Journal of Information Technology Education: Discussion Cases

INFORMING SCIENCE INSTITUTE

11/28/2013

GRANDON GILL

GROWING CHECKIO[1]

When the company started, its mission was clear: make it the best place in the world for coders. Now we need to decide how to get there.

Liza Avramenko, CEO of *CheckiO*, reflected on the progress of the past few years. The Boston-based company had developed a website designed to help Python programmers improve their skills and create more elegant code through a game-like environment. During that time frame it has signed up more than 20,000 users, about 2,500 of whom were active during a given month, averaging about 3 hours per week on the site. The site was free and had originally been launched mainly as a service by its principal developer and founder, Alexander Lyabah. What made the CheckiO website unique and engaging was the game metaphor upon which it was based. Users solved code-based problems of increasing complexity in order to increase their "level", a concept familiar to any active gamer. The site also offered forums for users to collaborate with each other and to receive expert assistance. Indeed, the inventor of the programming language Python had become a regular visitor and periodically presented awards and critiqued the code developed by community members. Users were also able to present their own challenges to other users. Certain faculty members teaching computer science courses at major university had also started to use the site indirectly—offering extra credit to students that reached a certain proficiency level.

Over the past year, however, a groundswell of academic and commercial interest in CheckiO had developed. As its visibility grew, its founders had applied to the *TechStars* startup accelerator, a high tech business incubator funded by venture capital and other sources, where it had received initial funding of $118,000 in February 2013. That had facilitated the company's move to the Boston area, near the famed *Massachusetts Institute of Technology* (MIT). Of particular interest to high tech players was CheckiO's community of developers ranked by skill level. The IT industry was experiencing a significant shortage of development talent, a situation that was only expected to get worse over time (see Exhibit 1). By engaging with the community, high tech firms anticipated that they might establish relationships with up-and-coming programmers that could ultimately lead to employment—thereby benefiting both parties.

Adding a commercial dimension would involve a number of decisions. What would be the role of corporate participants? What other language communities should be supported? How could the user base be expanded? What funding priorities should be established? How could they engage the academic community in CheckiO's mission? And, most importantly, how could the company retain its unique, supportive culture as its commercial dimension grew more important.

Programming Languages

At the heart of every information system is one or more *computers*, a term that may broadly be used to describe general purpose devices that transform digital inputs (programs and data) into more useful digital forms (sometimes referred to as information). Because these devices—such as the central processing unit (CPU) of a typical mainframe, desktop, laptop, or mobile device—are general purpose by design, the specific processes to be accomplished must be guided by a set of instructions. The process through which these instructions are created is referred to as programming.

Evolution of Programming Code

As a general rule, the instructions created by a developer—referred to as *source code* or just *code*—are far removed from the form in which they are actually used by a particular computer. In fact, over time the nature of the coding process has evolved considerably, going through a series of stages:

1. *Machine Language (1st Generation; 1GL)*: In the beginning days of computing (the late 1940s-early 1950s), programmers wrote code to be used directly by the specific model processor of a particular computer. This type of code is sometimes referred to as *binary* or *executable* code. It can be thought of as consisting of a series of 0s and 1s that are placed in a specific region of the computer's memory that has been designated to hold instructions to the processor.
2. *Assembly Language (2nd Generation; 2GL)*: Because working directly with 0s and 1s was tedious and vulnerable to hard-to-find errors, in the late 1950s it became more common for programmers to use assembly language, which was a mnemonic representation of machine code (e.g., JMP might tell the processor to jump a certain number of positions in the program, CMP might indicate a comparison operation, and so forth). It also allowed storage locations to be given names (referred to as *variables*) instead of physical addresses. When programmers wrote in assembly language, they needed a program called an *assembler* to translate the assembly language to the machine language that could be linked to other code, loaded and executed. Like machine language, assembly language was CPU specific, meaning that code developed for one model CPU would generally not run on another.
3. *Programming Languages (3rd Generation; 3GL)*: As the number of computer models increased and their use became more critical to organizations, inefficiency of programming the way a machine "thinks" became increasingly problematic. In the late 1950s, approaches to writing code that were independent of the system used to run the program were devised. While coding in these languages was still far different from natural language (i.e., the way we speak), they nevertheless represented a major step forward from assembly language. In addition, specific languages could be designed to be suitable for a particular set of tasks. For example, among the early languages COBOL was designed for business data process, FORTRAN for computations, BASIC for instruction, C for portability between machines, and so forth. The programming language counterpart to assembly language's assembler was known as a *compiler*, producing linkable binary code that could be loaded and executed. Although programming languages were not machine specific, their compilers were in the sense that they needed to generate binary code suitable for a particular model or model line of processors.
4. *Later Languages (4th and 5th Generations)*: After the third generation, the "generation" boundaries on later languages are fuzzy and not all that useful. Generally speaking, they tended to differ from "classic" 3GL's in two ways: 1) they were often interpreted rather than compiled, meaning that the source code would be translated and executed step by step using a program known as *interpreter*, rather than be compiled and run directly, 2) they were often embedded in other applications, such as the SQL language used to manage and query relational databases. Where such em-

bedded interpreted languages were specifically designed to automate tasks and were not critical to the function of the host application (such as the use of Microsoft Basic for Office or the use of javascript in web pages), such interpreted languages are commonly referred to as *scripting* languages.

One of the big advantages of compiled versus interpreted languages has always been speed of execution, since interpreted languages necessarily spend considerable time translating the source code while a program is running. As the performance of processors and memory have continued to improve (doubling every 18 months, according to Moore's Law), the relative performance benefits of compiling versus interpreting have declined accordingly. As a consequence, interpreters are becoming commonplace even for formerly compiled languages. Similarly, scripting languages have grown in popularity for the same reason.

Popular Languages

By the time of the case, literally hundreds of different programming languages were in use around the world. Some of these represented evolutionary improvements in existing languages (e.g., C, C++) or attempts to create more elegant versions of existing languages (e.g., Scala, intended to improve upon Java). Others were devised for specific tasks—such as building dynamic web pages—that barely existed a few decades before (e.g., PHP, javascript). Some were tightly tied to an underlying application (e.g., SQL), whereas others were specifically designed to support new computing architectures (e.g., Visual C#.NET).

