member. She blends the strengths of Python and Jupyter Notebooks to improve access to learning and education. Due to the worldwide foundation of the Python language, the hard work of the Jupyter team, and users worldwide, Carol was awarded the 2017 ACM Software System Award, recognizing the lasting influence...

[0:49:00]

MC: ...of Jupyter (at broad) collaboration to develop open-source tools for interactive computing with a language agnostic design. Thank you.

[APPLAUSE]

CAROL: Okay. First, welcome everybody for being here. It is my absolute pleasure to share this evening with you and some of the people that actually influenced my career in software development. So let's get through this. And for those of you that are watching us on the livestream, hello and welcome, and we hope you have a wonderful evening with us. We have a really distinguished panel of language creators, and I'd like to invite them to the stage as they read their bios. I'm going to keep the bios really short because I think most of them need no introduction whatsoever. And that'll save us some time for more juicy language design stuff.

[0:50:00]

CAROL: Okay. So first we have Guido Van Rossum, who developed the Python language that touches pretty much every corner of society today in some way. And [CROSSTALK]...

[APPLAUSE/CHEERS]

CAROL: James Gosling, who's the creator of Java, which is used across a wide variety of devices and deployments at scale. Welcome and please join us.

[APPLAUSE]

CAROL: Anders Hejlsberg, who has architected a number of languages over the past several decades, including Turbo Pascal, Delphi, C#, and some of you might know this little language, TypeScript.

[APPLAUSE/CHEERS]

CAROL: And last but certainly not least, Larry Wall, who combined his unique...

[0:51:00]

CAROL: ...perspective on text and computing with his strong background in linguistics to create Perl and its sister-sibling, Perl 6. Welcome.

[APPLAUSE/CHEERS]

CAROL: Okay. So just a couple housekeeping things about this evening's format, you're in for an engaging and hopefully very lively discussion, more lively than we were at dinner. I think that's possible, right? Okay. "The format...

JAMES: I would need more beer.

[LAUGHTER]

CAROL: More beer? Okay, we can do that. Keep it coming. Okay, so the format will be two sessions with a 15-minute intermission. What I'll do is I will start by asking one of the panelists a question, and hopefully the response will be, like, one to two minutes. And then we'll kind of throw it open for other panelists to respond...

[0:52:00]

CAROL: ...and hopefully get a really good discussion going. And the more discuss the less I have to ask questions, and the more in-depth we can go on the really cool stuff. After the 15-minute intermission we're going to take questions from the audience over a...

WOMAN: Twitter.

CAROL: Twitter, okay. So tweet out your questions. Hashtag is #puppyBDFL. And if you can get those in by 8:00 we'll kind of collate them and have some really great audience questions about language creation and language design. So I am going to invite you to sit back, enjoy a cup of strong Java, C# responses and insights, learn Perls of wisdom, and embrace the Python programming (zen).

[APPLAUSE/CHEERS]

CAROL: So enough of my bad puns.

[0:53:00]

CAROL: Let's get this thing started, and we'll start the celebration with a question for Guido about the principles of language design. Guido, you've mentioned before--perhaps somewhat jokingly--the Harry Potter theory of language design. What is this theory, and what do you see as the key principles of language design?

[BACKGROUND CONVERSATION]

GUIDO: Am I good? Okay. The Harry Potter theory of language design was a blog post I once wrote where I didn't know what - I cannot believe that every little detail that J.K. Rowling put in the first Harry Potter book was intended as some important plot point in part six or seven or (eight) maybe. And even...

[0:54:00]

GUIDO: ...if J.K. Rowling was such a genius that she had it actually all planned out who the pet rat was, who I think we found in book three was actually a wizard in disguise. But in book one it was just, like, a temporary diversion during the train ride, and otherwise you didn't hear much about the rat with the finger missing. In language design often that's exactly how things go. You choose some detail because you have to commit. You sort of - you have to pick key words, you have to choose a style of coding. Like, maybe you choose indentation instead of curly (inaudible)...

LARRY: Or maybe you don't.

[LAUGHTER]

GUIDO: You're a special case.

[LAUGHTER]

GUIDO: But whatever you do, in some way you're stuck...

[0:55:00]

GUIDO: ...with that. And you find new uses of that detail that you picked before you knew how important it would be. And sort of the craft of designing a language is on the one hand pick your initial set of choices so that there are a lot of possible continuations of the story. The other half of the art of language design is going back to your story and inventing creative ways of continuing it in a way that you had not anticipated.

CAROL: So I think each of you have languages that have spanned multiple decades. So what were some of the principles that maybe you (dealt with)?

JAMES: You know, so Java was sort of odd in that it didn't come out of...

[0:56:00]

JAMES: ...like, a personal passion project or something. It was from trying to build a prototype. You know, I mean, we were trying to understand a particular (domain), (inaudible) (embedded systems). And spent a lot of time talking to people who build software for embedded systems and trying - you know, spent a lot of time trying to understand (inaudible) and, you know, what was it in the whole process for them, what was it in how things evolved (inaudible), how did their systems fit into the universe? Because it wasn't - you know, because I was trying to worry - we were worried about things outside of datacenters. And this was a project that had about a dozen people on it. And my little...

CAROL: (It started)?

JAMES: Yeah. But it was - my little...

[0:57:00]

JAMES: ...piece was sort of, like - you know, we sort of figured out that part of the problem was that C has issues.

[LAUGHTER]

JAMES: And so I started out - so out of this, like, large pie of a project my slice was to make things a little easier from a programming language point of view and fix the (pain) points that came from the programming language part. And it started out as kind of do a better C; and it got out of control. The rest of the project really ended up just providing context. You know, the only thing out of that project that survived was Java.

[0:58:00]

JAMES: But the fact that it was sort of directed at a set of (pain) points that happened to be, you know, about people who were living outside of datacenters, and people who were sort of getting shredded by problems with networking, and security, and reliability, and, you know, they had to build things that ran in hostile environments like in homes, which any home with a child in it is a hostile environment.

[LAUGHTER]

CAROL: Speaking of hostile environments, Perl [CROSSTALK/LAUGHTER]... What were some of the guiding principles for designing Perl?

LARRY: (Inaudible)

CAROL: I love you, Larry.

