OPEN SOURCE YEARBOOK 2018

opensource.com/yearbook

Opensource.com publishes stories about creating, adopting, and sharing open source solutions. Visit Opensource.com to learn more about how the open source way is improving technologies, education, business, government, health, law, entertainment, humanitarian efforts, and more.

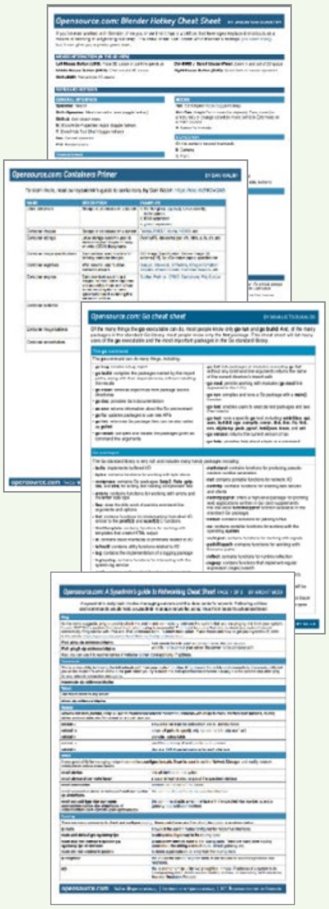
Submit a story idea: https://opensource.com/story

Email us: open@opensource.com

SUPPORTED BY RED HAT

Open Source Cheat Sheets

Visit our cheat sheets collection for free downloads, including:



Opensource.com: Blender Hotkey Cheat Sheet

Opensource.com: Containers Primer

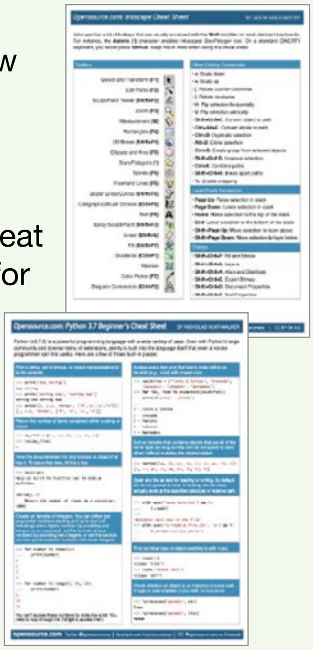
Opensource.com: Go cheat sheet

Opensource.com: A Sysadmin's guide to Networking Cheat Sheet

Blender: Discover the most commonly and frequently used hotkeys and mouse button presses.

Containers: Learn the lingo and get the basics in this quick and easy containers primer.

Go: Find out about many uses of the go executable and the most important packages in the Go standard library.



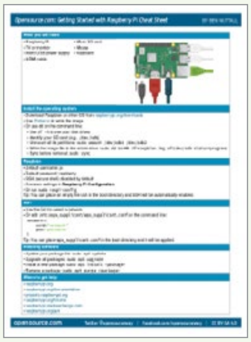
Opensource.com: Inkscape Cheat Sheet

Opensource.com: Python 3.7 Beginner's Cheat Sheet

Inkscape: Inkscape is an incredibly powerful vector graphics program that you can use to draw scaleable illustrations or edit vector artwork that other people have created.

Linux Networking: In this downloadable PDF cheat sheet, get a list of Linux utilities and commands for managing servers and networks.

Python 3.7: This cheat sheet rounds up a few built-in pieces to get new Python programmers started.



Opensource.com: Getting Started with Raspberry Pi Cheat Sheet

Raspberry Pi: See what you need to boot your Pi, how to install the operating system, how to enable SSH and connect to WiFi, how to install software and update your system, and links for where to get further help.

Opensource.com: Advanced SSH Cheat Sheet

SSH: Most people know SSH as a tool for remote login, which it is, but it can be used in many other ways.

We're looking for contributors

Here's how to write for Opensource.com.

Opensource.com publishes stories about creating, adopting, and sharing open source solutions. In 2018, we published more than 1,000 articles by talented people in tech from diverse backgrounds and with a variety of skills.

We welcome article proposals and submissions from subject matter users or experts on a range of topics [1] about open source. To view our editorial calendar and other writer resources, visit: opensource.com/writers [2]

- Alternatives: Open source vs. proprietary
- Automation: Ansible, Bash, Perl, and others
- Command line: Tools, tutorials, and tips for Linux
- Data science: Python and more
- DevOps: Tools and lessons learned
- Hardware: Tutorials for the Raspberry Pi, Arduino, and more
- IoT (Internet of Things): Home automation projects and more
- Java: Tutorials, tips, tools
- Linux: Getting started with a Linux command or distro
- Machine learning / AI: How to get started with a tool
- Multimedia: Music, audio, video, editing, 3D rendering, and more
- Networking: Tutorials, introductions, and recommendations
- Personal stories: Open source / Linux experiences, tips, and more
- Productivity tools: Linux, open source, and more
- Programming: Go, TensorFlow, Perl, JavaScript, MySQL, Rust, and more
- Python: Tutorials, tips, tools
- Systems administration: Sysadmin tools, tips, and more
- Tools: Tell us what it's for, how it works, and where to learn more
- You tell us: What's important for people in tech to know more about?

To send us your article idea or draft for review, use our webform [3] or email [4].

Before sending in your proposal, review our article submission and style guide [5] for best practices. If you are writing a technical or how-to article, review our technical style guide [6].

The Opensource.com team provides copy editing and lead images for your article. We also promote articles on our social media channels. If you haven't written before, learn more about how writing can change your career [7] for the better.

Have more questions? Email us at open@opensource.com.

Links

[1] https://opensource.com/article/19/1/write-for-us

[2] https://opensource.com/writers

[3] https://opensource.com/how-submit-article

[4] Email: open@opensource.com

[5] https://opensource.com/submission-style-guide

[6] http://stylepedia.net/

[7] https://opensource.com/life/15/7/7-big-reasons-contribute-opensourcecom

WRITE FOR US

7 big reasons to contribute to Opensource.com:

1 **Career benefits:** "I probably would not have gotten my most recent job if it had not been for my articles on Opensource.com."

2 **Raise awareness:** "The platform and publicity that is available through Opensource.com is extremely valuable."

3 **Grow your network:** "I met a lot of interesting people after that, boosted my blog stats immediately, and even got some business offers!"

4 **Contribute back to open source communities:** "Writing for Opensource.com has allowed me to give back to a community of users and developers from whom I have truly benefited for many years."

5 **Receive free, professional editing services:** "The team helps me, through feedback, on improving my writing skills."

6 **We're loveable:** "I love the Opensource.com team. I have known some of them for years and they are good people."

7 **Writing for us is easy:** "I couldn't have been more pleased with my writing experience."

Email us to learn more or to share your feedback about writing for us: https://opensource.com/story

Visit our Participate page to more about joining in the Opensource.com community: https://opensource.com/participate

FOLLOW US

Stay up on what's going on with Opensource.com by subscribing to our highlights newsletter: https://opensource.com/email-newsletter

Twitter @opensourceway: https://twitter.com/opensourceway

Facebook: https://www.facebook.com/opensourceway

FROM THE EDITOR

Dear Open Source Yearbook reader,

In 2018 we celebrated 20 years of “open source” and the 20-year anniversary of the Open Source Initiative (OSI). On Opensource.com, we were excited to join the celebration with an article by Christine Peterson, who coined the phrase “open source software,” and I think it’s fitting that we kick off the annual yearbook with this story. Then Opensource.com managing editor Jen Wike Huger rounds up 30 Linux installation tales in which our readers and writers tell their tales.

Throughout this issue, you’ll notice that old favorite technologies, including Python, Bash, Unix, GNOME, and Slackware, are still relevant today, while also leaving room for newer technologies and trends, including Flutter, Kubernetes, Raspberry Pi, AI, and serverless computing.

The past year also saw a lot of changes in open source communities. Diversity and inclusion efforts continued expanding, which is illustrated throughout these pages with articles on how to welcome newcomers, community metrics, how programmers in underrepresented countries can get ahead, gracefully receiving and giving code feedback, and a new film series that highlights women in technology.

We wrap up the 2018 Open Source Yearbook with a look back at pivotal moments in open source history, the anniversaries of Git and GNOME, an insider’s look at drafting the GPLv3 license, and 25 years of Slackware, and then we look ahead at 2019 conferences and resolutions for open source project maintainers.

In 2018, Opensource.com published 1,075 articles and welcomed more than 250 new writers. The annual Open Source Yearbook offers only a small snapshot of the larger open source story, and we’re not able to fit all the hundreds of articles and writers into these few pages.

As we begin our ninth year of Opensource.com, our team thanks our writers, readers, moderators, and community for sharing the stories, tools, and solutions that make up our wild and wonderful open source world.

Would you like to be part of Opensource.com? We’ll help you get started: http://opensource.com/story

Best regards,
Rikki Endsley
Opensource.com community manager

CONTENTS

WORKING

COLLABORATING

Best Couple

LEARNING

CREATING

OLD SCHOOL

FUTURE

All lead images by Opensource.com or the author under CC BY-SA 4.0 unless otherwise noted.

How I coined the term ‘open source’

BY CHRISTINE PETERSON

Christine Peterson finally publishes her account of that fateful day, 20 years ago.

FEBRUARY 3, 2018, was the 20th anniversary of the introduction of the term [1] “open source software.” As open source software grows in popularity and powers some of the most robust and important innovations of our time, we reflect on its rise to prominence.

I am the originator of the term “open source software” and came up with it while executive director at Foresight Institute. Not a software developer like the rest, I thank Linux programmer Todd Anderson for supporting the term and proposing it to the group.

This is my account of how I came up with it, how it was proposed, and the subsequent reactions. Of course, there are a number of accounts of the coining of the term, for example by Eric Raymond and Richard Stallman, yet this is mine, written on January 2, 2006.

It has never been published, until today.

The introduction of the term “open source software” was a deliberate effort to make this field of endeavor more understandable to newcomers and to business, which was viewed as necessary to its spread to a broader community of users. The problem with the main earlier label, “free software,” was not its political connotations, but that—to newcomers—its seeming focus on price is distracting. A term was needed that focuses on the key issue of source code and that does not immediately confuse those new to the concept. The first term that came along at the right time and fulfilled these requirements was rapidly adopted: open source.

This term had long been used in an “intelligence” (i.e., spying) context, but to my knowledge, use of the term with respect to software prior to 1998 has not been confirmed. The account below describes how the term open source software [2] caught on and became the name of both an industry and a movement.

Meetings on computer security

In late 1997, weekly meetings were being held at Foresight Institute to discuss computer security. Foresight is a nonprofit think tank focused on nanotechnology and artificial intelligence, and software security is regarded as central to the reliability and security of both. We had identified free software as a promising approach to improving software security and reliability and were looking for ways to promote it. Interest in free software was starting to grow outside the programming community, and it was increasingly clear that an opportunity was coming to change the world. However, just how to do this was unclear, and we were groping for strategies.

At these meetings, we discussed the need for a new term due to the confusion factor. The argument was as follows: those new to the term "free software" assume it is referring to the price. Old-timers must then launch into an explanation, usually given as follows: "We mean free as in freedom, not free as in beer." At this point, a discussion on software has turned into one about the price of an alcoholic beverage. The problem was not that explaining the meaning is impossible—the problem was that the name for an important idea should not be so confusing to newcomers. A clearer term was needed. No political issues were raised regarding the free software term; the issue was its lack of clarity to those new to the concept.

Releasing Netscape

On February 2, 1998, Eric Raymond arrived on a visit to work with Netscape on the plan to release the browser code under a free-software-style license. We held a meeting that night at Foresight's office in Los Altos to strategize and refine our message. In addition to Eric and me, active participants included Brian Behlendorf, Michael Tiemann, Todd Anderson, Mark S. Miller, and Ka-Ping Yee. But at that meeting, the field was still described as free software or, by Brian, "source code available" software.

While in town, Eric used Foresight as a base of operations. At one point during his visit, he was called to the phone to talk with a couple of Netscape legal and/or marketing staff. When he was finished, I asked to be put on the phone with them—one man and one woman, perhaps Mitchell Baker—so I could bring up the need for a new term. They agreed in principle immediately, but no specific term was agreed upon.

Between meetings that week, I was still focused on the need for a better name and came up with the term "open source software." While not ideal, it struck me as good enough. I ran it by at least four others: Eric Drexler, Mark Miller, and Todd Anderson liked it, while a friend in marketing and public relations felt the term "open" had been overused and abused and believed we could do better. He was right in theory; however, I didn't have a better idea, so I thought I would try to go ahead and introduce it. In hindsight, I should have simply proposed it to Eric Raymond, but I didn't know him well at the time, so I took an indirect strategy instead.

Todd had agreed strongly about the need for a new term and offered to assist in getting the term introduced. This was helpful because, as a non-programmer, my influence within the free software community was weak. My work in nanotechnology education at Foresight was a plus, but not enough for me to be taken very seriously on free software questions. As a Linux programmer, Todd would be listened to more closely.

The key meeting

Later that week, on February 5, 1998, a group was assembled at VA Research to brainstorm on strategy. Attending—in addition to Eric Raymond, Todd, and me—were Larry Augustin, Sam Ockman, and attending by phone, Jon "maddog" Hall.

The primary topic was promotion strategy, especially which companies to approach. I said little, but was looking for an opportunity to introduce the proposed term. I felt that it wouldn't work for me to just blurt out, "All you technical people should start using my new term." Most of those attending didn't know me, and for all I knew, they might not even agree that a new term was greatly needed, or even somewhat desirable.

Fortunately, Todd was on the ball. Instead of making an assertion that the community should use this specific new term, he did something less directive—a smart thing to do with this community of strong-willed individuals. He simply used the term in a sentence on another topic—just dropped it into the conversation to see what happened. I went on alert, hoping for a response, but there was none at first. The discussion continued on the original topic. It seemed only he and I had noticed the usage.

Not so—memetic evolution was in action. A few minutes later, one of the others used the term, evidently without noticing, still discussing a topic other than terminology. Todd and I looked at each other out of the corners of our eyes to check: yes, we had both noticed what happened. I was excited—it might work! But I kept quiet: I still had low status in this group. Probably some were wondering why Eric had invited me at all.

Toward the end of the meeting, the question of terminology [3] was brought up explicitly, probably by Todd or Eric. Maddog mentioned "freely distributable" as an earlier term, and "cooperatively developed" as a newer term. Eric listed "free software," "open source," and "sourceware" as the main options. Todd advocated the "open source" model, and Eric endorsed this. I didn't say much, letting Todd and Eric pull the (loose, informal) consensus together around the open source name. It was clear that to most of those at the meeting, the name change was not the most important thing discussed there; a relatively minor issue. Only about 10% of my notes from this meeting are on the terminology question.

But I was elated. These were some key leaders in the community, and they liked the new name, or at least didn't object. This was a very good sign. There was probably not much more I could do to help; Eric Raymond was far better positioned to spread the new meme, and he did. Bruce Perens signed on to the effort immediately, helping set up Opensource.org [4] and playing a key role in spreading the new term.

For the name to succeed, it was necessary, or at least highly desirable, that Tim O'Reilly agree and actively use it in his many projects on behalf of the community. Also helpful would be use of the term in the upcoming official release of the Netscape Navigator code. By late February, both O'Reilly & Associates and Netscape had started to use the term.

Getting the name out

After this, there was a period during which the term was promoted by Eric Raymond to the media, by Tim O'Reilly to business, and by both to the programming community. It seemed to spread very quickly.

On April 7, 1998, Tim O'Reilly held a meeting of key leaders in the field. Announced in advance as the first "Freeware Summit," [5] by April 14 it was referred to as the first "Open Source Summit." [6]

These months were extremely exciting for open source. Every week, it seemed, a new company announced plans to participate. Reading Slashdot became a necessity, even for those like me who were only peripherally involved. I strongly believe that the new term was helpful in enabling this rapid spread into business, which then enabled wider use by the public.

A quick Google search indicates that "open source" appears more often than "free software," but there still is substantial use of the free software term, which remains useful and should be included when communicating with audiences who prefer it.

A happy twinge

When an early account [7] of the terminology change written by Eric Raymond was posted on the Open Source Initiative website, I was listed as being at the VA brainstorming meeting, but not as the originator of the term. This was my own fault; I had neglected to tell Eric the details. My impulse was to let it pass and stay in the background, but Todd felt otherwise. He suggested to me that one day I would be glad to be known as the person who coined the name "open source software." He explained the situation to Eric, who promptly updated his site.

Coming up with a phrase is a small contribution, but I admit to being grateful to those who remember to credit me with it. Every time I hear it, which is very often now, it gives me a little happy twinge.

The big credit for persuading the community goes to Eric Raymond and Tim O'Reilly, who made it happen. Thanks to them for crediting me, and to Todd Anderson for his role throughout. The above is not a complete account of open source history; apologies to the many key players whose names do not appear. Those seeking a more complete account should refer to the links in this article and elsewhere on the net.

Links

[1] https://opensource.com/resources/what-open-source
[2] https://opensource.org/osd
[3] https://wiki2.org/en/Alternative_terms_for_free_software
[4] https://opensource.org/
[5] http://www.oreilly.com/pub/pr/636
[6] http://www.oreilly.com/pub/pr/796
[7] https://ipfs.io/ipfs/QmXoypizjW3WknFiJnKLwHCnL72vedxjQkDDP1mXWo6uco/wiki/Alternative_terms_for_free_software.html
[8] http://intelligence.org/
[9] http://blueribbonnano.org/
[10] https://nasasearch.nasa.gov/search?query=nanotech+briefs&affiliate=nasa&utf8=%E2%9C%93/
[11] https://www.foresight.org/Conferences/index.html
[12] https://www.foresight.org/about/fi_spons.html
[13] https://www.foresight.org/SrAssoc/spring2002/index.html
[14] http://www.oreilly.com/openbook/freedom/ch11.html
[15] https://www.foresight.org/UTF/Unbound_LBW/index.html
[16] https://www.foresight.org/SrAssoc/99Gathering/lta_toc.html

Author

Christine Peterson writes, lectures, and briefs the media on coming powerful technologies, especially nanotechnology, artificial intelligence, and longevity. She is cofounder and past president of Foresight Institute, the leading nanotech public interest group. Foresight educates the public, technical community, and policymakers on coming powerful technologies and how to guide their long-term impact.

She serves on the Advisory Board of the Machine Intelligence Research Institute [8], and has served on California's Blue Ribbon Task Force on Nanotechnology [9] and the Editorial Advisory Board of NASA's Nanotech Briefs [10].

She has often directed Foresight Conferences on Molecular Nanotechnology [11], organized Foresight Institute Feynman Prizes [12], and chaired Foresight Vision Weekends [13].

She lectures on technology topics to a wide variety of audiences, focusing on making complex fields understandable.

Her work is motivated by a desire to help Earth's environment and traditional human communities avoid harm and instead benefit from expected dramatic advances in technology. This goal of spreading benefits led to an interest in new varieties of intellectual property including open source software [14], a term she is credited with originating.

Wearing her for-profit hat, she chairs the Personalized Life Extension Conf erence series. In 1991 she coauthored *Unbounding the Future: the Nanotechnology Revolution* (Morrow, full text online [15]), which sketches nanotechnology's potential environmental and medical benefits as well as possible abuses. An interest in group process led to coauthoring *Leaping the Abyss: Putting Group Genius to Work* (knOwhere Press, 1997, full text online [16]) with Gayle Pergamit.

First time with Linux: 30 installation tales

BY JEN WIKE HUGER

The Linux kernel turns another year older on August 25.

THE LINUX [1] kernel turned another year older on Saturday, August 25. Twenty-six years ago it may have felt to the creator and BDFL [2] Linus Torvalds that Linux would only amount to satisfying the needs of one. But today we know it has changed the lives of many.

To celebrate, thirty of our readers share what their first Linux distro and installation was like. Some of their stories are magical, some maniacal. And, it's no surprise that the tension and passion of these Linux lovers is palpable.

Read on for their stories.

30 firsts with Linux

Gentoo

Steve Ovens [3] writes:

My first Linux kernel version was 2.6.3. It was Gentoo with Gnome 2. It took more than four days to compile on my computer at the time. I prayed there wasn't a power outage or failure halfway through. I remember spending all that time compiling the OS and getting to the desktop and thinking "Great! Now what?" Having used only Windows previous to this adventure with Gentoo, so I didn't really understand how getting software worked. I couldn't just download packages, and I wasn't particularly motivated to do anything in-depth. The system only lasted a few months before I gave up. I would come back Linux on 2.6.8 with Ubuntu 4.10 Warty Warthog. With this release, Linux gained a permanent foothold in my house, and it would eventually launch its domination of my infrastructure and eventually my personal computers. By 2006, I was running Linux on everything I could.

I was running Linux on everything I could.

Linux

Matthew Helmke [4] writes:

I installed Linux 2.2 on a server I did not own, and version 2.6 was the first that I successfully installed myself. My installation involved lots of invocations, muttering, driver source downloads, and compilation. In retrospect, I learned it was because I made a poor hardware choice at the time. My second was a breeze.

My second install was a breeze.

Jet Anderson [5] writes:

It was 1992 if I remember correctly. I had a spare box of parts sitting near my desk in a design company where I supported their fleet of Macs. I paid about $60 for the book "Yggradsil Computing 'Plug-n-Play'

How hard can this Linux thing be anyway?

Linux" with accompanying bootable installer CD. I thought... "How hard can this Linux thing be anyway? Besides, it's bootable!" So, I put in a graphics card, a hard drive (40 MB if I remember correctly), and installed some ram... a whole 4 MB. That should be plenty right? Everything went downhill from there. The graphics card was unsupported. Did I mention the drives were SCSI not IDE? Thankfully I had ethernet based access to the internet or I'd have been in the sad world of trying to configure a modem to download updated drivers for everything. After about a week of wrestling, I finally got the drive formatted, kernel installed, and X running. It felt like an incredibly heroic thing I'd just accomplished. Sitting there with my very first terminal window open I typed "dir" and got... "dir: command not found." It was time to start learning.

Steve Ellis [6] writes:

My first Linux install was 1.3 back in 1996 using the DR1 release of MkLinux on an early PowerPC 601 based Power Macintosh. The installer was remarkably like early versions of Red Hat Linux. We were given a CD at Apple's 1996 WWDC and it took me a couple of months to persuade my boss that I wouldn't totally destroy one of our developer workstations trying it out. Suddenly, I had a Linux/Unix workstation that was faster than our aging RS/6000 and pretty much never looked back from a Linux adoption perspective.

Gary Smith [7] writes:

My first Linux install was Yggdrasil Linux Plug and Play, in 1993. I still have the CDs for nostalgia's sake.

PowerPC Linux

John Anderson [8] writes:

My first Linux install was PowerPC Linux onto a PowerMac 7500. I downloaded the install media in my University lab and carried it home on a stack of zip disks, and I set up the PowerMac to dual-boot MacOS 9 and PPC Linux. I only had the one computer, and the thing I remember most is how incredibly overjoyed I was when I got PPP dial-up working on the Linux side so I could search for stuff without having to reboot back into MacOS!

I carried it home on a stack of zip disks.

Red Hat Linux

Don Watkins [9] writes:

My first Linux install was Red Hat 5.0. I don't know the kernel version. I think it came on floppies. I could not get a GUI and didn't like it. I thought it was like MS-DOS. The second install was with a CD on Red Hat 6.1, and that was kernel version 2.2.12-20, according to Wikipedia. I liked the install experience, and it was my first successful install with a GUI. I think it was GNOME, but I'm not sure.

Dave Neary [10] writes:

The first kernel I remember compiling was 2.0.32. It was on a Red Hat Linux 5.0, and I needed to recompile the kernel to get a driver for the ethernet card working. It took me a full day, with the help of a friend. It had fvwm95 as the window manager. It was a lot of fun and a great learning experience. I was on my hands and knees, with the back of the computer off, trying to see what the chip was on the ethernet card. As I recall, it was a D-Link DE-220. Then I remember having to figure out what the modeline was for my screen. I hedged my bets and got my computer dual booting with lilo. I don't think I could have done it without a friend who could help me figure out the next thing I needed to learn about!

I don't think I could have done it without a friend.

Greg Pittman [11] writes:

My first Linux was with Red Hat 5.1, or 2.0.34 kernel version, according to Wikipedia. The install went pretty well, it seemed to be well-documented, but it was all about the drivers: video chip drivers, printer drivers, modem drivers (remember modems?). Not only that, you had to manually edit the config file. This was on a Gateway laptop, where I had set up a dual-boot situation. Gateway was of course, no help, so you had to research it yourself and find somebody that talked about a driver for your hardware. Also, since I didn't have a modem working in Linux, I had to keep rebooting to Windows to do the research, download the drivers, and save them on a floppy. One of the happiest days of my life was about two weeks after the install when I FINALLY got X-windows working. Life was good.

Life was good.

Jay LaCroix [12] writes:

My first installation was Red Hat 7.1 or 7.2, probably 2002 or thereabouts. I was using it on a 300Mhz Pentium III PC with very limited resources. GNOME wouldnít even start on it, but I was able to get KDE to work. It was the installation I used while taking a Linux course at a community college.

Ben Cotton [13] writes:

My first Linux kernel version was 2.4.18 or so—whatever was shipping in Red Hat Linux 8 at the time. My first Linux install was a dual boot on my desktop my sophomore year of college. I'd been frustrated with my Windows 2000 installation and my friend suggested I try this "Linux" thing. It was a pretty basic desktop. I didn't really know what I was doing at the time, but I had just started learning my way around FreeBSD for my part-time job, so it felt pretty cool to run something

I didn't really know what I was doing at the time.

non-Windows. I remember taking my computer home for the summer and not being able to get the modem working, so I couldn't get online. My parents still had dial-up then.

Alan Formy-Duval [14] writes:

The first kernel I actually compiled myself seems like the most appropriate answer. I believe it was probably 2.0.32 or 2.0.33. I was running Red Hat Linux 5.X as my first distribution. My first installation was onto a basic Dell Optiplex desktop machine. There were always a few necessary steps after performing the installation. Those were to complete the network configuration, configure X-Windows, and compile the latest kernel. For the kernel, I would head over to kernel.org and download the latest version. I remember (mostly) the command used to run the compilation as something like "make dep clean bzlilo modules modules_install." It seems like the compile took an hour or two.

David Both [15] writes:

My first was probably kernel 2.0.32 in Red Hat Linux 5.0 in late 1997. My first install was long and slow on my IBM ThinkPad—which was even then quite old—with a CD. It required me to make a number of choices that I did not then understand, including ones about hardware and the list of software to install. As I recall, there were no groups of software that would install required prerequisites so after the basic installation I had to endure hours of dependency-hell to install a few additional top-level packages. Package management with RPM was a nightmare because it did not deal at all with finding and installing dependencies like YUM and DNF. It must have been even worse before RPM. I never did get the display on my ThinkPad to switch to graphics mode. But that was probably excellent as it forced me to learn how to use the Linux command line, and I have been a CLI fan ever since.

Chris Hermansen [16] writes:

My first kernel version would have been around 2.2.12. I remember getting Red Hat Linux on floppies shortly after returning home from my first visit to Chile in November 1999. If I recall correctly, the first install did not go at all well—it was on a Thinkpad. The second was on a desktop and it seems to me that worked just fine. The thing that struck me about that first brush with Linux, once I got past the huge packet of floppies, was how generally decent it was compared to my work machine which at the time was a Sun SPARCStation something or other.

Anderson Silva [17] writes:

Red Hat Linux 3.0.3 (Picasso), kernel 1.2.13. My first install was a total failure. I had to open up my Packard Bell computer, to see if I could identify if my CD-ROM was Primary or Secondary Master or Slave, as my BIOS wouldn't tell me, and Linux didn't auto-detect it, and then it died on the X install. I only tried the install again six months later, successfully. I've been running on Red Hat Linux/Fedora since then. A funny story: I remember jumping on an IRC server and going to ask for help. And I typed something like "I need help getting Xwindows working on my PC." I was kicked out of the channel with a reason: It is not Xwindows, but Xwindow... so, yeah... open source folks can be tough!

My first install was a total failure.

SCO Openserver

Jim Salter [18] writes:

A rough guess for which version is 1.2.0. I can't find a definitive answer on the kernel it shipped with, but 1.2.0 would have been pretty current when SCO OpenServer 5 shipped. The install was absolutely horrible. Did I mention it was SCO OpenServer? Horrible. This was the foundation of a very spendy telemarketing predictive dialing system, with SCO OpenServer 5.0 as the OS and a proprietary application running the phones over a T-1. Mostly I remember thinking that I was for absolute certain going to be a FreeBSD person, not a Linux person. And for quite a while, I was. There wasn't really any doubt in my mind then or now that FreeBSD was a superior system in pretty much every way in the mid-90s; but at some point in the 2000s Linux blew the doors off and has been gaining more steam ever since.

I remember thinking I was for absolute certain going to be a FreeBSD person, not a Linux person.

Slackware

Tony McCormick [19] writes:

It was exciting to be able to use a Unix like OS at home so I could run Perl and Bash scripts. During the installation, there were lots of 3 1/2" floppy drives and flipping through the Yggdrasil Linux book to figure out how to compile things. Getting dial-up working so I could login at to the office was *fun* too, but great.

Peter Czanik [20] writes:

My first Linux kernel version was 0.99.11 or 0.99.13. Of course, I don't remember by heart, but it was Slackware, and it was not yet kernel 1.0. It was a pretty basic installation, as my machine did not have much RAM. It was good enough for a text console and to learn the basics: bash, init scripts, server applications, reading tons of man pages. My first Linux install involved many floppy disks. And, I actually had to reinstall it a couple of times as DR-DOS (the other OS on the machine) and rearrange partition numbers on each boot.

Steve Morris [21] writes:

I was another of those early Slackware users; I picked it up at a local Comdex show in Vancouver. I rushed home and proceeded to install the 24 or so floppy disks on my PC. After what seemed like hours later, I was left with a command-line shell. All I could remember about that first experience is, "Now what do I do?" It lasted on the machine for about three months before it was mothballed and I purchased a copy of Red Hat Linux 5.0 on CD.

Now what do I do?

Kevin Cole [22] writes:

I started with an ancient Slackware distribution sold by a company called Trans-America. I cannot recall the kernel but the era was circa 1993-1994. The thing I remember most was that halfway through the install, I had to switch CDs and when I did, it said: "What's a CD?" (It could no longer find the driver.) Thankfully, even back then, there was an online community willing to help and someone lent me to a floppy that would allow me to continue. That first install was... "OMG, I've got a friggin' mainframe on my desk!" I wrote more about that adventure in this article [23].

Andy Thornton [24] writes:

Pre-version 2, I remember the buzz around a new 2.0 kernel coming out! I installed Slackware because I had met a mate in Boston who showed me around it, and I had to build a box when I got home. I used it entirely from the shell and it went on to be a MUD server I hosted out of my bedroom. I had a script to dial my internet provider during off-peak hours to save on bills (this was in the UK). I spent an entire weekend downloading the equivalent of 80 floppy disks on a 54k modem, and once I had it all down, I transferred it to a Jazz drive from Iomega so I had a copy. I would pray no one rang the house during the download as it would break and I would have to recover it or start all over again. I will admit, it was a fun weekend.

I would pray no one rang the house during the download.

Daniel Oh [25] writes:

My first install was Slackware, and it took a lot of time for me to stand up Linux. I got used to doing hands-on stuff with Windows, but it was so exciting to try out open source technology. There were many floppy drives involved with my first installation, and I had to use a few Linux books to learn how to build, install, and configure all matters of the OS programs. It took a lot of time but it was really fun for me.

Softlanding Linux System

Jim Hall [26] writes:

My first Linux distribution was Softlanding Linux System (SLS) 1.03, with Linux kernel 0.99 alpha patch level 11. Installation required a whopping 2 MB of RAM, or 4 MB if you wanted to compile programs, and 8 MB to run X Windows. Linux added modules in 1.0, so this was pre-modules. Everything had to be compiled in.

Michael Schulz [27] writes:

IIRC, my first version was 0.98-4 pl 10 on the SLS distro. There were twenty-four 1.44 MB floppy disks (remember those?) that I had to download with my super fast 9k6 US Robotics modem. Those were the days.

There were 24 floppy disks (remember those?)

Eric Eslinger [28] writes:

I installed some version of Linux in the 0.99 kernel version time (definitely prior to 1.0), using SLS, in or around 1993. It was a moment of spontaneous magic for me. A friend helped me do things like bring up an xterm, change my shell to the very cool csh, and learn how to set DISPLAY environment variables to run graphical applications on my personal computer while running the code on a server. It was transformative for me to have an OS I could not only understand but also write code for directly.

It was a moment of spontaneous magic for me.