Examples of some of the more widely used computing languages are presented in Exhibit 2. Generally speaking, once a developer was experienced in a particular programming language, the transition to other languages was relatively straightforward. There were some exceptions to this, however. These included:

- Lower level 3GLs, most notably C and C++, frequently employ variables that hold memory addresses (a.k.a. pointers) in order to make their compiled code more efficient. Programmers not familiar with this type of coding—which is prone to producing serious bugs when coded incorrectly—and the associated need to manage memory can find moving to this style of programming challenging.
- Object-oriented languages encourage a different style of application design and coding than earlier procedural languages such as BASIC, FORTRAN, COBOL, and C. The transition could be difficult for programmers, although this problem became less significant as object-oriented languages came to dominate programming (e.g., all languages in Exhibit 2 except C and Perl are object-oriented).
- Most programming languages (e.g., all those listed in Exhibit 2) require code written in terms of functions and procedures (in addition to objects). Some languages, such as Prolog and Haskell, take a radically different approach, constructing programs as a set of declarations about what is true. Moving in either direction between declarative and procedural programming paradigms can be difficult.

The question of what programming languages are most "popular" is a difficult one to answer. As illustrated in Exhibit 3, there are numerous ways in which the question might be answered—each of which yields a different result. Moving from left to right:

- *TIOBE Index*: Examines web queries and counts references.
- *Book Titles*: Counts titles listed at a major metropolitan bookstore in the U.S. Pacific Northwest.
- *Chat*: Counts references to languages made in *internet relay chat* rooms, a particularly good source of information on what is of interest to students and free-lance professionals.
- *Craigslist:* Based upon languages mentioned in job postings.

The situation is further complicated by the dynamic nature of language popularity, illustrated in Exhibit 4 (based upon the TIOBE Index). As computational capabilities and environments change, preferences for languages change. Moreover, languages with a strong academic following and user community (e.g., Python and Haskell) often generate greater visibility than their commercial use would imply.

The U.S. IT Workforce

Computer programming represented one of the many activities performed by a broader group of individuals characterized as the IT workforce. In addition to programmers, typical IT jobs ranged from help desk (a common entry-level position), through various technical support positions (e.g., database administrator, network administrator, web administrator), through various design positions (e.g., systems analyst, business analyst), all the way up to managers and chief information officers (CIO). What was common to nearly all these job categories were two things:

1. They all required some knowledge of programming, or at least knowledge of programming-like activities. Managing a database, a network or a website nearly always involved some scripting, at a minimum. Furthermore, it was nearly impossible to design a system or manage programmers if you did not know what programming was. Thus, even individuals who would never write code for a living needed some familiarity with the process.
2. Nearly all the IT jobs were experiencing a serious projected shortfall of qualified individuals, as noted earlier in Exhibit 1. In fact, of the 8 job categories a) requiring an undergraduate degree, b) expected to add *at least* 50,000 jobs over the next decade, and c) paying $75,000/year or more 5 were related to the IT workforce (see Exhibit 5).

According to the *U.S. Bureau of Labor Statistics*, in 2010 IT-related professions employed over 8 million people in the U.S. in 2010. These figures might well have been higher, however, had qualified workers been available. Over the previous 15 years, the field had experienced extraordinary volatility. First, in the late 1990s, it was spurred by growth of the Internet and by the need to address Y2K concerns. In the year 2000, however, the growth reversed itself, akin to the bursting of a bubble. First, the two digit formats used to record years in the early days of computing had been expected to cause serious system problems; for the most part, these problems did not materialize. Second, it became generally recognized that the valuations placed on typical Internet startups—most of which had never made money and did not have much, if any, revenue—had been fanciful. IT-related employment dropped dramatically as a result.

During the period after the bubble, there was also a broad perception that most programming jobs would be offshored to low cost locations such as Bangalore, in India. While India-based IT companies such as InfoSys did take on many U.S. clients, it is hard to make a strong case that an overall loss of programming jobs was the result. While declines in employment in this area did take place during the mid-2000s, this was the same period when IT jobs in general were falling due to the factors just mentioned. A 2012 report by the *Congressional Research Service* titled "Offshoring (or Offshore Outsourcing) and Job Loss among U.S. Workers" emphasizes the lack of hard data in this area, pointing to the lack of consensus among experts. The analysis is further complicated by the fact that firms often complained that they could not find the IT/programming talent that they needed. Indeed, the search for such talent often led firms to offshore in the first place.

More recently, the complaints about the lack of available programming talent in the U.S. had grown louder. In part, the problem was one of declines in new talent; according to a survey by *Computer Research Associates* cited in *the Denver Business Journal* in 2012, the number of U.S. computer science graduates had dropped from 21,000 to 10,000 between 2002 and 2010 and had only recently started to rise again.

Another aspect of the need for U.S. talent likely had to do with the fact that the nature of programming itself was transforming. With traditional 3GLs and development processes, the design process was largely complete before programming began. With a fully specified design, it was often feasible to transfer the actual work of programming to group separate from the designers. Hence, the feasibility of offshoring. As programming emphasis increasingly moved towards smaller app and component development, however, the activities of design and programming needed to be much more closely coordinated. An evolving set of development tools facilitated the process of intermingling design and programming, as did a the growth of new approaches to development, referred to as *agile methods*, which emphasized getting out working product as quickly as possible then refining it though a continuing series of redesign cycles. This type of development made large separation between designers and programmers undesirable.

The problem faced by businesses was figuring out how to acquire the local programming talent that they needed. The situation had grown increasingly desperate, with starting salaries often escalating to over $100,000/year when the right skills could be found. And finding the right skills was equally expensive. Search firms charged a fee equivalent to 3-6 months salary for finding suitable candidates. It was not unusual for a single hire to cost $50,000 in these fees. And even at that price, there was no assurance that the right candidate could be found.

CheckiO

The central premise upon which *CheckiO* was based was that performing a series of increasingly complex programming tasks in the context of a supportive community will help members of that community become far more skilled programmers, capable of writing code of increasing quality and elegance.

History

The company was founded by Alex Lyabah, the firm's Chief Technology Officer, in the Ukraine. Originally, it had been conceived as a hobby by Lyabah and a group of his friends and co-workers. After graduating university, Lyabah had started working for a bank and had found the work to be stifling in its routineness. The CheckiO site, initiated in 2011, became a way for him to challenge himself and others with interesting programming tasks in Python. Eventually, he left the bank and began to work as a consultant for companies around the globe. As he did so, the number of participants on the site—consisting of a simple text-based interface that did little more than list challenges—continued to grow, as did his outside programming activities.

In 2012, Lyabah hired Liza Avramenko to act as project manager for CheckiO and various other activities that he was working on. Avramenko was a recent MBA graduate from the *University of South Florida* in Tampa who had returned to the Ukraine after graduation. By that time, CheckiO had acquired over 1,500 registered users. Among her earliest actions was to bring in her friends' designers, with a known track record of successful game implementations, to dramatically upgrade the interface to make it graphical and more game-like (see Exhibit 6).