LARRY: Yeah. [LAUGHTER]

[0:59:00]

LARRY: That helps. Originally Perl was designed more coming at it as a linguist than a computer scientist. So I sort of almost actively ignored some of the computer science literature at the time and said, well, what can we just throw together in one pot that will work more like a natural language, instead of a, you know, instead of putting in a campus - university campus and deciding where all the walkways were going. We were just going to see where people want to walk and then put shortcuts in all those places, and build it more as a network, not as a, you know, terribly orthogonal, or computer-(science-y), or mathematical thing. And that turned out to be in the right place at the right time for bootstrapping a lot of the web. And...

[1:00:00]

LARRY: ...it also got used a lot for system administration. And so various principles that relate to trying to provide APIs to everything, and trying to be both a good text processing language linguistically, but also a good (glue) language is why it was kind of very useful when the HTML is text and, you know, database (inaudible). That needs (glue). And someone was in the right place at the right time. We were very fortunate to have that timing because in the '90s it became very stable, a lot of widespread use. A lot of people were - you know, the language itself was stabilizing in the form that it was. But there were a lot of issues, and still are. And so in the year 2000 we took a step back and...

[1:01:00]

LARRY: ...basically said, "We're going to break everything that needs breaking." So we kind of did the same thing, you know, as the Python (2), the three step, except instead of breaking a few things, we decided to break everything that needed breaking. So we came up with a whole (inaudible) of design principles in the 15 years since then. By the way, Perl 6 did come out two years ago; it's getting faster.

[CHEERS/APPLAUSE]

LARRY: And, you know, at the beginning we said we were going to do something impossible and fail at it, but be a very interesting failure. And so far it is proving to be that. But in the course of [CROSSTALK] - yeah. In the course of that we came up with (some) really interesting list of about sixty different design principles.

CAROL: Which he's going to read all of them right now.

[CROSSTALK/LAUGHTER]

LARRY: I'm not going to read - you know all of them - many of them already. You know, kill two birds with one stone, pick the right defaults, you know... There's lots like that.

[1:02:00]

LARRY: But we came up with some cute ones like, "You think that's cute today..."

[LAUGHTER]

CAROL: What about tomorrow?

LARRY: Yeah. You know, "Conserve your brackets because ASCII - even Unicode does not have enough brackets."

[LAUGHTER]

LARRY: "And don't reinvent object orientation poorly, which (inaudible) did." No. "(If something's) too strong for (inaudible), late binding sometimes cause your programs to be late."

[LAUGHTER]

LARRY: Some of the major ones are - you know, like, a lot of the stuff is Perl(fied) to screw over time. And, you know, there were a lot of weird magical variables. I blame (Chico). And so a great deal of the redesign was just saying, "Okay, what is the right peg..."

[1:03:00]

LARRY: "...to hang everything on? Is it object-oriented? Is it something in the (lexical) scope, or in the larger scope, or...?" You know, "What is the right peg to hang each piece of information on? And if we don't have that peg how do we create it?

CAROL: That's a great question. And since it's such a great question I'm going to put Anders on the spot. And Terence Parr had a quote about, "While programmers value simplicity, they more often value powerful functionality and amazing one-liners, incurring the cost of complexity. So...

LARRY: Works for us.

CAROL: (Inaudible) So what are your thoughts about simplicity versus complexity, and some of the principles that might guide you as you develop your languages?

ANDERS: I suppose I was always, like, in doing the language...

[1:04:00]

ANDERS: ...design that I've done, always, like, tried to make it so that there was only one way to do a particular thing. You know, some - a lot of languages have four different ways of doing something, and you pick the wrong one and then only weeks later do you realize you went down the wrong branch, and now you've got to back up. And so - but it's not always easy, right? And I think there is, like, often we're guilty of creating what I call - this thing I call "simplexity," which is, like, you take something complex and then you wrap a simple wrapper on top of it that's (hoping to make) the complexity go away, but what you're really creating is (simplexity). You know, it's just, like, a bad abstraction on top of another bad abstraction that is (inaudible). So - I don't know. I mean, it's...

LARRY: We think of that as picking the right default over the complex thing.

ANDERS: Yeah, but, I mean...

[1:05:00]

ANDERS: ...the thing about that - about language design is, like, any decision you make you have to live with forever. I mean, and in languages you can always add but you can never take away. And so you've got to actually as a language designer be very judicious about reasoning over not so much what to put in, but what not to put in, you know what I mean? Because people come to you all the time, "Oh, wouldn't it be great if you could do this, if - that, or you could...?" Well, yeah, but you can't fundamentally change the nature of the beast. If you created an empirical - an imperative programming language you can't turn it into a functional programming language. I mean, you can borrow from functional programming, but it is what it is and you've got to stay true to the nature of the beast or you've got to create a new beast, so to speak.

CAROL: Right. Great. Any other thoughts about...?

GUIDO: Can we interrupt yet?

CAROL: Excuse me?

[1:06:00]

GUIDO: Can we interrupt yet, or do you have more questions?

CAROL: You can totally interrupt anytime.

[LAUGHTER]

GUIDO: So Anders' sort of point about what do you leave out - and what I remember in the early days of Python's design, there were so many people who thought they have a good idea. And I wasn't always sufficiently critical to say no. And so there are a few things that didn't work out, but at the time I hadn't learned to say no. Like, I quickly enough did learn to say no, and I sort of developed, like, several lines of the fence. Like, the first line of the fence is, Hey, you think you need a new language feature, but actually you can already write that from Python.

[1:07:00]

GUIDO: If you need it a lot, write a function or a module. Well, if they say, "Yeah, but everybody needs that..." Well, nowadays we say, "Put it on the package repository." In those days I said, "Well, maybe you can propose a new standard (inaudible) module." That's a lot cheaper for the language design than a new language feature. And another line of the fence was, "Well, you can actually write extensions in C or in Fortran if you really care to," and you can help yourself (that). You can modify the behavior of the language in ways that aren't accessible when you just write the pure language, but you can still sort of - you can extend it in a way that doesn't fundamentally alter the core language. And if you've tried all those things and you've failed, then maybe...

[1:08:00]

GUIDO: ...you could argue for, "Well, we have to change something in the core language."