Yedidyah Bar David [29] writes:

My first kernel *booted* was 0.99.11 or so in SLS, but I failed to install it. My first kernel *used* was IIRC 0.99.10, from MCC Interim Linux, which managed to install with just 2 MB RAM (because it had a documented option to not use a ramdisk, but directly from a floppy). This install was hard and took several weeks! I remember that SLS (which is what was recommended to me at the time, in 1993) didn't install with 2 MB RAM. So, I consulted people, tried HJ Lu's Boot/Root floppies which did boot and work, then found MCC and installed it, which worked. Shortly thereafter, I compiled a kernel (0.99.14?), which took around 24 hours, then I bought another 2 MB RAM for my machine. Then, the compile took about an hour. I used MCC for some time, then tried SLS again. SLS's installation with 4 MB RAM was reasonable, but somewhat ugly with white on black and question prompts and answers. A few months later a friend told me he installed Slackware, and "it was so much nicer!," which indeed it was true. It was in color and fullscreen, though still in text mode. So, I moved to Slackware for two years, then finally installed Debian, and stayed with it until two years ago. Also, I used it on my first laptop as an employee at Red Hat. Three years later, I decided it was time to move to RHEL!

Ubuntu

Kedar Vijay Kulkarni [30] writes:

My first Linux was Ubuntu. I had to install it as it was mandated in our curriculum at MIT College of Engineering in Pune, India. The installation itself was very easy and straightforward. The only option I had to get right was "Install Ubuntu alongside Windows" in order to make sure I didn't wipe out my Windows partition. Something that I distinctly remember about my first Linux install was learning how to create a bootable USB drive and learning what dual partition is and looks like. I remember all the excitement of getting a whole new operating system entirely for free. Plus, the Unity GUI was a refreshing break from Windows XP.

Brian Whetten [31] writes:

My first Linux kernel version was 2.6.32 as part of Lucid Lynx Ubuntu version 10.04. My first Linux install was memorable because after helping others install Windows XP and older versions of MacOS, Linux installed the fastest. Even though I was only in junior high at the time, I knew that I had found something special. It was snappier than any computer I had used at school or home (with the exception of some video acceleration issues). I was just excited to finally have a machine I could freely learn and explore in, with no knowledge blocks and great searchable documentation. Blender ran incredibly well on it versus the comparable Mac my family-owned computer.

David Clinton [32] writes:

My first Linux experience was installing Ubuntu 7.10 (kernel version 2.6.22) from Windows XP using Wubi. That part was easy. Getting it to work as an LTSP server for a network of thin clients booting via PXE was considerably more complicated. I can't remember whether I actually got it all running from the Wubi version or whether that had to wait until my first full install, but the triumph of success was worth the effort... even if it first required diagnosing a flaky port on the old network switch I was using.

Kyle Conway [33] writes:

My first install was Ubuntu 8.04, Hardy Heron, which seems to have run Linux kernel 2.6.24. It was a WUBI install (the relative simplicity of an *.exe that wouldn't "break" my computer convinced me to give it a try). There's a great thread on the Ubuntu forums where I got the help I needed (getting my WUBI install back) and move fulltime to Linux. I'm coming up on a decade! It was unbelievable to me that I could download this at no cost, that it ran many of the applications that I already used (e.g. Firefox), that I could share it with others, that it had useful software installed by default, and that it didn't require license keys—it just worked.

It didn't require license keys—it just worked.

Links

[1] https://opensource.com//resources/linux
[2] https://opensource.com//article/18/7/bdfl
[3] https://opensource.com/users/stratusss
[4] https://opensource.com/users/matthew-helmke
[5] https://opensource.com/users/thatsjet
[6] https://opensource.com/users/steven-ellis
[7] https://opensource.com/users/greptile
[8] https://opensource.com/users/genehack
[9] https://opensource.com/users/don-watkins
[10] https://opensource.com/users/dneary
[11] https://opensource.com/users/greg-p
[12] https://opensource.com/users/jlacroix
[13] https://opensource.com/users/bcotton
[14] https://opensource.com/users/alanfdoss
[15] https://opensource.com/users/dboth
[16] https://opensource.com/users/clhermansen
[17] https://opensource.com/users/ansilva
[18] https://opensource.com/users/jim-salter
[19] https://opensource.com/users/tmccormick
[20] https://opensource.com/users/czanik
[21] https://opensource.com/users/smorris12
[22] https://opensource.com/users/kjcole
[23] https://opensource.com/life/15/11/my-open-source-story-kevin-cole
[24] https://opensource.com/users/andrew-thornton
[25] https://opensource.com/users/daniel-oh
[26] https://opensource.com/users/jim-hall
[27] https://opensource.com/users/mschulz
[28] https://opensource.com/users/eric-eslinger
[29] https://opensource.com/users/didib
[30] https://opensource.com/users/kkulkarn
[31] https://opensource.com/users/classywhetten
[32] https://opensource.com/users/dbclinton
[33] https://opensource.com/users/kreyc

Author

Jen has been an editor on the Opensource.com team for six years. In that time, she's worked with countless developers and engineers, helping them with the magic of turning their technical expertise and experience into written form. On any given day, you'll find her managing the website's publication schedule and editorial workflow (on kanban boards), as well as brainstorming the next big article.

15 open source applications for macOS

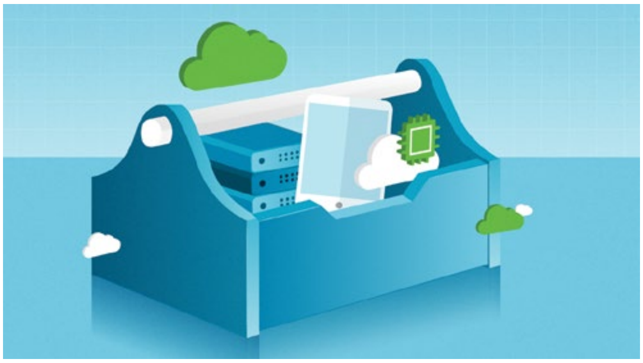
BY DON WATKINS

Dedicated open source users won't find it hard to use their favorite applications on non-Linux operating systems.

I USE OPEN SOURCE tools whenever and wherever I can. I returned to college a while ago to earn a master's degree in educational leadership. Even though I switched from my favorite Linux laptop to a MacBook Pro (since I wasn't sure Linux would be accepted on campus), I decided I would keep using my favorite tools, even on macOS, as much as I could.

Fortunately, it was easy, and no professor ever questioned what software I used. Even so, I couldn't keep a secret.

I knew some of my classmates would eventually assume leadership positions in school districts, so I shared information about the open source applications described below with many of my macOS or Windows-using classmates. After all, open source software is really about freedom and goodwill. I also wanted them to know that it would be easy to provide their students with world-class applications at little cost. Most of them were surprised and amazed because, as we all know, open source software doesn't have a marketing team except users like you and me.



My macOS learning curve

Through this process, I learned some of the nuances of macOS. While most of the open source tools worked as I was used to, others required different installation methods. Tools like yum [1], DNF [2], and [3] do not exist in the macOS world—and I really missed them.

Some macOS applications required dependencies and installations that were more difficult than what I was accustomed to with Linux. Nonetheless, I persisted. In the process, I learned how I could keep the best software on my new platform. Even much of macOS's core is open source [4].

Also, my Linux background made it easy to get comfortable with the macOS command line. I still use it to create and copy files, add users, and use other utilities [5] like cat, tac, more, less, and tail.

15 great open source applications for macOS

- The college required that I submit most of my work electronically in DOCX format, and I did that easily, first with OpenOffice [6] and later using LibreOffice [7] to produce my papers.
- When I needed to produce graphics for presentations, I used my favorite graphics applications, GIMP [8] and Inkscape [9].

- My favorite podcast creation tool is Audacity [10]. It's much simpler to use than the proprietary application that ships with the Mac. I use it to record interviews and create soundtracks for video presentations.
- I discovered early on that I could use the VideoLan [11] (VLC) media player on macOS.
- macOS's built-in proprietary video creation tool is a good product, but you can easily install and use OpenShot [12], which is a great content creation tool.
- When I need to analyze networks for my clients, I use the easy-to-install Nmap [13] (Network Mapper) and Wireshark [14] tools on my Mac.
- I use VirtualBox [15] for macOS to demonstrate Raspbian, Fedora, Ubuntu, and other Linux distributions, as well as Moodle, WordPress, Drupal, and Koha when I provide training for librarians and other educators.
- I make boot drives on my MacBook using Etcher.io [16]. I just download the ISO file and burn it on a USB stick drive.
- I think Firefox [17] is easier and more secure to use than the proprietary browser that comes with the MacBook Pro, and it allows me to synchronize my bookmarks across operating systems.
- When it comes to eBook readers, Calibre [18] cannot be beaten. It is easy to download and install, and you can even configure it for a classroom eBook server [19] with a few clicks.
- Recently I have been teaching Python to middle school students, I have found it is easy to download and install Python 3 and the IDLE3 editor from Python.org [20]. I have also enjoyed learning about data science and sharing that with students. Whether you're interested in Python or R, I recommend you download and install [21] the Anaconda distribution [22]. It contains the great iPython editor, RStudio, Jupyter Notebooks, and JupyterLab, along with some other applications.
- HandBrake [23] is a great way to turn your old home video DVDs into MP4s, which you can share on YouTube, Vimeo, or your own Kodi [24] server on macOS.

Links

[1] https://en.wikipedia.org/wiki/Yum_(software)
[2] https://en.wikipedia.org/wiki/DNF_(software)
[3] https://en.wikipedia.org/wiki/APT_(Debian)APT
[4] https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/OSX_Technology_Overview/SystemTechnology/SystemTechnology.html
[5] https://www.gnu.org/software/coreutils/coreutils.html
[6] https://www.openoffice.org/
[7] https://www.libreoffice.org/
[8] https://www.gimp.org/
[9] https://inkscape.org/en/
[10] https://www.audacityteam.org/
[11] https://www.videolan.org/index.html
[12] https://www.openshot.org/
[13] https://nmap.org/
[14] https://www.wireshark.org/
[15] https://www.virtualbox.org/
[16] https://etcher.io/
[17] https://www.mozilla.org/en-US/firefox/new/
[18] https://calibre-ebook.com/
[19] https://opensource.com/article/17/6/raspberrypi-ebook-server
[20] https://www.python.org/downloads/release/python-370/
[21] https://opensource.com/article/18/4/getting-started-anaconda-python
[22] https://www.anaconda.com/download/#macos
[23] https://handbrake.fr/
[24] https://kodi.tv/download

Author

Don is an educator, education technology specialist, entrepreneur, and open source advocate. He holds an M.A. in Educational Psychology and an MSED in Educational Leadership. As a Linux system administrator and CCNA, he is an expert at virtualization using Virtual Box. Follow him at @Don_Watkins

4 ways Flutter makes mobile app development delightful

BY EMILY FORTUNA

Open source mobile SDK simplifies and speeds iOS and Android app development.

I'M GOING TO LET YOU IN ON A SECRET: For years I hated mobile development. I wanted to like it—mobile was the future! It was cool! It was low-power! It was a way to connect with users whose first exposure to computers did not come from traditional desktop platforms! And yet… development was a slow, frustrating experience for me. Instead, I sequestered myself over in the *entirely problem-free* area of web development and mourned the disappearance of the HTML `blink` tag (kidding).

Then, I discovered Flutter [1], an open source mobile app SDK developed by Google that enables developers to use the same codebase to create mobile apps for iOS and Android.

Once I found Flutter, I found that mobile development could be *joyful*.

Yes, joyful.

How, you ask? To show you what I mean, let's walk through writing a very simple Flutter app that queries Stack Overflow. As the self-respecting open source developer that you are, I'm sure you want to keep on top of the questions people are asking about your software on Stack Overflow. This app allows you to search for questions about a specific topic.

Lightning-fast development cycle

Traditional compilers are trouble. You know how it goes: You hit "compile," and the next thing you know you're 10 tabs deep into cute kitten photos—and it's lunchtime. Fortunately, when I worked on the web, interpreters and Just In Time (JIT) compilers saved me from my kitten tendencies, making "edit, save, refresh" the name of the game.

Flutter takes this quick development idea one step further: "edit, save." Although it's not a web technology, you can see your changes on your mobile device's screen in *less than a second*, thanks to Flutter's hot reload.

Typically you get this fast development cycle by using fancy, dynamically typed scripting languages, with the downside of pushing errors to runtime rather than catching them beforehand at compile-time. The second common drawback is their performance is not as zippy as compiled languages. By using Dart as its programming language of choice, Flutter can sidestep both of these issues. Dart has a strong, sound type system that allows you to catch problems before demoing That Part of the Codebase With Less Than Ideal Test Coverage.

Second, Dart has two modes:

1. running in “interpreted” mode on the Dart virtual machine, which allows that joyful, hot reload experience, and;
2. compiled mode, which compiles your app down to native machine code when you’re ready to release your app.

Given these features, Dart is uniquely suited to provide developers with great development and release experiences for Flutter.

Finally, Dart was designed to be easy to learn, so if you’ve worked with any C-style language like Java, C++, or JavaScript, it will feel familiar.

Cool features, such as Streams and Futures

Time to start coding! Our app will use the Stack Overflow API to look for questions about Flutter that need responses, so that you, the intrepid open source project owner that you are, can help your community by keeping them informed. The simplest way to get that information in Dart is with an asynchronous request:

```
final url = 'https://api.stackexchange.com/2.2/questions?
order=desc&sort=activity&tagged=flutter&site=stackoverflow';
var result = await http.get(url);
print(result.body);
```

The result prints out some JSON, looking something like this:

```
{
  "items": [
    {
      "tags": [
        "android",
        "ios",
        "flutter"
      ],
      "owner": {
        "reputation": 1,
        ...
      },
      is_answered: false,
      "view_count": 1337,
      "title": "How to make a pretty Flutter app?"
...
}
```

In this code snippet, *http.get* is returning a `Future<Request>`, which means that the result will be available in the future of type `(Http)Request`. Even though we’re making a round trip to the server, we don’t need to pass in a callback; we can just use the `await` keyword to wait for a response. Flutter has `FutureBuilder` and `StreamBuilder` widgets to build corresponding UI components, given the results of a `Future` or a `Stream`.

A `Stream` is just like a `Future`, except that it can provide results asynchronously multiple times instead of just once. In our app, we’ll create a Stream where we can listen for updates on our Stack Overflow questions of interest. Since the Stack Overflow API doesn’t provide push notifications out of the box, we construct our own stream using a **StreamController** and add updated Stack Overflow information whenever we get it:

```
final controller = StreamController<List<String>>();

void refreshQuestions() async {
  var result = await http.get(url);
  Map decoded = json.decode(result.body);
  List items = decoded['items'];
  controller.add(items
      .where((item) => !item['is_answered'])
      .map<String>((item) => item['title'])
      .toList());
}
```

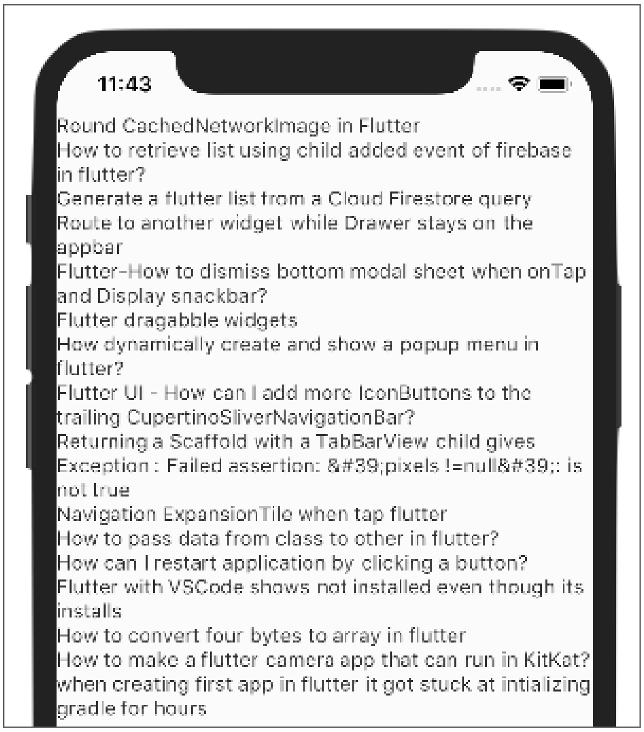
In `refreshQuestions`, we make a new call to the Stack Overflow API, then filter the results so we’re only looking at questions that have not been answered. From those results, we pull out the titles of the questions to display them in our app.

Flutter conveniently provides a `StreamBuilder` widget that can automatically update what the user sees in the app based on the contents of a stream. In this case, we provide the input stream source (`controller.stream`) and display different results depending on whether we successfully received data or not (in this case building terribly exciting `Text` widgets). `StreamBuilder` also conveniently takes care of unsubscribing itself from the stream and cleaning up after itself.

StreamBuilder also conveniently takes care of unsubscribing itself from the stream and cleaning up after itself.

```
StreamBuilder(
   stream: controller.stream,
   builder: (BuildContext context, AsyncSnapshot<List<String>>
snapshot) {
     if (snapshot.hasError)
       return Text('Error ${snapshot.error}');
     else if (snapshot.connectionState == ConnectionState.
              waiting) {
       return Text('Receiving questions...');
     }
     return Expanded(
         child: ListView(
             children: snapshot.data
                 .map<Widget>((info) => Text(info))
                 .toList()));
   });
```

The result of the previous code. (I don't understand why no design school would accept me.)



11:43

Round CachedNetworkImage in Flutter
How to retrieve list using child added event of firebase in flutter?
Generate a flutter list from a Cloud Firestore query
Route to another widget while Drawer stays on the appbar
Flutter-How to dismiss bottom modal sheet when onTap and Display snackbar?
Flutter dragabble widgets
How dynamically create and show a popup menu in flutter?
Flutter UI - How can I add more IconButtons to the trailing CupertinoSliverNavigationBar?
Returning a Scaffold with a TabBarView child gives Exception : Failed assertion: &#39;pixels !=null&#39;: is not true
Navigation ExpansionTile when tap flutter
How to pass data from class to other in flutter?
How can I restart application by clicking a button?
Flutter with VSCode shows not installed even though its installs
How to convert four bytes to array in flutter
How to make a flutter camera app that can run in KitKat?
when creating first app in flutter it got stuck at intializing gradle for hours

One technology for both iOS and Android

”Don't Repeat Yourself” is a common software engineering mantra, yet the mobile development world seems to be in denial. All too often, companies spin up separate iOS and Android app teams, each of which needs to solve the same problems twice. With Flutter, you can write in Dart and deploy natively to both iOS and Android. Scrolling behavior, system fonts, and other fundamental interaction components automatically default for the platform you're using. At a higher level, Flutter provides Cupertino and Material Design widget libraries to get the look and feel users expect on their platform of choice.

In our Stack Overflow app, we want to have a “Get New Results” button to see if there are new questions that need our attention. We'll write a `PlatformAdaptiveButton` whose behavior depends on the platform we're running on:

```
class PlatformAdaptiveButton extends StatelessWidget {
  final Widget child;
  final Widget icon;
  final VoidCallback onPressed;
  PlatformAdaptiveButton({Key key, this.child, this.icon, this.
                        onPressed})
      : super(key: key);

  @override
  Widget build(BuildContext context) {
    if (Theme.of(context).platform == TargetPlatform.iOS) {
      return CupertinoButton(
        child: child,
        onPressed: onPressed,
      );
    } else {
      return FloatingActionButton(
        icon: icon,
        onPressed: onPressed,
      );
    }
  }
}
```

Then, in our Flutter app, we can simply construct:

```
return PlatformAdaptiveButton(
   child: const Text('Refresh'),
   icon: const Icon(Icons.refresh),
   onPressed: refreshQuestions);
```

Which, when pressed, requests updates from the Stack Overflow API. Flutter's roadmap calls for more built-in ways to have platform adaptive components in your code, so stay tuned.

A handful of other app development systems provides cross-platform functionality, too: React Native, Xamarin, and Ionic, to name a few. With React Native and Ionic, you develop in JavaScript, which has the potential for less type safety (and therefore more unwanted surprises at runtime), and the code is interpreted or JITed. With Xamarin, you get strong type-safety guarantees with C# and, depending on the target platform, the code is compiled to native, JITed, or run on a virtual machine. Flutter compiles down to native machine code on both iOS and Android, giving it predictable, speedy performance.

Customization

”But Emily,” you say. “I work at an agency and I simply can't have all the apps I create look the same! I need them to look distinctive and add those stylish touches, like my signature, tasteful usage of Comic Sans!” Never fear, my aesthete friends. Flutter really shines in this area.

Because Flutter is drawing every pixel to the screen, *everything* is customizable. Don't like how that built-in `CupertinoButton` is behaving? Make a subclass and design it yourself. Think solid-color app bars are so passé? Write your own widget. In our Stack Overflow app, I wrote a custom app bar that has a custom font and a gradient color scheme to distinguish it from all those other boring app bars—and it's not even much code:

```
@override
Widget build(BuildContext context) {
  final double statusBarHeight = MediaQuery.of(context).padding.top;
  return Container(
   padding: EdgeInsets.only(top: statusBarHeight),
   height: statusBarHeight * 4,
   child: Center(
     child: Text(
       title,
       style: const TextStyle(
           color: Colors.white, fontFamily: 'Kranky', fontSize: 36.0),
     ),
   ),
   decoration: BoxDecoration(
     gradient: LinearGradient(
       colors: [
         Colors.deepOrange,
         Colors.orangeAccent,
       ],
     ),
   ),
  );
}
```

You can see the final result below.

All the code I wrote for this article can be found on GitHub at Stack Overflow Viewer [2].

Intrigued?

There is so much more we can do! However, further customization is left as an exercise to the reader.

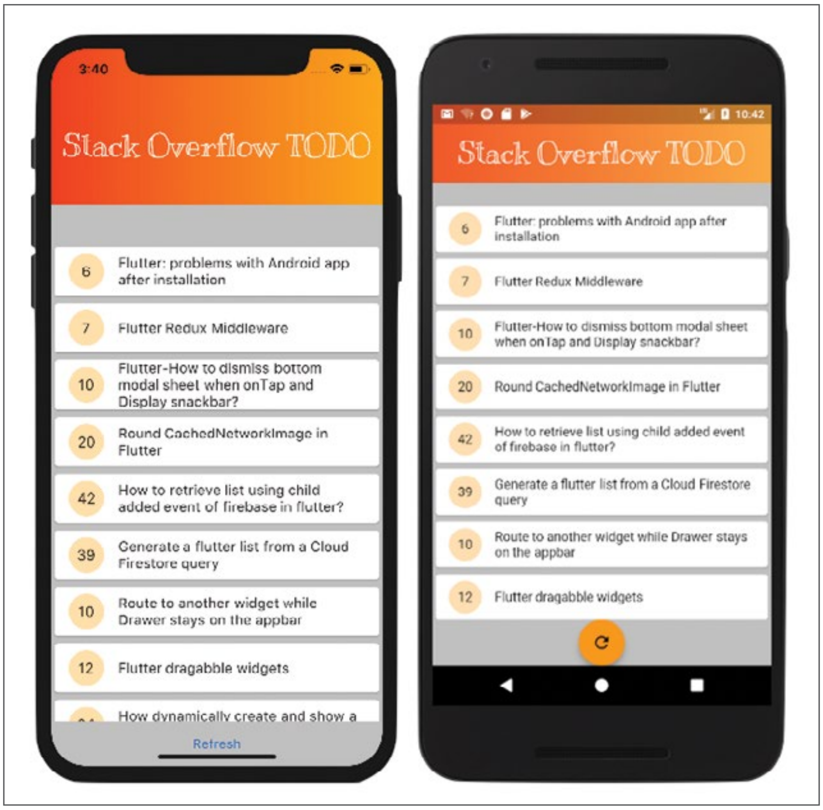
Links

[1] https://flutter.io

[2] https://github.com/efortuna/stack_overflow_viewer

Author

Emily Fortuna is a senior software engineer on the Dart team at Google. When not hacking on compilers and evangelizing the awesomeness of Flutter, she can be found working on improving fairness in machine learning or acting on the stage and screen. She is an avid member of the nerdy joke appreciation society.



Stack Overflow TODO

6 Flutter: problems with Android app after installation

7 Flutter Redux Middleware

10 Flutter-How to dismiss bottom modal sheet when onTap and Display snackbar?

20 Round CachedNetworkImage in Flutter

42 How to retrieve list using child added event of firebase in flutter?

39 Generate a flutter list from a Cloud Firestore query

10 Route to another widget while Drawer stays on the appbar

12 Flutter dragabble widgets

How dynamically create and show a

Stack Overflow TODO

6 Flutter: problems with Android app after installation

7 Flutter Redux Middleware

10 Flutter-How to dismiss bottom modal sheet when onTap and Display snackbar?

20 Round CachedNetworkImage in Flutter

42 How to retrieve list using child added event of firebase in flutter?

39 Generate a flutter list from a Cloud Firestore query

10 Route to another widget while Drawer stays on the appbar

12 Flutter dragabble widgets

Power(Shell) to the people

BY YEV BRONSHTEYN

Type less, write cleaner scripts, run consistently across platforms, and other reasons why Linux and OS X users can fall in love with PowerShell.

IN 2018, POWERSHELL CORE [1] became generally available [2] under an Open Source (MIT [3]) license. PowerShell is hardly a new technology. From its first release for Windows in 2006, PowerShell's creators sought [4] to incorporate the power and flexibility of Unix shells while remedying their perceived deficiencies, particularly the need for text manipulation to derive value from combining commands.

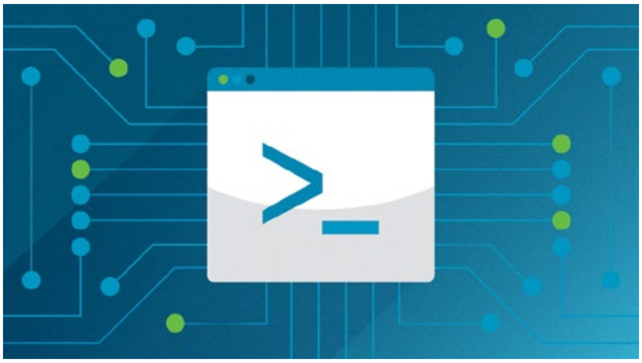
Five major releases later, PowerShell Core allows the same innovative shell and command environment to run natively on all major operating systems, including OS X and Linux. Some (read: *almost everyone*) may still scoff at the audacity and/or the temerity of this Windows-born interloper to offer itself to platforms that have had strong shell environments since time immemorial (at least as defined by a millennial). In this post, I hope to make the case that PowerShell can provide advantages to even seasoned users.

Consistency across platforms

If you plan to port your scripts from one execution environment to another, you need to make sure you use only the commands and syntaxes that work. For example, on GNU systems, you would obtain yesterday's date as follows:

```
date --date="1 day ago"
```

On BSD systems (such as OS X), the above syntax wouldn't work, as the BSD date utility requires the following syntax:



```
date -v -1d
```

Because PowerShell is licensed under a permissive license and built for all platforms, you can ship it with your application. Thus, when your scripts run in the target environment, they'll be running on the same shell using the same command implementations as the environment in which you tested your scripts.

Objects and structured data

*nix commands and utilities rely on your ability to consume and manipulate unstructured data. Those who have lived for years with `sed` `grep` and `awk` may be unbothered by this statement, but there is a better way.

Let's redo the yesterday's date example in PowerShell. To get the current date, run the `Get-Date` cmdlet (pronounced "commandlet"):

```
> Get-Date

Sunday, January 21, 2018 8:12:41 PM
```

The output you see isn't really a string of text. Rather, it is a string representation of a .Net Core object. Just like any other object in any other OOP environment, it has a type and most often, methods you can call.

Let's prove this:

```
> $(Get-Date).GetType().FullName
System.DateTime
```

The `$(...)` syntax behaves exactly as you'd expect from POSIX shells—the result of the evaluation of the command in parentheses is substituted for the entire expression. In PowerShell, however, the $ is strictly optional in such expressions. And, most importantly, the result is a .Net object, not text. So we can call the `GetType()` method on that object to get its type object (similar to `Class` object in Java), and the `FullName` property [5] to get the full name of the type.

So, how does this object-orientedness make your life easier?

First, you can pipe any object to the `Get-Member` cmdlet to see all the methods and properties it has to offer.

```
> (Get-Date) | Get-Member
PS /home/yevster/Documents/ArticlesInProgress> $(Get-Date) |
   Get-Member

   TypeName: System.DateTime

Name               MemberType   Definition
----               ----------   ----------
Add                Method       datetime Add(timespan value)
AddDays            Method       datetime AddDays(double value)
AddHours           Method       datetime AddHours(double value)
AddMilliseconds    Method       datetime AddMilliseconds(double
                                value)
AddMinutes         Method       datetime AddMinutes(double
                                value)
AddMonths          Method       datetime AddMonths(int months)
AddSeconds         Method       datetime AddSeconds(double
                                value)
AddTicks           Method       datetime AddTicks(long value)
AddYears           Method       datetime AddYears(int value)
CompareTo          Method       int CompareTo(System.Object
                                value), int ...
```

You can quickly see that the DateTime object has an `AddDays` that you can quickly use to get yesterday's date:

```
> (Get-Date).AddDays(-1)

Saturday, January 20, 2018 8:24:42 PM
```

To do something slightly more exciting, let's call Yahoo's weather service (because it doesn't require an API token) and get your local weather.

```
$city="Boston"
$state="MA"
$url="https://query.yahooapis.com/v1/public/
      yql?q=select%20*%20from%20weather.forecast%20
      where%20woeid%20in%20(select%20woeid%20from%20
      geo.places(1)%20where%20text%3D%22${city}%2C%20
      ${state}%22)&format=json&env=store%3A%2F%2Fdatatables.
      org%2Falltableswithkeys"
```

Now, we could do things the old-fashioned way and just run `curl $url` to get a giant blob of JSON, or...

```
$weather=(Invoke-RestMethod $url)
```

If you look at the type of `$weather` (by running `echo $weather.GetType().FullName`), you will see that it's a `PSCustomObject`. It's a dynamic object that reflects the structure of the JSON.

And PowerShell will be thrilled to help you navigate through it with its tab completion. Just type `$weather.` (making sure to include the ".") and press Tab. You will see all the root-level JSON keys. Type one, followed by a "`.`", press Tab again, and you'll see its children (if any).

Thus, you can easily navigate to the data you want:

```
> echo $weather.query.results.channel.atmosphere.pressure
1019.0

> echo $weather.query.results.channel.wind.chill
41
```

And if you have JSON or CSV lying around (or returned by an outside command) as unstructured data, just pipe it into the `ConvertFrom-Json` or `ConvertFrom-CSV` cmdlet, respectively, and you can have your data in nice clean objects.

Computing vs. automation

We use shells for two purposes. One is for computing, to run individual commands and to manually respond to their output. The other is automation, to write scripts that execute multiple commands and respond to their output programmatically.

A problem that most of us have learned to overlook is that these two purposes place different and conflicting requirements on the shell. Computing requires the shell to be laconic. The fewer keystrokes a user can get away with, the better. It's unimportant if what the user has typed is barely legible to another human being. Scripts, on the other hand, are code. Readability and maintainability are key. And here, POSIX utilities often fail us. While some commands do offer both laconic and readable syntaxes (e.g. `-f` and `--force`) for some of their parameters, the command names themselves err on the side of brevity, not readability.

PowerShell includes several mechanisms to eliminate that Faustian tradeoff.

First, tab completion eliminates typing of argument names. For instance, type `Get-Random -Mi`, press Tab and PowerShell will complete the argument for you: `Get-Random -Minimum`. But if you really want to be laconic, you don't even need to press Tab. For instance, PowerShell will understand

```
Get-Random -Mi 1 -Ma 10
```

because `Mi` and `Ma` each have unique completions.

You may have noticed that all PowerShell cmdlet names have a verb-noun structure. This can help script readability, but you probably don't want to keep typing `Get-` over and over in the command line. So don't! If you type a noun without a verb, PowerShell will look for a `Get-` command with that noun.

Caution: although PowerShell is not case-sensitive, it's a good practice to capitalize the first letter of the noun when you intend to use a PowerShell command. For example, typing date will call your system's date utility. Typing Date will call PowerShell's `Get-Date` cmdlet.

And if that's not enough, PowerShell has aliases to create simple names. For example, if you type `alias -name cd`, you will discover the `cd` command in PowerShell is itself an alias for the `Set-Location` command.

So to review—you get powerful tab completion, aliases, and noun completions to keep your command names short, automatic and *consistent* parameter name truncation, while still enjoying a rich, readable syntax for scripting.

So... friends?

There are just some of the advantages of PowerShell. There are more features and cmdlets I haven't discussed (check out Where-Object [6] or its alias ? if you want to make `grep` cry). And hey, if you really feel homesick, PowerShell will be happy to launch your old native utilities for you. But give yourself enough time to get acclimated in PowerShell's object-oriented cmdlet world, and you may find yourself choosing to forget the way back.