During 2012, as CheckiO traffic continued to grow, Lyabah and Avramenko sought local investors to fund taking the site to the next level. The reaction of the Ukrainian community was muted, seemingly preferring later stage investments with a clearer route to profitability. During this period, more or less on a whim, the two submitted an application to angel.co where a TechStars mentor found them and invited them to apply to an accelerator, self-described on their website as follows:

> TechStars is the #1 startup accelerator in the world. We're very selective – Although thousands of companies apply each year, we only invest our money and time in about ten companies per

> program location. We have selection rates lower than the Ivy League, so you have to be among the best of the best to earn investment from TechStars.

Fortuitously, the CheckiO was one of the few companies selected for the first level of TechStarts funding: $118,000, of which $18,000 was equity and $100,000 was a convertible debt note—a loan that TechStars could convert to equity if they later chose to do so. In addition to supplying funds, the accelerator provided assistance in moving the company to the Boston area and in making visa arrangements for its founders. In February 2013, the company made its move to a location near MIT in Cambridge, across the river from Boston. By graduation from the TechStars in May 2013, it had nearly 22,000 registered users of whom about 2,500 were active in a given month.

The CheckiO Site and Community

The CheckiO site was based around the Python programming language. The intended audience was not pure beginners—indeed, in order to sign up for an account, a potential new user needed to write a simple loop in Python—but ranged from novices familiar with the language to experts.

The activities on the site were based upon a game metaphor, with "islands" that hosted various challenges. As was common in many video games, users could "level up" by completing programming tasks. Level 4 and below represented novices, 5 and above indicated some level of competence, and double digit levels had to demonstrate true programming skill. Most islands were developed by CheckiO itself. There were, however, some islands with outside sponsors such as O'Reilly (a publisher of technology-related instructional materials) and Github (a web-based hosting solution for large software projects).

The Python functions and programs developed by users ran in a "sandbox" on the website that prevented users from interfering with each other or crashing the system. Python was particularly well-suited for this type of deployment since it was commonly used for scripting language and worked well on web-based deployments. CheckiO provided test data and code necessary to verify that the user's code met the specifications of the task. Eventually, the plan was to allow users to create their own challenge tasks. That feature had yet to be implemented, however.

Central to CheckiO's success was its strong user community. CheckiO users helped each other in completing challenges, provided other users with feedback on the elegance of their code, and answered questions on forums and through chat. One of those users was Guido van Rossum, the inventor of the Python language, who would periodically come online to critique the code developed by members and to act as a judge in programming competitions.

The CheckiO Business Model

One of the most unique aspects of CheckiO was its business model. Typically, websites made money through subscriptions, product sales, or advertising. This was not CheckiO's model. Instead, its plan was to generate revenue from matching users with firms seeking talented programmers.

On the surface, such matching sounded similar to the type of activities conducted by employment websites such as *Monster.com*. CheckiO provided a very different type and level of service, however. There were a number of problems associated with the typical employment agency or website. First, of course, they were highly dependent upon the information listed on the candidate's resume. Assuming that the credentials and skills listed were not an absolute lie—an all-too common situation—there remained a grey area, particularly when it came to programming proficiency. An individual with a single semester course in a particular language might judge him or herself proficient, while actually possessing only sufficient skills to inflict considerable damage on large and complex projects. To address this, firms typically needed to conduct a series of time consuming tests and interviews to screen out weak candidates.

There was also the opposite problem. In some cases, a job candidate might have formidable and transferable skills without having dealt with the specific technology preferred by the organization. As noted earlier, most programming languages involved a large common set of skills—a skilled Java programmer could achieve high levels of proficiency in C# in a matter of weeks, and vice-versa. For most employers, the specifics of an individual's background were much less important than the broader programming skills set. This was particularly true when programmers were in short supply. The problem was that not specifying any criteria in screening candidates led to a pool that was far too large and, on average, too unprepared.

The final issue was particularly serious in the current environment of programmer shortages. Even if a candidate with the "right" skill set could be found and hired, there was no guarantee that he or she would remain with the hiring firm for long. With alternative offers likely to be pouring in, unless the programmer felt a high level of "fit" with the company, his or her job tenure was likely to be short-lived. Such fit depended not only upon the organizational culture, but also upon the nature of the projects that tended to be undertaken by the organization. Some individuals liked to work in a structured environment, writing well specified code; others liked to experiment and felt comfortable designing on the fly. Unless candidates had a clear sense of the portfolio of projects under development at the hiring firm, the risk of poor fit was considerable.

The CheckiO solution addressed both the concerns. Employers were to acquire islands in the CheckiO game world. On those islands, they would present their own set of challenges—ideally, challenges that would mirror the types of programming activities that were typical within their organization. CheckiO users could come to the island to undertake the challenge tasks. Through both their performance on these tasks and, more generally, through the leveling up process, the skill set of the user could be determined. In the meantime, the user—a potential job candidate—would develop a clear view of what working for the organization would be like based upon the nature of the challenges provided. If there appeared to be a fit, the organization could invite the user to apply (or the user could request an application). While this process would not substitute for organization's formal hiring process, it would provide a valuable source of additional information. At least that was the theory, since the company had yet to begin this part of its plan.

Current Situation

By June 2013, the company had received expressions of interest from a number of organizations interested in participating as sponsors and had also heard from faculty members at several universities interested in how the site might be adapted for the purpose of teaching programming. Pursuing these opportunities would not be without cost, however. Avramenko, with the help of TechStars, had already begun to explore acquiring the funding necessary for expansion—a figure just below $1,000,000. In order to be an attractive investment, the company needed to better articulate its direction. That would involve making decisions in a number of areas that included:

- Language support
- Community building
- Pricing
- Defining its relationship with academia

Each of these issues had its own subtleties.

Language Support

The Python language had been a natural choice for the initial CheckiO site. Its popularity was growing on the web, it was open source in origin (eliminating the need to pay licensing fees), and it was well suited to run in a protected sandbox. It was also widely used for instructional purposes, making it a good platform for programming skill acquisition. As suggested by the right-most bar of Exhibit 3, however, it was not one of the most widely used programming languages in industry. For this reason, it was clear that CheckiO needed to expand into alternative languages. Furthermore, a reasonable case could be made for implementing a CheckiO site for nearly any of the languages listed in Exhibit 2 (with the possible exception of the non-object-oriented C and Perl).

This particular decision would be Lyabah's to make, in consultation with the community. His immediate thought was to implement Javascript next, and then Java. Next in the queue, he thought to build a site for a more exotic language—such as Scala or Haskell—that had a growing user community that was eager to develop and polish its skills.