ANDERS: Yeah, I'll interrupt, too. I find that whenever I get feedback on whatever it is programming language I've been working on, it's - people come to you with a particular instance of a problem. And it's your job as a language designer to tease out the class of problem they're talking about, and then try to understand whether there's a feature there underneath that you could put in place that is broadly applicable, not just to that one problem, but, "Oh look, this could solve this, and this, and this problem as well." And so you're putting in place a class, not an instance, so to speak. That's, like - and if all you're talking about is an instance then you're probably better off leaving it out (inaudible). But it's a subjective thing, but...

LARRY: One of our principles is, "Save your money for college (inaudible)."

[1:09:00]

[LAUGHTER]

LARRY: We actually found this in the early design of Perl 6. We asked for RFCs--I expected about 20 suggestions; we got 361.

CAROL: Whoa.

LARRY: So we had this in spades. We - and, you know, it's completely overwhelming, until I hit upon this principle, you know, which Anders alluded to, which is basically ignore the proposed solution.

ANDERS: Yeah.

LARRY: But there is a (pain) point underneath. And if you look at more - enough of these RFCs and they have the same (pain) point, there is something you should (address). There's some fundamental unification issue underneath, or something that just is funky that could be fixed. And especially if you're doing a complete redesign, then you can think about that sort of thing.

JAMES: Yeah, and working with (inaudible). It's really hard. (Inaudible)...

[1:10:00]

JAMES: ...before Java, which (inaudible). You know, there were a whole lot of problems where - you know, it's clear that there was a problem there. There were, like, twenty or thirty worldwide academics who were absolutely at the top of their league for understanding the issues. Every one of them had two entirely different suggestions. They couldn't agree with themselves. You get, like, 40 different people in a room. The model (inaudible) and they've all got radically different ideas. And, you know, (inaudible), you know, it was their (inaudible) for (inaudible). And for some reason, you know, they - it's actually important to do the science project, you know? So, like, with generics...

[1:11:00]

JAMES: ...in Java, before Java was first released, it was really clear that this was, like, a big issue. The - and, you know, Bill Joy and I came about as close to spilling blood on this topic as I've gotten at (work). And, you know, I would much rather do something - or much rather leave something out than do something stupid.

LARRY: Yeah. One of our principles: "Plan to be smarter later."

JAMES: Yeah. Well, but that means - that means - the opening answer is no, right? And one of the things that happened with, like, a whole bunch of different topics in the evolution of Java was that they kind of turned into competitions. And generics and (closures) were both probably the most hard-fought...

[1:12:00]

JAMES: ...competitions with people writing PhD theses on the topic. And I was, like, trying them all out. And, I mean, some had no shortage of smart people at the time. But it's really hard to be, you know, a planet full of grad students...

CAROL: So speaking of grad students, at some point I think most of you were grad students, correct? No?

LARRY: In linguistics.

CAROL: In linguistics? Okay. Well, okay. But at some point you all, probably 20, 30 years ago, decided, "Okay, I'm going to write my own language." And things were very different back then. Internet was sort of starting, maybe not even there, single... I think we had computers. We had phones, right? Cell phones...

LARRY: Nobody cared [CROSSTALK] - nobody cared what you did with the eighth bit.

[CROSSTALK/LAUGHTER]

[1:13:00]

ANDERS: Oh, I think you're that (inaudible).

CAROL: But these are languages that persisted through many changes and, you know, on different hardware. When you started designing your language what was the key goal that you were trying to meet, versus using some other existing language?

JAMES: Well, I mean, for me it's never been a key goal. I mean, one of the secrets of success is try to solve as many problems at once as you can. And, you know, if there's just one problem there's probably an easier way than a new fucking programming language, right? [LAUGHTER] Right?

GUIDO: If you think that way you'll never have a new programming language. There's always a better solution.

JAMES: [CROSSTALK] Well, it depends on how much you hate...

[1:14:00]

JAMES: ...chasing down memory corruption bugs.

[LAUGHTER]

LARRY: The three virtues of a program are laziness, impatience, and hubris.

[CHEERS/LAUGHTER]

LARRY: It takes hubris to...

ANDERS: Yeah, the world needs another programming language [CROSSTALK] hole in its head.

LARRY: Don't look at (inaudible) code.

ANDERS: But the thing about programming languages that I think a lot of people don't realize is that every programming language is about 90% the same and maybe 10% new, if you're lucky. And a lot of people get very focused on the 10% new and forget about the 90% the same that makes it a great language, right? I mean, there's a lot of hard, boring work in creating programming language, lot of semantics that you've just got to worry about. And, like, everyone loves to talk about syntax. And syntax is the easy part; semantics is the hard part.

[1:15:00]

ANDERS: You know, like, how the types work, and what are all the supported promotions, and what are the different kinds of type constructors that the language has, etc., etc. These are the hard things to design, but not the things that people... People love to (inaudible) on the syntax. You know, "Should it be a colon or a comma?" you know? And it's, like, oh my God, let's have a long thread about that. [LAUGHTER]

CAROL: So talking about people having opinions, and syntax, and typing, these languages don't all take the same approach to typing. Maybe we'll start with Guido and talk about typing in Python, and then kind of work our way around.

GUIDO: Typing in Python evolved at various points during the language design.

[1:16:00]

GUIDO: I mean, I just remembered that when Python first happened INT was not a class; INT was a little conversion function. There was an internal object (type) which was really just kind of a (V-table) that represented integer objects. And there was a built in function if you needed to convert a (string) to an integer, and that was... We had a bunch of those functions, and we realized that we had made a mistake--we had given users classes that were different from the built-in object types. And for a long time it was, like, oh well, like, the real stuff is implemented in C. And the user writes a little bit of glue on top...

[1:17:00]

GUIDO: ...of that. And when I found out that we had 80 different competing web frameworks it was sort of time to realize that people were writing much larger programs, and that a different approach to types was needed. And that's where we sort of re-invented the whole approach to types in Python. And there was a bunch of cleanup that didn't happen until Python 3 actually. But one of the things we introduced, and we were lucky that there weren't--actually despite those 80 web frameworks--there weren't enough users that we were completely stymied by backwards compatibility yet, like we are now. So we just - one day we changed the function INT into a designator for the class INT.

