Free and Open: A (rough) Primer

Most things "open" now refer back to "open source software" and model themselves in some way on the goals and practices of open source software communities. These, in turn started as something of a marketing effort for an older cousin, the "free software" movement.

This primer is meant for those familiar with daily use of computing devices at home and especially at work, but who generally are not involved in software development.

I've tried to highlight in **bold text** the take-home message of some sections in case one is already familiar with most concepts reviewed in that section.

Software

So, we'll talk about software for just a little bit.

These days, most folk have a pretty good conception of what we usually mean by software, but to review: We understand that these devices we work with–desktop computers still for some of us, laptop computers for many of us, phones for most of us, but also increasingly many things all around us, like cars and speakers and thermostats and doorbells–can be made to do different things if we install on them different apps.
These apps, or, slightly less colloquially "applications" are what we mean by software. Sometimes these apps come pre-instatalled, sometimes we add them from a store. Mostly, we download apps through network connections. Occasionally even now, some are installed through a type of removable medium, usually a keychain flash drive. Some of us remember older forms of installation media like DVD's, CD's, possibly floppy disks or even tapes.

Whatever its manner of distribution, software is the portable, fungible, mutable part in any computing device, the essence of the ability for a computing device to be customized in nearly limitless variety.

The broad and growing spread of computing devices into so many aspects of our lives is one big reason we care how software is made.

Source code

At a place like RIT, many of us understand that this software needs to be made by someone, known variously by the terms "programmer" or "coder" or maybe "software engineer" or "developer". There's a lot that goes into making software, but at the most fundamental level, we talk about writing software. We talk about writing it as we might talk about writing a poem or a play or a novel. Writing software is a craft that has to be learned and practiced. It can be art. Like other

art, its pursuit can invoke passion in the truest sense of the word, driving those who seek to cross the gulf between conception and realization variously through cycles of suffering and exultation, through despair, awe, wonder, delight.

We write software in any of a large and growing number of artificial languages, programming languages. The form of software that people can read and write is the "source" (or more fully, the "source code" to which "open source" refers) written in one of these languages. Once written, source code is transformed in various ways into something computers can read and act upon. This transformed version is sometimes referred to as "machine code" or "object code" or often just as the "binary" or "binaries".

Changing the way software works, whether fixing bugs or adding new features, involves (usually still, but with AI not entirely) humans looking at and modifying this source code, then once again transforming this now-changed version into machine-readable binaries. Again, a hint of the comparison to literature persists, in that we speak of using tools called "editors" to make these changes to the human-readable source.

Free and open source software communities

We can talk about software, and we can talk about who makes it. With this distinction in mind, I'll note that there are two distinct movements, one whose adherents prefer to talk about "free software" in which the "free" refers to freedom, not price. The other movement refers to "open source" and prefers to dwell on what they see as several pragmatic benefits of freedom while assiduously avoiding using the word "freedom" itself.

Though we might talk about these groups of people using different terms, the digital artifacts with which they concern themselves are nearly always the same. So, we tend to see terms like "Free and Open Source Software" (FOSS) or sometimes "Free/Libre/Open Source Software" (FLOSS) as a catchall term including the work of both groups but, as these things often go, satisfying neither group fully.

Software is made from source code, so we care who care who can read that source code better to understand what that software does, and who can change that source code to change how the software, and thus all our computing devices, operate.

Free and open source software criteria

Software is free and open if it is available to its users under certain criteria. There is broad agreement about these criteria, expressed at more or less length by the Open Source Initiative's (OSI's) Open Source Defintion (OSD), by the Debian project's Free Software Guidelines (DFSG) from which the OSD was first

derived, and the Free Software Foundation and GNU Project's four freedoms. I usually summarize these as "the four R's"

- Run
- Read
- Revise
- Redistribute

That is to say, free and open source software is **only** considered free and open if those who use it 1) can **use** it for any purpose (run), 2) can inspect or **study** the source code to learn how it works (read), 3) can make any changes they want to the source code in order to **improve** the software for their own purposes (revise) and 4) whether they can **share** either the original (upstream) version of the source code verbatim or their changed (derivative) version to other people (downstream).