Alternatively, the company might postpone the decision, putting its energies towards continuing to refine the current Python environment, then make the choice of next language supported dependent upon the demand from corporate sponsors. For example, one faculty member noted that many companies had complained of the lack of capable .NET programmers. If an organization was willing to pay extra in order to bring such a site up quickly, would it make sense to change priorities? And what about the costs associated with implementing languages that might involve licensing fees?

Of course, it was not just the organizations that would determine a new language's attractiveness. Part of CheckiO's attraction was its ability to attract users. Certain languages—such as Microsoft's .NET suite—seemed to lack the grassroots enthusiasm generated by open source and emerging languages. To what extent should "grassroots enthusiasm" be factored into the decision? And how could it possibly be measured?

Community Building

When discussing CheckiO's success, Avramenko gave considerable credit to its user community. As a free and open site, users had been eager to provide other users with support and advice. In doing so, they made what would otherwise have been an impossibly large support task manageable. Ensuring that the community's culture remained strong as the site became more commercial was therefore a critical priority.

One way to continue to encourage community spirit was through added functionality. At the top of the CheckiO build list was providing a mechanism through which users could post their own challenge tasks. This was not as straightforward as it might initially sound, since a task needed to include clear and unambiguous instructions/documentation as well as having well designed test code/data—both of these seemed to be areas of weakness for many programmers. Challenge tasks also had to be designed to fit in with the overall theme of the site. If care was not taken, the result could be a rapid degradation of the overall site—causing both lack of coherence and an abundance of bugs. There was no particular reason that these issues could not be addressed. They simply meant that care needed to be taken and a lot of testing conducted.

More broadly, there could be considerable benefits—both in terms of community building and in terms of the evolution of the site—to allowing particularly skilled community members to participate in further developing the code that ran the site itself. This community-based approach had led to some spectacular successes in the open source world, with community-developed/enhanced products such as Apache (web

server), Blender (3D graphics), Linux (operating system), and MySQL (database) rivaling or exceeding the capabilities of competing proprietary commercial products.

To build such a development community, some or all of CheckiO would likely need to be licensed as open source. Open source could be tricky, however. Nearly all licenses required that source code be freely provided in additional to binary (executable) code and that the license agreement be included with any code distribution. Beyond that, licenses came in a variety of flavors. Examples of license terms included:

- *Restrictions on modifications.* Some open source could be freely modified, others could not.
- *Restrictions on patenting.* Some open source licenses allowed the originator to patent the code, others did not.
- *Restrictions on linking to other code.* Some open source did not allow the code to be incorporated into non-open source applications or applications that lacked the same linking restriction. This type of license was sometimes referred to as "copyleft" license, since it tended to force abandonment of the copyright of any code that used it.
- *Restriction on resale.* Open source code was sometimes packaged alone or with other code to be distributed to users who wished to avoid hassle (e.g., RedHat Linux). Some licenses allowed for such repackaging, while others prohibited it.

Some descriptions of key open source terms are provided in Exhibit 7. Exhibit 8 summarizes some of the characteristics of popular open source licenses.

Lyabah saw a distinct tradeoff with respect to making CheckiO completely or partially open source. On the positive side, the greater the percentage of open source, the greater the involvement of the community in enhancing the site. On the negative side, as soon as software became open source, it ceased to be a tangible asset. Generally speaking, companies involved in open source generated revenue through providing support services. Software product companies, on the other hand, could generate revenues through sales and licensing that did not depend on the number of employees who were billing customers. As a result, product companies tended to be more highly valued by investors.

Whatever balance Lyabah decided upon, considerable care needed to be taken in getting the right license. Similar caution would also be necessary that any open source components of the site were not inadvertently exposed to open source code with more aggressive (i.e., copyleft) licenses unless the decision was made to take the entire project open source.

Pricing

If CheckiO's revenue model were to be based primarily around selling "islands" to organizations seeking programmers, how to price those islands would be challenging. There were several issues, in particular, that needed to be considered:

1. Setup costs for a particular island were likely to be relatively high, while ongoing operating costs would be relatively trivial
2. Because the islands were supposed to mirror the type of project typical of the sponsor, there could be considerable variation in the difficulty associated with setting up a particular island.
3. Different organizations were going to have very different hiring needs. How could a pricing policy be established that was fair to both large and smaller organizations?

Broadly speaking there were two pricing philosophies that might be considered. The first was a flat per-island setup fee followed, possibly, by a monthly subscription fee. At the other extreme, a fee might be

levied for each individual hired through a connection made on the site. In many ways, the latter would be a better reflection of the site's value to the customer—recalling the high fees associated with recruiters. It would also provide valuable data on the site's effectiveness that could be used in marketing to other sponsors. On the other hand, it would be highly vulnerable to cheating, with sponsors not reporting (or paying a fee for) hires. Avramenko wondered which approach made the greatest sense, or what—if any—balance was appropriate.

Relationship with Academia

The final area where decisions needed to be made involved the relationship of the site with academic institutions. One professor had already indicate that he had started using the site as a source of extra credit for students—setting a floor of 'C' for the grade of any student who reached Level 5 or higher. In many ways, academics represented attractive potential partners in that:

- They were used to teaching programming and could probably become a source of valuable insights
- They could direct a large body of users to the site. In just the previous two years, many schools had experienced a sudden jump in computer science majors—doubling and even tripling in size since they reached a low ebb 2010. At Harvard, for example, the introductory programming course had just recently surpassed economics as the largest undergraduate course.
- They could become instrumental in adapting the project to new languages and paradigms.
- They were highly motivated, particularly since, over the past decade, many hundreds of millions of grant dollars had been allocated to improving programming instruction (and attracting more women into the field) from sources such as the *National Science Foundation*.

Academic institutions could also become a source of potential revenue, although how this could be implemented would require some thought. As implied by the previously mentioned comment from the professor, the site could be used effectively in conjunction with a programming course. In fact, for some highly motivated students, it could well become a virtual substitute for a course—and a highly effective one at that. Thinking in these terms raised a variety of issues, however:

1. *How could the participation of a particular student be verified?* In some online game environments, individuals from low wage countries had actually been hired to acquire virtual currency for money. It would be easy to imagine a similar situation occurring in which U.S. students paid to have someone else "level them up". What types of approaches could be employed to reduce this problem?
2. *Customizing the environment?* Faculty members often became attached to a particular textbook or self-made exercises. They would be likely to resist any approach to teaching that required them to abandon these. What facilities could be provided to allow faculty members to customize the world to their own specifications?
3. *Creating an "introductory" version?* CheckiO was not intended to be a substitute for elementary programming instruction. It was, however, at the most introductory levels that tools that supported student learning, such as CheckiO, were most needed. How hard would it be to create a version of the tool aimed a true beginners?
4. *Addressing privacy concerns?* Legally, universities were bound to preserve student privacy, particularly as it relates to anything involving grades. Ensuring that the system did not expose users to each other's performance (which might include their individual levels) could require significant reworking of the environment.