[1:18:00]

GUIDO: And sort of it followed that calling the class would mean constructing an instance of the class, so that if you had simple code that wrote INT (inaudible) some expression (inaudible), it would work exactly the same way. It would have the same effect. But the workings inside were completely different. There was one particular file that sort of implemented the sort of basic functionality of types in the language. And in Python 1 it was, like, 50 lines of code. And at some point I rewrote it and it was 5000 lines of code.

CAROL: Wow. That's some (inaudible). What about types in Java?

JAMES: So I've got this long history of caring about...

[1:19:00]

JAMES: ...things like performance and building robust software, and often that comes out about... You know, I'm much more worried about what it takes to build production-quality software than about what it takes to just, like, do, like, a quick thing. And Java's not a great language for quick things, but it's... You know, for me one of the things that I love to do, and maybe I'm weird, but being able to do automatic theorem-proving on (hunks) of code. And a type system is a really great (part) to be one of the foundations of theorem-proving. And theorem-proving for hunks of code turns out to be really...

[1:20:00]

JAMES: ...useful for things like building off of (inaudible) compilers, and doing ahead of time correctness-jumping, trying to be able to theorem-prove away as many things as possible. You know, so, like, you know, one of the not-well-known things about Java is that, you know, in the Java spec, you know, it's array subscript checking is always on. But, you know, it's only conceptually always on, right? The truth is that there are - there's more than (enough hooks) for the compiler to theorem-prove away almost all index checks. And same thing with, like, NULL pointer checks and all kinds of things that look like they're heavyweight, but they're really...

[1:21:00]

JAMES: ...(cheap). You know, so one of the things that I really cared about at the time was that A+B should almost always be, you know, at most one instruction. Bonus points for zero instructions.

[LAUGHTER]

CAROL: [CROSSTALK] Makes it a lot faster.

JAMES: Well, the thing about zero instructions is often you can, like, fold them into some weird addressing (mode), right?

CAROL: Yeah.

JAMES: And so in kind of the universe I tend to live, being able to look at A+B and realize it's that instruction, that all feeds off of the type system. And sometimes you can do this with sort of optimistic (inaudible) (compilation), and depending on how far you want to push that. So, like, the...

[1:22:00]

JAMES: ...JavaScript engine in Chrome is absolutely astonishing for the kind of gore that they go through to optimistically compile JavaScript code. But it's also very hard to do those kinds of tricks if you're trying to get into small footprint devices, you know? So, you know, there are Java compilers that work for machines that only have, like, 50K.

CAROL: Right.

JAMES: And, you know, to do that kind of compaction and distillation you need every kind of (hook) that gives you every little - every last drop of information. And the earlier you know it the better a job you can do.

CAROL: Right. So Anders, speaking of lots of information that one gets, TypeScript gives you a lot of flexibility and power.

[1:23:00]

CAROL: What's your general view about typing?

ANDERS: Well, let me actually... It's funny that you mention (CPU) cycles and counting how many of whatever... I remember when I wrote Turbo Pascal, which was all written in Z80 assembly code - and back in those days you could literally look up the intel manual, or the Zilog, or whatever, you know, and see how many clock cycles every instruction took. And actually it would work out just like that.

CAROL: I remember that.

ANDERS: And now everything takes zero clock cycles except when it takes a thousand clock cycles. [LAUGHTER] When you hit the memory wall... And it is absolutely impossible to reason about how CPUs execute (your) code today. That's just one...

JAMES: Well, it's not impossible. It's just...

ANDERS: [CROSSATLK] No, it's not. But it's much harder.

JAMES: Oh yeah. But it is a complete pain in the ass.

ANDERS: Yes, it is.

JAMES: And they don't give you manuals that are good enough, right?

ANDERS: No - no - no.

JAMES: If you want to actually understand it you need to get the chip diagrams.

ANDERS: But with respect...

[1:24:00]

ANDERS: ...to types, I've always worked on programming languages that have type systems. But it's interesting how it's gone from being type systems for the code generator's sake, or type systems for, you know, generating errors, to now I almost view type systems as a tooling feature. And that's really sort of the thing that has been interesting (inaudible) bit for TypeScript, it is, you know... First of all, starting with a dynamically-typed programming language like JavaScript, and then trying to add a type system on top of it that faithfully reflects the semantics of the programming language... And the reason we're doing it is actually not because we think type systems are interesting, but because if you think about what it is that powers programmer productivity today... Like everyone loves their IDE, like, whatever IDE you're using. I hope it's Visual Studio Code.

[1:25:00]

ANDERS: But if it's not, you know, I'm sure you're all, like, accustomed to things like statement completion, and refactoring, and code navigation, and go to definition, and so forth. And if you think about it, the things that - the thing that powers that is semantic knowledge of your code. And the thing that provides the semantic knowledge of your code is a compiler with a type system. And once you add types you can actually dramatically increase productivity, which in some ways seems counterintuitive, right? I thought, like, dynamic languages were easier to approach because you got rid of the types, which was, like, a bother all the time. Well, it turns out that you can actually be more productive by adding types if you do it in a non-intrusive manner, and if you work hard on doing good type inference and so forth, you know?

JAMES: [CROSSTALK] So I wanted to jump in here.

CAROL: Let's let Larry jump in and then (inaudible).

[1:26:00]

LARRY: I'll probably take over the whole [CROSSTALK]...

JAMES: So I really, really, really believe in that point, in the power tools for power geeks thing. And one of the things that drives me nuts is the "real men use VI" movement. And, you know, it's really - I just want to punch people who are, like, "Well, I'm a real..."

CAROL: There is no violence on stage.

JAMES: ..."I'm a real developer because I use VI." And it's, like, you know...

ANDERS: "...because I do it the hard way."

JAMES: I think IDEs make language developers lazy.

[AUDIENCE RESPONSE]

CAROL: Wow. You should expand on that, Larry.

JAMES: IDEs let me get a lot more done a lot faster. I mean, I'm not - I'm really not into proving my manhood.

[1:27:00]

JAMES: [CROSSTALK] I'm into getting things done.

CAROL: Case and point. But I'm going to take the devil's advocate here for a second.

GUIDO: Emax.

CAROL: Something like - Emax, okay, yeah. The Jupyter Notebooks--a lot of science people - data scientists get a lot done in actually a pretty simplistic IDE with a dynamic language most of the time. So...