These criteria have been honed over the past several decades, in the face of various attempts to exploit the work of free and open source software communities by watering them down. These attempts have driven most who embrace these principles to be wary of proposed changes. Expect advocates to insist on all of these criteria and to react coolly, at best, to arguments or suggestions in favor of exceptions or changes.

In a nutshell, we use a single term to talk about the process of taking a project, making a copy, and then working on that copy as if that project were your own. We say that one has the right to make a 'fork' of that project.

Intellectual property

Some free software activists point out that the term "intellectual property" (IP) covers a range of legal concepts that are in many important ways different from each other. Out of concern that using this one term for disparate areas might (or worse, might intentionally) cause confusion, they discourage use of the phrase. I use it because the term has such wide currency, and because most of the concepts involved do have similarities important to this discussion. Various terms lumped together under IP include copyright, trademark, patents, and trade secrets.

What ties them together is the idea that they secure, in some fashion and to some degree, exclusive access to an intangible good, increasingly these days manifested either partially or wholly in digital form. The law has various ways of granting exclusive access to IP's creators. Those creators then have various avenues through which they can arrange to pass along, either in whole or in part and often for a fee, that access. These arrangements often take the form either of contracts or licenses.

As an expedient, we'll use copyright as an example. It is the oldest and clearest example of how intellectual property laws–and the agreements made under those laws–shape the rise and spread of free and open approaches.

Intellectual property law sharply restricts who can access source code and what they can do with it. So, we care aboutlicense agreementst that allow us to work more freely and in the open.

IP comes to software

In a span of not much more than 5 or 6 years roughly from the time of the passage of the Copyright Act of 1976 to the Apple v. Franklin Supreme Court decision in 1983, the creative nature of software development came into its due, legally speaking.

In that same period, the Bayh-Dole Act (1980) opened the door for government contractors to commercialize the results of federally-funded research, setting the stage particularly for universities to establish technology transfer offices. What's more, in 1982, the consent decree breaking up AT&T's Bell System removed restraints established in the 1950s prevented software developed at AT&T being sold as a going concern. These were all huge developments with many implications and effects, but from them all arises the threads of the free and open source software movements.

Computing industry changes from the 1960s, both legally with the invalidation of patents IBM relied upon to dominate the industry, and technically, as the miniaturization most spectacularly exemplified by the Apollo Guidance Computer (AGC) made its way into devices affordable by hobbyists, came to fruition in the late 1970s with the rise of industries around so-called 8-bit computing (not the least of which were today's powerhouses Microsoft and Apple).

The ferment from this commercialization of computing disrupted cloisters of early computing enthusiasts on each coast, at MIT and at Berkeley. These communities worked on the cutting edge hardware of the day, mainframes and minicomputers, rather than the small, new, relatively cheap, but relatively underpowered 8-bit hobbyist computers. Dismay at the creation of an almost Montague versus Capulet divide at the MIT AI lab over who had access to what source code for what computer for what purpose led to the establishment of the GNU project and later the Free Software Foundation, who in turn created and supported the use of the GNU Public Licenses (the GPL licenses) in their various forms. Also at MIT, a number of projects under the Project Athena umbrella were first developed and deployed which are still in use, or whose direct descendents are still in use, including the X11 window system and the Kerberos trusted-third-party authentication (information security) system. These other projects tended to carry a free and open source software license often referred to as the MIT license and were adopted by other universities and research centers.

Meanwhile, at Berkeley, enthusiasts had gotten access to what was at the time illegal-to-commercialize AT&T Unix source code. In ship-of-Theseus fashion the University of California, Berkeley hackers proceded to add and remove parts to it until not much of what came from AT&T was left.

Free and Open Source software emerges

Once the Bell System breakup removed blockers to commercialization for AT&T Unix, AT&T did in fact try to commercialize Unix often by licensing it to workstation manufacturers like HP, IBM, and Sun, leading to the so-called Unix Wars of the 1980s and to lawsuits over what people could do with the code that had passed through Berkeley.

Eventually, all the code AT&T had supplied had been removed from the collection of software the people at Berkeley had been working on, and the resulting operating system, known as the Berkeley Software Distributing (BSD), lives on as FreeBSD, NetBSD, OpenBSD, DragonflyBSD and so on. Like Linux, the BSDs often find use in small embedded systems.