5. *Deciding who should pay for the service?* The choice here was students or the institution (although some sites also gave the option of allowing the professor to pay for a class).

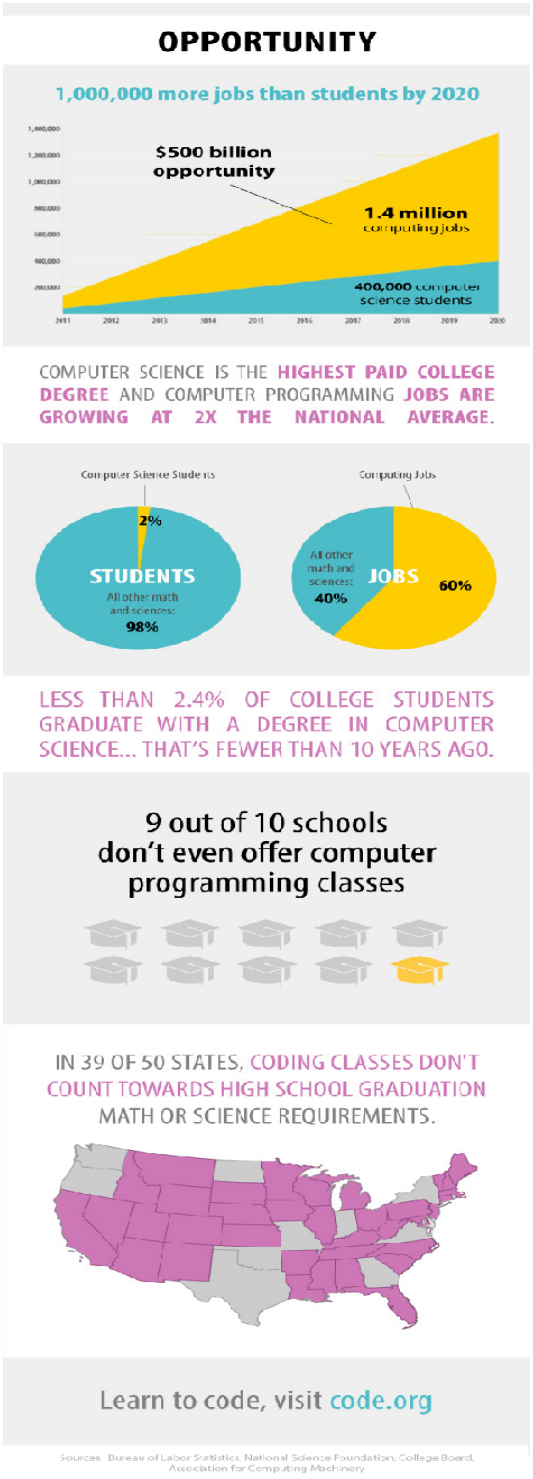
It might also be possible to involve textbook publishers in some way. As a group, the thriving used book market (enabled by global sites such as Amazon.com) had made textbook publishing much less profitable and, to make up for lost sales, the prices of new textbooks had risen to levels that were almost unconscionable (over $200/book for some classes).

Creating linkages with academic institutions could also lead to a share of grant revenues and, more importantly, research findings supporting the value of CheckiO in improving the skills of programmers. Care would need to be taken in fashioning such relationships, however, since many grants and projects had effect of muddying the waters on intellectual property rights (much like open source projects).

The Next Step

In order to create a credible business model to ensure the success of the next round of financing, Avramenko and Lyabah felt that they needed greater clarity on at least some of these issues. The most immediate question was which needed to be addressed right away and which were better postponed until the resource picture was clarified.