JAMES: They could get a lot more done.

[LAUGHTER]

CAROL: I don't know - I don't know.

ANDERS: No, but I think typeset - I mean, it's not just - a type system can help you. First of all, if you're uninitiated and you want to know, "Here's my (fu)," and now I say "(fu dot)," and that - the IDE can show you, "What can I type next?" right, as opposed to, "I've got to go read manuals and figure it out or know it all," right? I mean, the original developer of whatever piece of code might know it all. But then...

[1:28:00]

ANDERS: ...people move on, new people come, you know, there's - here's this project, there was no documentation written about it. And if you think about it, types are documentation, too, right? I mean, and there are just - there are so many things about, like - like, adding them that down the line give you increased productivity [CROSSTALK].

CAROL: Okay. So there's productivity and then there is...

ANDERS: But that's...

CAROL: ...maintainability.

ANDERS: Oh, well, both.

CAROL: And maybe any different thoughts about maintaining some...?

LARRY: I didn't get to talk about types yet.

CAROL: Oh, well, talk about types. You don't want to leave Perl types out.

LARRY: Well, the story is very different for Perl 5 and Perl 6, is the thing. As you know, Perl 5 just sort of grew over time. And the whole idea - it was sort of a ticklish idea. Everything - you can pretend everything is a string, even if it's a number or a (floating point) internally; it's just all, you know, interchangeable.

[1:29:00]

LARRY: And that works out for bootstrapping people to a language quite nicely. And it's a feature that we wanted to keep in Perl 6, but what we discovered in Perl 5 as part of the redesign was it's fine if the new user is confused about the interchangeability of (inaudible), to use a technical term, of your (inaudible) types. But it's not so good if the computer is confused about things. [LAUGHTER] You know, the Perl 1 engine was written way back in the dark ages when you had (inaudible) a bunch of stuff in, and it cheated a lot. So part of the redesign when we did the whole Perl 6 thing, we wanted to do object-oriented programming better than these languages, and we wanted it to be functional programming, better than most functional programming languages. To do that you have to have a fundamental, very sound type system...

[1:30:00]

LARRY: ...a sound (inaudible) object model underneath. And you really have to take seriously these slogans like, "Everything is an object. Everything is a closure. Everything..." You know, in Perl 6 even (inaudible) our closures. And you've got to optimize hard to get them away from there. But I - and I also agree with this idea that the types also are a cultural - I talked about pegs - hanging things on pegs. They're one of the pegs we didn't have something to hang on in Perl 5, and now we can, you know, hang all the - all the meta-information. And they could just run the little program. They don't even have to have an idea; they can say, "Oh, this object (inaudible)," you know, they could do their own introspection of the whole thing. And people just do that all the time. By the way, we do also have an IDE [LAUGHTER].

CAROL: Good points all around. So...

[1:31:00]

CAROL: ...most of the time programmers actually spend maintaining code versus writing new code out in the wild. And what things or elements of a programming language make it easier to maintain code? And maybe we'll start with Guido.

GUIDO: Well, I've found that - [BREAK/BLACKOUT] - TypeScript is actually incredibly useful, and so we're adding a very similar idea to Python. We're adding it in a slightly different way because we have a different (inaudible)... [MIC FREQUENCY] I'm sorry, I turned my chair because my neck was, like, turned (inaudible). Sorry.

CAROL: All right, it's the ghost.

[1:32:00]

GUIDO: Anyway, [CROSSTALK] I sort of learned a painful lesson that for small programs dynamic typing is great. For large programs you have to have a more disciplined approach. And it helps if the language actually gives you that discipline rather than telling you, "Well, you can do whatever you want."

LARRY: Yeah, that was part of our scale-up idea for Perl 6, that the types would help with programming in the large. Because we did notice both limitations of, you know, (loose) typing.

JAMES: Yeah, about 20 years ago I started working really heavily on refactoring engines, and, you know, being able to do, like, largescale refactoring on, like, millions of lines of code...

[1:33:00]

JAMES: ...at once. And that really changes the way you think about maintaining code. You know, because there's - you know, the world is filled with libraries that have things like stupid variable names, stupid method names. Because, you know, when you started out it meant one thing. But now you realize that you really didn't really understand what you were doing in the beginning, and so this - this method name really should be different. But nobody ever renames methods, because it's really hard to go over a piece of code and rename exactly the right variable. But if you've got a good refactoring engine you just type, you know, meta-control R, type in the new name. You know, hundreds and hundreds of files later--which takes about, you know, maybe 30 seconds if it's a lot of files--you're done.

[1:34:00]

GUIDO: And so you basically rewrite volume one of a seven-volume series.

[LAUGHTER]

ANDERS: No, but this is actually, like, (inaudible) you're describing... And (inaudible) was the genesis of the TypeScript project, which was these enormous JavaScript code pieces that we were seeing customers, right, and then in-house projects. And they were getting bigger and bigger because JavaScript engines were getting good enough for you to write really big programs. But maintaining them was impossible, because it turned into write-only code. You know, you dared not touch it once you had written it because it worked. And then - but, you know, there's this property called "text." And I really want to change it to be, I don't know, like, some other name. Except there is, like, a million other properties called "text." And if I just, like, do a global search and replace for "text," then, oh my God, it's, like, it all... So I can't change anything. But if you have semantic...

[1:35:00]

ANDERS: ...understanding of, like, "this property called 'text' is different from that other property over here called 'text,'" and if you want to rename this one that doesn’t mean you want to rename that one. But that takes for something like a type system to be in place. And, you know, once you start adding that you add documentation to the code and you add these capabilities where we're now seeing people, like, with regularity refactor hundreds of thousands of lines of JavaScript code, you know, like, in minutes. And it just works afterwards. And people are, like, amazed that it's possible. But it's true that it takes a bit more work to begin with. And it's maybe not right if you're writing five lines of code. Okay, that may be more bother than you really care to. But, you know, it starts to pay off pretty quickly.

LARRY: (Inaudible) (in addition) to types, really good lexical scoping helps with refactoring. It fits back into this...

[1:36:00]

LARRY: ..."what's the right peg to hang this on?" And often it's lexically-scoped or dynamically-scoped somehow. And they're doing those right and not interfering with threading and all sorts of interesting questions (inaudible).