In parallel to the emergence of BSD as free software, other projects, like the GNU Project and the Linux kernel project were independently re-implementing pieces of a Unix-like operating system, released under a family of licenses known as the GNU General Public Licenses (GPLs).

One big thing added to Unix at Berkeley was networking code. Linux, BSD, and other free and open source software grew massively with the rise of computer networking in the mid 1990s due to how flexibly they could be adapted and deployed. One of the very early Internet-age IPOs was that of web-browser maker Netscape. Soon, though, Netscape foundered on the shoals of Microsoft's late but aggressive entry into browsers, leading in turn to anti-trust cases against Microsoft and to Netscape throwing the IP shackles off their browser by announcing the availability of their browsers source code under free software licensing terms. Netscape's move quickly prompted discussion as to how to bring the benefits of software freedom to businesses without ensnaring discussions over the ambiguities in the English word *free*. Christine Peterson came up with the idea of calling software licensed this way "open source software". When the term was introduced in a meeting shortly after Netscape's announcement, it was adopted and began to spread rapidly. The dominant computer companies of today were either founded, or pivoted towards their current networking-dominant forms, over the successive decade.

Myths

Myth: If you find source code on the Internet without a copyright statement or a license, you can use it however you want.

related Myth: You have to register a work for it to be copyrighted.

related Myth: You have to put a copyright statement or symbol on your source code for it to be copyrighted.

Busted: Creative works are not "born free". Copyright adheres to creative works as soon as they are fixed in a "tangible" medium. Computer storage devices count as a tangible medium. So, as soon as you write a creative work, it is *automatically* copyrighted. This idea comes from the Berne Convention for the Protection of Literary and Artistic Works, first established at a meeting in Switzerland in the 19th century. The United States did not join the Berne Convention until a century later, in 1989. Misconceptions about how copyright works in the US might still be circulating from when they were true, prior to 1989, a time still in living memory for some of us.

One only has the freedom to use a work if one has been given a license to use it, either through a contract or a purchase or as a gift from the free and open source software development communities.

One *can* register the copyright on a work, and one can add a copyright statement to a work to make clear to whom the copyright belongs and the date of the copyright. These efforts can improve one's position in the event of a dispute, but they are not necessary for basic copyright restrictions to come into being.

Myth: Open source is "free" which means you can't make money with it.

Busted: The biggest tech companies–scratch that, the biggest companies, full stop–all use free and open source software to make money.

One misunderstanding at the root of this myth is that software is always made to be sold. This is true for some software, but not all of it. A great deal of software written is written for various forms of "in house" use and is never sold, as such, and certainly never as a standalone retail product. Entirely different considerations apply to the production and use of in-house software than for retail software. In house software typically performs functions common to a business sector or to running a business more generally that it is not a competitive differentiator. Rather than being a revenue center, most software lives in cost centers. Minimizing the cost of producing, deploying and maintaining that software come to the forefront. Being able to share the production and maintenance burden with other organizations becomes a huge win for in-house software. This shared effort is widely facilitated by free and open approaches.

Myth: Free and open source software is poor quality.

Busted: Sturgeon's Law–90% of everything is crap–applies. A great many one-off projects, either experiments or toys or abandoned student projects, are increasingly available given how easy and cheap it is to share code. This sort of software always was written, but now more people are writing it, and more of what people write is visible. The advantage of this great diversity is that it makes room for experimental or niche applications that might otherwise never have

been developed, or never have found an audience. Not every piece of software has to be a slickly-produced mass-makret best-seller to have value to someone.

Beyond that, though, are best in class pieces of software like the Linux kernel, the Apache or nginx web servers, a vast array of programming languages and tools, and so much more. Very often the FOSS packages available merely work differently from their non-free alternatives, even if they work just as well, or represent a different set of design trade-offs. These differences may read to someone invested in a non-free tool as deficits even if the free software is cheaper and more flexible to use and deploy. This bias against the free tool becomes more pronounced when the costs of negotiating license agreements for the proprietary software are hidden from or subsidized for the end user (cf, systems of centrally-administered site licenses and educational loss-leader type discounts).