Exhibit 1: Projected Demand for Computer Science and IT Specialists



Source: Code.org website

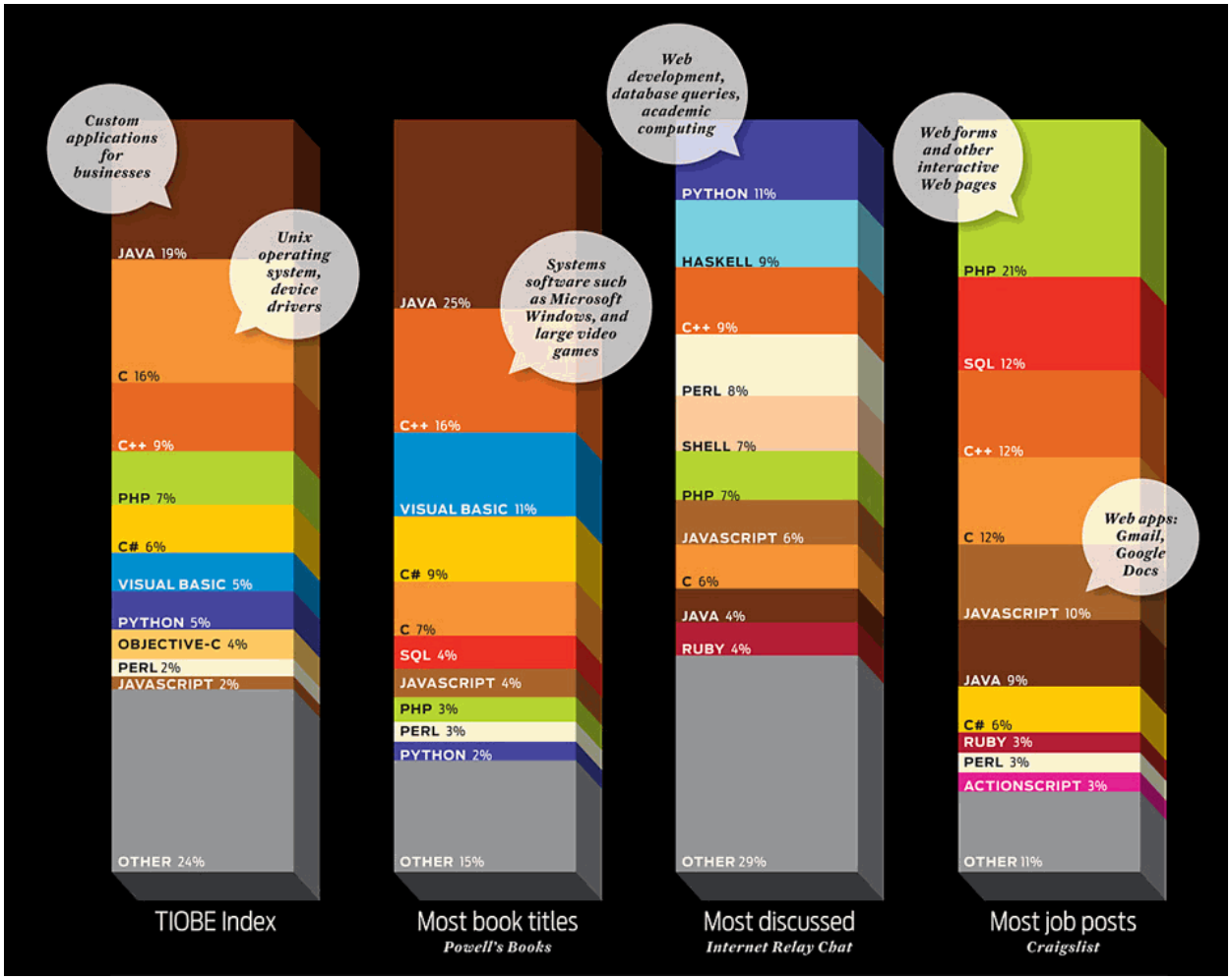
Exhibit 2: Common Programming Languages

Language	Type and Platform	Comments
C	Compiled. Compilers exist for most platforms and operating systems.	Originally developed by AT&T in early 1970s as a portable language that produced highly efficient machine code. Used mainly in operating system and low level system applications demanding high performance.
C++	Compiled. Compilers exist for most platforms and operating systems.	Object-oriented version of C that became popular in the 1980s. Used primarily for high performance applications such as games and in support of legacy applications.
C#	Compiled to interpreted intermediate language on Windows platforms. The open source Mono platform supports	(Pronounced C-sharp). Object-oriented language similar to C++ and Java introduced around the year 2000 to support Microsoft's new .NET framework intended to integrate Windows desktop and web-based programming. Has become the preferred language for .NET programming, particularly popular among small and medium enterprises.
Java	Compiled to run on a virtual machine supported by a wide range of platforms, from set-top cable boxes to mainframes.	Developed by Sun and introduced in the 1990s, Java's virtual machine architecture was intended to support "write-once/run anywhere", although in its early days it was also characterized as "write once/debug everywhere". Commonly used in web-based applications and in large enterprise applications.
Javascript	Scripting language supported by nearly all web browsers	Also originally developed by Sun, Javascript is quite different from its namesake. The principle language used to create dynamic web pages on the client side (i.e., it is run by the user's browser, not the server). Its functionality is intentionally limited (e.g., highly restricted access to memory and the user's file system) to prevent hackers from taking control of a user's system. Advanced libraries, such as JQuery, have dramatically increased its capabilities.
Objective C	Compiled language supported by iPhone and iPad platform	Apple-specific object-oriented language similar to Java and C# made popular by the ubiquity of the iPhone.
Perl	Interpreted language that runs on web servers	Designed to perform operations on web servers, has largely been supplanted by PHP.
PHP	Interpreted language that runs on web servers	Fully functional object-oriented language that is designed to run under a web server. Provides strong text processing and database access functionality.
Python	Scripting language, with interpreter and compiler support	Flexible object-oriented language that is often used as the basis for scripting support in other applications. Particularly popular in the educational community, where it frequently used in introductory programming courses.

Language	Type and Platform	Comments
Ruby	Interpreted and scripting	Open source object-oriented language principally used as a server-side language, often combined with a web-based environment referred to as “Ruby-on-Rails”
Visual Basic	Compiled to interpreted intermediate language on Windows platforms	A close analog to C# when used in .NET environment, with only minor differences in syntax. Earlier versions (6.0 and before) were intended primarily for building Windows desktop applications and were not compatible with .NET versions.