CAROL: So I'm going to throw one more question out before the break in three minutes. And we made sort of a reference to Donald Knuth, one of the Turing winners, in this last section of conversation. And he wrote, "Premature optimization is the root of all evil." So what's your approach to optimizing code and gaining efficiency, say, in Python?

GUIDO: Well, I'm terrible at that so I leave it to others.

[LAUGHTER]

CAROL: That is spoken like a very wise language-creator.

GUIDO: And they've (lift) up to the task. I mean, Python's hash table implementation has been rewritten so many times...

[1:37:00]

GUIDO: ...I don't understand any part of it. [LAUGHTER] But it is so much better than the thing I actually copied out of Knuth 30 years ago.

CAROL: You copied it out of (Clue)?

LARRY: "Knuth."

CAROL: Oh, Knuth, oh okay.

JAMES: Yeah, so I kind of 50% believe that, because there are - you know, the premature optimization in terms of algorithms and code, that part of it I believe. But, you know, people often don't really think about the role of data structures in optimization. And if a data structure is exposed you have to - you know, and you decide that you, you know, this algorithm is a problem, but, you know, the reason that it's a problem is because, you know, this data structure was just slapped together...

[1:38:00]

JAMES: ...and they thought, oh, there'd only ever be, like, ten items in the list, or ten items in the set. And then you start, you know, having, you know, million-item sets all the time. And all of a sudden, you know, your little linked list is not going to hack it, right?

LARRY: Or (inaudible).

JAMES: Or - yeah. And so if you can - you know, it's, like, you know, hide early; hide often, right? Never tell the truth about what your data structures are.

[LAUGHTER]

CAROL: and on that note we're going to take... You want a quick (point)?

ANDERS: I just wanted to say that I think, like with anything, the important thing is to make sure that you actually have the right data before you start optimizing, right? But too often you have a hunch that, oh, I think if I...

[1:39:00]

ANDERS: ...need - I need to do this, you know? I need to change this data structure from being a linked list to being [CROSSTALK] or whatever. And - but if you imagine, then you discover it doesn't matter. And - but really there's this other thing that you haven't even thought about that matters enormously, and maybe you should go look over there. So get proof, get - use a profiler, figure out where is the time actually spent? I am continually surprised and I write compilers for a living, and I'm always surprised about where time is spent.

CAROL: All right. And that - with that we're out of time for this section. We're going to take a quick 15-minute break, and when we come back the really hard questions are going to come out, because the audience questions that have been submitted are going to be the ones that we move into. So thank you.

[CHEERING/APPLAUSE]

CAROL: [CROSSTALK] You guys have been wonderful.

[1:40:00]

[BEGIN PART 2]

[2:00:55]

M: ...each and every one of you has a community that is formed around the languages that you've created. I'm curious if you would talk to ways in which you've seen the design choices you've made for your language shape the communities that formed around those languages.

[CROSSTALK]

LARRY: It really helps if you're trying to make everybody - you know, what's your slogan here, "Everybody...?"

CAROL: "Everybody comfortable?"

LARRY: No.

CAROL: [LAUGHTER] "...feel valued and supported..."

LARRY: Make, you know, all Americans I heard, you know, feel, like, comfortable programming.

CAROL: Oh, CS for All, yes, to make...

LARRY: Yeah, that was it.

CAROL: Yes.

LARRY: I'm getting senile. But...

CAROL: The sign was a clue.

LARRY: ...it's not CS for All if it's just Americans.

CAROL: No.

LARRY: And there are many underserved...

[2:02:00]

LARRY: ...populations in the world. And what we've really discovered, that it really helps build international camaraderie and community to have world-class, literally, support for Unicode.

CAROL: Yes.

LARRY: So I would encourage any language designers out there, don't do this halfway stuff with, you know, code points. Go (inaudible).

CAROL: I completely agree. Guido, you had something you were going to say?

GUIDO: I was going to try and say that I (don't know) how my language design choices necessarily affected the community, but my choices about how to interact with communities certainly have affected the language design (inaudible) deeply. I was somehow imbued with the importance of user-centered language design...

[2:03:00]

GUIDO: ...because Python borrows a little bit from (inaudible) (ABC) (inaudible). Its authors had a bunch of things right, and one of those was listen to the users; ask the users what they think. And then (inaudible) suggested solutions and you first figure out for the program (list) (inaudible) you do give the users a voice. And I've sort of taken that out and let the users help me design the language. And now they're ready to take over.

ANDERS: I was going to say, I always felt like it's important as a language designer that you don't create unnecessary work for your community. Because it's very easy to, in the name of purity or whatever, go change something or strive for the perfect language. And I've always said, like, I mean...

[2:04:00]

ANDERS: ...show me the perfect language and I'll show you a language with no users, you know? Every language has imperfections in it, and it has them because it has evolved. And the hard part of language design is actually not version 1-0 - it's, like, all the versions that come after, where you're trying to sneak new things in without actually causing extra burdens on people who don't necessarily care about the things that you're trying to sneak in, right? And so that's the hard part of language design I think is the evolution of the language and keeping your community with you as you evolve.

JAMES: Yeah, and so we Java we, like, always followed that. We've maybe been a little bit excessive about it. I mean, I've got binaries that I compiled, like, twenty years ago...

[2:05:00]

JAMES: ...that still run. And it's, you know, on machines that were never invented when the code was compiled. And you only get there with crazy discipline and a tolerance for living with your mistakes. But it makes the - you know, it also ends up selecting who your users are, because in the Java universe pretty much everybody is really disciplined. Beca- you know, it's kind of like mountain climbing. You know, you don't get - dare get sloppy with your gear when you're mountain climbing, because it has a clear price. And, you know, when - you said something about - Larry, you said something about Unicode. Java's had Unicode since '92.

[2:06:00]

JAMES: So, you know, it's - you know, trying to be inclusive... [CROSSTALK] Yeah. And that really made a difference. I mean, lots of folks thought it was, like, the lamest thing ever. But that was during my, you know, Lord of Java phase, which...

LARRY: That was (a great decision).

JAMES: Yeah, but that phase of my life ended in about '95.