Links

[1] https://github.com/PowerShell/PowerShell/blob/master/README.md

[2] https://blogs.msdn.microsoft.com/powershell/2018/01/10/powershell-core-6-0-generally-available-ga-and-supported/

[3] https://spdx.org/licenses/MIT

[4] http://www.jsnover.com/Docs/MonadManifesto.pdf

[5] https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/properties

[6] https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/where-object?view=powershell-6

Author

Yev Bronshteyn is a software engineer with an open source governance bend, and an occasional developer outreacher. All opinions are my own. Brain droppings here.

Running a Python application on Kubernetes

BY JOANNAH NANJEKYE

This step-by-step tutorial takes you through the process of deploying a simple Python application on Kubernetes.

KUBERNETES is an open source platform that offers deployment, maintenance, and scaling features. It simplifies management of containerized Python applications while providing portability, extensibility, and self-healing capabilities.

Whether your Python applications are simple or more complex, Kubernetes lets you efficiently deploy and scale them, seamlessly rolling out new features while limiting resources to only those required.

In this article, I will describe the process of deploying a simple Python application to Kubernetes, including:

- Creating Python container images
- Publishing the container images to an image registry
- Working with persistent volume
- Deploying the Python application to Kubernetes

Requirements

You will need Docker, kubectl, and this source code [1].

Docker is an open platform to build and ship distributed applications. To install Docker, follow the official documentation [2]. To verify that Docker runs your system:

```
$ docker info
Containers: 0
Images: 289
Storage Driver: aufs
 Root Dir: /var/lib/docker/aufs
 Dirs: 289
Execution Driver: native-0.2
Kernel Version: 3.16.0-4-amd64
Operating System: Debian GNU/Linux 8 (jessie)
WARNING: No memory limit support
WARNING: No swap limit support
```

kubectl is a command-line interface for executing commands against a Kubernetes cluster. Run the shell script below to install kubectl:

```
curl -LO https://storage.googleapis.com/kubernetes-release/
  release/$(curl -s https://storage.googleapis.com/kubernetes-
  release/release/stable.txt)/bin/linux/amd64/kubectl
```

Deploying to Kubernetes requires a containerized application. Let's review containerizing Python applications.

Containerization at a glance

Containerization involves enclosing an application in a container with its own operating system. This full machine virtualization option has the advantage of being able to run an application on any machine without concerns about dependencies.

Roman Gaponov's article [3] serves as a reference. Let's start by creating a container image for our Python code.

Create a Python container image

To create these images, we will use Docker, which enables us to deploy applications inside isolated Linux software containers. Docker is able to automatically build images using instructions from a Docker file.

This is a Docker file for our Python application:

```
FROM python:3.6
MAINTAINER XenonStack

# Creating Application Source Code Directory
RUN mkdir -p /k8s_python_sample_code/src

# Setting Home Directory for containers
WORKDIR /k8s_python_sample_code/src

# Installing python dependencies
COPY requirements.txt /k8s_python_sample_code/src
RUN pip install --no-cache-dir -r requirements.txt

# Copying src code to Container
COPY . /k8s_python_sample_code/src/app

# Application Environment variables
ENV APP_ENV development

# Exposing Ports
EXPOSE 5035

# Setting Persistent data
VOLUME ["/app-data"]

# Running Python Application
CMD ["python", "app.py"]
```

This Docker file contains instructions to run our sample Python code. It uses the Python 3.5 development environment.

Build a Python Docker image

We can now build the Docker image from these instructions using this command:

```
docker build -t k8s_python_sample_code .
```

This command creates a Docker image for our Python application.

Publish the container images

We can publish our Python container image to different private/public cloud repositories, like Docker Hub, AWS ECR, Google Container Registry, etc. For this tutorial, we'll use Docker Hub.

Before publishing the image, we need to tag it to a version:

```
docker tag k8s_python_sample_code:latest k8s_python_sample_
  code:0.1
```

Push the image to a cloud repository

Using a Docker registry other than Docker Hub to store images requires you to add that container registry to the local Docker daemon and Kubernetes Docker daemons. You can look up this information for the different cloud registries. We'll use Docker Hub in this example.

Execute this Docker command to push the image:

```
docker push k8s_python_sample_code
```

Working with CephFS persistent storage

Kubernetes supports many persistent storage providers, including AWS EBS, CephFS, GlusterFS, Azure Disk, NFS, etc. I will cover Kubernetes persistence storage with CephFS.

To use CephFS for persistent data to Kubernetes containers, we will create two files:

persistent-volume.yml

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: app-disk1
  namespace: k8s_python_sample_code
spec:
  capacity:
  storage: 50Gi
  accessModes:
  - ReadWriteMany
  cephfs:
  monitors:
    - "172.17.0.1:6789"
  user: admin
  secretRef:
    name: ceph-secret
  readOnly: false
```

persistent_volume_claim.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: appclaim1
  namespace: k8s_python_sample_code
spec:
  accessModes:
  - ReadWriteMany
  resources:
  requests:
    storage: 10Gi
```

We can now use kubectl to add the persistent volume and claim to the Kubernetes cluster:

```
$ kubectl create -f persistent-volume.yml
$ kubectl create -f persistent-volume-claim.yml
```

We are now ready to deploy to Kubernetes.

Deploy the application to Kubernetes

To manage the last mile of deploying the application to Kubernetes, we will create two important files: a service file and a deployment file.

Create a file and name it `k8s_python_sample_code.service.yml` with the following content:

```
apiVersion: v1
kind: Service
metadata:
  labels:
  k8s-app: k8s_python_sample_code
  name: k8s_python_sample_code
  namespace: k8s_python_sample_code
spec:
  type: NodePort
  ports:
  - port: 5035
  selector:
  k8s-app: k8s_python_sample_code
```

Create a file and name it `k8s_python_sample_code.deployment.yml` with the following content:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: k8s_python_sample_code
  namespace: k8s_python_sample_code
spec:
  replicas: 1
  template:
  metadata:
    labels:
    k8s-app: k8s_python_sample_code
  spec:
    containers:
    - name: k8s_python_sample_code
      image: k8s_python_sample_code:0.1
      imagePullPolicy: "IfNotPresent"
      ports:
      - containerPort: 5035
      volumeMounts:
        - mountPath: /app-data
          name: k8s_python_sample_code
     volumes:
         - name: <name of application>
           persistentVolumeClaim:
             claimName: appclaim1
```

Finally, use kubectl to deploy the application to Kubernetes:

```
$ kubectl create -f k8s_python_sample_code.deployment.yml
  $ kubectl create -f k8s_python_sample_code.service.yml
```

Your application was successfully deployed to Kubernetes.

You can verify whether your application is running by inspecting the running services:

```
kubectl get services
```

May Kubernetes free you from future deployment hassles!

Want to learn more about Python? Nanjekye's book, Python 2 and 3 Compatibility [4] *offers clean ways to write code that will run on both Python 2 and 3, including detailed examples of how to convert existing Python 2-compatible code to code that will run reliably on both Python 2 and 3.*

Links

[1] https://github.com/jnanjekye/k8s_python_sample_code/tree/master
[2] https://docs.docker.com/engine/installation/
[3] https://hackernoon.com/docker-tutorial-getting-started-with-python-redis-and-nginx-81a9d740d091
[4] https://www.apress.com/gp/book/9781484229545

Author

Straight outta 256, Joannah chooses results over reasons. She is a passionate aviator. Show me the code!

Blockchain: Not just for cryptocurrency

BY KATE CHAPMAN

There's a lot more to blockchain than Bitcoin.

I DON'T REMEMBER the first time I heard about blockchain. I do, however, remember when I began to hear about it frequently. A couple of years ago, I was working on building tools for community land rights [1], when our partners and people at conferences began to ask us about it. A colleague and I sat down and said, "we need to figure out this blockchain thing," because we didn't even know how it was relevant, let alone what problems it might be able to fix. Before we started our research, I used to describe blockchain as "the technology that powers Bitcoin." Although this is accurate, it's not very instructive because most people haven't really considered how Bitcoin works.

Blockchain basics

A blockchain is a distributed set of data that uses cryptography to verify and secure that information. Each piece of data in a blockchain is called a block, and the blockchain is the entire set of that data.

A blockchain is a distributed set of data that uses cryptography to verify and secure that information.

Rather than having a central database server to store the data, everyone involved in the blockchain has a copy of the information. This enables each involved party to verify that an individual block is accurate using hashing and cryptography. Each block is created from a hash of some information. Anyone who has that same information can create the same hash to verify the block; however, they cannot go backward from the hash to re-create the data the block is about. Each person updating the blockchain uses a key that verifies that they are who they say they are.

Public vs. private blockchains

We have said everyone has a copy of the blockchain, but we haven't talked about who is "everyone." In a public blockchain, it could literally be *everyone*, as anyone can participate. Bitcoin and other cryptocurrencies are examples of public blockchains. Anyone can obtain Bitcoin (although whether they have enough knowledge or if it is practical is another story). They can purchase Bitcoin through another currency, sell something and get paid in Bitcoin, or mine Bitcoin themselves.

Private blockchains define who can participate [2]. A participant can either be approved by whoever set up the blockchain or through a set of rules that define if someone is approved. Private blockchains permit uses that might not work in a public blockchain, such as a bank verifying someone's identity [3].

Advanced blockchain applications

Advanced blockchain moves beyond simply recording and verifying transactions. Ethereum [4] is an example of an advanced use case. Because code can be executed on the Ethereum blockchain, it enables applications beyond those of a simple blockchain. One such use case is smart contracts [5]. Let's say you want to buy an item from me, but we don't know each other. Instead of just trusting each other, we could utilize blockchain technology to enable the transaction by using the following steps:

1. You put the agreed upon payment into an account.
2. Code is executed verifying that the payment is there.
3. I ship the item to you.
4. You verify that the item arrived.
5. Payment is released into my account.

All these steps could be turned into algorithms and run to verify each step in the transaction. When individuals sell something small, it's uncommon to make up a traditional contract, although they could. Much more complicated contracts, such as buying a house or executing a will, could be codified in the same way, using algorithms that verify the execution of the contract.

Future blockchain applications

Blockchain shows a lot of promise, but it is not without warnings. When many people participate in a blockchain, the transaction costs can become quite high. Bitcoin is already running into these issues [6]. Many non-cryptocurrency applications are being tested in promising pilots, but none has yet reached scale.

One thesis discussed by Oxford Internet Institute professor Vili Lehdonvirta is that blockchain will have its own governance issues [7], and if governance issues are fixed, blockchain might not be needed at all. He might have a point, as many of the issues I saw in the land rights sector were attempts to skirt existing governance issues like corruption. If those issues were fixed, there would be little need for the technology.

Governance challenges are something many of us in open source are all too familiar with. Unfortunately, there are seldom easy solutions.

Links

[1] https://opensource.com/article/17/1/land-rights-documentation-Cadasta
[2] https://www.ibm.com/blogs/blockchain/2017/05/the-difference-between-public-and-private-blockchain/
[3] https://medium.com/blockchain-review/private-blockchain-or-database-whats-the-difference-523e7d42edc
[4] https://www.ethereum.org/
[5] https://bitsonblocks.net/2016/02/01/a-gentle-introduction-to-smart-contracts/
[6] https://www.wired.com/story/bitcoin-global-warming/
[7] http://blogs.oii.ox.ac.uk/policy/the-blockchain-paradox-why-distributed-ledger-technologies-may-do-little-to-transform-the-economy/

Author

Kate Chapman has worked at the intersection of technology and nonprofits for the last decade. Currently, she is a principal at Cascadia Technical Mentorship, a consultancy she started with Chris Daley. Kate specializes in formalizing communities into organizational structures, product management, technical strategy, and building open communities. Her mantra is "people before data." Kate also serves as the chairperson of the Board of Directors of the OpenStreetMap Foundation and is a board member of the Software Freedom Conservancy.

Best Couple

Coupled commands with control operators in Bash

BY DAVID BOTH

Add logic to the command line with control operators in compound commands.

BEST COUPLE 2018

SIMPLE COMPOUND COMMANDS—such as stringing several commands together in a sequence on the command line—are used often. Such commands are separated by semicolons, which define the end of a command. To create a simple series of shell commands on a single line, simply separate each command using a semicolon, like this:

```
command1 ; command2 ; command3 ; command4 ;
```

You don't need to add a final semicolon because pressing the Enter key implies the end of the final command, but it's fine to add it for consistency.

All the commands will run without a problem—as long as no error occurs. But what happens if an error happens? We can anticipate and allow for errors using the && and || control operators built into Bash. These two control operators provide some flow control and enable us to alter the code-execution sequence. The semicolon and the `newline` character are also considered to be Bash control operators.

The && operator simply says "if command1 is successful, then run command2." If command1 fails for any reason, command2 won't run. That syntax looks like:

```
command1 && command2
```

This works because every command returns a code to the shell that indicates whether it completed successfully or failed during execution. By convention, a return code (RC) of 0 (zero) indicates success and any positive number indicates some type of failure. Some sysadmin tools just return a 1 to indicate any failure, but many use other positive numerical codes to indicate the type of failure.

The Bash shell's $? variable can be checked very easily by a script, by the next command in a list of commands, or even directly by a sysadmin. Let's look at RCs. We can run a simple command and immediately check the RC, which will always pertain to the last command that ran.

```
[student@studentvm1 ~]$ ll ; echo "RC = $?"
total 284
-rw-rw-r--  1 student student   130 Sep 15 16:21 ascii-program.sh
drwxrwxr-x  2 student student  4096 Nov 10 11:09 bin
<snip>
drwxr-xr-x. 2 student student  4096 Aug 18 10:21 Videos
RC = 0
[student@studentvm1 ~]$
```

This RC is 0, which means the command completed successfully. Now try the same command on a directory where we don't have permissions.

```
[student@studentvm1 ~]$ ll /root ; echo "RC = $?"
ls: cannot open directory '/root': Permission denied
RC = 2
[student@studentvm1 ~]$
```

This RC's meaning can be found in the **ls** command's man page [1].

Let's try the && control operator as it might be used in a command-line program. We'll start with something simple: Create a new directory and, if that is successful, create a new file in it.

We need a directory where we can create other directories. First, create a temporary directory in your home directory where you can do some testing.

```
[student@studentvm1 ~]$ cd ; mkdir testdir
```

Create a new directory in `~/testdir`, which should be empty because you just created it, and then create a new, empty file in that new directory. The following command will do those tasks.

```
[student@studentvm1 ~]$ mkdir ~/testdir/testdir2 && touch ~/
  testdir/testdir2/testfile1
[student@studentvm1 ~]$ ll ~/testdir/testdir2/
total 0
-rw-rw-r-- 1 student student 0 Nov 12 14:13 testfile1
[student@studentvm1 ~]$
```

We know everything worked as it should because the `testdir` directory is accessible and writable. Change the permissions on `testdir` so it is no longer accessible to the user `student` as follows:

```
[student@studentvm1 ~]$ chmod 076 testdir ; ll | grep testdir
d---rwxrw-. 3 student student  4096 Nov 12 14:13 testdir
[student@studentvm1 ~]$
```

Using the `grep` command after the long list (`ll`) shows the listing for `testdir`. You can see that the user `student` no longer has access to the `testdir` directory. Now let's run almost the same command as before but change it to create a different directory name inside `testdir`.

```
[student@studentvm1 ~]$ mkdir ~/testdir/testdir3 && touch ~/
  testdir/testdir3/testfile1
mkdir: cannot create directory '/home/student/testdir/
  testdir3': Permission denied
[student@studentvm1 ~]$
```

Although we received an error message, using the && control operator prevents the `touch` command from running because there was an error in creating `testdir3`. This type of command-line logical flow control can prevent errors from compounding and making a real mess of things. But let's make it a little more complicated.

The **||** control operator allows us to add another command that executes when the initial program statement returns a code larger than zero.

```
[student@studentvm1 ~]$ mkdir ~/testdir/testdir3 && touch
  ~/testdir/testdir3/testfile1 || echo "An error occurred while
  creating the directory."
mkdir: cannot create directory
  '/home/student/testdir/testdir3': Permission denied
An error occurred while creating the directory.
[student@studentvm1 ~]$
```

Our compound command syntax using flow control takes this general form when we use the && and **||** control operators:

```
preceding commands ; command1 && command2 || command3 ;
  following commands
```

The compound command using the control operators may be preceded and followed by other commands that can be related to the ones in the flow-control section but which are unaffected by the flow control. All of those commands will execute without regard to anything that takes place inside the flow-control compound command.

These flow-control operators can make working at the command line more efficient by handling decisions and letting us know when a problem has occurred. I use them directly on the command line as well as in scripts.

You can clean up as the root user to delete the directory and its contents.

```
[root@studentvm1 ~]# rm -rf /home/student/testdir
```

Links

[1] http://man7.org/linux/man-pages/man1/ls.1.html

Author

David Both is a Linux and Open Source advocate who resides in Raleigh, North Carolina. He has been in the IT industry for over forty years and taught OS/2 for IBM where he worked for over 20 years. While at IBM, he wrote the first training course for the original IBM PC in 1981. He has taught RHCE classes for Red Hat and has worked at MCI Worldcom, Cisco, and the State of North Carolina. He has been working with Linux and Open Source Software for almost 20 years. David has written articles for OS/2 Magazine, Linux Magazine, Linux Journal and OpenSource.com. His article "Complete Kickstart," co-authored with a colleague at Cisco, was ranked 9th in the Linux Magazine Top Ten Best System Administration Articles list for 2008.

Ribbon Image: Graphics Provided by vecteezy.com

How to avoid humiliating newcomers: A guide for advanced developers

BY A. JESSE

To sustain an open source community's growth, we need to welcome new developers. Unfortunately, we are not always a welcoming bunch.

EVERY YEAR IN NEW YORK CITY, a few thousand young men come to town, dress up like Santa Claus, and do a pub crawl. One year during this SantaCon event, I was walking on the sidewalk and minding my own business, when I saw an extraordinary scene. There was a man dressed up in a red hat and red jacket, and he was talking to a homeless man who was sitting in a wheelchair. The homeless man asked Santa Claus, "Can you spare some change?" Santa dug into his pocket and brought out a $5 bill. He hesitated, then gave it to the homeless man. The homeless man put the bill in his pocket.

In an instant, something went wrong. Santa yelled at the homeless man, "I gave you $5. I wanted to give you one dollar, but five is the smallest I had, so you oughtta be grateful. This is your lucky day, man. You should at least say thank you!"

This was a terrible scene to witness. First, the power difference was terrible: Santa was an able-bodied white man with money and a home, and the other man was black, homeless, and using a wheelchair. It was also terrible because Santa Claus was dressed like the very symbol of generosity! And he was behaving like Santa until, in an instant, something went wrong and he became cruel.

This is not merely a story about Drunk Santa, however; this is a story about technology communities. We, too, try to be generous when we answer new programmers' questions, and every day our generosity turns to rage. Why?

My cruelty

I'm reminded of my own bad behavior in the past. I was hanging out on my company's Slack when a new colleague asked a question.

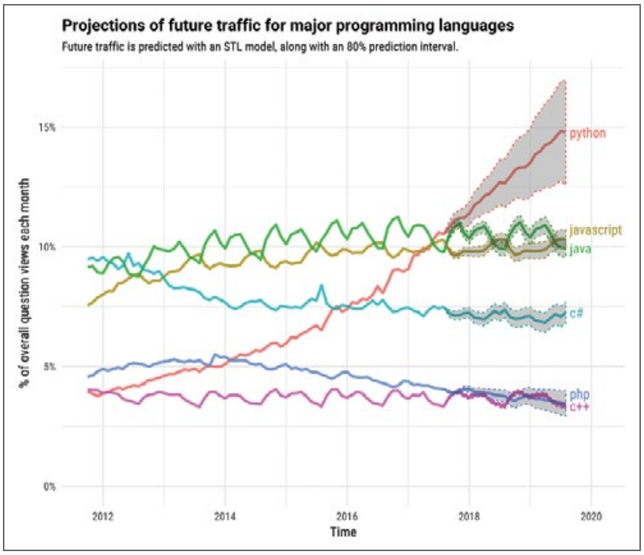
New Colleague: Hey, does anyone know how to do such-and-such with MongoDB?
Jesse: That's going to be implemented in the next release.
New Colleague: What's the ticket number for that feature?
Jesse: I memorize all ticket numbers. It's #12345.
New Colleague: Are you sure? I can't find ticket 12345.

He had missed my sarcasm, and his mistake embarrassed him in front of his peers. I laughed to myself, and then I felt terrible. As one of the most senior programmers at MongoDB, I should not have been setting this example. And yet, such behavior is commonplace among programmers everywhere: We get sarcastic with newcomers, and we humiliate them.

Why does it matter?

Perhaps you are not here to make friends; you are here to write code. If the code works, does it matter if we are nice to each other or not?

A few months ago on the Stack Overflow blog, David Robinson showed that Python has been growing dramatically [1], and it is now the top language that people view questions about on Stack Overflow. Even in the most pessimistic forecast, it will far outgrow the other languages this year.

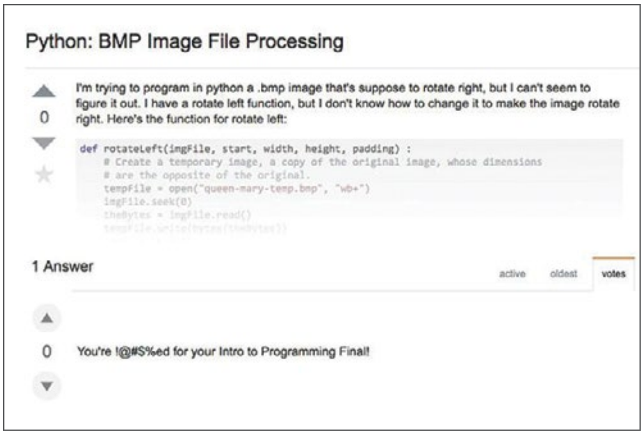


If you are a Python expert, then the line surging up and to the right is good news for you. It does not represent competition, but *confirmation*. As more new programmers learn Python, our expertise becomes ever more valuable, and we will see that reflected in our salaries, our job opportunities, and our job security.

But there is a danger. There are soon to be more new Python programmers than ever before. To sustain this growth, we must welcome them, and we are not always a welcoming bunch.

The trouble with Stack Overflow

I searched Stack Overflow for rude answers to beginners' questions, and they were not hard to find.



Python: BMP Image File Processing

I'm trying to program in python a .bmp image that's suppose to rotate right, but I can't seem to figure it out. I have a rotate left function, but I don't know how to change it to make the image rotate right. Here's the function for rotate left:

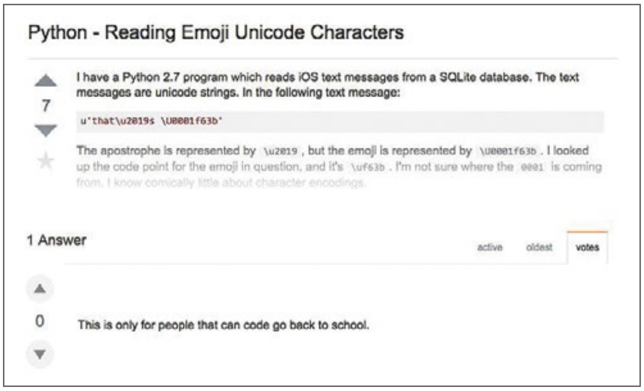
```
def rotateLeft(imgFile, start, width, height, padding) :
    # Create a temporary image, a copy of the original image, whose dimensions
    # are the opposite of the original.
    tempFile = open("queen-mary-temp.bmp", "wb+")
    imgFile.seek(0)
    theBytes = imgFile.read()
    tempFile.write(bytes(theBytes))
```

1 Answer

You're !@#S%ed for your Intro to Programming Final!

The message is plain: If you are asking a question this stupid, you are doomed. Get out.

I immediately found another example of bad behavior:



Python - Reading Emoji Unicode Characters

I have a Python 2.7 program which reads iOS text messages from a SQLite database. The text messages are unicode strings. In the following text message:

```
u'that\u2019s \U0001f63b'
```

The apostrophe is represented by `\u2019`, but the emoji is represented by `\U0001f63b`. I looked up the code point for the emoji in question, and it's `\uf63b`. I'm not sure where the `0001` is coming from. I know comically little about character encodings.

1 Answer

This is only for people that can code go back to school.

Who has never been confused by Unicode in Python? Yet the message is clear: You do not belong here. Get out.

Do you remember how it felt when you needed help and someone insulted you? It feels terrible. And it decimates the community. Some of our best experts leave every day because they see us treating each other this way. Maybe they still program Python, but they are no longer participating in conversations online. This cruelty drives away newcomers, too, particularly members of groups underrepresented in tech who might not be confident they belong. People who could have become the great Python programmers of the next generation, but if they ask a question and somebody is cruel to them, they leave.

This is not in our interest. It hurts our community, and it makes our skills less valuable because we drive people out. So, why do we act against our own interests?

Why generosity turns to rage

There are a few scenarios that really push my buttons. One is when I act generously but don't get the acknowledgment I expect. (I am not the only person with this resentment: This is probably why Drunk Santa snapped when he gave a $5 bill to a homeless man and did not receive any thanks.)

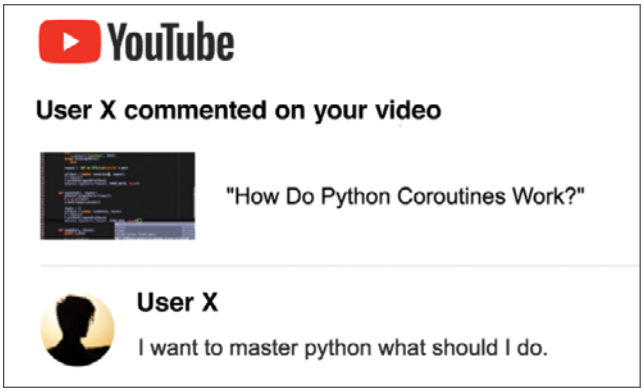
Another is when answering requires more effort than I expect. An example is when my colleague asked a question on Slack and followed-up with, "What's the ticket number?" I had judged how long it would take to help him, and when he asked for more help, I lost my temper.

These scenarios boil down to one problem: I have expectations for how things are going to go, and when those expectations are violated, I get angry.

I've been studying Buddhism for years, so my understanding of this topic is based in Buddhism. I like to think that the Buddha discussed the problem of expectations in his first tech talk when, in his mid-30s, he experienced a breakthrough after years of meditation and convened a small conference to discuss his findings. He had not rented a venue, so he sat under a tree. The attendees were a handful of meditators the Buddha had met during his wanderings in northern India. The Buddha explained that he had discovered four truths:

- First, that to be alive is to be dissatisfied—to want things to be better than they are now.
- Second, this dissatisfaction is caused by wants; specifically, by our expectation that if we acquire what we want and eliminate what we do not want, it will make us happy for a long time. This expectation is unrealistic: If I get a promotion or if I delete 10 emails, it is temporarily satisfying, but it does not make me happy over the long-term. We are dissatisfied because every material thing quickly disappoints us.
- The third truth is that we can be liberated from this dissatisfaction by accepting our lives as they are.
- The fourth truth is that the way to transform ourselves is to understand our minds and to live a generous and ethical life.

I still get angry at people on the internet. It happened to me recently, when someone posted a comment on a video I published about Python co-routines [2]. It had taken me months of research and preparation to create this video, and then a newcomer commented, "I want to master python what should I do."



YouTube

User X commented on your video

"How Do Python Coroutines Work?"

User X

I want to master python what should I do.

This infuriated me. My first impulse was to be sarcastic, "For starters, maybe you could spell Python with a capital P and end a question with a question mark." Fortunately, I recognized my anger before I acted on it, and closed the tab instead. Sometimes liberation is just a Command+W away.

What to do about it

If you joined a community with the intent to be helpful but on occasion find yourself flying into a rage, I have a method to prevent this. For me, it is the step when I ask myself, "Am I angry?" Knowing is most of the battle. Online, however, we can lose track of our emotions. It is well-established that one reason we are cruel on the internet is because, without seeing or hearing the other person, our natural empathy is not activated. But the other problem with the internet is that, when we use computers, we lose awareness of our bodies. I can be angry and type a sarcastic message without even knowing I am angry. I do not feel my heart pound and my neck grow tense. So, the most important step is to ask myself, "How do I feel?"

If I am too angry to answer, I can usually walk away. As Thumper learned in *Bambi* [3], "If you can't say something nice, don't say nothing at all."

The reward

Helping a newcomer is its own reward, whether you receive thanks or not. But it does not hurt to treat yourself to a glass of whiskey or a chocolate, or just a sigh of satisfaction after your good deed.

But besides our personal rewards, the payoff for the Python community is immense. We keep the line surging up and to the right. Python continues growing, and that makes our own skills more valuable. We welcome new members, people who might not be sure they belong with us, by reassuring them that there is no such thing as a stupid question. We use Python to create an inclusive and diverse community around writing code. And besides, it simply feels good to be part of a community where people treat each other with respect. It is the kind of community that I want to be a member of.

The three-breath vow

There is one idea I hope you remember from this article: To control our behavior online, we must occasionally pause and notice our feelings. I invite you, if you so choose, to repeat the following vow out loud:

I vow
to take three breaths
before I answer a question online.

This article is based on a talk, Why Generosity Turns To Rage, and What To Do About It, that Jesse gave at PyTennessee in February.

Links

[1] https://stackoverflow.blog/2017/09/06/incredible-growth-python/
[2] https://www.youtube.com/watch?v=7sCu4gEjH5I
[3] https://www.youtube.com/watch?v=nGt9jAkWie4

Author

A. Jesse is a staff engineer at MongoDB in New York City. He wrote Motor, the async MongoDB Python driver, and is the lead developer of the MongoDB C Driver. He contributes to PyMongo, asyncio, Python and Tornado. He studies at the International Center for Photography and practices at the Village Zendo.

Community metrics: The challenge behind the numbers

BY ILDIKO VANCSA

Although metrics are an important way to understand community members' effectiveness, they're only one piece of the puzzle.

WE ARE ALL OBSESSED with the numbers and statistics we can measure in our lives. We are concerned about our health, so we monitor our weight, blood pressure, and calorie intake. We also observe ourselves and our work environments to evaluate our efficiency and team dynamics. This mindset of focusing on the numbers carries over to how we evaluate open source communities.

Why are metrics important?

Open source communities, like the human body, are complex organizations with commonalities as well as unique operational characteristics and dynamics. By their nature, open source projects make a lot of data available, not just related to the source code, but also about contributors' processes and actions. This information gives us a better picture of a project's ecosystem and how it changes over time.

When evaluating community health and progress, communities typically look at metrics on contributions, diversity, and adoption of the artifacts they produce. Metrics can also be very helpful to find bottlenecks and identify changes in the balance of the ecosystem. Metrics can provide insights into community health, growth, and overall dynamics—but only if we use them wisely.

Why looking beyond the numbers is crucial

Although metrics are widely used and essential to understanding the community, being careful about how we use the numbers is important. There are no magic "healthy" numbers in open source community metrics. In fact, numbers can be misleading unless you look further into the details and context. You could get an incomplete picture, for example, if you only count code contributions and overlook valuable documentation and tests in other parts of repositories.

Finally, repeatedly collecting and publishing the same metrics can cause people to try to game the system, resulting in unhealthy community behavior. Judging a community's health purely on numbers could lead to false conclusions and inappropriate follow-up actions, so how can we do better?

> Metrics can provide insights into community health, growth, and overall dynamics—but only if we use them wisely.

Case study: Code reviews

Code reviews are highly encouraged in both corporate environments and open source projects to identify and fix problems before they go live. Code reviewers learn the most about the code and changes in the software, and project maintainers rely on steady contributors' opinions before merging new changes. So how do metrics come into the picture?

Measuring the number of positive and negative code reviews over a specific period (e.g., a quarter of a year or per release cycle) is easy. Many open source projects publish these activity metrics with options to filter results on things such as data about one contributor or all contributors working for the same company.

Although the tools used by open source projects are accessible by anyone (which means anyone can extract the numbers), publishing these metrics on a dashboard may lead to gaming them over time. For example, people may try to have the most reviews, thinking it will speed up their acceptance to the community, or companies may encourage employees to generate higher numbers to improve their reputation with customers.

The unfortunate consequence of trying to increase these numbers quickly is that the quality of the code reviews drops. One example is a negative review in which the reviewer just repeats what the automated testing system pointed out. Another is a reviewer simply saying he or she agrees with previous reviewers, which adds nothing to the discussion. Or even less helpful, the reviewer merely adds a "+1" mark (which means the change looks good) on as many open changes as possible with no meaningful comment.

There are multiple problems with these behaviors. These meaningless reviews are disturbing for active contributors trying to help code authors get the highest quality changes merged. Not to mention that people who aren't trying to help maintain the project, rather trying only to boost their statistics in the open dashboards, are annoying to regular contributors. Also, people who abuse the system like this are easy to recognize, and often their reputation goes down once they are identified.

How to use metrics better

Education is important to address these challenges. The success of an open source project depends upon a group of people who care about the topic and the technology working to maintain the source code, tests, and documentation. Metrics are important to get an overall picture of the balance of the ecosystem, which is not driven by any single metric, rather a combination of multiple key performance indicators (KPIs).

When we look at metrics, like the number of code reviews, we must always look beyond the number itself and understand how we can use the data for growth and reflection on whether we are moving in the right direction.

We need to ask key questions to identify which metrics we should look into and how to combine them to gain meaningful information. For example:

- Why is a data point important for us (or our managers)?
- What does it mean to have a higher or lower number?
- And what do changes over time say?

Or to look back at an earlier example:

- What does the ration of negative and positive reviews mean?

Measuring only a single set of metrics and thinking only the numbers matter is a bad idea. Instead, dig deeper and look behind the numbers.

Author

Ildiko started her journey with virtualization during the university years and has been in connection with this technology different ways since then. She started her career at a small research and development company in Budapest, where she was focusing on areas like system management and business process modelling and optimization. Ildiko got in touch with OpenStack when she started to work in the cloud project at Ericsson in 2013. She was a member of the Ceilometer and Aodh core teams, now she drives NFV related feature development activities in projects like Nova and Cinder. Beyond code and documentation contributions she is also very passionate about on boarding and training activities, which is one of her focus areas within the OpenStack Foundation.

6 ways programmers from underrepresented countries can get ahead

BY IVANGE LARRY

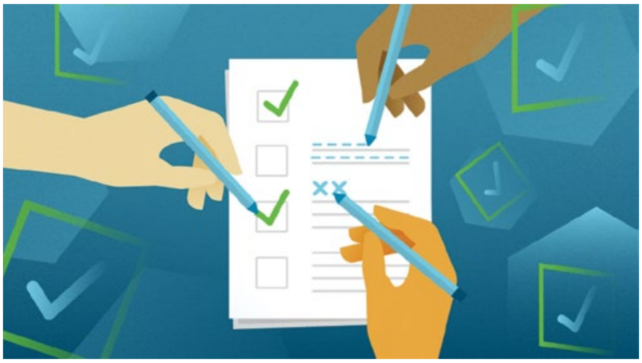
It's harder for programmers from less-privileged nations trying to achieve success alongside people from countries with many material advantages.

BECOMING A PROGRAMMER from an underrepresented community like Cameroon is tough. Many Africans don't even know what computer programming is—and a lot who do think it's only for people from Western or Asian countries.

I didn't own a computer until I was 18, and I didn't start programming until I was a 19-year-old high school senior, and had to write a lot of code on paper because I couldn't be carrying my big desktop to school. I have learned a lot over the past five years as I've moved up the ladder to become a successful programmer from an underrepresented community. While these lessons are from my experience in Africa, many apply to other underrepresented communities, including women.

1. Learn how to code

This is obvious: To be a successful programmer, you first have to be a programmer. In an African community, this may not be very easy. To learn how to code you need a computer and probably internet, too, which aren't very common for Africans to have. I didn't own a desktop computer until I was 18 years old—and I didn't own a laptop until I was about 20, and some may have still considered me privileged. Some students don't even know what a computer looks like until they get to the university.



You still have to find a way to learn how to code. Before I had a computer, I used to walk for miles to see a friend who had one. He wasn't very interested in it, so I spent a lot of time with it. I also visited cybercafes regularly, which consumed most of my pocket money.

Take advantage of local programming communities, as this could be one of your greatest sources of motivation. When you're working on your own, you may feel like a ninja, but that may be because you do not interact much with other programmers. Attend tech events. Make sure you have at least one friend who is better than you. See that person as a competitor and work hard to beat them, even though they may be working as hard as you are. Even if you never win, you'll be growing in skill as a programmer.

Take advantage of local programming communities, as this could be one of your greatest sources of motivation.

2. Don’t read too much into statistics

A lot of smart people in underrepresented communities never even make it to the “learning how to code” part because they take statistics as hard facts. I remember when I was aspiring to be a hacker, I used to get discouraged about the statistic that there are far fewer black people than white people in technology. If you google the “top 50 computer programmers of all time,” there probably won’t be many (if any) black people on the list. Most of the inspiring names in tech, like Ada Lovelace, Linus Torvalds, and Bill Gates, are white.

Growing up, I always believed technology was a white person’s thing. I used to think I couldn’t do it. When I was young, I never saw a science fiction movie with a black man as a hacker or an expert in computing. It was always white people. I remember when I got to high school and our teacher wrote that programming was part of our curriculum, I thought that was a joke—I wondered, “since when and how will that even be possible?” I wasn’t far from the truth. Our teachers couldn’t program at all.

Statistics also say that a lot of the amazing, inspiring programmers you look up to, no matter what their color, started coding at the age of 13. But you didn’t even know programming existed until you were 19. You ask yourself questions like: How am I going to catch up? Do I even have the intelligence for this? When I was 13, I was still playing stupid, childish games—how can I compete with this?

This may make you conclude that white people are naturally better at tech. That’s wrong. Yes, the statistics are correct, but they’re just statistics. And they can change. Make them change. Your environment contributes a lot to the things you do while growing up. How can you compare yourself to someone whose parents got him a computer before he was nine—when you didn’t even see one until you were 19? That’s a 10-year gap. And that nine-year-old kid also had a lot of people to coach him.

How can you compare yourself to someone whose parents got him a computer before he was nine—when you didn’t even see one until you were 19?

You can be a great software engineer regardless of your background. It may be a little harder because you may not have the resources or opportunities people in the western world have, but it’s not impossible.

3. Have a local hero or mentor

I think having someone in your life to look up to is one of the most important things. We all admire people like Linus Torvalds and Bill Gates but trying to make them your role models can be demotivating. Bill Gates began coding at age 13 and formed his first venture at age 17. I’m 24 and still trying to figure out what I want to do with my life. Those stories always make me wonder why I’m not better yet, rather than looking for reasons to get better.

Having a local hero or mentor is more helpful. Because you’re both living in the same community, there’s a greater chance there won’t be such a large gap to discourage you. A local mentor probably started coding around the age you did and was unlikely to start a big venture at a very young age.

I’ve always admired the big names in tech and still do. But I never saw them as mentors. First, because their stories seemed like fantasy to me, and second, I couldn’t reach them. I chose my mentors and role models to be those near my reach. Choosing a role model doesn’t mean you just want to get to where they are and stop. Success is step by step, and you need a role model for each stage you’re trying to reach. When you attain a stage, get another role model for the next stage.

Success is step by step, and you need a role model for each stage you’re trying to reach.

You probably can’t get one-on-one advice from someone like Bill Gates. You can get the advice they’re giving to the public at conferences, which is great, too. I always follow smart people. But advice that makes the most impact is advice that is directed to you. Advice that takes into consideration your goals and circumstances. You can get that only from someone you have direct access to.

I’m a product of many mentors at different stages of my life. One is Nyah Check [1], who was a year ahead of me at the university, but in terms of skill and experience, he was two to three years ahead. I heard stories about him when I was still in high school. He made people want to be great programmers, not just focus on getting a 4.0 GPA. He was one of the first people in French-speaking Africa to participate in Google Summer of Code [2]. While still at the university, he traveled abroad more times than many lecturers would dream of—without spending a dime. He could write code that even our course instructors couldn’t understand. He co-founded Google Developer Group Buea [3] and created an elite programmers club that helped many students learn to code. He started a lot of other communities, like the Docker Buea meetup [4] that I’m the lead organizer for.

These things inspired me. I wanted to be like him and knew what I would gain by becoming friends with him. Discussions with him were always very inspiring—he talked about programming and his adventures traveling the world for conferences. I learned a lot from him, and

I think he taught me well. Now younger students want to be around me for the same reasons I wanted to learn from him.

4. Get involved with open source

If you're in Africa and want to gain top skills from top engineers, your best bet is to join an open source project. The tech ecosystem in Africa is small and mostly made of startups, so getting experience in a field you love might not be easy. It's rare for startups in Africa to be working with machine learning, distributed computing, or containers and technologies like Kubernetes. Unless your passion is web development, your best bet is joining an open source project. I've learned most of what I know by being part of the OpenMRS [5] community. I've also contributed to other open source projects including LibreHealth [6], Coala [7], and Kubernetes [8]. Along with gaining tech skills, you'll be building your network of influential people. Most of my peers know about Linus Torvalds from books, but I have a picture with him.

Participate in open source outreach programs like Google Summer of Code, Google Code-in [9], Outreachy [10], or Linux Foundation Networking Internships [11]. These opportunities help you gain skills that may not be available in startups.

I participated in Google Summer of Code twice as a student, and I'm now a mentor. I've been a Google Code-in org admin, and I'm volunteering as an open source developer. All these activities help me learn new things.

5. Take advantage of diversity programs while you can

Diversity programs are great, but if you're like me, you may not like to benefit very much from them. If you're on a team of five and the basis of your offer is that you're a black person and the other four are white, you might wonder if you're really good enough. You won't want people to think a foundation sponsored your trip because you're black rather than because you add as much value as anyone else. It's never only that you're a minority—it's because the sponsoring organization thinks you're an exceptional minority. You're not the only person who applied for the diversity scholarship, and not everyone that applied won the award. Take advantage of diversity opportunities while you can and build your knowledge base and network.

When people ask me why the Linux Foundation sponsored my trip to the Open Source Summit, I say: "I was invited to give a talk at their conference, but they have diversity scholarships you can apply for." How cool does that sound?

Attend as many conferences as you can—diversity scholarships can help. Learn all you can learn. Practice what you learn. Get to know people. Apply to give talks. Start small. My right leg used to shake whenever I stood in front of a crowd to give a speech, but with practice, I've gotten better.

6. Give back

Always find a way to give back. Mentor someone. Take up an active role in a community. These are the ways I give back to my community. It isn't only a moral responsibility—it's a win-win because you can learn a lot while helping others get closer to their dreams.

I was part of a Programming Language meetup organized by Google Developer Group Buea where I mentored 15 students in Java programming (from beginner to intermediate). After the program was over, I created a Java User Group to keep the Java community together. I recruited two members from the meetup to join me as volunteer developers at LibreHealth, and under my guidance, they made useful commits to the project. They were later accepted as Google Summer of Code students, and I was assigned to mentor them during the program. I'm also the lead organizer for Docker Buea, the official Docker meetup in Cameroon, and I'm also Docker Campus Ambassador.

Taking up leadership roles in this community has forced me to learn. As Docker Campus Ambassador, I'm supposed to train students on how to use Docker. Because of this, I've learned a lot of cool stuff about Docker and containers in general.

Links

[1] https://github.com/Ch3ck
[2] https://summerofcode.withgoogle.com/
[3] http://www.gdgbuea.net/
[4] https://www.meetup.com/Docker-Buea/?_cookie-check=EnOn1Ct-CS4o1YOw
[5] https://openmrs.org/
[6] https://librehealth.io/
[7] https://coala.io/#/home
[8] https://kubernetes.io/
[9] https://codein.withgoogle.com/archive/
[10] https://www.outreachy.org/
[11] https://wiki.lfnetworking.org/display/LN/LF+Networking+Internships

Author

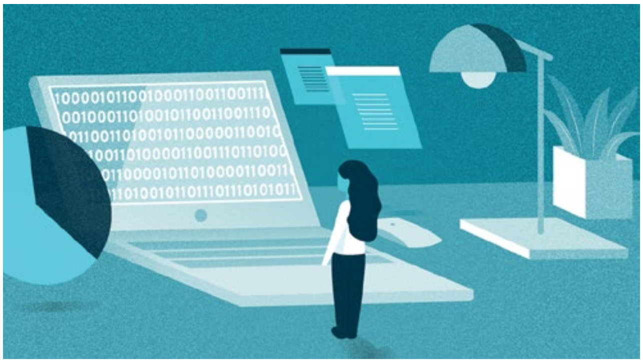
A final-year computer engineering student, Larry has been an open source enthusiast since 2015. In 2016, he was accepted for a Google Summer of Code internship with OpenMRS, continuing with LibreHealth in 2017. He has contributed to a number of open source projects, including OpenMRS, LibreHealth, Coala, and Kubernetes. Currently a Google Summer of Code mentor with OpenMRS and LibreHealth, Larry serves in Docker Buea (the official Docker meetup for Cameroon), lead and Docker ambassador at the University of Buea, and founder and lead of the Buea Java User Group (BueaJUG). He is also a GDG organizer.

10 principles of resilience for women in tech

BY JENNIFER CLOER

We need everyone at the table, in the lab, at the conference and in the boardroom.

BEING A WOMAN in tech is pretty damn cool. For every headline about what Silicon Valley thinks of women [1], there are tens of thousands of women building, innovating, and managing technology teams around the world. Women are helping build the future despite the hurdles they face, and the community of women and allies growing to support each other is stronger than ever. From BetterAllies [2] to organizations like Girls Who Code [3] and communities like the one I met recently at Red Hat Summit [4], there are more efforts than ever before to create an inclusive community for women in tech.



But the tech industry has not always been this welcoming, nor is the experience for women always aligned with the aspiration. And so we're feeling the pain. Women in technology roles have dropped from its peak in 1991 at 36% to 25% today, according to a report by NCWIT [5]. Harvard Business Review estimates [6] that more than half of the women in tech will eventually leave due to hostile work conditions. Meanwhile, Ernst & Young recently shared a study [7] and found that merely 11% of high school girls are planning to pursue STEM careers.

We have much work to do, lest we build a future that is less inclusive than the one we live in today. We need everyone at the table, in the lab, at the conference, and in the boardroom.

I've been interviewing both women and men for more than a year now about their experiences in tech, all as part of The Chasing Grace Project [8], a documentary series about women in tech. The purpose of the series is to help recruit and retain female talent for the tech industry and to give women a platform to be seen, heard, and acknowledged for their experiences. We believe that compelling story can begin to transform culture.

What Chasing Grace taught me

What I've learned is that no matter the dismal numbers, women want to keep building and they collectively possess a resilience unmatched by anything I've ever seen. And this is inspiring me. I've found a power, a strength, and a beauty in every story I've heard that is the result of resilience. I recently shared with the attendees at the Red Hat Summit Women's Leadership Luncheon the top 10 principles of resilience I've heard from throughout my interviews so far. I hope that by sharing them here the ideas and concepts can support and inspire you, too.

1. Practice optimism
When taken too far, optimism can give you blind spots. But a healthy dose of optimism allows you to see the best in people and situations and that positive energy comes back to you 100-fold. I haven't met a woman yet as part of this project who isn't an optimist.

2. Build mental toughness
When I recently asked a 32-year-old tech CEO, who is also a single mom of three young girls, what being a CEO required she said *mental toughness*. It really summed up what I'd heard in other words from other women, but it connected with me on another level when she proceeded to tell me how caring for her daughter—who was born with a hole in heart—prepared her for what she would encounter as a tech CEO. Being mentally tough to her means fighting for what you love, persisting like a badass, and building your EQ as well as your IQ.

> I haven't met a woman yet as part of this project who isnít an optimist.

3. Recognize your power
Most of the women I've interviewed don't know their own power and so they give it away unknowingly. Too many women have told me that they willingly took on the housekeeping roles on their teams—picking up coffee, donuts, office supplies, and making the team dinner reservations. Usually the only woman on their teams, this put them in a position to be seen as less valuable than their male peers who didn't readily volunteer for such tasks. All of us, men and women, have innate powers. Identify and know what your powers are and understand how to use them for good. You have so much more power than you realize. Know it, recognize it, use it strategically, and don't give it away. It's yours.

4. Know your strength
Not sure whether you can confront your boss about why you haven't been promoted? You can. You don't know your strength until you exercise it. Then, you're unstoppable. Test your strength by pushing your fear aside and see what happens.

5. Celebrate vulnerability
Every single successful women I've interviewed isn't afraid to be vulnerable. She finds her strength in acknowledging where she is vulnerable and she looks to connect with others in that same place. Exposing, sharing, and celebrating each other's vulnerabilities allows us to tap into something far greater than simply asserting strength; it actually builds strength—mental and emotional muscle. One women with whom we've talked shared how starting her own tech company made her feel like she was letting her husband down. She shared with us the details of that conversation with her husband. Honest conversations that share our doubts and our aspirations is what makes women uniquely suited to lead in many cases. Allow yourself to be seen and heard. It's where we grow and learn.

6. Build community
Building community seems like a no-brainer in the world of open source, right? But take a moment to think about how many minorities in tech, especially those outside the collaborative open source community, don't always feel like part of the community. Many women in tech, for example, have told me they feel alone. Reach out and ask questions or answer questions in community forums, at meetups, and in IRC and Slack. When you see a woman alone at an event, consider engaging with her and inviting her into a conversation. Start a meetup group in your company or community for women in tech. I've been so pleased with the number of companies that host these groups. If it doesn't exists, build it.

> If it doesn't exist, build it.

7. Celebrate victories
One of my favorite Facebook groups is TechLadies [9] because of its recurring hashtag #YEPIDIDTHAT. It allows women to share their victories in a supportive community. No matter how big or small, don't let a victory go unrecognized. When you recognize your wins, you own them. They become a part of you and you build on top of each one.

8. Be curious
Being curious in the tech community often means asking questions: How does that work? What language is that written in? How can I make this do that? When I've managed teams over the years, my best employees have always been those who ask a lot of questions, those who are genuinely curious about things. But in this context, I mean be curious when your gut tells you something doesn't seem right. *The energy in the meeting was off. Did he/she just say what I think he said?* Ask questions. Investigate. Communicate openly and clearly. It's the only way change happens.

9. Harness courage
One women told me a story about a meeting in which the women in the room kept being dismissed and talked over. During the debrief roundtable portion of the meeting, she

called it out and asked if others noticed it, too. Being a 20-year tech veteran, she'd witnessed and experienced this many times but she had never summoned the courage to speak up about it. She told me she was incredibly nervous and was texting other women in the room to see if they agreed it should be addressed. She didn't want to be a "troublemaker." But this kind of courage results in an increased understanding by everyone in that room and can translate into other meetings, companies, and across the industry.

10. Share your story

Share your experience with a friend, a group, a community, or an industry. Be empowered by the experience of sharing your experience. Stories change culture. When people connect to a compelling story, they begin to change behaviors. When people act, companies and industries begin to transform.

> When people connect to a compelling story, they begin to change behaviors.

If you would like to support The Chasing Grace Project [8], email Jennifer Cloer to learn more about how to get involved: jennifer@wickedflicksproductions.com

Links

[1] http://www.newsweek.com/2015/02/06/what-silicon-valley-thinks-women-302821.html

[2] https://opensource.com/article/17/6/male-allies-tech-industry-needs-you

[3] https://twitter.com/GirlsWhoCode

[4] http://opensource.com/tags/red-hat-summit

[5] https://www.ncwit.org/sites/default/files/resources/womenintech_facts_fullreport_05132016.pdf

[6] http://www.latimes.com/business/la-fi-women-tech-20150222-story.html

[7] http://www.ey.com/us/en/newsroom/news-releases/ey-news-new-research-reveals-the-differences-between-boys-and-girls-career-and-college-plans-and-an-ongoing-need-to-engage-girls-in-stem

[8] https://www.chasinggracefilm.com/

[9] https://www.facebook.com/therealTechLadies/

Author

Jennifer's career has been dedicated to telling the stories that have defined a generation of technology developers, from Linux creator Linus Torvalds to the men and women who started Creative Commons and Google's first I/O Conference. For more than 15 years, Jennifer has been a woman in tech and has been recognized for her storytelling acumen by BusinessInsider, who ranked her among the best PR people in tech for her video storytelling works. She is the creator and executive producer of The Chasing Grace Project (http://www.chasinggracefilm.com) and co-founder of Wicked Flicks, a film/video production house working with companies and individuals to affect change through original content. She is also founder and lead consultant at reTHINKit PR. She was most recently VP of communications at The Linux Foundation, where she oversaw brand storytelling and team of PR, social media and video production professionals. Prior to that, she was vice president at Page One PR and held posts both in house and at agency in communications. In addition to her BusinessInsider recognition, CIO.com identified her as one of the most influential women in open source.

6 tips for receiving feedback on your open source contributions

BY VM (VICKY) BRASSEUR

Receiving feedback can be hard. These tips will help.

IN THE FREE AND OPEN SOURCE software world, there are few moments as exciting or scary as submitting your first contribution to a project. You've put your work out there and now it's subject to review and feedback by the rest of the community.

Not to put it too lightly, but feedback is great. Without feedback we keep making the same mistakes. Without feedback we can't learn and grow and evolve. It's one of the keys that makes free and open source collaboration work.

Unfortunately, most of us have a hard time receiving feedback, let alone accepting it. We identify too closely with our contribution, such that criticisms of it—no matter how valid—are taken personally and put us on the defensive.

It doesn't help that most of us also have a hard time giving feedback, often delivering criticisms without empathy or in ways that are directed more at the person than at their contribution.

Both receiving and giving feedback are skills that can be learned and honed through practice. As you enter into this world of free and open source contributions, I encourage you to remember these tips:

1. **You are not your contribution.** Even if the person providing the feedback is unskilled at it, and their criticisms come across as personally directed, try not to take their comments in that way. Try to focus on the aspects of their feedback that relate directly to your contribution, then guide the feedback conversation toward these elements.
2. **It's not personal.** Problems found with your contribution are not problems found with you. You've put a lot of time and effort into that contribution, so naturally you feel a bit attached to it and that's OK. It's right to feel pride in what you've created and accomplished. But it's better to recognize that there's always a way to improve your contribution. Collaborate with those providing feedback to help evolve the contribution, the project, your knowledge, and your skills.
3. **Feedback is a gift.** When people provide feedback on your contribution, they're freely sharing their knowledge and experience with you. You can use this feedback to grow into a more skilled contributor, then one day pay that gift forward as you provide feedback to others. This is part of the beneficial cycle that allows free and open source to grow.
4. **Feedback and questions help make you better at what you do.** That's because feedback and questions help you see things you never have before and expand your mind and experiences in ways you never anticipated. None of

us are perfect. None of us are all knowing. All of us have been in your position before: feeling excited at the newness but more than a little lost in it as well. It's OK. Ask questions. Ask for feedback. It's the only way not to feel lost, and we all want to help you.

Collaborate with those providing feedback to help evolve the contribution, the project, your knowledge, and your skills.

5. **If you get angry at some feedback, step away for a bit to cool off before responding.** It happens: A piece of feedback will get under your skin. Perhaps it was the way it was phrased. Maybe it's dismissing an implementation about which you have strong opinions. Or maybe the person who gave the feedback is just an indelicate chowderhead. Like I said: It happens. Just because you're angry does not mean you have to react immediately. Replying in the heat of the moment rarely ends well for anyone involved. Take time to cool off before responding. Go for a walk. Play with your pets or your kids. Spend some time on a hobby or other project. Fire up a good movie or video game. Whatever it takes, give yourself space from the offending comment. Once you've had the time to cool off and think it over more, then you can *respond* rather than *react*.
6. **Always Assume Good Intent.** Above all, always assume good intent with all feedback. No matter how poorly a piece of feedback may be delivered, the person providing it is still giving you that gift of their knowledge and experience. They're not (usually) doing it to show off; they want the best for the project, for the contribution, and for you. Respect that and them and help them help you provide the best contribution you can. They mean well. Do you?

These tips will help you keep the perspective needed to get the most out of the feedback you'll receive on your first contribution. But what if you're the one *providing* the feedback? The next article [1] in this series has you covered there, too.

Adapted from Forge Your Future with Open Source [2] *by VM (Vicky) Brasseur, Copyright © 2018 The Pragmatic Programmers LLC. Reproduced with the permission of the publisher.*

Links

[1] https://opensource.com/article/18/10/4-best-practices-giving-open-source-code-feedback

[2] http://www.pragprog.com/titles/vbopens

Author

VM (aka Vicky) spent most of her 20 years in the tech industry leading software development departments and teams, and providing technical management and leadership consulting for small and medium businesses. Now she leverages nearly 30 years of free and open source software experience and a strong business background to advise companies about free/open source, technology, community, business, and the intersections between them.

She is the author of Forge Your Future with Open Source, the first book to detail how to contribute to free and open source software projects. Think of it as the missing manual of open source contributions and community participation. The book is published by The Pragmatic Programmers and is now available at https://fossforge.com.

Vicky is the Vice President of the Open Source Initiative, a moderator and author for opensource.com, an author for Linux Journal, and a frequent and popular speaker at free/open source conferences and events. She's the proud winner of the Perl White Camel Award (2014) and the O'Reilly Open Source Award (2016). She blogs about free/open source, business, and technical management at {anonymous => 'hash'};

4 best practices for giving open source code feedback

BY VM (VICKY) BRASSEUR

A few simple guidelines can help you provide better feedback.

IN THE PREVIOUS ARTICLE I gave you tips for how to *receive* feedback [1], especially in the context of your first free and open source project contribution. Now it's time to talk about the other side of that same coin: *providing* feedback.

If I tell you that something you did in your contribution is "stupid" or "naive," how would you feel? You'd probably be angry, hurt, or both, and rightfully so. These are mean-spirited words that when directed at people, can cut like knives. Words matter, and they matter a great deal. Therefore, put as much thought into the words you use when leaving feedback for a contribution as you do into any other form of contribution you give to the project. As you compose your feedback, think to yourself, "How would I feel if someone said this to me? Is there some way someone might take this another way, a less helpful way?" If the answer to that last question has even the chance of being a yes, backtrack and rewrite your feedback. It's better to spend a little time rewriting now than to spend a lot of time apologizing later.

When someone does make a mistake that seems like it should have been obvious, remember that we all have different experiences and knowledge. What's obvious to you may not be to someone else. And, if you recall, there once was a time when that thing was not obvious to you. We all make mistakes. We all typo. We all forget commas, semicolons, and closing brackets. Save yourself a lot of time and effort: Point out the mistake, but leave out the judgement. Stick to the facts. After all, if the mistake is that obvious, then no critique will be necessary, right?

Point out the mistake, but leave out the judgement. Stick to the facts.

1. **Avoid ad hominem comments.** Remember to review only the contribution and not the person who contributed it. That is to say, point out, "*the contribution* could be more efficient here in this way…" rather than, "*you* did this inefficiently." The latter is *ad hominem* feedback. *Ad hominem* is a Latin phrase meaning "to the person," which is where your feedback is being directed: to the person who contributed it rather than to the contribution itself. By providing feedback on the person you make that feedback personal, and the contributor is justified in taking it personally. Be careful when crafting your feedback to make sure you're addressing only the contents of the contribution

and not accidentally criticizing the person who submitted it for review.

2. **Include positive comments.** Not all of your feedback has to (or should) be critical. As you review the contribution and you see something that you like, provide feedback on that as well. Several academic studies—including an important one by Baumeister, Braslavsky, Finkenauer, and Vohs [2]—show that humans focus more on negative feedback than positive. When your feedback is solely negative, it can be very disheartening for contributors. Including positive reinforcement and feedback is motivating to people and helps them feel good about their contribution and the time they spent on it, which all adds up to them feeling more inclined to provide another contribution in the future. It doesn't have to be some gushing paragraph of flowery praise, but a quick, "Huh, that's a really smart way to handle that. It makes everything flow really well," can go a long way toward encouraging someone to keep contributing.

> When your feedback is solely negative, it can be very disheartening for contributors.

3. **Questions are feedback, too.** Praise is one less common but valuable type of review feedback. Questions are another. If you're looking at a contribution and can't tell why the submitter did things the way they did, or if the contribution just doesn't make a lot of sense to you, asking for more information acts as feedback. It tells the submitter that something they contributed isn't as clear as they thought and that it may need some work to make the approach more obvious, or if it's a code contribution, a comment to explain what's going on and why. A simple, "I don't understand this part here. Could you please tell me what it's doing and why you chose that way?" can start a dialogue that leads to a contribution that's much easier for future contributors to understand and maintain.
4. **Expect a negotiation.** Using questions as a form of feedback implies that there will be answers to those questions, or perhaps other questions in response. Whether your feedback is in question or statement format, you should expect to generate some sort of dialogue throughout the process. An alternative is to see your feedback as incontrovertible, your word as law. Although this is definitely one approach you can take, it's rarely a good one. When providing feedback on a contribution, it's best to collaborate rather than dictate. As these dialogues arise, embracing them as opportunities for conversation and learning on both sides is important. Be willing to discuss their approach and your feedback, and to take the time to understand their perspective.

The bottom line is: Don't be a jerk. If you're not sure whether the feedback you're planning to leave makes you sound like a jerk, pause to have someone else review it before you click *Send*. Have empathy for the person at the receiving end of that feedback. While the maxim is thousands of years old, it still rings true today that you should try to do unto others as you would have them do unto you. Put yourself in their shoes and aim to be helpful and supportive rather than simply being right.

Adapted from Forge Your Future with Open Source [3] *by VM (Vicky) Brasseur, Copyright © 2018 The Pragmatic Programmers LLC. Reproduced with the permission of the publisher.*

Links

[1] https://opensource.com/article/18/10/6-tips-receiving-feedback

[2] https://www.msudenver.edu/media/content/sri-taskforce/documents/Baumeister-2001.pdf

[3] http://www.pragprog.com/titles/vbopens

Author

VM (aka Vicky) spent most of her 20 years in the tech industry leading software development departments and teams, and providing technical management and leadership consulting for small and medium businesses. Now she leverages nearly 30 years of free and open source software experience and a strong business background to advise companies about free/open source, technology, community, business, and the intersections between them.

She is the author of Forge Your Future with Open Source, the first book to detail how to contribute to free and open source software projects. Think of it as the missing manual of open source contributions and community participation. The book is published by The Pragmatic Programmers and is now available at https://fossforge.com.

Vicky is the Vice President of the Open Source Initiative, a moderator and author for opensource.com, an author for Linux Journal, and a frequent and popular speaker at free/open source conferences and events. She's the proud winner of the Perl White Camel Award (2014) and the O'Reilly Open Source Award (2016). She blogs about free/open source, business, and technical management at {anonymous => 'hash'};

8 unusual FOSS tools for agile teams

BY MARIANNE FEIFER AND JEN KRIEGER

In this list, there are no project management apps, no checklists, and no integrations with GitHub. Just simple ways to organize your thoughts and promote team communication.

YOU MIGHT BE FAMILIAR with the expression: *So many tools, so little time.* In order to try to save you some time, I've outlined some of my favorite tools that help agile teams work better. If you are an *agilist*, chances are you're aware of similar tools, but I'm specifically narrowing down the list to tools that appeal to open source enthusiasts.

Caution! These tools are a little different than what you may be expecting. There are no project management apps—there is a great article [1] on that already—so there are no checklists, no integrations with GitHub, just simple ways to organize your thoughts and promote team communication.

Building teams

In an industry where most people are used to giving and receiving negative feedback, it's rare to share positive feedback with coworkers. It's not surprising—while some enjoy giving compliments, many people struggle with telling someone "way to go" or "couldn't have done this without you." But it never hurts to tell someone they're doing a good job, and it often influences people to work better for the team. Here are two tools that help you share kudos with your coworkers.

- Management 3.0 [2] has a treasure trove of free resources [3] for building teams. One tool we find compelling is the concept of Feedback Wraps (and not just because it inspires us to think about burritos). Feedback Wraps [4] is a six-step process to come up with effective feedback for anyone; you might think it is designed for negative feedback, but we find it's perfect for sharing positive comments.
- Happiness Packets [5] provides a way to share anonymous positive feedback with people in the open source community. It is especially useful for those who aren't comfortable with such a personal interaction or don't know the people they want to reward. Happiness Packets offers a public archive [6] of comments (from people who agree to share them), so you can look through and get warm fuzzies and ideas on what to say to others if you are struggling to find your own words. As a bonus, its code of conduct process prevents anyone from sending nasty messages.

Understanding why

Definitions are hard. In the agile world, keys to success include defining personas, the purpose of a feature, or the product vision, and ensuring the entire agile team understands *why* they are doing the work they are doing. We are a little disappointed by the limited number of open source tools available that help product managers and owners do their jobs.

One that we highly respect and use frequently to teach teams at Red Hat is the Product Vision Board. It comes from product management expert Roman Pichler, who offers numerous tools and templates [7] to help teams develop a better understanding of “the why.” (Note that you will need to provide your email address to download these files.)

- The Product Vision Board [8] template guides teams by asking simple but effective questions to prompt them to think about *what* they are doing *before* they think about *how* they are going to do it.
- We also like Roman’s Product Management Test [9]. This is a simple and quick web form that guides teams through the traditional role of a product manager and helps uncover where there may be gaps. We recommend that product management teams periodically complete this test to reassess where they fall.

Visualizing work

Have you ever been working on a huge assignment, and the steps to complete it are all jumbled up in your head, out of order, and chaotic? Yeah, us, too. Mind mapping is a technique that helps you visually organize all the thoughts in your head. You don’t need to start out understanding how everything fits together—you just need your brain, a whiteboard (or a mind-mapping tool), and some time to think.

- Our favorite open source tool in this space is Xmind3 [10]. It’s available for multiple platforms (Linux, MacOS, and Windows), so you can easily share files back and forth with other people. If you need to have the latest & greatest, there is an updated version [11], which you can download for free if you don’t mind sharing your email.
- If you like more variety in your life, Eduard Lucena offers three additional options [12] in *Fedora Magazine*. You can find information about these tools’ availability in Fedora and other distributions on their project pages.
- Labyrinth [13]
- View Your Mind [14]
- FreeMind [15]

As we wrote at the start, there are many similar tools out there.

Links

[1] https://opensource.com/business/16/3/top-project-management-tools-2016
[2] https://management30.com/
[3] https://management30.com/leadership-resource-hub/
[4] https://management30.com/en/practice/feedback-wraps/
[5] https://happinesspackets.io/
[6] https://www.happinesspackets.io/archive/
[7] http://www.romanpichler.com/tools/
[8] http://www.romanpichler.com/tools/vision-board/
[9] http://www.romanpichler.com/tools/romans-product-management-test/
[10] https://sourceforge.net/projects/xmind3/?source=recommended
[11] http://www.xmind.net/
[12] https://fedoramagazine.org/three-mind-mapping-tools-fedora/
[13] https://people.gnome.org/~dscorgie/labyrinth.html
[14] http://www.insilmaril.de/vym/
[15] http://freemind.sourceforge.net/wiki/index.php/Main_Page

Author

Marianne Feifer has been part of agile teams for the past 15 years. Mostly using Scrum methodology, she participated as a scrum team member as the original (and only) technical writer for ManageIQ, www.manageiq.org. In 2015, she became a Certified Scrum Master, and used those skills with her team at Red Hat. She has adapted those methods to work with a team of developers from a variety of geographic locations. Most recently, she has stepped into the Product Owner role as one of the leads in an effort to sustain and manage customer satisfaction from the engineering perspective.

Marianne prides herself in her ability to communicate effectively with all levels of software organizations across multiple disciplines, including engineering, sales, support, and business units. She appreciates the challenge of connecting people to keep the business moving and the developers working.

Of all the Agile Principles, she values “people and relationships over process” the most.

Jen Krieger is Chief Agile Architect at Red Hat. Most of her 20+ year career has been in software development representing many roles throughout the waterfall and agile lifecycles. At Red Hat, she led a department-wide DevOps movement focusing on CI/CD best practices. Most recently, she worked with with the Project Atomic & OpenShift teams. Now Jen is guiding teams across the company into agility in a way that respects and supports Red Hat’s commitment to Open Source.

Is BDFL a death sentence?

BY JASON BAKER

What happens when a Benevolent Dictator For Life moves on from an open source project?

IN 2018, Guido van Rossum [1], creator of the Python [2] programming language and Benevolent Dictator For Life [3] (BDFL) of the project, announced his intention to step away.

Below is a portion of his message, although the entire email [4] is not terribly long and worth taking the time to read if you're interested in the circumstances leading to van Rossum's departure.

> I would like to remove myself entirely from the decision process. I'll still be there for a while as an ordinary core dev, and I'll still be available to mentor people—possibly more available. But I'm basically giving myself a permanent vacation from being BDFL, and you all will be on your own.
>
> After all that's eventually going to happen regardless—there's still that bus lurking around the corner, and I'm not getting younger... (I'll spare you the list of medical issues.)
>
> I am not going to appoint a successor.
>
> So what are you all going to do? Create a democracy? Anarchy? A dictatorship? A federation?

It's worth zooming out for a moment to consider the issue at a larger scale. How an open source project is governed can have very real consequences on the long-term sustainability of its user and developer communities alike.

BDFLs tend to emerge from passion projects, where a single individual takes on a project before growing a community around it. Projects emerging from companies or other large organization often lack this role, as the distribution of authority is more formalized, or at least more dispersed, from the start. Even then, it's not uncommon to need to figure out how to transition from one form of project governance to another as the community grows and expands.

But regardless of how an open source project is structured, ultimately, there needs to be some mechanism for deciding how to make technical decisions. Someone, or some group, has to decide which commits to accept, which to reject, and more broadly what direction the project is going to take from a technical perspective.

Surely the Python project will be okay without van Rossum. The Python Software Foundation [5] has plenty of formalized structure in place bringing in broad representation from across the community. There's even been a humorous April Fools Python Enhancement Proposal [6] (PEP) addressing the BDFL's retirement in the past.

That said, it's interesting that van Rossum did not heed the fifth lesson of Eric S. Raymond from his essay, *The Mail Must Get Through* [7] (part of *The Cathedral & the Bazaar* [8]), which stipulates: "When you lose interest in a program, your last duty to it is to hand it off to a competent successor." One could certainly argue that letting the community pick its own leadership, though, is an equally valid choice.

What do you think? Are projects better or worse for being run by a BDFL? What can we expect when a BDFL moves on? And can someone truly step away from their passion project after decades of leading it? Will we still turn to them for the hard decisions, or can a community smoothly transition to new leadership without the pitfalls of forks or lost participants?

Can you truly stop being a BDFL? Or is it a title you'll hold, at least informally, until your death?

Links

[1] https://en.wikipedia.org/wiki/Guido_van_Rossum
[2] https://opensource.com/resources/python
[3] https://en.wikipedia.org/wiki/Benevolent_dictator_for_life
[4] https://www.mail-archive.com/python-committers@python.org/msg05628.html
[5] https://www.python.org/psf-landing/
[6] https://www.python.org/dev/peps/pep-0401/
[7] http://www.catb.org/esr/writings/homesteading/cathedral-bazaar/ar01s02.html
[8] https://opensource.com/life/16/5/19-years-later-cathedral-and-bazaar-still-moves-us

Author

Jason uses technology to make the world more open.†He is a Linux desktop enthusiast, map/geospatial nerd, Raspberry Pi tinkerer, Data analysis and visualization geek, occasional coder, cloud nativist, and civic tech and open government booster.

10 Hello World programs for your Raspberry Pi

BY BEN NUTTALL

"Hello world" is the beginning of everything when it comes to computing and programming.

"HELLO WORLD" is the beginning of everything when it comes to computing and programming. It's the first thing you learn in a new programming language, and it's the way you test something out or check to see if something's working because it's usually the simplest way of testing simple functionality.

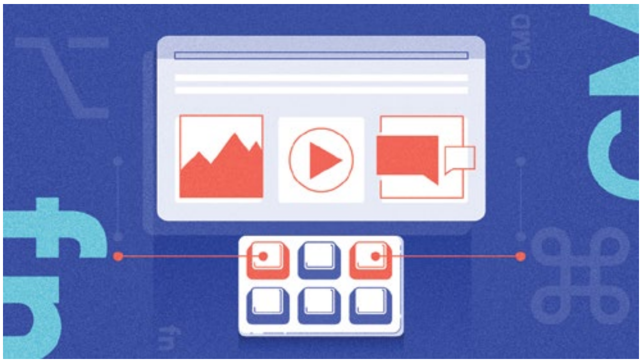
Warriors of programming language wars often cite their own language's "hello world" against that of another, saying theirs is *shorter* or *more concise* or *more explicit* or something. Having a nice simple readable "hello world" program makes for a good intro for beginners learning your language, library, framework, or tool.

I thought it would be cool to create a list of as many different "hello world" programs as possible that can be run on the Raspberry Pi [1] using its Raspbian operating system, but without installing any additional software than what comes bundled when you download it from the Raspberry Pi website. I've created a GitHub repository [2] of these programs, and I've explained 10 of them for you here.

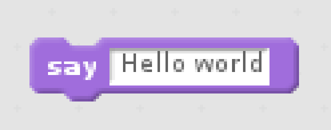
1. Scratch

Scratch [3] is a graphical block-based programming environment designed for kids to learn programming skills without having to type or learn the synax of a programming language. The "hello world" for Scratch is simple—and very visual!

1. Open **Scratch 2** from the main menu.
2. Click **Looks**.
3. Drag a **say Hello!** block into the workspace on the right.

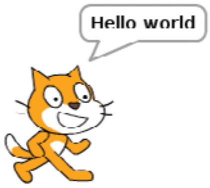


4. Change the text to `Hello world`.



say Hello world

5. Click on the block to run the code.



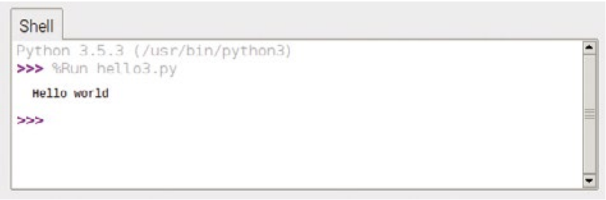
2. Python

Python [4] is a powerful and professional language that's also great for beginners—and it's lots of fun to learn. Because one of Python's main objectives was to be readable and stick to simple English, its "hello world" program is as simple as possible.

1. Open **Thonny Python IDE** from the main menu.
2. Enter the following code:

```
print("Hello world")
```

3. Save the file as `hello3.py`.
4. Click the **Run** button.



```
Python 3.5.3 (/usr/bin/python3)
>>> %Run hello3.py
  Hello world
>>>
```

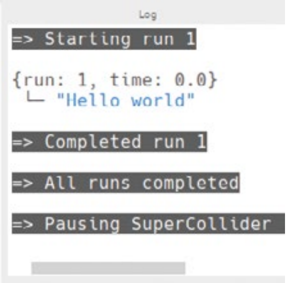
3. Ruby/Sonic Pi

Ruby [5] is another powerful language that's friendly for beginners. Sonic Pi [6], the live coding music synth, is built on top of Ruby, so what users actually type is a form of Ruby.

1. Open **Sonic Pi** from the main menu.
2. Enter the following code:

```
puts "Hello world"
```

3. Press **Run**.



```
=> Starting run 1

{run: 1, time: 0.0}
 └─ "Hello world"

=> Completed run 1

=> All runs completed

=> Pausing SuperCollider
```

Unfortunately, "hello world" does not do Sonic Pi justice in the slightest, but after you've finished this article you should check out its creator Sam Aaron live coding [7], and see the tutorials on the Sonic Pi website [8].

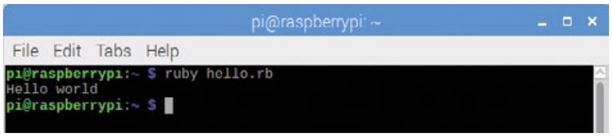
Alternatively, to using the Sonic Pi application for this example, you can write Ruby code in a text editor and run it in the terminal:

1. Open **Text Editor** from the main menu.
2. Enter the following code:

```
puts "Hello world"
```

3. Save the file as `hello.rb` in the home directory.
4. Open **Terminal** from the main menu.
5. Run the following command:

```
ruby hello.rb
```



```
pi@raspberrypi:~ $ ruby hello.rb
Hello world
pi@raspberrypi:~ $
```

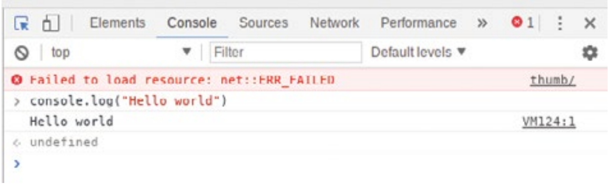
4. JavaScript

This is a bit of a cheat as I just make use of client-side JavaScript [9] within the web browser using the Web Inspector console, but it still counts!

1. Open **Chromium Web Browser** from the main menu.
2. Right-click the empty web page and select **Inspect** from the context menu.
3. Click the **Console** tab.
4. Enter the following code:

```
console.log("Hello world")
```

5. Press **Enter** to run.



```
Failed to load resource: net::ERR_FAILED                thumb/
> console.log("Hello world")
  Hello world                                           VM124:1
<- undefined
>
```

You can also install NodeJS on the Raspberry Pi, and write server-side JavaScript, but that's not available in the standard Raspbian image.

5. Bash

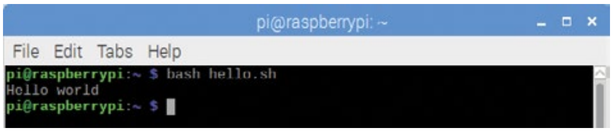
Bash [10] (Bourne Again Shell) is the default Unix shell command language in most Linux distributions, including Raspbian. You can enter Bash commands directly into a terminal window, or script them into a file and execute the file like a programming script.

1. Open **Text Editor** from the main menu.
2. Enter the following code:

```
echo "Hello world"
```

3. Save the file as `hello.sh` in the home directory.
4. Open **Terminal** from the main menu.
5. Run the following command:

```
bash hello.sh
```



```
pi@raspberrypi:~ $ bash hello.sh
Hello world
pi@raspberrypi:~ $
```

Note you'd usually see a "hashbang" at the top of the script (`#!/bin/bash`), but because I'm calling this script directly using the `bash` command, it's not necessary (and I'm trying to keep all these examples as short as possible).

You'd also usually make the file executable with `chmod +x`, but again, this is not necessary as I'm executing with `bash`.

6. Java

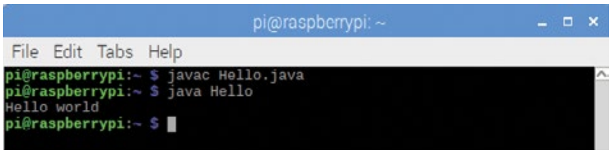
Java [11] is a popular language in industry, and is commonly taught to undergraduates studying computer science. I learned it at university and have tried to avoid touching it since then. Apparently, now I do (very small amounts of) it for fun...

1. Open **Text Editor** from the main menu.
2. Enter the following code:

```
public class Hello {
        public static void main(String[] args) {
            System.out.println("Hello world");
        }
    }
```

3. Save the file as `Hello.java` in the home directory.
4. Open **Terminal** from the main menu.
5. Run the following commands:

```
javac Hello.java
java Hello
```



```
pi@raspberrypi:~ $ javac Hello.java
pi@raspberrypi:~ $ java Hello
Hello world
pi@raspberrypi:~ $ 
```

I could *almost* remember the “hello world” for Java off the top of my head, but not quite. I always forget where the `String[] args` bit goes, but it’s obvious when you think about it...

7. C

C is a fundamental low-level programming language. It’s what many programming languages are written in. It’s what operating systems are written in. See for yourself—take a look at the source for Python [12] and the Linux kernel [13]. If that looks a bit hazy, get started with “hello world”:

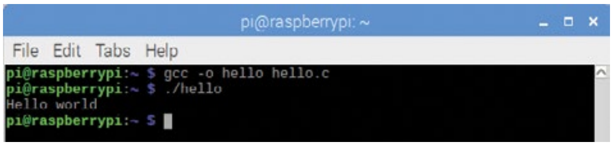
1. Open **Text Editor** from the main menu.
2. Enter the following code:

```
#include <stdio.h>

int main() {
    printf("Hello world\n");
}
```

3. Save the file as `hello.c` in the home directory.
4. Open **Terminal** from the main menu.
5. Run the following commands:

```
gcc -o hello hello.c
./hello
```



```
pi@raspberrypi:~ $ gcc -o hello hello.c
pi@raspberrypi:~ $ ./hello
Hello world
pi@raspberrypi:~ $ 
```

Note that in the previous examples, only one command was required to run the code (e.g., `python3 hello.py` or `ruby hello.rb`) because these languages are interpreted rather than compiled. (Actually Python is compiled at runtime but that’s a minor detail.) C code is compiled into byte code and the byte code is executed.

If you’re interested in learning C, the Raspberry Pi Foundation publishes a book Learning to code with C [14] written by one of its engineers. You can buy it in print or download for free.

8. C++

C’s younger bother, C++ (that’s C incremented by one...) is another fundamental low-level language, with more advanced language features included, such as classes. It’s popular in a range of uses, including game development, and chunks of your operating system will be written in C++ too.

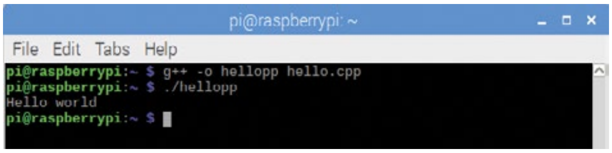
1. Open **Text Editor** from the main menu.
2. Enter the following code:

```
#include <iostream>
using namespace std;

int main() {
    cout << "Hello world\n";
}
```

3. Save the file as `hello.cpp` in the home directory.
4. Open **Terminal** from the main menu.
5. Run the following commands:

```
g++ -o hellopp hello.cpp
./hellocpp
```



```
pi@raspberrypi:~ $ g++ -o hellopp hello.cpp
pi@raspberrypi:~ $ ./hellopp
Hello world
pi@raspberrypi:~ $ 
```

Readers familiar with C/C++ will notice I have not included the main function return values in my examples. This is intentional as to remove boilerplate, which is not strictly necessary.

9. Perl

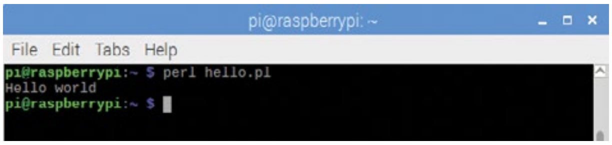
Perl [15] gets a lot of stick for being hard to read, but nothing much gets in the way of understanding its “hello world” program. So far, so good!

1. Open **Text Editor** from the main menu.
2. Enter the following code:

```
print "Hello world\n"
```

3. Save the file as `hello.pl` in the home directory.
4. Open **Terminal** from the main menu.
5. Run the following command:

```
perl hello.pl
```



```
pi@raspberrypi:~ $ perl hello.pl
Hello world
pi@raspberrypi:~ $ 
```

Again, I learned Perl at university, but unlike Java, I have managed to *successfully* avoid using it.

10. Python extras: Minecraft and the Sense HAT emulator

So that's nine different programming languages covered, but let's finish with a bit more Python. The popular computer game Minecraft is available for Raspberry Pi, and comes bundled with Raspbian. A Python library allows you to communicate with your Minecraft world, so open Minecraft and a Python editor side-by-side for some fun hacking your virtual world with code.

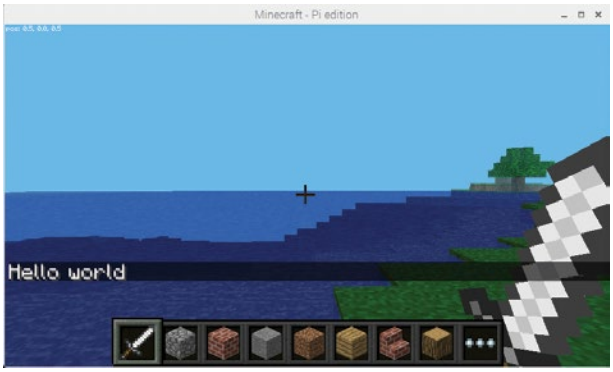
1. Open **Minecraft Pi** [16] from the main menu.
2. Create and enter a Minecraft world.
3. Press **Tab** to release your focus from the Minecraft window.
4. Open **Thonny Python IDE** from the main menu.
5. Enter the following code:

```
from mcpi.minecraft import Minecraft

mc = Minecraft.create()

mc.postToChat("Hello world")
```

6. Save the file as `hellomc.py`.
7. Click the **Run** button.



Hello world

Read more about hacking Minecraft with Python in my article Getting started with Minecraft Pi [16].

Finally, let's look at the Sense HAT Emulator [17]. This tool provides a graphical representation of the Sense HAT [18], an add-on board for Raspberry Pi made especially to go to space for reasons explained in this article [19].

The `sense_emu` Python library is identical to the `sense_hat` library except that its commands get executed in the emulator rather than on a physical piece of hardware. Because the Sense HAT includes an 8x8 LED display, we can use its `show_message` function to write "hello world".

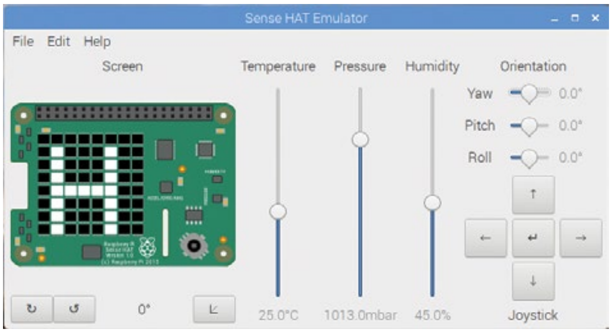
1. Open another tab in Thonny and enter the following code:

```
from sense_emu import SenseHat

sense = SenseHat()

sense.show_message("Hello world")
```

2. Save the file as `sense.py`.
3. Click the **Run** button.



More

That's it! I hope you learned something new, and have fun trying out new "hello world" programs on your Raspberry Pi!

You can find more on the GitHub repository [20] —and feel free to suggest more in an issue, or send me a pull request with your contribution.

Links

[1] https://opensource.com/resources/raspberry-pi
[2] https://github.com/bennuttall/hello-world-raspberry-pi
[3] https://opensource.com/sitewide-search?search_api_views_fulltext=scratch
[4] https://opensource.com/tags/python
[5] https://opensource.com/sitewide-search?search_api_views_fulltext=ruby
[6] http://sonic-pi.net/
[7] https://www.youtube.com/watch?v=KJPdbp1An2s
[8] http://sonic-pi.net/
[9] https://opensource.com/tags/javascript
[10] https://opensource.com/sitewide-search?search_api_views_fulltext=bash
[11] https://opensource.com/tags/java
[12] https://github.com/python/cpython/
[13] https://github.com/torvalds/linux
[14] https://www.raspberrypi.org/magpi/issues/essentials-c-v1/
[15] https://opensource.com/tags/perl
[16] https://opensource.com/life/15/5/getting-started-minecraft-pi
[17] https://opensource.com/life/16/9/coding-raspberry-pi-web-emulator
[18] https://opensource.com/life/15/10/exploring-raspberry-pi-sense-hat
[19] https://opensource.com/education/15/4/uk-students-compete-chance-have-their-raspberry-pi-code-run-space
[20] https://github.com/bennuttall/hello-world-raspberry-pi

Author

Ben Nuttall is the Raspberry Pi Community Manager. In addition to his work for the Raspberry Pi Foundation, he's into free software, maths, kayaking, GitHub, Adventure Time, and Futurama. Follow Ben on Twitter @ben_nuttall.

How to **get started in AI**

BY GORDON HAFF

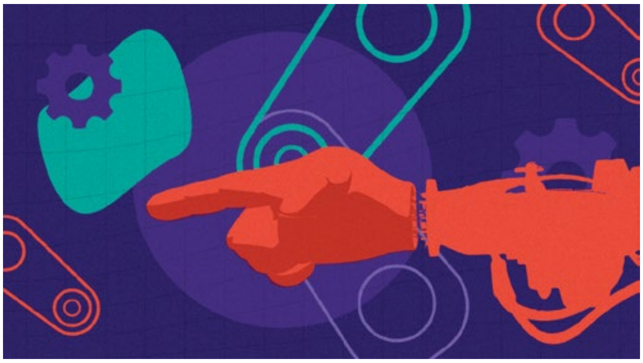
Before you can begin working in artificial intelligence, you need to acquire some human intelligence.

I'VE BOTH ASKED and been asked about the best way to learn more about artificial intelligence (AI). What should I read? What should I watch? I'll get to that. But, first, it's useful to break down this question, given that AI covers a lot of territory.

One important distinction to draw is between the research side of AI and the applied side. Cassie Kozyrkov of Google drew this distinction [1] in a talk at the recent O'Reilly Artificial Intelligence Conference in London, and it's a good one.

Research AI is rather academic in nature and requires a heavy dose of math across a variety of disciplines before you even get to those parts that are specific to AI. This aspect of AI focuses on the algorithms and tools that drive the state of AI forward. For example, what neural network structures might improve vision recognition results? How might we make unsupervised learning a more generally useful approach? Can we find ways to understand better how deep learning pipelines come up with the answers they do?

Applied AI, on the other hand, is more about using existing tools to obtain useful results. Open source has played a big role here in providing free and often easy-to-use software in a variety of languages. Public cloud providers have also devoted a lot of attention to providing machine learning services, models, and datasets that make the on-ramp to getting started with AI much simpler than it would be otherwise.



I'll add at this point that applied AI practitioners shouldn't treat their tools as some sort of black box that spits out answers for mysterious reasons. At a minimum, they need to understand the limits and potential biases of different techniques, models, and data collection approaches. It's just that they don't necessarily need to delve deeply into all the theory underpinning every part of their toolchain.

Although it's probably less important for working in AI on a day-to-day basis, it's also useful to understand the broader context of AI. It goes beyond the narrow scope of deep learning on neural networks that have been so important to the gains made in reinforcement learning and supervised learning to date. For example, AI is often viewed as a way to *augment* (rather than replace) human judgment and decisions. But the handoff between machine and human has its own pitfalls.

With that background, here are some study areas and resources you may find useful.

Research AI

In a lot of respects, a list of resources for research AI mirror those in an undergraduate (or even graduate) computer science program that's focused on AI. The main difference is that the syllabus you draw up may be more interdisciplinary than more traditionally focused university curricula.

Where you start will depend on your computer science and math background.

If it's minimal or rusty, but you still want to develop a deep understanding of AI fundamentals, you'll benefit from taking some math courses to start. There are many options on massive online open courses (MOOCs) like the nonprofit edX [2] platform and Coursera [3]. (Both platforms charge for certifications, but edX makes all the content available for free to people just auditing the course.)

Typical foundational courses could include:

- MIT's Calculus courses [4], starting with differentiation
- Linear Algebra [5] (University of Texas)
- Probability and statistics, such as MIT's Probability—The Science of Uncertainty and Data [6]

To get deeper into AI from a research perspective, you'll probably want to get into all these areas of mathematics and more. But the above should give you an idea of the general branches of study that are probably most important before delving into machine learning and AI proper.

In addition to MOOCs, resources such as MIT OpenCourseWare [7] provide the syllabus and various supporting materials for a wide range of mathematics and computer science courses.

With the foundations in place, you can move onto more specialized courses in AI proper. Andrew Ng's AI MOOC, from when he was teaching at Stanford, was one of the early courses to popularize the whole online course space. Today, his Neural Networks and Deep Learning [8] is part of the Deep Learning specialization at Coursera. There are corresponding programs on edX. For example, Columbia offers an Artificial Intelligence MicroMasters [9].

In addition to courses, a variety of textbooks and other learning material are also available online. These include:

- *Neural Networks and Deep Learning* [10]
- *Deep Learning* [11] from MIT Press by Ian Goodfellow and Yoshua Bengio and Aaron Courville

Applied AI

Applied AI is much more focused on using available tools than building new ones. Some appreciation of the mathematical underpinnings, especially statistics, is still useful—arguably even necessary—but you won't be majoring in that aspect of AI to the same degree you would in a research mode.

Programming is a core skill here. While different programming languages can come into play, a lot of libraries and toolsets—such as PyTorch [12]—rely on Python, so that's a good skill to have. Especially if you have some level of programming background, MIT's Introduction to Computer Science and Programming Using Python [13], based on its on-campus 6.001 course, is a good primer. If you're truly new to programming, Charles Severance's Programming for Everybody (Getting Started with Python) [14] from the University of Michigan doesn't toss you into the deep end of the pool the way the MIT course does.

The R programming language [15] is also a useful skill to add to your toolbox. While it's less used in machine learning (ML) per se, it's common for a variety of other data science tasks, and applied AI/ML and data science often blend in practice. For example, many tasks associated with organizing and cleaning data apply equally whatever analysis techniques you'll eventually use. A MOOC sequence like Harvard's Data Science certificate [16] is an example of a set of courses that provide a good introduction to working with data.

Another open source software library you're likely to encounter if you do any work with AI is TensorFlow [17]. It was originally developed by researchers and engineers from the Google Brain team within Google's AI organization. Google offers a variety of tutorials [18] to get started with TensorFlow using the high-level Keras API. You can run TensorFlow locally as well as online in Google Cloud.

In general, all of the big public cloud providers offer online datasets and ML services that can be an easy way to get started. However, especially as you move beyond "play" datasets and applications, you need to start thinking seriously about the degree to which you want to be locked into a single provider.

Datasets for your exploratory learning projects are available from many different sources. In addition to the public cloud providers, Kaggle [19] is another popular source and also a good learning resource more broadly. Government data is also increasingly available in digital form. The US Federal Government's Data.gov [20] claims over 300,000 datasets. State and local governments also publish data on everything from restaurant health ratings to dogs' names.

Miscellany

I'll close by noting that AI is a broad topic that isn't just about math, programming, and data. AI as a whole touches many other fields, including cognitive psychology, linguistics, game theory, operations research, and control systems. Indeed, a concern among at least some AI researchers today is that the field has become too fixated on a small number of techniques that have become powerful and interesting only quite recently because of the intersection of processing power and big data. Many longstanding problems in understanding how humans learn and reason remain largely unsolved. Developing at least some appreciation for these broader problem spaces will better enable you to place AI within a broader context.

One of my favorite examples is the Humans and Autonomy Lab [21] at Duke. The work in this lab touches on all the challenges of humans working with machines, such as how autopilots can create "Children of the Magenta" [22] who are unable to take control quickly if the automation fails. A basic brain-science course, such as MIT's Introduction to Psychology [23], provides some useful context for the relationship between human intelligence and machine intelligence. Another course in a similar vein, but taught by the late Marvin Minsky from MIT's Electrical Engineering and Computer Science department, is The Society of Mind [24].

If there's one key challenge to learning about AI, it's not that raw materials and tools aren't readily available. It's that there are so many of them. My objective hasn't been to give you a comprehensive set of pointers. Rather, it's been to both point out the different paths you can take and provide you with some possible starting points. Happy learning!

Links

[1] https://www.youtube.com/watch?v=RLtI7r3QUyY
[2] https://www.edx.org/
[3] https://www.coursera.org/
[4] https://www.edx.org/course/calculus-1a-differentiation
[5] https://www.edx.org/course/linear-algebra-foundations-to-frontiers
[6] https://courses.edx.org/courses/course-v1:MITx+6.431x+3T2018/course/
[7] https://ocw.mit.edu/index.htm
[8] https://www.coursera.org/learn/neural-networks-deep-learning
[9] https://www.edx.org/micromasters/columbiax-artificial-intelligence
[10] http://neuralnetworksanddeeplearning.com/
[11] http://www.deeplearningbook.org/
[12] https://pytorch.org/
[13] https://www.edx.org/course/introduction-to-computer-science-and-programming-using-python
[14] https://www.coursera.org/learn/python
[15] https://www.r-project.org/about.html
[16] https://www.edx.org/professional-certificate/harvardx-data-science
[17] https://www.tensorflow.org/
[18] https://www.tensorflow.org/tutorials/
[19] https://www.kaggle.com/
[20] https://www.data.gov/
[21] https://hal.pratt.duke.edu/
[22] https://99percentinvisible.org/episode/children-of-the-magenta-automation-paradox-pt-1/
[23] https://ocw.mit.edu/courses/brain-and-cognitive-sciences/9-00sc-introduction-to-psychology-fall-2011/
[24] https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-868j-the-society-of-mind-fall-2011/

Author

Gordon Haff is Red Hat technology evangelist, is a frequent and highly acclaimed speaker at customer and industry events, and helps develop strategy across Red Hatís full portfolio of cloud solutions. He is the co-author of Pots and Vats to Computers and Apps: How Software Learned to Package Itself in addition to numerous other publications. Prior to Red Hat, Gordon wrote hundreds of research notes, was frequently quoted in publications like The New York Times on a wide range of IT topics, and advised clients on product and marketing strategies. Earlier in his career, he was responsible for bringing a wide range of computer systems, from minicomputers to large UNIX servers, to market while at Data General. Gordon has engineering degrees from MIT and Dartmouth and an MBA from Cornellís Johnson School.

7 open source platforms to get started with serverless computing

BY DANIEL OH

Serverless computing is transforming traditional software development. These open source platforms will help you get started.

THE TERM *serverless* [1] has been coming up in more conversations recently. Letís clarify the concept, and those related to it, such as *serverless computing* and *serverless platform*.

Serverless is often used interchangeably with the term *FaaS* (Functions-as-a-Service). But serverless doesnít mean that there is no server. In fact, there are many servers–*serverful*–because a public cloud provider provides the servers that deploy, run, and manage your application.

Serverless computing is an emerging category that represents a shift in the way developers build and deliver software systems. Abstracting application infrastructure away from the code can greatly simplify the development process while introducing new cost and efficiency benefits. I believe serverless computing and FaaS will play an important role in helping to define the next era of enterprise IT, along with cloud-native services and the hybrid cloud [2].

Serverless platforms provide APIs that allow users to run code functions (also called *actions*) and return the results of each function. Serverless platforms also provide HTTPS endpoints to allow the developer to retrieve function results. These endpoints can be used as inputs for other functions, thereby providing a sequence (or chaining) of related functions.

On most serverless platforms, the user deploys (or creates) the functions before executing them. The serverless platform then has all the necessary code to execute the functions when it is told to. The execution of a serverless function can be invoked manually by the user via a command, or it may be triggered by an event source that is configured to activate the function in response to events such as cron job alarms, file uploads, or many others.

7 open source platforms to get started with serverless computing

- Apache OpenWhisk [3] is a serverless, open source cloud platform that allows you to execute code in response to events at any scale. Itís written in the Scala language. The framework processes the inputs from triggers like HTTP requests and later fires a snippet of code on either JavaScript or Swift.
- Fission [4] is a serverless computing framework that enables developers to build functions using Kubernetes. It

allows coders to write short-lived functions in any programming language and map them with any event triggers, such as HTTP requests.

- IronFunctions [5] is a serverless computing framework that offers a cohesive microservices platform by integrating its existing services and embracing Docker. Developers write the functions in Go language.
- Fn Project [6] is an open source container-native serverless platform that you can run anywhere–on any cloud or on-premise. Itís easy to use, supports every programming language, and is extensible and performant.
- OpenLambda [7] is an Apache-licensed serverless computing project, written in Go and based on Linux containers. The primary goal of OpenLambda is to enable exploration of new approaches to serverless computing.
- Kubeless [8] is a Kubernetes-native serverless framework that lets you deploy small bits of code without having to worry about the underlying infrastructure. It leverages Kubernetes resources to provide autoscaling, API routing, monitoring, troubleshooting, and more.
- OpenFaas [9] is a framework for building serverless functions with Docker and Kubernetes that offers first-class support for metrics. Any process can be packaged as a function, enabling you to consume a range of web events without repetitive boilerplate coding.

Kubernetes is the most popular platform to manage serverless workloads and microservice application containers, using a finely grained deployment model to process workloads more quickly and easily. With Knative Serving [10], you can build and deploy serverless applications and functions on Kubernetes and use Istio [11] to scale and support advanced scenarios such as:

- Rapid deployment of serverless containers
- Automatic scaling up and down to zero
- Routing and network programming for Istio components
- Point-in-time snapshots of deployed code and configurations

Knative [12] focuses on the common tasks of building and running applications on cloud-native platforms for orchestrating source-to-container builds, binding services to event ecosystems, routing and managing traffic during deployment, and autoscaling workloads. Istio is an open platform to connect and secure microservices (effectively a service mesh control plane to the Envoy proxy [13]) and has been designed to consider multiple personas interacting with the framework, including developers, operators, and platform providers.

For example, you can deploy a JavaScript serverless workload using Knative Serving on a local Minishift [14] platform with the following code snippets:

```
## Dockerfile
FROM bucharestgold/centos7-s2i-nodejs:10.x
WORKDIR /opt/app-root/src
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 8080 3000
CMD ["npm", "start"]


## package.json
{
  "name": "greeter",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "express": "~4.16.0"
  }
}


## app.js
var express = require("express");
var app = express();

var msg = (process.env.MESSAGE_PREFIX || "") + "NodeJs::Knative
           on OpenShift";

app.get("/", function(req, res, next) {
  res.status(200).send(msg);
});

app.listen(8080, function() {
  console.log("App started in port 8080");
});


## service.yaml
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: greeter
spec:
  configuration:
    revisionTemplate:
      spec:
        container:
          image: dev.local/greeter:0.0.1-SNAPSHOT
```

Build your Node.js serverless application and deploy the service on local Kubernetes platform. Install Knative, Istio, Knative Serving on Kubernetes (or Minishift) [15] as prerequisites.

1. Attach to the Docker daemon using the following the commands:

   ```
   (minishift docker-env) && eval(minishift oc-env)
   ```

2. Build a serverless application container image using the following the commands with Jib [16]:

```
./mvnw -DskipTests clean compile jib:dockerBuild
```

3. Deploy a serverless service such as Minishift to your Kubernetes cluster:

```
kubectl apply -f service.yaml
```

Conclusion

The example above shows where and how to start developing the serverless application with a cloud-native platform such as Kubernetes, Knative Serving, and Istio.

Links

[1] https://enterprisersproject.com/article/2018/9/what-serverless

[2] https://enterprisersproject.com/hybrid-cloud

[3] https://openwhisk.apache.org/

[4] https://github.com/fission/fission

[5] https://github.com/iron-io/functions

[6] https://fnproject.io/

[7] https://open-lambda.org/

[8] https://kubeless.io/

[9] https://docs.openfaas.com/

[10] https://github.com/knative/serving

[11] https://istio.io/

[12] https://github.com/knative/

[13] https://www.envoyproxy.io/

[14] https://github.com/minishift/minishift

[15] https://github.com/knative/docs/blob/master/install/Knative-with-Minishift.md

[16] https://github.com/GoogleContainerTools/jib

Author

Daniel is a DevOps evangelist, CNCF ambassador, developer, speaker, writer, and Opensource.com author.

The current state of Linux video editing 2018

BY SETH KENLON

Linux is a big deal in modern movie-making. Whether you're a hobbyist or a professional, you can find Linux software that meets your needs.

IT'S PRETTY WELL KNOWN THAT LINUX is a big deal in modern movie making. Linux is the standard base, a literal industry standard [1] for digital effects but, like all technology with momentum, it seems that the process of cutting footage still defaults mostly to a non-Linux platform. Slowly, however, as artists seek to simplify and consolidate the post-production pipeline, Linux video editing is gaining in popularity.

It can be difficult to talk about video editing objectively because it means so many different things to different people. For instance, to some people a video editing application *must* be able to generate fancy animated title sequences, while professional users balk at the idea of doing serious work on titles in their video editor. It's not unlike the debate over professional SLR cameras that happened when digital cameras in phones became contenders for serious photography.

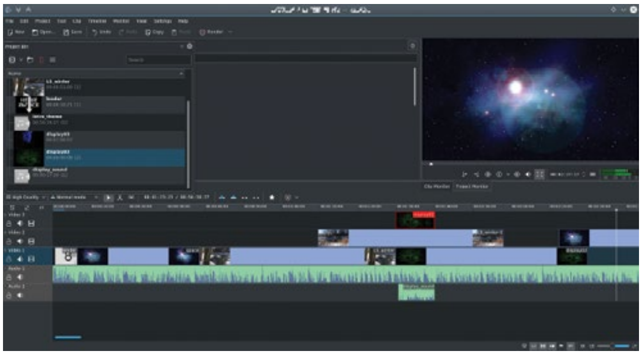
For this reason, a pragmatic overview of a Linux-based video editor needs two broad qualifiers: How it performs for home users, and how it might integrate into a professional pipeline.

Defining key terms

- **Independent:** For the purposes of this article, I'll call a workflow that begins and ends with either one video editing software or one computer system either "independent" or "hobbyist." In other words, an independent or hobbyist filmmaker is likely to use one application to do video editing, maybe a few other applications for specialized tasks like audio sweetening or motion graphics, and then they're done. Their project is exported and delivered.
- **Professional integration:** A "professional" editor probably also uses only one application to edit video, but that's because they're a cog in a larger machine. A professional editor might get their footage from a producer or director, and when they're done they probably aren't exporting the final version that their audiences are going to see, but they'll pass their work on to audio engineers, VFX artists, and colorists.

Top pro pick: Kdenlive

Kdenlive [2] is the best-in-class professional open source editing application, hands-down. As long as you run a stable version of Kdenlive on a stable Linux OS, use reasonable file formats, and keep your work organized, you'll have a reliable, professional-quality editing experience.



Strengths

- The interface is intuitive for anyone who has ever used a professional-style editing application.
- The way you work in Kdenlive is natural and flexible, allowing you to use both of the major styles of editing: cutting by numbers and just mousing around in the timeline [3].
- Kdenlive has plenty of capabilities beyond just cutting up footage. It can do some advanced visual effects, like masking [4], all manner of composting (see this [5], this [6], and this [7]), color correction [8], offline "proxy" editing [9], and much much more.

Weaknesses

- The greatest weakness of open source editing is also its greatest strengths: Kdenlive lets you throw nearly anything you want at it, even if that sometimes means its performance suffers. You should resist the urge to take advantage of this flexibility and instead manage your assets and formats smartly. Instead of using an MP3, convert the MP3 to WAV first (which is what other editors do for you, but they do it "behind the scenes"). Don't throw in an animated GIF without first breaking it out into a series of images. And so on. Gaining flexibility means you gain the responsibility for maintaining a sensible media library.
- The interface, while accounting for both "traditional" editing styles and the "modern" style of treating the timeline as a sort of scratchpad, wouldn't really satisfy an editor who wants to cut by numbers. Currently, there's no way, for instance, to modify or move clips with quick number-pad entries (typing +6, for instance, has no effect on a video region's placement in the timeline).

Independent

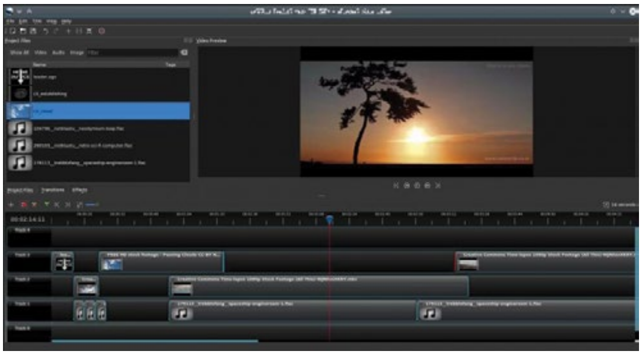
- If anything, Kdenlive could be overkill for home users who aren't accustomed to professional-style editing. Basic operations of the interface are mostly intuitive, but new editors might feel that there's a learning curve for advanced operations (like layered composting [10] and offline editing [11]).
- On the other hand, it scales down well. You can use a fraction of its features and find it a pretty simple, mostly intuitive editor.
- And for serious home editors and independent movie makers, Kdenlive is worth learning [12] and using [13], and it is likely to satisfy all requirements. It may not always be a drop-in replacement if you're transitioning from some other editor, but it's familiar enough to keep the learning curve manageable.

Professional integration

- If you're working in a production environment with an established workflow, then any change to your editor requires adaptation.
- Kdenlive saves projects as an XML file, so it's possible to convert an existing edit decision list (EDL) to a Kdenlive project file, although there aren't any official auto-converters available yet, so round trips (i.e., returning to the original application) out of Kdenlive would require intervention. Alternately, round trips can be done with lossless clip exports, which can be reintegrated into a project after whatever has been applied from the external application.
- The same holds true for audio [14]. You can render audio to a file and import into an external digital audio workstation (DAW), but currently there's no native, built-in audio-export target for popular formats like Open Media Framework (OMF).
- For the most part, as long as your pipeline isn't perilously rigid, Kdenlive can exist within any professional environment. It can output video, audio, and image sequences, and it's hard to imagine a workflow where such generic output isn't acceptable.

Hobbyist pick: OpenShot

OpenShot [15] is a simple but robust video editor. If you're not interested in learning the finer details on how to edit video, then OpenShot is for you. It doesn't scale up; a professional editor will find it restrictive, but for a quick and easy edit, OpenShot is a great choice on any OS.



Strengths

- OpenShot is focused. It understands exactly what its audience wants: the ability to make attractive videos with minimal fuss. Its interface is intuitive, and what you can't immediately figure out from context, you can access with a right-click.

- The most common transition, a crossfade, is available by overlapping the edges of two clips. This is such a simple and obvious trick, but it cuts down on so many mouse clicks that you'll wonder why all video editors don't do that.
- It's also a very conservative application. You won't see a new OpenShot release every month, and that's a good thing. You can download OpenShot as an AppImage today and use it for the next year or more. It's a beautiful, comfortable, simple piece of software.

Weaknesses

- A hobbyist's strengths are a pro's weaknesses. It's a deliberately simplified system, and little conveniences like the auto-crossfades are unwelcome to a professional editor who doesn't necessarily want clips to crossfade when they overlap.
- OpenShot doesn't have a very robust engine for real-time effects. Too many dynamic effects severely slow playback.

Independent

- An independent or hobbyist editor with simple needs will find OpenShot perfect. It's an easy install, it has all the usual benefits of open source multimedia (near indifference to codecs, no false limitations or paywalls for advanced features).

Professional integration

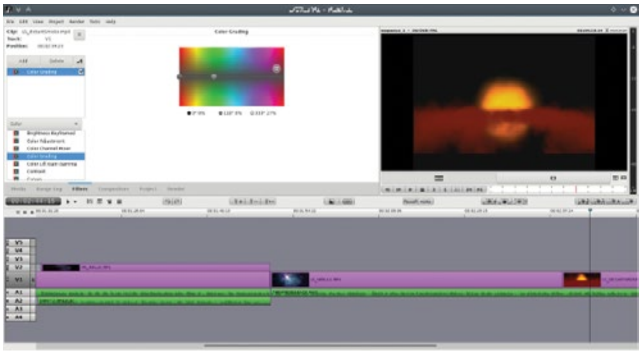
- Integrating OpenShot with a larger pipeline is possible, but only in the sense that it can output generic video and audio files and image sequences. Its project file format, however, is also open source, and it saves into a JSON format that theoretically could be leveraged for an EDL, but there's no built-in exporter for that.

Everything else

Kdenlive and OpenShot are my top picks, the open source editors an editor ought to turn to for a quick fix, but there are, of course, several others to look at.

Flowblade [16]

Flowblade is a simplified video editor that focuses on the editorial process. If you're an experienced editor and just want to get down to business, or you 're a hobbyist who needs little more than an interface to assemble video clips in sequence, then Flowblade's minimal interface may appeal to you.



Strengths

- A no-frills, stable application for quick, no-nonsense cutting.
- Its workflow favors a traditional cutting style: mark in, mark out, dump into timeline. Rinse and repeat.
- This makes it slightly less convenient to stumble around your project in search of a good edit, but that's what makes it so efficient and smooth when you know what you want.
- A professional-level editor who lives to count frames and edit on the keyboard will love Flowblade.

Weaknesses

- Flowblade's interface is arguably overly simple.
- At the time of this writing, its keyboard shortcuts are not user-definable (although it's written in Python, so an editor fluent in Python can adjust preferences by brute force).

Independent

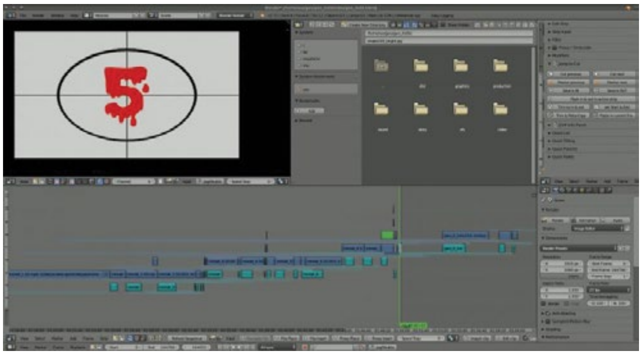
- Many of the "obvious" things a hobbyist would expect from a video editor just don't happen in Flowblade. For instance, moving a clip once it's in the timeline requires activation of an "overwrite" mode, since otherwise clips "float" left.

Professional integration

- In addition to generic video and audio files, Flowblade can export to MLT XML for use with the open source multimedia framework [17] that powers it, as well a plain text, parseable EDL. Additionally, Flowblade's project format is plain text and could be used to extract information for a custom EDL format.
- These options don't provide specialized hooks into specific applications, but it's certainly enough of a variety that a simple converter should be able to import the information.

Blender [18]

Blender excels at efficiency. Once you know how to interact with its interface, you can accomplish amazing things amazingly quickly. Transferring this kind of efficiency over to video editing is a dream come true.



Strengths

- By default, Blender's video sequence editor (VSE) is, from what I can tell, optimized for only the most basic "editing" tasks. This makes sense, given that in the animation and VFX world, there isn't generally excess footage. Artists work on shots that have already been finalized, so the only editing task after all the animation is done is to reintegrate shots into the final cut of the movie. Luckily, though, there are several plugins (such as Easy-Logging [19] and the Blender Velvets [20]) in active development to apply traditional editing interface conventions to Blender's VSE mode, and they manage to transform Blender into a very usable video editing software.

- Blender is stable, fully cross-platform, popular, and under steady development. Using it to edit video isn't exactly common, but the application as a framework for multimedia work is robust and reliable.

Weaknesses

- If you're expecting a traditional editing platform, Blender's weaknesses are many. Its interface can be confusing, and the UI is unconventional as a video editor, at best. Even with VSE plugins and personal customizations, the interface is mostly utilitarian.
- Blender's rendering engines are backends for 3D model rendering. Rendering a video sequence, especially with effects (like color correction, which one would expect to have on each clip in a primary editing application) applied to each clip, takes far longer (10x as long from Kdenlive and Flowblade, in my most recent tests) than rendering from any other video editor. This might be partly because the Blender interface offers no control over FFmpeg [21] threads.
- The VSE lacks integration with the rest of Blender. You cannot, for instance, attach clips from your VSE edit into the node editor and apply fancy effects. In Blender's internal pipeline, the VSE is definitely a separate process.

Independent

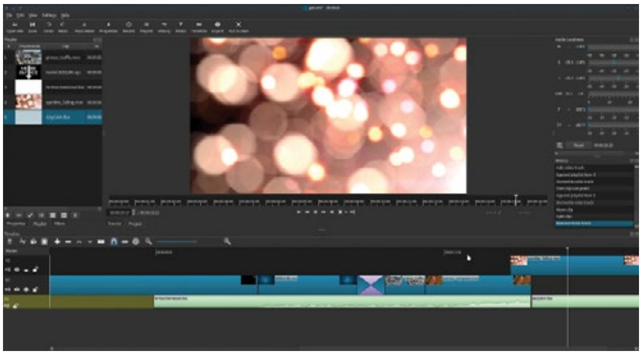
- A hobbyist who knows nothing about Blender will find a steep learning curve. Even with VSE add-ons to make the VSE act more like a "normal" application, anything beyond basic cuts and sequencing just doesn't work the way most users would expect.
- Like all powerful applications, however, Blender is by all means worth knowing. In terms of application design, it's one of the best examples, outside of Emacs, of combining internal logic and consistency with endless extensibility to produce a powerful, unstoppable force of computational wonder.

Professional integration

- Depending on your industry, your production house may already be using Blender, if not for video editing then for animation or motion graphics.
- There are several EDL export [22] add-ons available, and Blender's seamless integration with Python makes it trivial for a technically minded editor or support staff to export whatever information is necessary to blend Blender into any pipeline.

Shotcut [23]

Shotcut is a video editor being developed by Dan Dennedy, an MLT [24] co-founder and current project lead. It is designed from the ground up to be cross-platform and leverages new technologies like WebVfx (visual effects created with web technologies) and Movit (GPU image processing).



Strengths

- Shotcut is using the latest in open source technology to provide performance unlike any other open source video editor. Its real-time effects are smooth as is, and they will get even better once it's offloaded onto the GPU.
- The interface is mostly familiar, although some liberties are taken in the interest of progress. One wonders if mobile devices are on the roadmap, because much of the interface design would work well on a tablet or a large phone screen.
- Shotcut is JACK-aware, so tethering it to a pro audio application like Ardour is trivial.

Weaknesses

- Shotcut is a little progressive, so there's a learning curve involved where its interface implements something different than the de facto standard. For instance, the workflow in a traditional editor is: bring a clip into your bin, open that clip from the bin, mark in and out, and put it in the timeline. With Shotcut, however, there's no internal import process to populate your bin ("playlist," in Shotcut terminology). You can either drag and drop from your file manager or you can open a clip and add that clip to your playlist, or you can bypass the playlist entirely and just add it to your timeline.
- It's less esoteric, for example, there's no way to group select several clips in the timeline to move them. You can insert clips in front of them, but editors used to using their timeline as a scratchpad with lots of groups of edited scenes might find this limitation troublesome.
- The effect stack is still a work in progress. Important effects, like a chromakey (green screen), are missing. They're being added as the dev team perfects their interfaces and functionality.

Independent

- For basic editing, Shotcut is a breeze. It's uncluttered, relatively lightweight, and functional. It's got everything you need and doesn't offer a lot of options you probably don't intend to use.
- In its current state, it doesn't scale up. When you hit its ceiling, you'll have to move to another application. For some, this might be when they suddenly realize they need to do complex composites (to be fair, it's arguable that complex

composites shouldn't be done in a video editing application at all, but that doesn't change expectations), while for others it will be small interface preferences, like Shotcut's inability to dynamically create a new audio track when dragging an audio-only clip into a timeline with only one video track.

Professional integration

- Shotcut isn't production-ready yet, but since a true professional is more than the sum of the tools, it could be used in a professional setting. Shotcut can export an EDL, and it stores its project files as MLT XML, so you could extract information for a custom EDL format as needed.

Non-open editors

There's a handful of cross-platform editors that are *not* open source. However, they can run on an otherwise open stack (in other words, they are fully Linux-compatible), which is a pretty common paradigm in the professional film world.

A not insignificant advantage to these closed-source solutions is that a team of editors can use the same software regardless of the OS they're running.

Lightworks [25]

A long-time editing solution in Hollywood, Lightworks is now free to download. While its natural approach to editing defers to a traditional film workflow, working in the timeline is possible and new features are constantly being added to make sandboxing in the timeline comfortable. The free version is basically a complete solution for serious editng, but if you pay for a subscription you "unlock" better codec support and a few effects (which are, awkwardly, not cross-platform).

Strengths

- Nobody would call Lightworks the industry standard, but it is an Emmy award winner and has a long history of professional use before it became no-cost software independent of its hardware stack. It's a robust application with some serious pro features, such as timeline effects, codec support, lots of export formats, and a unique but efficient interface.
- It's a technical editing environment. It's very aware of editing decisions and timecode and frame numbers, so if you are a professional editor who needs to know that your edit can conform later in the pipeline, Lightworks won't let you down.
- Real-time effects are well supported in Lightworks, so performance is as good as your system specs provide.

Weaknesses

- It's not open source. Its development team announced many years ago [26] that the code would be released in Q3 of 2012; now the official stance in the forums [27] is that "Lightworks is freemium software."
- Furthermore, Lightworks is not a lightweight application. It expects a powerful rig, and at a certain point, it bottoms out and just plain won't run.
- Lightworks' default editing style in many ways mimics the traditional film-editing process. Its timeline is designed for keyboard and shuttle control. Hobbyists or editors who were trained to do their editing with the mouse might find Lightworks a little difficult to get used to. With each new version, the timeline gets a little more mouse-friendly, but the overall design is somewhat technical.

Independent

- Lightworks is probably overkill for the hobbyist. It works well, but there's a learning curve and an emphasis on precision and professionalism that will probably get in the way for people who just want to edit.

Professional integration

- Lightworks exports to a number of formats, such as OMF and AAF, so it's prepared to communicate with whatever's next in your pipeline. If it doesn't export to what you need, it does export to a variety of video and audio formats.

Da Vinci Resolve [28]

Coming from Da Vinci's color correction suite, and once tied to a proprietary hardware suite, Resolve is a cross-platform editor distributed for $0.

Strengths

- Da Vinci has been an industry standard for decades, and while Resolve is technically relatively new, many professionals in the industry have some familiarity with the system in general.

Weaknesses

- Resolve, like Lightworks, has hefty hardware requirements. If your system doesn't meet its requirements, it doesn't run. There's no lightweight mode, even if you just want to do some basic edits.
- Resolve is not open source.

Independent

- Resolve is probably overkill for hobbyists, but its interface is flexible and allows for several editing styles. Its interface is fairly intuitive; if you've used a video-editing application before, you can probably figure out Resolve with an afternoon and a few online tutorial videos.

Professional integration

- Da Vinci exports to several exchange formats as well as video, audio, and image sequences.

Hiero [29]

Hiero isn't, strictly speaking, a video editor, but a show viewer. However, it's set up such that clips can be arranged and adjusted, so it sometimes gets used as a video editing solution by artists familiar with other Foundry tools.

All the rest

Of course, there are still more options. Some, like Pitivi [30] and Cinelerra [31], are less active and less stable now than they may have once been, others, like Avidemux [32], are limited in scope, and still others, like using FFmpeg directly, are just too niche to cover.

The point is that there are *plenty* of very good video editing solutions for Linux. All you have to do is choose one, and get creative.

Links

[1] http://vfxplatform.com
[2] https://kdenlive.org/
[3] https://opensource.com/life/15/6/mastering-timeline-kdenlive
[4] https://opensource.com/life/15/11/basic-masking-kdenlive
[5] https://opensource.com/life/15/3/creating-split-screen-shots-kdenlive
[6] https://opensource.com/life/15/12/10-kdenlive-tools
[7] https://opensource.com/life/15/4/layered-compositing-kdenlive
[8] https://opensource.com/life/11/11/kdenlive-part-4-colour-correction
[9] https://opensource.com/life/16/1/offline-editing-kdenlive
[10] https://opensource.com/life/15/4/layered-compositing-kdenlive
[11] https://opensource.com/life/16/1/offline-editing-kdenlive
[12] https://opensource.com/resources/ebook/video-editing
[13] https://opensource.com/life/15/3/creating-split-screen-shots-kdenlive
[14] https://opensource.com/life/15/9/audio-and-video-xjadeo
[15] https://www.openshot.org/
[16] https://jliljebl.github.io/flowblade/
[17] https://www.mltframework.org
[18] https://www.blender.org/
[19] http://easy-logging.net/
[20] http://blendervelvets.org
[21] http://ffmpeg.org
[22] https://github.com/tin2tin/ExportEDL
[23] https://shotcut.org/
[24] http://mltframework.org
[25] https://www.lwks.com/
[26] https://editshare.wordpress.com/tag/editshareflow/page/3/
[27] https://www.lwks.com/index.php?option=com_kunena&func=view&catid=42&id=164255&Itemid=81
[28] https://www.blackmagicdesign.com/products/davinciresolve/
[29] https://www.foundry.com/products/hiero
[30] http://www.pitivi.org/
[31] http://cinelerra.org/
[32] http://avidemux.sourceforge.net/

Author

Seth Kenlon is an independent multimedia artist, free culture advocate, and UNIX geek. He has worked in the film and computing industry, often at the same time. He is one of the maintainers of the Slackware-based multimedia production project, http://slackermedia.info

6 pivotal moments in open source history

BY DAVE NEARY

Here's how open source developed from a printer jam solution at MIT to a major development model in the tech industry today.

OPEN SOURCE has taken a prominent role in the IT industry today. It is everywhere from the smallest embedded systems to the biggest supercomputer, from the phone in your pocket to the software running the websites and infrastructure of the companies we engage with every day. Let's explore how we got here and discuss key moments from the past 40 years that have paved a path to the current day.

1. RMS and the printer

In the late 1970s, Richard M. Stallman (RMS) [1] was a staff programmer at MIT. His department, like those at many universities at the time, shared a PDP-10 computer and a single printer. One problem they encountered was that paper would regularly jam in the printer, causing a string of print jobs to pile up in a queue until someone fixed the jam. To get around this problem, the MIT staff came up with a nice social hack: They wrote code for the printer driver so that when it jammed, a message would be sent to everyone who was currently waiting for a print job: "The printer is jammed, please fix it." This way, it was never stuck for long.

In 1980, the lab accepted a donation of a brand-new laser printer. When Stallman asked for the source code for the printer driver, however, so he could reimplement the social hack to have the system notify users on a paper jam, he was told that this was proprietary information. He heard of a researcher in a different university who had the source code for a research project, and when the opportunity arose, he asked this colleague to share it—and was shocked when they refused. They had signed an NDA, which Stallman took as a betrayal of the hacker culture.

The late '70s and early '80s represented an era where software, which had traditionally been given away with the hardware in source code form, was seen to be valuable. Increasingly, MIT researchers were starting software companies, and selling licenses to the software was key to their business models. NDAs and proprietary software licenses became the norms, and the best programmers were hired from universities like MIT to work on private development projects where they could no longer share or collaborate.

As a reaction to this, Stallman resolved that he would create a complete operating system that would not deprive users of the freedom to understand how it worked, and would allow them to make changes if they wished. It was the birth of the free software movement.

2. Creation of GNU and the advent of free software

By late 1983, Stallman was ready to announce his project and recruit supporters and helpers. In September 1983, he announced the creation of the GNU project [2] (GNU stands for GNU's Not Unix—a recursive acronym). The goal of the project was to clone the Unix operating system to create a system that would give complete freedom to users.

In January 1984, he started working full-time on the project, first creating a compiler system (GCC) and various operating system utilities. Early in 1985, he published "The GNU Manifesto [3]," which was a call to arms for programmers to join the effort, and launched the Free Software Foundation in order to accept donations to support the work. This document is the founding charter of the free software movement.

3. The writing of the GPL

Until 1989, software written and released by the Free Software Foundation [4] and RMS did not have a single license. Emacs was released under the Emacs license, GCC was released under the GCC license, and so on; however, after a company called Unipress forced Stallman to stop distributing copies of an Emacs implementation they had acquired from James Gosling (of Java fame), he felt that a license to secure user freedoms was important.

The first version of the GNU General Public License was released in 1989, and it encapsulated the values of copyleft (a play on words—what is the opposite of copyright?): You may use, copy, distribute, and modify the software covered by the license, but if you make changes, you must share the modified source code alongside the modified binaries. This simple requirement to share modified software, in combination with the advent of the internet in the 1990s, is what enabled the decentralized, collaborative development model of the free software movement to flourish.

4. "The Cathedral and the Bazaar"

By the mid-1990s, Linux was starting to take off, and free software had become more mainstream—or perhaps "less fringe" would be more accurate. The Linux kernel was being developed in a way that was completely different to anything people had been seen before, and was very successful doing it. Out of the chaos of the kernel community came order, and a fast-moving project.

In 1997, Eric S. Raymond published the seminal essay, "The Cathedral and the Bazaar [5]," comparing and contrasting the development methodologies and social structure of GCC and the Linux kernel and talking about his own experiences with a "bazaar" development model with the Fetchmail project. Many of the principles that Raymond describes in this essay will later become central to agile development and the DevOps movement—"release early, release often," refactoring of code, and treating users as co-developers are all fundamental to modern software development.

This essay has been credited with bringing free software to a broader audience, and with convincing executives at software companies at the time that releasing their software under a free software license was the right thing to do. Raymond went on to be instrumental in the coining of the term "open source" and the creation of the Open Source Institute.

"The Cathedral and the Bazaar" was credited as a key document in the 1998 release of the source code for the Netscape web browser Mozilla. At the time, this was the first major release of an existing, widely used piece of desktop software as free software, which brought it further into the public eye.

5. Open source

As far back as 1985, the ambiguous nature of the word "free", used to describe software freedom, was identified as problematic by RMS himself. In the GNU Manifesto, he identified "give away" and "for free" as terms that confused zero price and user freedom. "Free as in freedom," "Speech not beer," and similar mantras were common when free software hit a mainstream audience in the late 1990s, but a number of prominent community figures argued that a term was needed that made the concept more accessible to the general public.

After Netscape released the source code for Mozilla in 1998 (see #4), a group of people, including Eric Raymond, Bruce Perens, Michael Tiemann, Jon "Maddog" Hall, and many of the leading lights of the free software world, gathered in Palo Alto to discuss an alternative term. The term "open source" was coined by Christine Peterson [6] to describe free software, and the Open Source Institute was later founded by Bruce Perens and Eric Raymond. The fundamental difference with proprietary software, they argued, was the availability of the source code, and so this was what should be put forward first in the branding.

Later that year, at a summit organized by Tim O'Reilly, an extended group of some of the most influential people in the free software world at the time gathered to debate various new brands for free software. In the end, "open source" edged out "sourceware," and open source began to be adopted by many projects in the community.

There was some disagreement, however. Richard Stallman and the Free Software Foundation continued to champion the term "free software," because to them, the fundamental difference with proprietary software was user freedom, and the availability of source code was just a means to that end. Stallman argued that removing the focus on freedom would lead to a future where source code would be available, but the user of the software would not be able to avail of the freedom to modify the software. With the advent of web-deployed software-as-a-service and open source firmware embedded in devices, the battle continues to be waged today.

6. Corporate investment in open source—VA Linux, Red Hat, IBM

In the late 1990s, a series of high-profile events led to a huge increase in the professionalization of free and open source software. Among these, the highest-profile events were the IPOs of VA Linux and Red Hat in 1999. Both companies had massive gains in share price on their opening days as publicly traded companies, proving that open source was now going commercial and mainstream.

Also in 1999, IBM announced that they were supporting Linux by investing $1 billion in its development, making is less risky to traditional enterprise users. The following year, Sun Microsystems released the source code to its cross-platform office suite, StarOffice, and created the OpenOffice.org [7] project.

The combined effect of massive Silicon Valley funding of open source projects, the attention of Wall Street for young companies built around open source software, and the market credibility that tech giants like IBM and Sun Microsystems brought had combined to create the massive adoption of open source, and the embrace of the open development model that helped it thrive have led to the dominance of Linux and open source in the tech industry today.

Links

[1] https://en.wikipedia.org/wiki/Richard_Stallman

[2] https://groups.google.com/forum/#!original/net.unix-wizards/8twfRPM79u0/1xlglzrWrU0J

[3] https://www.gnu.org/gnu/manifesto.en.html

[4] https://www.fsf.org/

[5] https://en.wikipedia.org/wiki/The_Cathedral_and_the_Bazaar

[6] https://opensource.com/article/18/2/coining-term-open-source-software

[7] http://www.openoffice.org/

Author

Dave Neary is a member of the Open Source and Standards team at Red Hat, helping make Open Source projects important to Red Hat be successful. Dave has been around the free and open source software world, wearing many different hats, since sending his first patch to the GIMP in 1999.

13 Git tips for Git's 13th birthday

BY JOHN SJ ANDERSON

Make your revision-control experience more useful and powerful with these 13 tricks and tips for Git.

GIT [1], the distributed revision-control system that's become the default tool for source code control in the open source world, turned 13 on April 7. One of the more frustrating things about using Git is how much you need to know to use it effectively. This can also be one of the more awesome things about using Git, because there's nothing quite like discovering a new tip or trick that can streamline or improve your workflow.

In honor of Git's 13th birthday, here are 13 tips and tricks to make your Git experience more useful and powerful, starting with some basics you might have overlooked and scaling up to some real power-user tricks!

1. Your ~/.gitconfig file

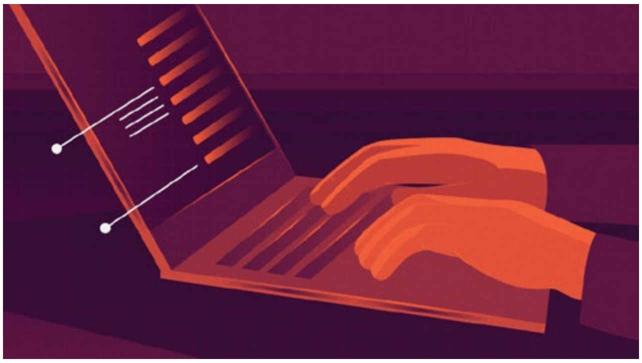
The first time you tried to use the `git` command to commit a change to a repository, you might have been greeted with something like this:

```
*** Please tell me who you are.
Run
  git config --global user.email "you@example.com"
  git config --global user.name "Your Name"
to set your account's default identity.
```

What you might not have realized is that those commands are modifying the contents of `~/.gitconfig`, which is where Git stores global configuration options. There are a vast array of things you can do via your `~/.gitconfig` file, including defining aliases, turning particular command options on (or off!) on a permanent basis, and modifying aspects of how Git works (e.g., which diff algorithm `git diff` uses or what type of merge strategy is used by default). You can even conditionally include other config files based on the path to a repository! See `man git-config` for all the details.

2. Your repo's .gitconfig file

In the previous tip, you may have wondered what that `--global` flag on the `git config` command was doing. It tells Git to update the "global" configuration, the one found in `~/.gitconfig`. Of course, having a global config also



implies a *local* configuration, and sure enough, if you omit the `--global` flag, `git config` will instead update the repository-specific configuration, which is stored in `.git/config`.

Options that are set in the `.git/config` file will override any setting in the `~/.gitconfig` file. So, for example, if you need to use a different email address for a particular repository, you can run `git config user.email "also_you@example.com"`. Then, any commits in that repository will use your other email address. This can be super useful if you work on open source projects from a work laptop and want them to show up with a personal email address while still using your work email for your main Git configuration.

Pretty much anything you can set in `~/.gitconfig`, you can also set in `.git/config` to make it specific to the given repository. In any of the following tips, when I mention adding something to your `~/.gitconfig`, just remember you could also set that option for just one repository by adding it to `.git/config` instead.

3. Aliases

Aliases are another thing you can put in your `~/.gitconfig`. These work just like aliases in the command shell—they set up a new command name that can invoke one or more other commands, often with a particular set of options or flags. They're super useful for longer, more complicated commands you use frequently.

You can define aliases using the `git config` command—for example, running `git config --global --add alias.st status` will make running `git st` do the same thing as running `git status`—but I find when defining aliases, it's frequently easier to just edit the `~/.gitconfig` file directly.

If you choose to go this route, you'll find that the `~/.gitconfig` file is an INI file [2]. INI is basically a key-value file format with particular sections. When adding an alias, you'll be changing the `[alias]` section. For example, to define the same `git st` alias as above, add this to the file:

```
[alias]
st = status
```

(If there's already an `[alias]` section, just add the second line to that existing section.)

4. Aliases to shell commands

Aliases aren't limited to just running other Git subcommands—you can also define aliases that run other shell commands. This is a fantastic way to deal with a recurring, infrequent, and complicated task: Once you've figured out how to do it once, preserve the command under an alias. For example, I have a few repositories where I've forked an open source project and made some local modifications that don't need to be contributed back to the project. I want to keep up-to-date with ongoing development work in the project but also maintain my local changes. To accomplish this, I need to periodically merge the changes from the upstream repo into my fork—which I do by using an alias I call `upstream-merge`. It's defined like this:

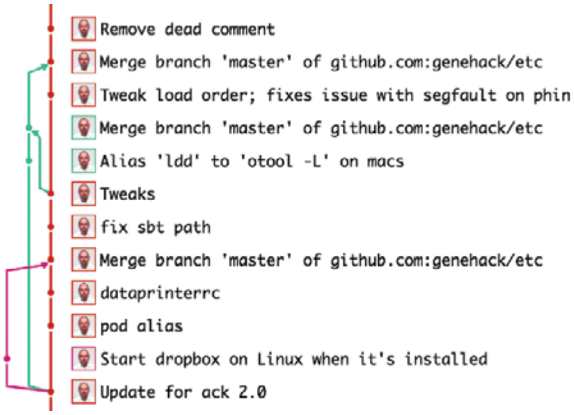
```
upstream-merge = !"git fetch origin -v && git fetch upstream -v
  && git merge upstream/master && git push"
```

The `!` at the beginning of the alias definition tells Git to run the command via the shell. This example involves running a number of `git` commands, but aliases defined in this way can run *any* shell command.

(Note that if you want to copy my `upstream-merge` alias, you'll need to make sure you have a Git remote named `upstream` pointed at the upstream repository you've forked from. You can add this by running `git remote add upstream <URL to repo>`.)

5. Visualizing the commit graph

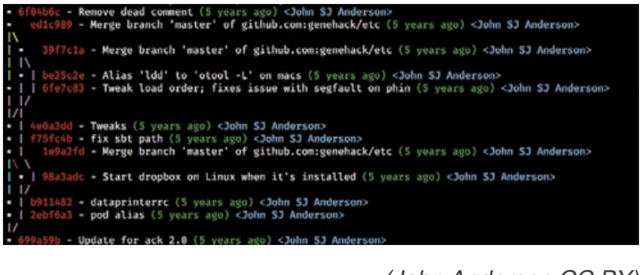
If you work on a project with a lot of branching activity, sometimes it can be difficult to get a handle on all the work that's happening and how it's all related. Various GUI tools allow you to get a picture of different branches and commits in what's called the "commit graph." For example, here's a section of one of my repositories visualized with the GitLab [3] commit graph viewer:



```
Remove dead comment
Merge branch 'master' of github.com:genehack/etc
Tweak load order; fixes issue with segfault on phin
Merge branch 'master' of github.com:genehack/etc
Alias 'ldd' to 'otool -L' on macs
Tweaks
fix sbt path
Merge branch 'master' of github.com:genehack/etc
dataprinterrc
pod alias
Start dropbox on Linux when it's installed
Update for ack 2.0
```

(John Anderson CC BY)

If you're a dedicated command-line user or somebody who finds switching tools to be distracting, it's nice to get a similar view of the commit graph from the command line. That's where the `--graph` argument to the `git log` command comes in:



```
* 6f04b6c - Remove dead comment (5 years ago) <John SJ Anderson>
*   ed1c989 - Merge branch 'master' of github.com:genehack/etc (5 years ago) <John SJ Anderson>
|\
| *   39f7c1a - Merge branch 'master' of github.com:genehack/etc (5 years ago) <John SJ Anderson>
| |\
| * | be25c2e - Alias 'ldd' to 'otool -L' on macs (5 years ago) <John SJ Anderson>
* | | 6fe7c83 - Tweak load order; fixes issue with segfault on phin (5 years ago) <John SJ Anderson>
| |/
|/|
* | 4e0a3dd - Tweaks (5 years ago) <John SJ Anderson>
* | f75fc4b - fix sbt path (5 years ago) <John SJ Anderson>
* |   1e9a2fd - Merge branch 'master' of github.com:genehack/etc (5 years ago) <John SJ Anderson>
|\ \
| * | 98a3adc - Start dropbox on Linux when it's installed (5 years ago) <John SJ Anderson>
| |/
* | b911482 - dataprinterrc (5 years ago) <John SJ Anderson>
* | 2ebf6a3 - pod alias (5 years ago) <John SJ Anderson>
|/
* 699a59b - Update for ack 2.0 (5 years ago) <John SJ Anderson>
```

(John Anderson CC BY)

This is the same section of the same repo visualized with the following command:nklll

```
git log --graph --pretty=format:'%Cred%h%Creset
  -%C(yellow)%d%Creset %s %Cgreen(%cr)
  %C(bold blue)<%an>%Creset' --abbrev-commit --date=relative
```

The `--graph` option adds the graph to the left side of the log, `--abbrev-commit` shortens the commit SHAs [4], `--date=relative` expresses the dates in relative terms, and the `--pretty` bit handles all the other custom formatting. I have this aliased to `git lg`, and it is one of my top 10 most frequently run commands.

6. A nicer force-push

Sometimes, as hard as you try to avoid it, you'll find that you need to run `git push --force` to overwrite the history on a remote copy of your repository. You may have gotten some feedback that caused you to do an interactive rebase, or you may simply have messed up and want to hide the evidence.

One of the hazards with force pushes happens when somebody else has made changes on top of the same branch in the remote copy of the repository. When you force-push your rewritten history, those commits will be lost. This is where `git push --force-with-lease` comes in—it will not allow you to force-push if the remote branch has been updated, which will ensure you don't throw away someone else's work.

7. git add -N

Have you ever used `git commit -a` to stage and commit all your outstanding changes in a single move, only to discover after you've pushed your commit that `git commit -a` ignores newly added files? You can work around this by using the `git add -N` (think "notify") to tell Git about newly added files you'd like to be included in commits before you actually commit them for the first time.

8. git add -p

A best practice when using Git is to make sure each commit consists of only a single logical change—whether that's a fix for a bug or a new feature. Sometimes when you're working, however, you'll end up with more than one commit's worth of change in your repository. How can you manage to divide things up so that each commit contains only the appropriate changes? `git add --patch` to the rescue!

This flag will cause the `git add` command to look at all the changes in your working copy and, for each one, ask if you'd like to stage it to be committed, skip over it, or defer the decision (as well as a few other more powerful options you can see by selecting ? after running the command). `git add -p` is a fantastic tool for producing well-structured commits.

9. git checkout -p

Similar to `git add -p`, the `git checkout` command will take a `--patch` or `-p` option, which will cause it to present each "hunk" of change in your local working copy and allow you to discard it—basically reverting your local working copy to what was there before your change.

This is fantastic when, for example, you've introduced a bunch of debug logging statements while chasing down a bug. After the bug is fixed, you can first use `git checkout -p` to remove all the new debug logging, then you `git add -p` to add the bug fix. Nothing is more satisfying than putting together a beautiful, well-structured commit!

10. Rebase with command execution

Some projects have a rule that each commit in the repository must be in a working state—that is, at each commit, it should be possible to compile the code or the test suite should run without failure. This is not too difficult when you're working on a branch over time, but if you end up needing to rebase for whatever reason, it can be a little tedious to step through each rebased commit to make sure you haven't accidentally introduced a break.

Fortunately, `git rebase` has you covered with the `-x` or `--exec` option. `git rebase -x <cmd>` will run that command after each commit is applied in the rebase. So, for example, if you have a project where `npm run tests` runs your test suite, `git rebase -x npm run tests` would run the test suite after each commit was applied during the rebase. This allows you to see if the test suite fails at any of the rebased commits so you can confirm that the test suite is still passing at each commit.

11. Time-based revision references

Many Git subcommands take a revision argument to specify what part of the repository to work on. This can be the SHA1 of a particular commit, a branch name, or even a symbolic name like `HEAD` (which refers to the most recent commit on the currently checked out branch). In addition to these simple forms, you can also append a specific date or time to mean "this reference, at this time."

This becomes very useful when you're dealing with a newly introduced bug and find yourself saying, "I *know* this worked yesterday! What changed?" Instead of staring at the output of `git log` trying to figure out what commit was changed when, you can simply run `git diff HEAD@{yesterday}`, and see all the changes that have happened since then. This also works with longer time periods (e.g., `git diff HEAD@{'2 months ago'}`) as well as exact dates (e.g., `git diff HEAD@{'2010-01-01 12:00:00'}`).

You can also use these date-based revision arguments with *any* Git subcommand that takes a revision argument. Find full details about which format to use in the man page for `gitrevisions`.

12. The all-seeing reflog<!--12-->

Have you ever rebased away a commit, then discovered there was something in that commit you wanted to keep? You may have thought that information was lost forever and would need to be recreated. But if you committed it in your local working copy, it was added to the reference log (reflog), and you should still be able to access it.

Running `git reflog` will show you a list of all the activity for the current branch in your local working copy and also give you the SHA1 of each commit. Once you've found the commit you rebased away, you can run `git checkout <SHA1>` to check out that commit, copy any information you need, and run `git checkout HEAD` to return to the most recent commit in the branch.

13. Cleaning up after yourself

Whoops! It turns out my basic math skills aren't quite up to the same level as my Git ones. Git was originally released in 2005, which means it turned 13 this year, not 12. To make up for the mistake, here's a 13th tip to bring us up to a baker's dozen [5].

If you use a branching-based workflow, overtime on a long-lived project, unless you're fastidious about cleaning up as each branch is merged, eventually you will end up with a bunch of branches. This can make finding a branch of interest difficult, and you won't be able to see the forest for the... branches, if you will. Even worse, if you have a number of active branches in play, it can be really tedious to figure out whether a branch has been merged (and can be safely deleted) or if it still remains unmerged and should be left alone. Fortunately, Git has your back here: Just run `git branch --merged` to get a list of branches that have been merged into your current branch, or `git branch --merged <branch-name></branch-name>` to find ones that have been merged into some other branch. By default, this will list branches in your local working copy, but if you include `--remote` or `-r` into the command, it will also list merged branches that only exist on the remote.

Important note: if you plan on using the output from `git branch --merged` to clean up those merged branches, you should be aware it will also include the current branch in the output (because, after all, the current branch is merged to the current branch!). Make sure you exclude that branch from anything destructive (or if you forgot to, see tip #12 to learn how the reflog can help you get your branch back, hopefully...)

That's all folks!

Hopefully at least one of these tips has taught you something new about Git, a 13-year-old project that's continuing to innovate and add new features. What's your favorite Git trick?

Links

[1] https://git-scm.com/
[2] https://en.wikipedia.org/wiki/INI_file
[3] https://gitlab.com/
[4] https://en.wikipedia.org/wiki/Secure_Hash_Algorithms
[5] https://en.wikipedia.org/wiki/Dozen#Baker's_dozen

Author

John is the VP of Technology for Infinity Interactive, a technology consultancy and bespoke software development shop. When he's not madly trying to keep up with the pace of change in Javascript development, maintaining Perl modules, or tweaking his Emacs config, he likes to play around with new languages like Swift and write about himself in the third person.

Happy birthday, GNOME: 8 reasons to love this Linux desktop

BY JAY LACROIX

On GNOME's 21st birthday, we highlight some of the features we enjoy the most.

GNOME has been my favorite desktop environment [1] for quite some time. While I always make it a point to check out other environments from time to time, there are some aspects of the GNOME desktop that are hard to live without. While there are many great desktop environments out there, GNOME [2] feels like home to me. Here are some of the features I enjoy most about GNOME.

Stability

Having a stable working environment is the most important aspect of a desktop for me. After all, the feature set of an environment doesn't matter at all if it crashes constantly and you lose work. For me, GNOME is rock-solid. I have heard of others experiencing crashes and instability, but it always seems to be due to either the user running GNOME on unsupported hardware or due to faulty extensions (more on that later). On my end, I run GNOME primarily on hardware that is known to be well-supported in Linux (System76 [3], for example). I also have a few systems that are not as well supported (a custom-built desktop and a Dell Latitude laptop), and I actually don't have any issues there either. For me, GNOME is rock-solid. I have compared stability in other well-known desktop environments, and I had unfortunate results. Nothing comes close to GNOME when it comes to stability.

Extensions

I really enjoy being able to add additional functionality to my environment. I don't necessarily require any extensions, because I am perfectly fine with stock-GNOME with no extensions whatsoever. However, having the ability to add a few things here and there, is welcome. GNOME features various extensions to do things such as add a weather display to your panel, and much more. This adds a level of customization that is not typical of other environments. That said, proceed with caution. Sometimes extensions are of varying quality and may lead to stability issues. I find though that if you only install extensions you absolutely need, and you make sure they're kept up to date (and aren't abandoned by the developer) you'll generally be in good shape.

Activities overview

Activities overview is quite possibly the easiest feature to use in GNOME, and it's barely detailed enough to justify its own section in this article. However, when I use other desktop environments, I miss this feature the most.

The thing is, I am very busy, with multiple projects going on at any one time, and dozens of different windows open. To access the activities overview, I simply press the Super key. Immediately, my workspace is "zoomed out" and I see all of my windows side-by-side. This is often a faster way to locate a window that is hidden behind others, and a good way overall to see what exactly is running on any given workspace.

When using other desktop environments, I will often find myself pressing the Super key out of habit, only to remember that I'm not using GNOME at the time. There are ways of achieving similar behavior in other environments (such as installing and tweaking Compiz), but in GNOME this feature is built-in.

Dynamic workspaces

While working, I am not sure up-front how many workspaces I will need. Sometimes I can be working on three projects at a time, or as many as ten. With most desktop environments, I can access the settings screen and add or remove workspaces as needed. But with GNOME, I have exactly as many workspaces as I need at any given time. Every time I open applications on a workspace, I am given another blank one that I can switch to in order to start another project. Typically, I keep all windows related to a specific project on their own workspace, so it makes it very easy to locate my workflow for a given project.

Other desktop environments have really good implementations of the concept of workspaces, but GNOME's implementation works best for me.

Simplicity

Another thing I love about GNOME is that it's simple and straight to the point. By default, there is only one panel, and it's at the top of the screen. This panel shows you a small amount of information, such as the date, time, and battery usage. GNOME 2 had two panels, so seeing GNOME stripped down to a single panel is welcome and saves room on the screen. Most of the things you don't need to see all the time are hidden within the Activities overview, leaving you with the maximum amount of screen space for the application(s) you are working on. GNOME just stays out of the way and lets you focus on getting your work done, and stays away from fancy widgets and desktop gadgets that just aren't necessary.

In addition, GNOME has really great support for keyboard shortcuts. Most of GNOME's features I can access without needing to touch my mouse, such as SUPER+Page Up and Super Page Down to switch workspaces, Super+Up arrow to maximize windows, etc. In addition, I am able to easily create my own keyboard shortcuts for all of my favorite applications.

GNOME Boxes

GNOME's Boxes app is an underrated gem. This utility makes it very easy to spin up a virtual machine, which is a godsend among developers and those that like to test configurations on multiple distributions and platforms. With Boxes, you can spin up a virtual machine at any time, and it will even automate the installation process for you. For example, if you want a new Ubuntu VM, you simply choose Ubuntu as your desired platform, fill out your username and any related information, and you will have a new Ubuntu VM in a few minutes. When you're done with it, you can power it down or trash it.

For me, I do a lot of DevOps-style work as well as system administration. Being able to test a configuration on a virtual machine before deploying to another environment is great. Sure, you can do the exact same thing in VirtualBox, and VirtualBox is a great piece of software. However, Boxes is built right into GNOME, and desktop environments generally don't offer their own solution for virtualization.

GNOME Music

While I work, I have difficulty tuning out noise in my environment. Therefore, I like to listen to music while I complete projects and tune out the rest of the world. GNOME's Music app is very simplistic and works very well. With most of the music industry gravitating toward streaming music online, and many once-popular open source music players [4] becoming abandoned projects, it's nice to see GNOME support a built-in music player that can play my music collection. It's great to listen to my music collection while I work, and it helps me zone-in to what I am doing.

GNOME Games

When work is done for the day, it's time to play! There's nothing like playing a classic game such as Final Fantasy VI or Super Metroid after a hard day's work. The thing is, I am a huge fan of classic gaming, and I have 22 working gaming consoles and somewhere near 1,000 physical games in my collection. But I may not always have a moment to hook up one of my retro-consoles, so GNOME Games allows me quick-access to emulated versions of my collection. In addition to that, it also works with Libretro cores as well, so it seems to me that the developers of this application have really thought-out what fans of classic gaming like me are looking for in a frontend for gaming.

Links

[1] https://opensource.com/article/18/8/how-navigate-your-gnome-linux-desktop-only-keyboard

[2] https://opensource.com/article/17/8/reasons-i-come-back-gnome

[3] https://opensource.com/article/16/12/open-gaming-news-december-31

[4] https://opensource.com/article/18/6/open-source-music-players

Author

Jay LaCroix is a technologist from Michigan, with a focus on Linux and open-source software. Using Linux since 2002, Jay has been a die-hard fan ever since. He is currently a Senior Solutions Architect and freelance consultant and enjoys training and empowering others to use Linux and to make the most of this amazing software. In his free time, Jay is an author of books such as Linux Mint Essentials, Mastering Linux Network Administration and most recently, Mastering Ubuntu Server. In addition, Jay creates instructional Linux videos at www.learnlinux.tv.

An insider's look at drafting the **GPLv3 license**

BY RICHARD FONTANA

On the 11th anniversary of the GPLv3 license, learn about its lasting contributions to free and open source software.

LAST YEAR, I missed the opportunity to write about the 10th anniversary of GPLv3 [1], the third version of the GNU General Public License. GPLv3 was officially released by the Free Software Foundation (FSF) on June 29, 2007—better known in technology history as the date Apple launched the iPhone. Now, one year later, I feel some retrospection on GPLv3 is due. For me, much of what is interesting about GPLv3 goes back somewhat further than 11 years, to the public drafting process in which I was an active participant.

In 2005, following nearly a decade of enthusiastic self-immersion in free software, yet having had little open source legal experience to speak of, I was hired by Eben Moglen to join the Software Freedom Law Center as counsel. SFLC was then outside counsel to the FSF, and my role was conceived as focusing on the incipient public phase of the GPLv3 drafting process. This opportunity rescued me from a previous career turn that I had found rather dissatisfying. Free and open source software (FOSS) legal matters would come to be my new specialty, one that I found fascinating, gratifying, and intellectually rewarding. My work at SFLC, and particularly the trial by fire that was my work on GPLv3, served as my on-the-job training.

GPLv3 must be understood as the product of an earlier era of FOSS, the contours of which may be difficult for some to imagine today. By the beginning of the public drafting process in 2006, Linux and open source were no longer practically synonymous, as they might have been for casual observers several years earlier, but the connection was much closer than it is now.

Reflecting the profound impact that Linux was already having on the technology industry, everyone assumed GPL version 2 was the dominant open source licensing model. We were seeing the final shakeout of a Cambrian explosion of open source (and pseudo-open source) business models. A frothy business-fueled hype surrounded open source (for me most memorably typified by the Open Source Business Conference) that bears little resemblance to the present-day embrace of open source development by the software engineering profession. Microsoft, with its expanding patent portfolio and its competitive opposition to Linux, was commonly seen in the FOSS community as an existential threat, and the SCO litigation [2] had created a cloud of legal risk around Linux and the GPL that had not quite dissipated.

That environment necessarily made the drafting of GPLv3 a high-stakes affair, unprecedented in free software history. Lawyers at major technology companies and top law firms scrambled for influence over the license, convinced that GPLv3 was bound to take over and thoroughly reshape open source and all its massive associated business investment.

A similar mindset existed within the technical community; it can be detected in the fears expressed in the final paragraph of the Linux kernel developers' momentous September 2006 denunciation [3] of GPLv3. Those of us close to the FSF knew a little better, but I think we assumed the new license would be either an overwhelming success or a resounding failure—where "success" meant something approximating an upgrade of the existing GPLv2 project ecosystem to GPLv3, though perhaps without the kernel. The actual outcome was something in the middle.

I have no confidence in attempts to measure open source license adoption, which have in recent years typically been used to demonstrate a loss of competitive advantage for copyleft licensing. My own experience, which is admittedly distorted by proximity to Linux and my work at Red Hat, suggests that GPLv3 has enjoyed moderate popularity as a license choice for projects launched since 2007, though most GPLv2 projects that existed before 2007, along with their post-2007 offshoots, remained on the old license. (GPLv3's sibling licenses LGPLv3 and AGPLv3 never gained comparable popularity.) Most of the existing GPLv2 projects (with a few notable exceptions like the kernel and Busybox) were licensed as "GPLv2 or any later version." The technical community decided early on that "GPLv2 or later" was a politically neutral license choice that embraced both GPLv2 and GPLv3; this goes some way to explain why adoption of GPLv3 was somewhat gradual and limited, especially within the Linux community.

During the GPLv3 drafting process, some expressed concerns about a "balkanized" Linux ecosystem, whether because of the overhead of users having to understand two different, strong copyleft licenses or because of GPLv2/GPLv3 incompatibility. These fears turned out to be entirely unfounded. Within mainstream server and workstation Linux stacks, the two licenses have peacefully coexisted for a decade now. This is partly because such stacks are made up of separate units of strong copyleft scope (see my discussion of related issues in the container setting [4]). As for incompatibility inside units of strong copyleft scope, here, too, the prevalence of "GPLv2 or later" was seen by the technical community as neatly resolving the theoretical problem, despite the fact that nominal license upgrading of GPLv2-or-later to GPLv3 hardly ever occurred.

I have alluded to the handwringing that some of us FOSS license geeks have brought to the topic of supposed copyleft decline. GPLv3 has taken its share of abuse from critics as far back as the beginning of the public drafting process, and some, predictably, have drawn a link between GPLv3 in particular and GPL or copyleft disfavor in general.

I have viewed it somewhat differently: Largely because of its complexity and baroqueness, GPLv3 was a lost opportunity to create a strong copyleft license that would appeal very broadly to modern individual software authors and corporate licensors. I believe individual developers today tend to prefer short, simple, easy to understand, minimalist licenses, the most obvious example of which is the MIT License [5].

Some corporate decisionmakers around open source license selection may naturally share that view, while others may associate some parts of GPLv3, such as the patent provisions or the anti-lockdown requirements, as too risky or incompatible with their business models. The great irony is that the characteristics of GPLv3 that fail to attract these groups are there in part because of conscious attempts to make the license appeal to these same sorts of interests.

How did GPLv3 come to be so baroque? As I have said, GPLv3 was the product of an earlier time, in which FOSS licenses were viewed as the primary instruments of project governance. (Today, we tend to associate governance with other kinds of legal or quasi-legal tools, such as structuring of nonprofit organizations, rules around project decision making, codes of conduct, and contributor agreements.)

GPLv3, in its drafting, was the high point of an optimistic view of FOSS licenses as ambitious means of private regulation. This was already true of GPLv2, but GPLv3 took things further by addressing in detail a number of new policy problems—software patents, anti-circumvention laws, device lockdown. That was bound to make the license longer and more complex than GPLv2, as the FSF and SFLC noted apologetically in the first GPLv3 rationale document [6].

But a number of other factors at play in the drafting of GPLv3 unintentionally caused the complexity of the license to grow. Lawyers representing vendors' and commercial users' interests provided useful suggestions for improvements from a legal and commercial perspective, but these often took the form of making simply worded provisions more verbose, arguably without net increases in clarity. Responses to feedback from the technical community, typically identifying loopholes in license provisions, had a similar effect.

The GPLv3 drafters also famously got entangled in a short-term political crisis—the controversial Microsoft/Novell deal [7] of 2006—resulting in the permanent addition of new and unusual conditions in the patent section of the license, which arguably served little purpose after 2007 other than to make license compliance harder for conscientious patent-holding vendors. Of course, some of the complexity in GPLv3 was simply the product of well-intended attempts to make compliance easier, especially for community project developers, or to codify FSF interpretive practice. Finally, one can take issue with the style of language used in GPLv3, much of which had a quality of playful parody or mockery of conventional software license legalese; a simpler, straightforward form of phrasing would in many cases have been an improvement.

The complexity of GPLv3 and the movement towards preferring brevity and simplicity in license drafting and unambitious license policy objectives meant that the substantive text of GPLv3 would have little direct influence on later FOSS legal drafting. But, as I noted with surprise and delight [8] back in 2012, MPL 2.0 adapted two parts of GPLv3: the 30-day cure and 60-day repose language from the GPLv3 termination provision, and the assurance that downstream upgrading to a later license version adds no new obligations on upstream licensors.

The GPLv3 cure language has come to have a major impact, particularly over the past year. Following the Software Freedom Conservancy's promulgation, with the FSF's support, of the Principles of Community-Oriented GPL Enforcement [9], which calls for extending GPLv3 cure opportunities to GPLv2 violations, the Linux Foundation Technical Advisory Board published a statement [10], endorsed by over a hundred Linux kernel developers, which incorporates verbatim the cure language of GPLv3. This in turn was followed by a Red Hat-led series of corporate commitments [11] to extend the GPLv3 cure provisions to GPLv2 and LGPLv2.x noncompliance, a campaign to get individual open source developers to extend the same commitment, and an announcement by Red Hat that henceforth GPLv2 and LGPLv2.x projects it leads will use the commitment language directly in project repositories. I discussed these developments in a recent blog post [12].

One lasting contribution of GPLv3 concerns changed expectations for how revisions of widely-used FOSS licenses are done. It is no longer acceptable for such licenses to be revised entirely in private, without opportunity for comment from the community and without efforts to consult key stakeholders. The drafting of MPL 2.0 and, more recently, EPL 2.0 reflects this new norm.

Links

[1] https://www.gnu.org/licenses/gpl-3.0.en.html

[2] https://en.wikipedia.org/wiki/SCO%E2%80%93Linux_disputes

[3] https://lwn.net/Articles/200422/

[4] https://opensource.com/article/18/1/containers-gpl-and-copyleft

[5] https://opensource.org/licenses/MIT

[6] http://gplv3.fsf.org/gpl-rationale-2006-01-16.html

[7] https://en.wikipedia.org/wiki/Novell#Agreement_with_Microsoft

[8] https://opensource.com/law/12/1/the-new-mpl

[9] https://sfconservancy.org/copyleft-compliance/principles.html

[10] https://www.kernel.org/doc/html/v4.16/process/kernel-enforcement-statement.html

[11] https://www.redhat.com/en/about/press-releases/technology-industry-leaders-join-forces-increase-predictability-open-source-licensing

[12] https://www.redhat.com/en/blog/gpl-cooperation-commitment-and-red-hat-projects?source=author&term=26851

Author

Richard is Senior Commercial Counsel on the Products and Technologies team in Red Hat's legal department. Most of his work focuses on open source-related legal issues.

Revisiting the Unix philosophy in 2018

BY MICHAEL HAUSENBLAS

The old strategy of building small, focused applications is new again in the modern microservices environment.

IN 1984, ROB PIKE AND BRIAN W. KERNIGHAN published an article called "Program Design in the Unix Environment [1]" in the AT&T Bell Laboratories Technical Journal, in which they argued the Unix philosophy, using the example of BSD's **cat -v** implementation. In a nutshell that philosophy is: Build small, focused programs—in whatever language—that do only one thing but do this thing well, communicate via **stdin/stdout**, and are connected through pipes.

Sound familiar?

Yeah, I thought so. That's pretty much the definition of microservices [2] offered by James Lewis and Martin Fowler:

> In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API.

While one *nix program or one microservice may be very limited or not even very interesting on its own, it's the combination of such independently working units that reveals their true benefit and, therefore, their power.

*nix vs. microservices

The following table compares programs (such as **cat** or **lsof**) in a *nix environment against programs in a microservices environment.

	***nix**	**Microservices**
Unit of execution	program using **stdin/stdout**	service with HTTP or gRPC API
Data flow	Pipes	?
Configuration & parameterization	Command-line arguments, environment variables, config files	JSON/YAML docs
Discovery	Package manager, man, **make**	DNS, environment variables, OpenAPI

Let's explore each line in slightly greater detail.

Unit of execution

The unit of execution in *nix (such as Linux) is an executable file (binary or interpreted script) that, ideally, reads input from **stdin** and writes output to **stdout**. A microservices setup deals with a service that exposes one or more communication interfaces, such as HTTP or gRPC APIs. In both cases, you'll find stateless examples (essentially a purely functional behavior) and stateful examples, where, in addition to the input, some internal (persisted) state decides what happens.

Data flow

Traditionally, *nix programs could communicate via pipes. In other words, thanks to Doug McIlroy [3], you don't need to create temporary files to pass around and each can process virtually endless streams of data between processes. To my knowledge, there is nothing comparable to a pipe standardized in microservices, besides my little Apache Kafka-based experiment from 2017 [4].

Configuration and parameterization

How do you configure a program or service—either on a permanent or a by-call basis? Well, with *nix programs you essentially have three options: command-line arguments, environment variables, or full-blown config files. In microservices, you typically deal with YAML (or even worse, JSON) documents, defining the layout and config-

uration of a single microservice as well as dependencies and communication, storage, and runtime settings. Examples include Kubernetes resource definitions [5], Nomad job specifications [6], or Docker Compose [7] files. These may or may not be parameterized; that is, either you have some templating language, such as Helm [8] in Kubernetes, or you find yourself doing an awful lot of **sed -i** commands.

Discovery

How do you know what programs or services are available and how they are supposed to be used? Well, in *nix, you typically have a package manager as well as good old man; between them, they should be able to answer all the questions you might have. In a microservices setup, there's a bit more automation in finding a service. In addition to bespoke approaches like Airbnb's SmartStack [9] or Netflix's Eureka [10], there usually are environment variable-based or DNS-based approaches [11] that allow you to discover services dynamically. Equally important, OpenAPI [12] provides a de-facto standard for HTTP API documentation and design, and gRPC [13] does the same for more tightly coupled high-performance cases. Last but not least, take developer experience (DX) into account, starting with writing good Makefiles [14] and ending with writing your docs with (or in?) **style** [15].

Pros and cons

Both *nix and microservices offer a number of challenges and opportunities

Composability

It's hard to design something that has a clear, sharp focus and can also play well with others. It's even harder to get it right across different versions and to introduce respective error case handling capabilities. In microservices, this could mean retry logic and timeouts—maybe it's a better option to outsource these features into a service mesh? It's hard, but if you get it right, its reusability can be enormous.

Observability

In a monolith (in 2018) or a big program that tries to do it all (in 1984), it's rather straightforward to find the culprit when things go south. But, in a