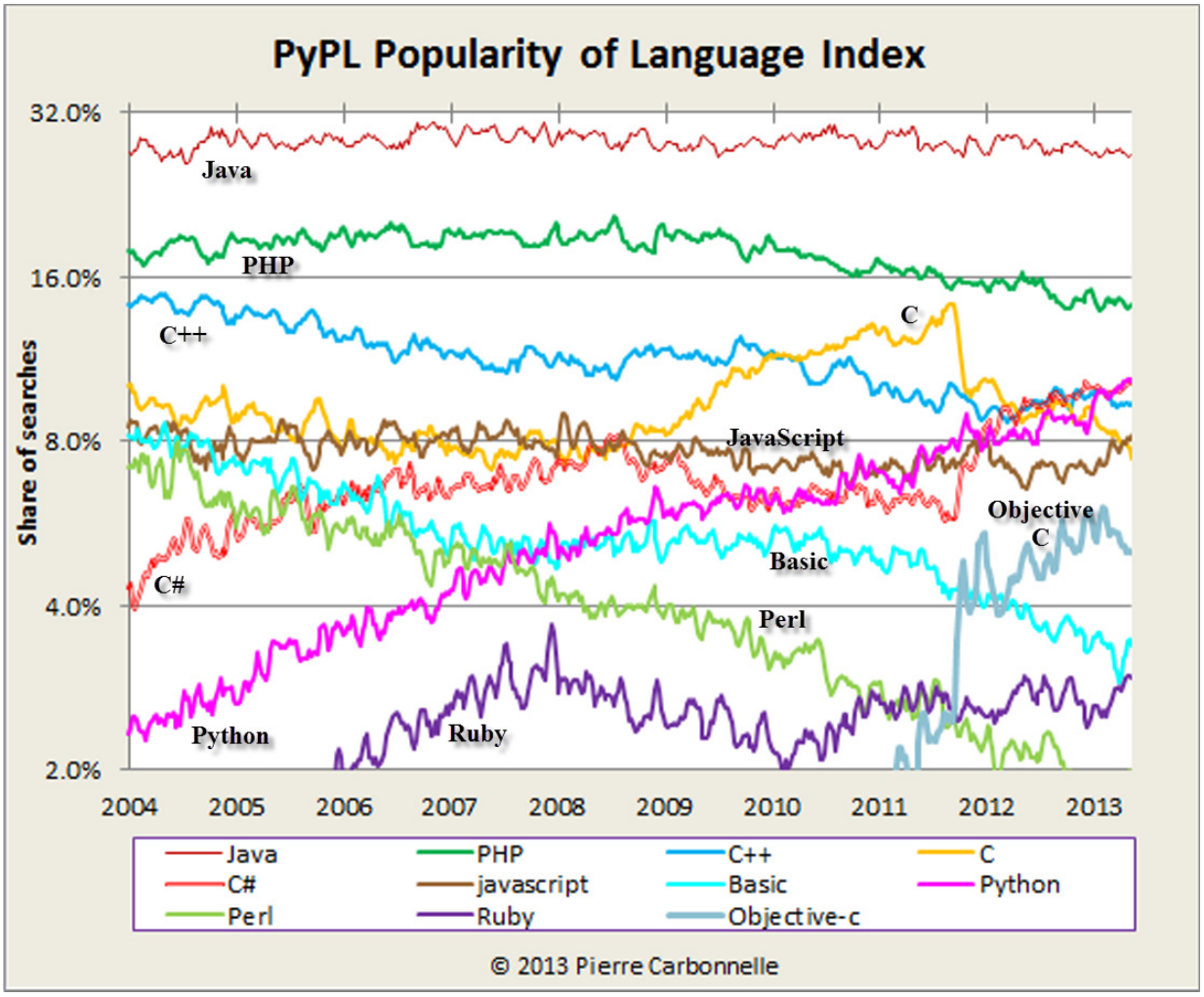
Source: Compiled by case writer

Exhibit 3: Popularity of Programming Languages



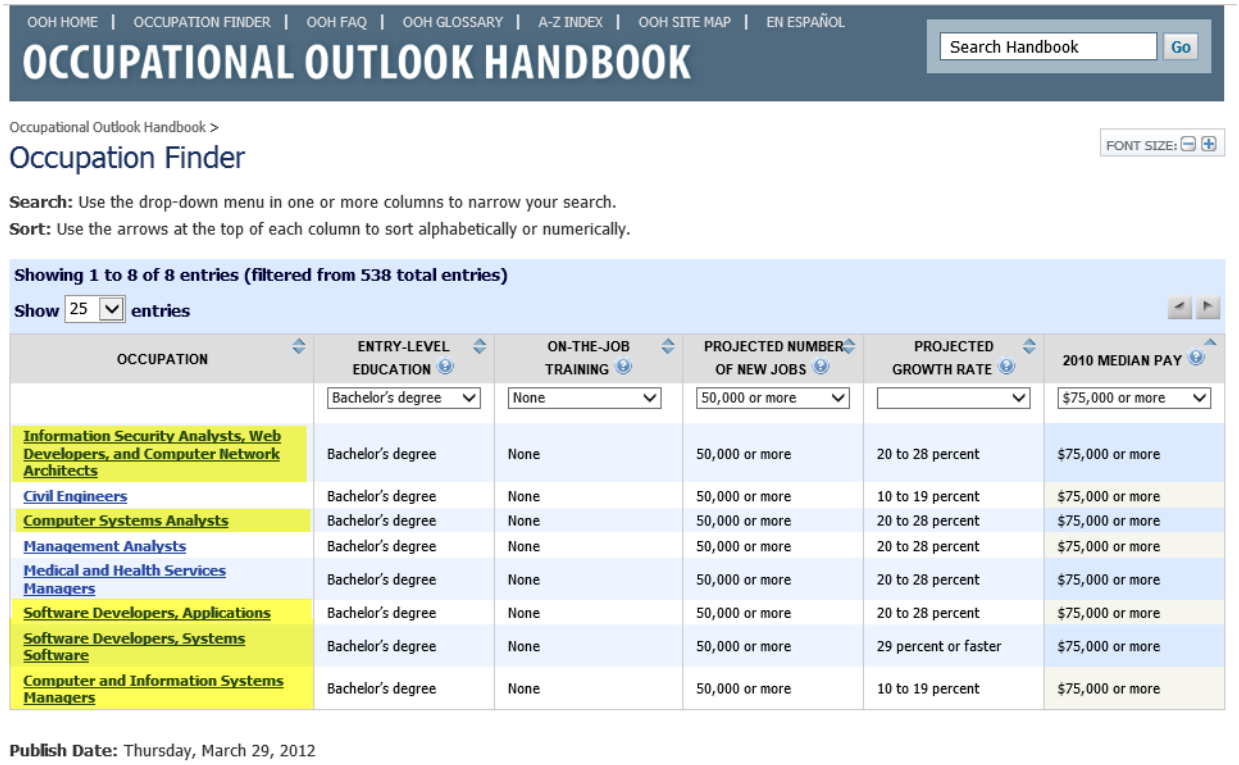
Source: http://spectrum.ieee.org/at-work/tech-careers/the-top-10-programming-languages

Exhibit 4: Language Popularity Over Time



Source: https://sites.google.com/site/pydatalog/pypl/PyPL-PopularitY-of-Programming-Language

Exhibit 5: Occupational Outlook



OOH HOME | OCCUPATION FINDER | OOH FAQ | OOH GLOSSARY | A-Z INDEX | OOH SITE MAP | EN ESPAÑOL

OCCUPATIONAL OUTLOOK HANDBOOK

Search Handbook Go

Occupational Outlook Handbook >

Occupation Finder

FONT SIZE:

Search: Use the drop-down menu in one or more columns to narrow your search.
Sort: Use the arrows at the top of each column to sort alphabetically or numerically.

Showing 1 to 8 of 8 entries (filtered from 538 total entries)

Show 25 **entries**

OCCUPATION	ENTRY-LEVEL EDUCATION	ON-THE-JOB TRAINING	PROJECTED NUMBER OF NEW JOBS	PROJECTED GROWTH RATE	2010 MEDIAN PAY
	Bachelor's degree	None	50,000 or more		$75,000 or more
Information Security Analysts, Web Developers, and Computer Network Architects	Bachelor's degree	None	50,000 or more	20 to 28 percent	$75,000 or more
Civil Engineers	Bachelor's degree	None	50,000 or more	10 to 19 percent	$75,000 or more
Computer Systems Analysts	Bachelor's degree	None	50,000 or more	20 to 28 percent	$75,000 or more
Management Analysts	Bachelor's degree	None	50,000 or more	20 to 28 percent	$75,000 or more
Medical and Health Services Managers	Bachelor's degree	None	50,000 or more	20 to 28 percent	$75,000 or more
Software Developers, Applications	Bachelor's degree	None	50,000 or more	20 to 28 percent	$75,000 or more
Software Developers, Systems Software	Bachelor's degree	None	50,000 or more	29 percent or faster	$75,000 or more
Computer and Information Systems Managers	Bachelor's degree	None	50,000 or more	10 to 19 percent	$75,000 or more

Publish Date: Thursday, March 29, 2012

Source: http://www.bls.gov/ooh/occupation-finder.htm?pay=&education=Bachelor%E2%80%99s+degree&training=&newjobs=50%2C000+or+more&growth=29+percent+or+faster&submit=GO