LARRY: I'd like to say something about the stability thing and as it relates to community. And that is we did this perfect - tried to do perfect backward compatibility thing all the way up through Perl 5 and did a really good job of it. It's one of the most stable things... There's still Perl 1 owners that (inaudible). That eventually runs into the problem that, you know, you can't (inaudible). You can't fix your mistakes. So one of the mistakes that we made was kind of a meta mistake...

[2:07:00]

LARRY: ...and that was assuming that you had to always have that kind of stability and not change anything. How would it be to design a language where the language itself could be evolved from within by the community and be the language we want 25 or 50 years from now? This ties into what Paul Graham was talking about, a hundred-year language. And we kind of took that seriously--what is it that prevents languages from evolving, not having things (inaudible) right? But fundamentally the Perl 6 compiler is written in Perl 6. Most of the runtime system is written in Perl 6. The users can extend it. Perl 6 itself does not care whether things are built in or not; it all works the same. So most of the built-ins are written as if they were user code. So users, even if...

[2:08:00]

LARRY: ...we say, "No, we don't want that in the language right now, but put it in a module," you can turn classes into monitors with about that much code. You can implement an actor model in classes with about that much code. You can mutate - adding operators is just trivial. When it does, it does everything perfectly one pass. It drops into the sub-compiler. And with the end it comes back out. So there's never any ambiguity as to what language you are in. And so because the scoping of the exact language here, and it lexically-scoped, any lexical scope can say, "Use whatever language I want." A future version of Perl 6, you could say, "Use COBOL, use Python." If someone - well, actually people have written Python parsers in Perl 6. I don't think anybody's written a COBOL parser yet.

[2:09:00]

[LAUGHTER]

LARRY: But we'd like to think that, you know, all languages are sub-languages of Perl 6 now. So I think that [LAUGHTER] - I think that they're... (Inaudible) semantics, which are hard. I agree semantics are hard. But I think there is a way forward. [CROSSTALK] I think there is a way forward to get the stability and the evolution, and I think we're trying for that.

CAROL: Awesome. Do we want to take another audience question? All right. Thanks.

M: Hi.

CAROL: Can you talk into the mic like you're eating the mic?

M: Hello? Hi. Yeah, can you hear me?

CAROL: Yeah.

M: Great. There's been a lot of talk tonight about the evolvability of languages and the trouble with implementing things that might not be backwards-compatible. What's something that you wish you could have implemented with our language, but for that...

[2:10:00]

M: ...or maybe another reason it wasn't possible?

GUIDO: Well, I have a (inaudible) I am sort of jealous of because it's appearing in more and more other languages. It's (inaudible)-matching. And I cannot come up with the right keyword because all the interesting keywords are already very popular method names for other forms of (inaudible)-matching.

CAROL: Naming's hard.

ANDERS: Yeah. Well, I'll go. My favorite is always, like, the $2 billion mistake of - or the billion-dollar mistake of having NULL in the language. And since JavaScript has both NULL and undefined it's the $2 billion mistake. But, I mean, if - and, I mean, what's done is done, right? And now...

[2:11:00]

ANDERS: ...you know, I spend a significant portion of my time actually working on ways to make code NULL-safe, to... And who in here has ever had a NULL pointer exception, right? I mean, there you go. It is by far the most problematic part of language design. And it's a single value that [LAUGHTER] if only that wasn't there, imagine all the problems we wouldn't have, right, if type systems (were) designed that way. And some type systems are, and some type systems are getting there. But, boy, trying to retrofit that on top of a type system that has NULL in the first place is quite an undertaking.

LARRY: The features I wanted to add were negative features. I think all of us as language designers have borrowed things... You know, we all steal from each other's languages all the time. And...

[2:12:00]

LARRY: ...often we steal good things. But for some reason we also steal bad things. You know...

CAROL: Like what?

LARRY: Like regular-expression syntax.

CAROL: Oh, yeah. [LAUGHTER] I'll give you that one.

LARRY: Like the C-precedence table.

CAROL: Ah, okay, another.

LARRY: Okay. These are things that I could not fix in Perl 5, and we did fix in Perl 6. Because...

CAROL: Oh yay, awesome. All right, we do seem to be doing pretty well with the audience. So how about another audience question?

M: Hi, hello there. So, well, what I wanted to ask is you can serve notice these sort of (inaudible) effect in regards to tech and programming throughout time to (inaudible) different paradigms. There's a certain time where people were, like, "Are we going to do things (imperatively)? Are we going to go..."

[2:13:00]

M: "...object-oriented or functional?" And for example, right now there's a whole bunch of languages that are sort of, like, very aggressively taking that standpoint of being very pro-being friendly with concurrency or being very over-jealous about (inaudible) with memory. And I think that that kind of pendulum effect happens because technology has actually been evolving throughout time. Our machines are, like, beefier and we have more memory now. So the ways that we designed programming languages now are probably a lot different than they were 20 or 30 years ago. So considering that, where do you think, or how do you think the programming languages ten years into the future are going to look like? Or in your opinion where is language design headed to, like, in the future?

JAMES: So my favorite candidate for that...

[2:14:00]

JAMES: ...is that, you know, these, you know, major breaks and things like that always happen because, you know, something major happens in the underlying technology. And, you know, one of the areas of underlying technology that is kind of like a computer that I think is really underserved is writing code for GPUs, right? I mean, right now there are no languages worth a crap--and that's a technical (term), trademark [LAUGHTER]--that work well on GPUs. And, you know, maybe because there's, like, a limited number of algorithms that people are willing to - are really interested in, but they're really important ones. Or maybe it's because they're inherently all mathematical and the number of people who care about writing numerical code...

[2:15:00]

JAMES: ...is small. I'm one of those though, and - you know? And so every now and then I spend time thinking about, you know, what would you do to make GPU programming really easy?

LARRY: We need a champion in Perl 6 to hook up our (vector) (inaudible) to the GPUs.

JAMES: Yeah, but it's more than just vector operators. I mean, that's kind of my problem with the whole thing is that most of these things are just libraries of hand-coded assembly that do vector operators. And surely there's something more clever than that.

ANDERS: Maybe I'll just add with language design, you know, one of the things that's interesting, you look at, like, all of us old geezers sitting up here, and we're proof-positive that languages move slowly. And, I mean, it's not, you know, like - a lot of people make the mistake of thinking that languages move...