```
yes | tr \\n x | head -c 450m | grep n
```

or a request path in a microservices setup that involves, say, 20 services, how do you even *start* to figure out which one is behaving badly? Luckily we have standards, notably OpenCensus [16] and OpenTracing [17]. Observability still might be the biggest single blocker if you are looking to move to microservices.

Global state

While it may not be such a big issue for *nix programs, in microservices, global state remains something of a discussion. Namely, how to make sure the local (persistent) state is managed effectively and how to make the global state consistent with as little effort as possible.

Wrapping up

In the end, the question remains: Are you using the right tool for a given task? That is, in the same way a specialized *nix program implementing a range of functions might be the better choice for certain use cases or phases, it might be that a monolith is the best option [18] for your organization or workload. Regardless, I hope this article helps you see the many, strong parallels between the Unix philosophy and microservices—maybe we can learn something from the former to benefit the latter.

Links

[1] http://harmful.cat-v.org/cat-v/
[2] https://martinfowler.com/articles/microservices.html
[3] https://en.wikipedia.org/wiki/Douglas_McIlroy
[4] https://speakerdeck.com/mhausenblas/distributed-named-pipes-and-other-inter-services-communication
[5] http://kubernetesbyexample.com/
[6] https://www.nomadproject.io/docs/job-specification/index.html
[7] https://docs.docker.com/compose/overview/
[8] https://helm.sh/
[9] https://github.com/airbnb/smartstack-cookbook
[10] https://github.com/Netflix/eureka
[11] https://kubernetes.io/docs/concepts/services-networking/service/#discovering-services
[12] https://www.openapis.org/
[13] https://grpc.io/
[14] https://suva.sh/posts/well-documented-makefiles/
[15] https://www.linux.com/news/improve-your-writing-gnu-style-checkers
[16] https://opencensus.io/
[17] https://opentracing.io/
[18] https://robertnorthard.com/devops-days-well-architected-monoliths-are-okay/

Author

Michael is a Developer Advocate for Kubernetes and OpenShift at Red Hat where he helps appops to build and operate apps. His background is in large-scale data processing and container orchestration and he's experienced in advocacy and standardization at W3C and IETF. Before Red Hat, Michael worked at Mesosphere, MapR and in two research institutions in Ireland and Austria. He contributes to open source software incl. Kubernetes, speaks at conferences and user groups, and shares good practices around cloud native topics via blog posts and books.

The oldest, active Linux distro, Slackware, turns 25

BY BEN COTTON

Slackware boasts a unique history and a loyal user base.

PATRICK VOLKERDING didn't set out to create a Linux distribution. He just wanted to simplify the process of installing and configuring Softlanding Linux System [1]. But when SLS didn't pick up his improvements, Volkerding decided to release his work as Slackware [2]. On July 17, 1993, he announced version 1.0. A quarter century and 30-plus versions later, Slackware is the oldest actively maintained Linux distribution.

For many early Linux users, Slackware was their introduction. One user told me her first Linux install was Slackware—and she had to use a hex editor to fix the partition tables so that Slackware would install. Support for her hardware was added in a later release. Another got his start building the data center that would power one of the first internet-enabled real estate sites. In the mid-1990s, Slackware was one of the easiest distributions to get and didn't require a lot of effort to get IP masquerading to work correctly. A third person mentioned going to sleep while a kernel compile job ran, only to find out it had failed when he woke up.

All of these anecdotes would suggest a hard-to-use operating system. But Slackware fans don't see it that way. The project's website says the two top priorities are "ease of use and stability." For Slackware, "ease of use" means simplicity. Slackware does not include a graphical installer. Its package manager does not perform any dependency resolution. This can be jarring for new users, particularly within the last few years, but it also enables a deeper understanding of the system.

The different take on ease of use isn't the only thing unique about Slackware. It also does not have a public bug tracker, code repository, or well-defined method of community contribution. Volkerding and a small team of contributors maintain the tree in a rolling release called "-current" and publish a release when it meets the feature and stability goals they've set.

As the oldest distro around, Slackware has been very influential. The earliest releases of SUSE Linux [3] were based on Slackware, and distributions such as Arch Linux [4] can be seen as philosophical heirs to Slackware. And while its popularity may have fallen over the years—the slightly younger Debian [5] has 10x the number of subscribers on its sub-Reddit, for example—it remains an active project with a loyal fan base. So happy 25th birthday, Slackware, and here's to 25 more!

Links

[1] https://en.wikipedia.org/wiki/Softlanding_Linux_System
[2] http://www.slackware.com/
[3] https://en.wikipedia.org/wiki/SUSE_Linux_Enterprise
[4] https://www.archlinux.org/
[5] https://www.debian.org/

Author

Ben Cotton is a meteorologist by training, but weather makes a great hobby. Ben works as a the Fedora Program Manager at Red Hat. He co-founded a local open source meetup group, and is a member of the Open Source Initiative and a supporter of Software Freedom Conservancy. Find him on Twitter (@FunnelFiasco) or at FunnelFiasco.com.

Top 8 Python conferences to attend in 2019

BY NICHOLAS HUNT-WALKER

Resolve to expand your Python knowledge and network at these events.

THERE ARE A LOT of reasons to go to tech conferences [1], and even more of a reason to go to a conference focused specifically on your chosen programming language. My favorite is Python [2].

Rather than rehash all the various reasons why conferences are great and you should attend, I'll go right into which Python conferences you might want to show up to in 2019.

PyTennessee

- Conference homepage [3]
- Twitter [4]
- Dates: February 9-10, 2019
- Location: Nashville, Tennessee
- Registration: Open; $100 individual, $200 corporate

Started in 2014, PyTennessee is an annual regional conference held every February in Tennessee. It features much of what you'd expect from a conference: keynotes, regular speakers, tutorials, and some company exhibitions.

The talks cover topics from software development to data science and machine learning. Often, speakers will also talk about developer-adjacent topics, such as developing emotional intelligence as developers and better ways to build engineering communities. There's a Game Night/After Party on the Saturday and the conference provides some programming education through its Young Coders program.

This conference is a nonprofit event facilitated by TechFed Nashville, an organization with the mission to support and grow grassroots tech talent in Tennessee through educational events and groups.

PyCascades

- Conference homepage [5]
- Twitter [6]
- YouTube [7]
- Dates: February 23-24, 2019
- Location: Seattle, Washington
- Registration: Open; $82.57 student, $110.10 individual, $220.20 corporate

PyCascades debuted this year in Vancouver, BC, Canada, as a regional PyCon in the Pacific Northwest. Its organizing team includes members from Python user groups in Vancouver, Seattle, and Portland. I'm quite partial to this event, as I was able to give the talk that spurred my first posts to Opensource.com [8]!

Last year's conference was fairly small, with final sales peaking just under 400 people. The 2019 iteration in Seattle is set for a larger guest list, but the general feel of a small regional conference is still the goal.

Like PyTennessee, the talks encompass a variety of skill levels (beginner to advanced) as well as technical topics (web dev, device dev, tech education, data science, and machine learning). Last year's conference also featured several talks on tech-adjacent topics like understanding racial bias in your software and navigating unconscious bias in your personal and professional interactions.

One of the big differences at PyCascades is that there is no dedicated Q&A portion for any talk (at least there wasn't last year). There are instead breaks after each talk where speakers can engage directly with people who have questions or comments while the next speakers prep to begin their talks.

The talks occur in a single track, eliminating the need to make tough decisions about which to attend and which to leave behind. Last year, it finished with a day of sprints, where participants gathered together to collaborate on and push changes to various Python-focused open source projects.

PyCon US

- Conference homepage [9]
- Twitter [10]
- YouTube (2018) [11]
- Dates: May 1-9, 2019
- Location: Cleveland, Ohio
- Registration: Open; $125 student, $400 individual, $700 corporate, $150 per tutorial

PyCon is the big Python conference that brings Pythonistas together from around the US and often from around the world. While the regional conferences may last two or three days, PyCon exceeds a week, with a variety of events and workshops—talks are only a small bit.

Prior to the three main days of talks and keynotes that typically define a conference, PyCon features two days of tutorials on a variety of Python projects that often are given by the creators or maintainers of the projects. During the main conference days, there are dozens of talks organized by topic, a massive exposition hall for conference sponsors and vendors often looking to hire or educate about their products and services, an academic poster session, a dedicated job fair, and a benefit auction. The final four days are full of sprints on various projects that are looking to attract developers of all experience levels.

All in all, PyCon is a great place to learn about new products and techniques, teach others from your own experience, and meet dozens of new people at all experience levels throughout the world of Python development.

PyOhio

- Conference homepage [12]
- Twitter [13]
- YouTube [14]
- Dates: July 27-28, 2019
- Location: Columbus, Ohio
- Registration: Open; Free!

PyOhio is unique in one sense because it's the only event on this list that's completely free. The organizers ask for donations, but there's no barrier to entry if you can't manage one. It's also special in that, while they do have corporate sponsors, there are no exhibitors.

PyOhio is all about the talks and sprints. The evening before the conference's two days of talks and tutorials, there's an opening reception with sprints, with more sprints at the end of the first day. The talks are the same variety of Python-focused presentations that we've come to know and love, featuring speakers from all over.

On its own, it's a great conference to attend. It's even better if your budget for conference attendances is tight but you still want to be part of the community.

DjangoCon

- Conference homepage [15]
- Twitter [16]
- YouTube [17]
- Dates: September 22-27, 2019
- Location: TBD
- Registration: Not open yet; 2018's fees were: $295 discounted, $595 individual, $795 corporate, $195 per tutorial

While the name may imply a strong focus on the Python web framework Django, DjangoCon also includes a fair bit of content that appeals to all Python developers. It is still very strong on Django and features tutorials, talks, and sprints for the framework and its related packages and practices.

The first day is filled with 3.5-hour tutorials at various levels. The next three days feature keynotes and two talk tracks about various topics in Django and Python development.

If you're on the job hunt, this is a great time to meet companies with interest in Django enthusiasts. If you're hiring, you can find Django users at all levels everywhere you turn.

The last two days feature sprints, where you can work alongside many project maintainers and team leaders in the Django community.

PyGotham

- Conference homepage [18]
- Twitter [19]
- YouTube (2018) [20]
- Dates (likely): Early October 2019
- Location: New York City
- Registration: Not open yet; 2018's fees were: $75 academic, $200 individual, $300 corporate

Founded in 2011 and based in New York City, PyGotham features two full days of talks and social events for Python enthusiasts of all types. This conference is focused heavily on the talks, with no sprints or exhibitors to speak of. There are tutorials, but they're specifically for young coders (age 10 and older).

The talk topics range across all types of Python topics, from web dev, to natural language processing, to testing, to Python fundamentals. If you attend PyGotham, you'll be getting a great dose of useful tricks and great lessons about the language.

North Bay Python

- Conference homepage [21]
- Twitter [22]
- YouTube [23]
- Dates (likely): Early November 2019
- Location: Petaluma, California
- Registration: Not open yet; 2018's fees were: $50 individual, $100 individual supporter, $200 corporate, $500 individual sponsor

North Bay Python is a two-day, single-track, nonprofit Python conference held in downtown Petaluma, California. Its goal is to appeal to every type of Python user, from students to field experts. Like PyCascades, it's a fairly small conference; 2018's event hosted about 400 attendees.

The conference features over 20 sessions from experienced presenters hailing from across the US and other countries speaking on a wide mix of topics. Sessions include updates from leading Python community members on the state of major projects, technical explorations of how Python and Python-based systems work in practice, and thought-provoking explorations and lessons about people-focused, tech-adjacent topics like team development, diversity and inclusion, and communication skills.

PyData

- Conference homepage [24]
- Twitter [25]
- YouTube [26]
- Dates: Various [27] (next one is Jan. 9-11, 2019)
- Location: Various (next one is in Miami)
- Registration: Open (Miami); $85 student, $235 individual

While the other conferences on this list range across various areas of Python software development, PyData is specifically geared toward users and developers of data analysis tools. The global PyData network promotes discussions of best practices, new approaches, and emerging technologies for data management, processing, analytics, and visualization. In its focus on all things data, this conference also includes other languages big in analytics, such as R and Julia.

The upcoming event in Miami has a specific call for presentations addressing some of the concerns and strengths of South Florida, including climate and weather modeling, ecological data, civic data science, digital humanities, and biomedical data.

See our list of 40 top Linux and open source conferences in 2019 [28] for more upcoming events to attend. Are you proposing a talk for a 2019 event? Maybe the topic would also make a great article for Opensource.com. Send your story idea to us at open@opensource.com.

Links

[1] https://opensource.com/life/16/2/attending-technical-conferences-whats-big-deal

[2] https://www.python.org/

[3] https://www.pytennessee.org/

[4] https://twitter.com/PyTennessee

[5] https://2019.pycascades.com/

[6] https://twitter.com/pycascades

[7] https://www.youtube.com/channel/UCtWI06j1EADmEOGj2iJhSyA

[8] https://opensource.com/users/nhuntwalker

[9] https://us.pycon.org

[10] https://twitter.com/pycon

[11] https://www.youtube.com/channel/UCsX05-2sVSH7Nx3zuk3NYuQ

[12] https://www.pyohio.org

[13] https://twitter.com/pyohio

[14] https://www.youtube.com/channel/UCYqdrfvhGxNW3vXebypqXoQ

[15] https://djangocon.us

[16] https://twitter.com/djangocon

[17] https://www.youtube.com/channel/UC0yY6a79pPY9J0ShIHRf6yw

[18] https://pygotham.org

[19] https://twitter.com/PyGotham

[20] https://www.youtube.com/channel/UCTse88P9vOnsMaovjNl1YBA

[21] https://northbaypython.org/

[22] https://twitter.com/northbaypython

[23] https://www.youtube.com/channel/UCLc1vUexbRTIRBJcUG9U6ug

[24] https://pydata.org/

[25] https://twitter.com/PyData

[26] https://www.youtube.com/channel/UCOjD18EJYcsBog4lozkF_7w

[27] https://pydata.org/events/

[28] https://opensource.com/article/18/12/top-2019-conferences

Author

Nicholas Hunt-Walker is a software developer at Starbucks' department of Emerging Technology. His specialty is in using Python for development, but he dabbles heavily in JavaScript as well. Formerly a graduate student in astronomy at the University of Washington, he studied the structure of our galaxy by looking at the positions and properties of evolved stars before using his skills as a scientist to dip into data science. Driven by curiosity and the possibility of building something cool and interesting, Hunt-Walker loves web/software development because it offers a blank canvas, and data analysis because it gives him the opportunity to be curious in a structured and systematic way while also learning something new. Email him at nhuntwalker@gmail.com.

40 top Linux and open source conferences in 2019

BY RIKKI ENDSLEY

We've rounded up a few favorite picks for conferences to attend in 2019.

EVERY YEAR OPENSOURCE.COM editors, writers, and readers attend open source-related conferences and events hosted around the world. As we started planning our 2019 schedules, we rounded up a few top picks for the year.

Which conferences do you plan to attend in 2019? If you don't see your conference on this list, be sure to add it to our community conference calendar [1]. (And for more events to attend, check out The Enterprisers Project [2] list of business leadership conferences worth exploring in 2019 [3].)

Are you proposing a talk for a 2019 event? Maybe the topic would also make a great article for Opensource.com. Send your story idea to us at open@opensource.com. Check the links at the bottom of this page for lots of examples from 2018 events.

linux.conf.au (LCA) [4]
Christchurch, New Zealand
January 21 – 25, 2019
The Linux of Things-themed event will explore the use of free open source software and hardware for internet of things devices, along with security concerns, privacy, and legal aspects, environmental impacts, everyday communication, health, ethics, and more.

DevConf.cz [5]
Brno, Czechia
January 25 – 27, 2019
A free Red Hat-sponsored community conference for developers, admins, DevOps engineers, testers, documentation writers and other contributors to open source technologies such as Linux, middleware, virtualization, storage, cloud, and mobile where FLOSS communities sync, share, and hack on upstream projects together.

FOSDEM [6]
Brussels, Belgium
February 2 – 3, 2019
FOSDEM is a free event for software developers to meet, share ideas, and collaborate.

PyCon Namibia 2019 [7]
Windhoek, Namibia
February 19 – 22, 2019
Namibia's international open source software conference returns for its fifth edition.

PyCascades 2019 [8]
Seattle, WA, USA
February 23 – 25, 2019
PyCascades is a two-day Python conference held in the scenic coastal city of Seattle, Washington (USA).

SCALE 17x [9]
Pasadena, CA, USA
March 7 – 10, 2019
SCaLE is the largest community-run open-source and free software conference in North America. It is held annually in the greater Los Angeles area.

Open Source Leadership Summit [10]
Half Moon Bay, CA, USA
March 12 – 14, 2019
The Linux Foundation's Open Source Leadership Summit is where industry leaders convene to drive digital transformation with open source technologies and learn how to collaboratively manage the largest shared technology investment of our time.

Chemnitzer Linux-Tage [11]
Chemnitz, Germany
March 16 – 17, 2019
Annual Linux and free (libre) software event

NetDev [12]
Prague, Czech Republic
March 20 – 22, 2019
Netdev 0x13 is a conference of the netdev community, by the netdev community, for the netdev community. The focus is on Linux kernel networking and user space utilization of the interfaces to the Linux kernel networking subsystem.

PyCon SK 2019 [13]
Bratislava, Slovakia
March 22 – 25, 2019
The PyCon SK 2019 conference, which will take place in Bratislava, is the annual gathering for the community using and developing the open source Python programming language. It is organized by the volunteers from the SPy o.z., civic association dedicated to advancing and promoting Python and other open source technologies and ideas.

LibrePlanet [14]
Boston, MA, USA
March 23 – 24, 2019
LibrePlanet is an annual conference hosted by the Free Software Foundation for free software enthusiasts and anyone who cares about the intersection of technology and social justice.

Open edX [15]
San Diego, CA, USA
March 26 – 29, 2019
The Open edX Conference is produced in collaboration with the Open edX community to discuss topics such as the Open edX learning platform, new research in online learning best practices, and new approaches to collaborative learning.

FLISoL [16]
Locations across Latin America
April 27, 2019
Annual Latin American free software installation festival

OpenStack Summit [17]
Denver, CO, USA
April 29 – May 5, 2019
In addition to OpenStack-related sessions, the event features Kata Containers, Ansible, Ceph, Kubernetes, ONAP, and more projects. Featured topics include CI/CD, container infrastructure, edge computing, HPC/GPU/AI, open source community, private and hybrid cloud, public cloud, telecom and NFV.

PyCon USA [18]
Cleveland, OH, USA
May 1 – 9, 2019
The PyCon 2019 conference, which will take place in Cleveland, is the largest annual gathering for the community using and developing the open-source Python programming language. It is produced and underwritten by the Python Software Foundation, the 501(c)(3) nonprofit organization dedicated to advancing and promoting Python.

Red Hat Summit 2019 [19]
Boston, MA, USA
May 7 – 9, 2019
Red Hat Summit is an open source technology event to showcase the latest and greatest in cloud computing, platform, virtualization, middleware, storage, and systems management technologies.

Libre Graphics Meeting [20]
Saarbruecken, Germany
May 29 – June 2, 2019
The Libre Graphics Meeting (LGM) is an annual meeting on free and open source software for graphics.

OSCON [21]
Portand, OR, USA
July 15 – 18, 2019
OSCON focuses on leading-edge software development incorporating AI, cloud technology, and distributed computing.

Open Source Summit Japan [22]
Tokyo, Japan
July 17 – 19, 2019
Open Source Summit Japan is a conference for technologists and open source industry leaders to collaborate and share information, learn about the latest in open source technologies and find out how to gain a competitive advantage by using innovative open solutions.

GopherCon USA [23]
San Diego, CA, USA
July 24 – 27, 2019
North American event dedicated to the Go programming language

PyCon AU 2019 [24]
Sydney, Australia
August 2 – 7, 2019
PyCon Australia (PyCon AU) is the national conference for the Python Programming Community, bringing together professional, student, and enthusiast developers with a love for developing with Python.

Open Source Summit North America [25]
San Diego, CA, USA
August 21 – 23, 2019
Open Source Summit is a conference for developers, architects and other technologists—as well as open source community and industry leaders—to collaborate, share information, learn about the latest technologies, and gain a competitive advantage by using innovative open solutions.

All Things Open [26]
Raleigh, NC, USA
October 13 – 15, 2019
A conference exploring open source, open tech and open web in the enterprise

LISA19 [27]
Portland, OR, USA
October 28 – 30, 2019
LISA is the premier conference for operations professionals, where sysadmins, systems engineers, IT operations professionals, SRE practitioners, developers, IT managers, and academic researchers share real-world knowledge about designing, building, securing, and maintaining the critical systems of our interconnected world.

KubeCon+CloudNativeCon [28]
San Diego, CA, USA
November 18 – 21, 2019
The Cloud Native Computing Foundation's flagship conference gathers adopters and technologists from leading open source and cloud native communities.

Devopsdays [29]
Location: Around the world
Dates: Year-round
Devopsdays is a worldwide series of technical conferences covering topics of software development, IT infrastructure operations, and the intersection between them. Each event is run by volunteers from the local area.

Embedded Recipes [30]
Paris, France
Dates: TBD
The open source embedded conference

EuroPython [31]
Location: TBD
Dates: TBD
The European Python Conference

FISL 19 [32]
Porto Alegre, Brazil
Dates: TBD
International free software conference

GopherCon BR [33]
Florian—polis, Brazil
Dates: TBD
Latin American event dedicated to the Go programming language

Kernel Recipes [34]
Paris, France
Dates: TBD
Informal conference about the Linux kernel

LatinoWare 16 [35]
Foz do Iguaçu, Brazil
Dates: TBD
Latin American FOSS conference

LVEE [36]
Grodno, Belarus
Dates: TBD
International conference of developers and users of free / open source software

Ohio Linux Fest [37]
Columbus, OH, USA
Dates: TBD
The Ohio LinuxFest is a grassroots conference for the GNU/Linux/Open Source Software/Free Software community that started in 2003 as a large inter-LUG (Linux User Group) meeting and has grown steadily since. It is a place for the community to gather and share information about Linux and Open Source Software.

Open Hardware Summit [38]
Location: TBD
Dates: TBD
The Open Hardware Summit is the annual conference organized by the Open Source Hardware Association a 501(c)(3) not for profit charity. It is the world's first comprehensive conference on open hardware; a venue and community in which we discuss and draw attention to the rapidly growing Open Source Hardware movement.

ORConf [39]
City TBD, Europe
Dates: TBD
ORConf is an annual conference for open source digital, semiconductor and embedded systems designers and users.

OWASP AppSec EU [40]
City TBD, Europe
Dates: TBD
An application security conference for European developers and security experts

SeaGL [41]
Seattle, WA, USA
Dates: TBD
SeaGL is a grassroots technical conference dedicated to spreading awareness and knowledge about the GNU/Linux community and free/libre/open-source software/hardware.

SouthEast LinuxFest [42]
Charlotte, NC, USA
Dates: TBD
The SouthEast LinuxFest is a community event for anyone who wants to learn more about Linux and open source Software. It is part educational conference and part social gathering. Like Linux itself, it is shared with attendees of all skill levels to communicate tips and ideas, and to benefit all who use Linux and open source Software.

OWASP AppSec USA [43]
City TBD, USA
Dates: TBD
An application security conference for developers and security experts

2018 conference-inspired articles include:

All Things Open [44]
LISA18 [45]
OpenStack Summit [46]
OSCON [47]
Red Hat Summit [48]

Links

[1] https://opensource.com/resources/conferences-and-events-monthly
[2] https://enterprisersproject.com
[3] https://enterprisersproject.com/article/2018/11/business-leadership-conferences-worth-exploring-2019
[4] https://linux.conf.au/
[5] https://devconf.info/cz
[6] https://fosdem.org/2019/
[7] https://na.pycon.org/
[8] https://2019.pycascades.com/
[9] https://www.socallinuxexpo.org/scale/17x
[10] https://events.linuxfoundation.org/events/open%20source-leadership-summit-2019/
[11] https://chemnitzer.linux-tage.de/2019/en
[12] https://netdevconf.org/0x13/
[13] https://2019.pycon.sk/en/
[14] https://libreplanet.org/2019/
[15] https://con.openedx.org/
[16] https://flisol.info/
[17] https://www.openstack.org/summit/denver-2019/
[18] https://us.pycon.org/2019/
[19] https://www.redhat.com/en/summit/2019
[20] https://libregraphicsmeeting.org/2019/
[21] https://conferences.oreilly.com/oscon/oscon-or
[22] https://events.linuxfoundation.org/events/open-source-summit-japan-2019/
[23] https://www.gophercon.com/
[24] https://2019.pycon-au.org/
[25] https://events.linuxfoundation.org/events/open-source-summit-2019/
[26] https://allthingsopen.org/
[27] https://www.usenix.org/conference/lisa19
[28] https://events.linuxfoundation.org/events/kubecon-cloudnativecon-north-america-2019/
[29] https://www.devopsdays.org/
[30] https://embedded-recipes.org/2018/
[31] https://ep2018.europython.eu/en/
[32] http://fisl18.softwarelivre.org/index.php/en/
[33] https://2018.gopherconbr.org/
[34] https://kernel-recipes.org/en/2018/
[35] https://latinoware.org/
[36] https://lvee.org/en/main
[37] https://ohiolinux.org/
[38] https://www.oshwa.org/
[39] https://orconf.org/
[40] https://2018.appsec.eu/
[41] https://seagl.org
[42] http://www.southeastlinuxfest.org/
[43] https://2018.appsecusa.org/
[44] https://opensource.com/tags/all-things-open
[45] https://opensource.com/tags/lisa
[46] https://opensource.com/tags/openstack-summit
[47] https://opensource.com/tags/oscon
[48] https://opensource.com/tags/red-hat-summit

Author

Rikki Endsley is a community architect and editor for Opensource.com. In the past, she worked as the community evangelist on the Open Source and Standards (OSAS) team at Red Hat; a freelance tech journalist; community manager for the USENIX Association; associate publisher of *Linux Pro Magazine*, *ADMIN*, and *Ubuntu User*; and as the managing editor of *Sys Admin* magazine and *UnixReview.com*. Follow her on Twitter at: @rikkiends.

5 resolutions for open source project maintainers

BY BEN COTTON

No matter how you say it, good communication is essential to strong open source communities.

I'M GENERALLY not big on New Year's resolutions. I have no problem with self-improvement, of course, but I tend to anchor around other parts of the calendar. Even so, there's something about taking down this year's free calendar and replacing it with next year's that inspires some introspection.

In 2017, I resolved to not share articles on social media until I'd read them. I've kept to that pretty well, and I'd like to think it has made me a better citizen of the internet. For 2019, I'm thinking about resolutions to make me a better open source software maintainer.

Here are some resolutions I'll try to stick to on the projects where I'm a maintainer or co-maintainer.

1. Include a code of conduct

Jono Bacon included "not enforcing the code of conduct" in his article "7 mistakes you're probably making [1]." Of course, to *enforce* a code of conduct, you must first *have* a code of conduct. I plan on defaulting to the Contributor Covenant [2], but you can use whatever you like. As with licenses, it's probably best to use one that's already written instead of writing your own. But the important thing is to find something that defines how you want your community to behave, whatever that looks like. Once it's written down and enforced, people can decide for themselves if it looks like the kind of community they want to be a part of.

2. Make the license clear and specific

You know what really stinks? Unclear licenses. "This software is licensed under the GPL" with no further text doesn't tell me much. Which version of the GPL [3]? Do I get to pick? For non-code portions of a project, "licensed under a Creative Commons license" is even worse. I love the Creative Commons licenses [4], but there are several different licenses with significantly different rights and obligations. So, I will make it very clear which variant and version of a license applies to my projects. I will include the full text of the license in the repo and a concise note in the other files.

Sort of related to this is using an OSI-approved license [5]. It's tempting to come up with a new license that says exactly what you want it to say, but good luck if you ever need to enforce it. Will it hold up? Will the people using your project understand it?

3. Triage bug reports and questions quickly

Few things in technology scale as poorly as open source maintainers. Even on small projects, it can be hard to find the time to answer every question and fix every bug. But that doesn't mean I can't at least acknowledge the person. It doesn't have to be a multi-paragraph reply. Even just labeling the GitHub issue shows that I saw it. Maybe I'll get to it right away. Maybe I'll get to it a year later. But it's important for the community to see that, yes, there is still someone here.

4. Don't push features or bug fixes without accompanying documentation

For as much as my open source contributions over the years have revolved around documentation, my projects don't reflect the importance I put on it. There aren't many commits I can push that don't require some form of documentation. New features should obviously be documented at (or before!) the time they're committed. But even bug fixes should get an entry in the release notes. If nothing else, a push is a good opportunity to also make a commit to improving the docs.

5. Make it clear when I'm abandoning a project

I'm really bad at saying "no" to things. I told the editors I'd write one or two articles for Opensource.com [6] and here I am almost 60 articles later. Oops. But at some point, the things that once held my interests no longer do. Maybe the project is unnecessary because its functionality got absorbed into a larger project. Maybe I'm just tired of it. But it's unfair to the community (and potentially dangerous, as the recent event-stream malware injection [7] showed) to leave a project in limbo. Maintainers have the right to walk away whenever and for whatever reason, but it should be clear that they have.

Links

[1] https://opensource.com/article/17/8/mistakes-open-source-avoid
[2] https://www.contributor-covenant.org/
[3] https://opensource.org/licenses/gpl-license
[4] https://creativecommons.org/share-your-work/licensing-types-examples/
[5] https://opensource.org/
[6] http://Opensource.com
[7] https://arstechnica.com/information-technology/2018/11/hacker-backdoors-widely-used-open-source-software-to-steal-bitcoin/

Author

Ben Cotton is a meteorologist by training, but weather makes a great hobby. Ben works as a the Fedora Program Manager at Red Hat. He co-founded a local open source meetup group, and is a member of the Open Source Initiative and a supporter of Software Freedom Conservancy. Find him on Twitter (@FunnelFiasco) or at FunnelFiasco.com.