[2:16:00]

ANDERS: ...at the same speed as, like, hardware or, like, all of the other technologies that we live with. But languages are much more like math, and much more like the human brain, and, you know, and it all evolves slowly. And we're still programming in languages that were invented 50 years ago. Like functional - all of the principles of functional programming were thought of more than 50 years ago. I do think one of the things that is luckily happening is that, like as Larry said, everyone's borrowing from everyone. And languages are becoming more multi-paradigmed. Yeah, I think it's wrong to talk about, "Oh, I only like object-oriented programming languages," or "I only like imperative-programming," or "functional-programming." I mean, it's important to look at where is the research, or where are the new - where's the new thinking and where are new paradigms that are interesting, and then try to incorporate them, but do so tastefully in a sense, and work them into whatever is there already.

[2:17:00]

ANDERS: And I think we're all learning a lot from functional-programming languages these days; I certainly feel like I am. Because a lot of interesting research has happened there. But functional programming is imperfect, and no one writes pure functional programs. I mean, because they don't exist. So it's all about, like, how can you tastefully sneak in mutation in ways that you can better reason about, as opposed to mutation and free-threading for everyone, you know? And that's, like, just a recipe for disaster, right? So...

M: (Inaudible) on the functional (inaudible)?

[LAUGHTER]

ANDERS: All languages are imperfect, let's just settle it right there, you know? [LAUGHTER]

M: (Inaudible) all languages, I think in the late '70s I heard a saying that said, "We don't know what the language of the future will look like, but we know what it will be called, and it'll be called 'Fortran.'"

[LAUGHTER]

[2:18:00]

M: I started (wanting to take that) as a (humbling) lesson.

LARRY: Yes. We've been thinking a lot about the parallels and concurrency issue. This is one of those issues where we were waiting to be smarter. We did borrow, I mean, steal, a bunch of ideas from Pascal in terms of lazy lists and things like that. But in terms of how you actually are going to deal with the end of Moore's Law--or if not the end of Moore's Law, at least, you know, multi CPUs, many - you know, many (cores)--when you've got a thousand (cores) what are you going to do with them? And so a lot of our early design on Perl 6 was - we didn't understand how we wanted to do it yet, but we knew very - we knew very deeply that we didn't want...

[2:19:00]

LARRY: ...to do anything that would prevent it. So a lot of the early design decisions were, you know, just saying, "Well, we have to keep this open." Then when the right smart people came along they said, "Look, what you really have to look at is things that are composable." Threads are not composable--you can't take two threads and turn it into a third thread. What you want is things like we stole from Go--we stole promises and channels. From C# we stole functional-reactional programming. And with these sort of primitives, which end up being duals of your regular list-processing kind of primitives--they just happen to be working on events--and you can have loops - event loops that work just like regular loops, except they're running on... And you - same control flow, so that is easy to learn. There are ways of sneaking these things into a language, at least in the ways that we understand right now, that...

[2:20:00]

LARRY: ...I heard, was it - who was it talking about adding fibers to their runtime? I think it was a Java news item. But lightweight threads that can - you - as Erlang has done - has shown, you know, you can run 100,000 threads in your process and...

CAROL: That's another language that's been around for a long time.

LARRY: Oh yeah, and we stole from them, too. We like their pattern-matching. But...

JAMES: Well, one of the meta-theorems is that all good language design is theft.

LARRY: Of course. So, yeah, I mean, we're thinking about it. We don't know - have all the answers yet. But I think all these languages, you know, long-term tend to converge on similar solutions.

CAROL: So very sadly we are at the last question. I know, it went so quickly, and unfortunately all good things must end, except for maybe good languages.

[2:21:00]

CAROL: So I'm going to start and maybe quickly wrap up. As you look back over the long lives of the languages that you've written, what have you found most rewarding?

LARRY: The people.

CAROL: The people?

LARRY: By far.

[APPLAUSE]

ANDERS: Yeah, absolutely. Oh, I - yeah. Yeah, I mean, that's what keeps me coming back to work every day is, like, the incredible excitement that the community shows you. And I am so grateful for all the people who have used the... I mean, there's nothing more rewarding than have people just, like, on Twitter or wherever it is go, "Oh my God, this is so great." And it's, like, yeah - that makes you want to just keep doing more of it. I mean, it's...

JAMES: Oh yeah, every - you know, when somebody comes up to me, you know, in the street somewhere and says, you know, "Thanks for giving me a career. Can I have a selfie?" [LAUGHTER]...

[2:22:00]

JAMES: ...you know, that's, like - that's the kind of, like, you know, light-and-dark thing. But...

LARRY: Being famous is not all...

JAMES: ...it's really wonderful.

LARRY: ...being famous is not all great. I mean, I've ha- it's happened to me in Silicon Valley at a movie theater or the tire shop--"You're Larry Wall."

JAMES: Yeah, well, what's really weird is when your kids are with you when that happens. [LAUGHTER]

LARRY: But yes, it's when the - when people said, "You gave me a career. You gave me a livelihood. You fed my family." That's the best.

[CROSSTALK]

LARRY: And not just - we're not talking about a star-relationship here, of each of us to those individuals; we're talking about the second-order community 2.0 kind of effects. It's one thing to see them, you know, getting help from you; it's another thing to see them helping each other.

[2:23:00]

LARRY: That's a whole level above even the individual relationships I feel, to see a community that is learning how to love, you know, kind of building the little bit of heaven on earth. There's just nothing like it.

[APPLAUSE]

CAROL: Completely agree. Guido, you get the last word.

GUIDO: I love learning from (inaudible).

CAROL: And we love learning from you as well. And with that I want to say we've all learned so much from all of our wonderful panelists tonight. I want to thank you for allowing me to share and celebrate this moment with you. The little fifth-grade kid in me that was plinking away on BASIC at Bell Labs, who later watched the PC cell phone, internet, web and Cloud computing change our world, never dreamed that I would be chatting with four individuals who profoundly impacted our world by doing what they...

[2:24:00]

CAROL: ...love, creating and building languages. Please join me in a huge round of applause to thank Anders, James, Larry, and Guido for inspiring this evening.

[CHEERS/APPLAUSE]

ANDERS: And thank you, Carol.

[2:24:22]

END (LANGUAGE-CREATORS.MP4)