Exhibit 6: Example CheckiO Welcome Screens

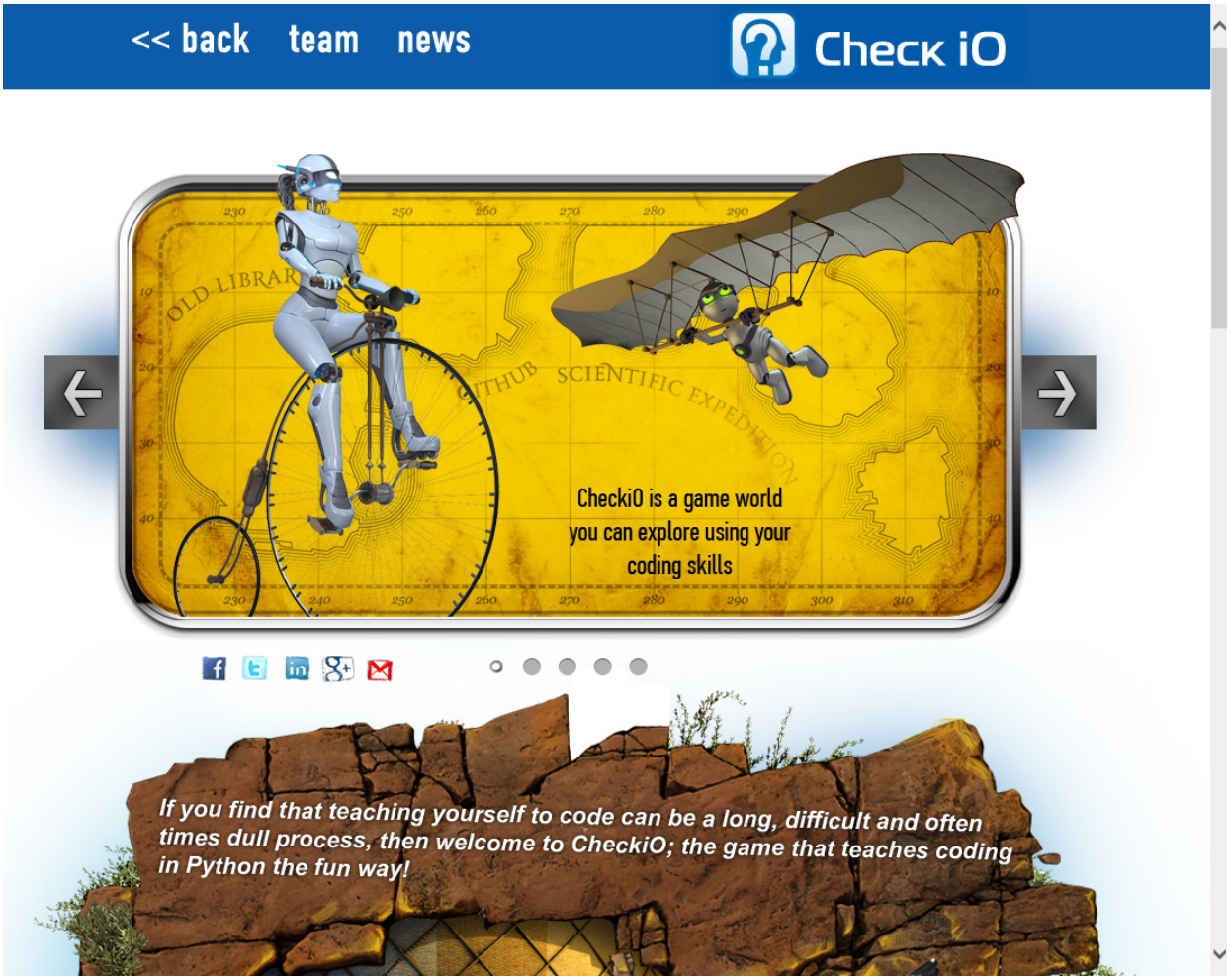


Exhibit 7: Open Source Licensing

Basics of Open Source

Can Open Source software be used for commercial purposes?

Absolutely. All Open Source software can be used for commercial purpose; the Open Source Definition guarantees this. You can even sell Open Source software.

However, note that *commercial* is not the same as *proprietary*. If you receive software under an Open Source license, you can always use that software for commercial purposes, but that doesn't always mean you can place further restrictions on people who receive the software from you. In particular, so-called copyleft-style Open Source licenses require that when you distribute the software, you do so under the same license you received it under.

Can I restrict how people use an Open Source licensed program?

No. The freedom to use the program for any purpose is part of the Open Source Definition. Open source licenses do not discriminate against fields of endeavor. Note that nearly all Open Source licenses also state that there is no warranty: you can't sue if it blows up your computer or destroys your data, even if it was the program's fault. (Some companies may sell you a warranty separately, for a fee, but that is not part of the open source license, it's just your private contract with that company.)

Can I stop "evil people" from using my program?

No. The Open Source Definition specifies that Open Source licenses may not discriminate against persons or groups. Giving everyone freedom means giving evil people freedom, too. Fortunately, there are other laws that constrain the behavior of evil people.

What is "free software" and is it the same as "open source"?

"Free software" and "open source software" are two terms for the same thing: software released under licenses that guarantee a certain, specific set of freedoms.

The term "free software" is older, and is reflected in the name of the Free Software Foundation (FSF), an organization founded in 1985 to protect and promote free software. The term "open source" was coined in 1998 by a group of people — the founders of the Open Source Initiative (OSI) — who also supported the development and distribution of free software, but who disagreed with the FSF about how to promote it, and who felt that software freedom was primarily a practical matter rather than an ideological one (see for example the entry *"How is `open source' related to `free software'?"*from the OSI's original 1998 FAQ page).

Following the coining of the term "open source", some of those who adopted it did so because they too had philosophical differences with the FSF about the reasons *why* to promote such software, while others who adopted the term did so because of differences of opinion with the FSF about tactically *how* to support such software, even while sharing an ideological motivation. These two groups can and do overlap, of course, and some people use both terms, choosing according to context and audience.

One of the tactical concerns most often cited by adopters of the term "open source" was the ambiguity of the English word "free", which can refer either to freedom or to mere monetary price; this ambiguity was also given by the OSI founders as a reason to prefer the new term (see "What Does `free' Mean, Anyway?", and similar language on the marketing for hackerspage, both from the original 1998 web site).

Furthermore, the FSF uses a shorter, four-point definition of software freedom when evaluating licenses, while the OSI uses a longer, ten-point definition. The two definitions lead to the same result in practice, but use superficially different language to get there.

This history has led to occasional confusion about the relationship between the two terms. Sometimes people mistakenly assume that users of the term "open source" do not intend to communicate a philosophical point of view via that term, even though some (but not all) speakers actually do use it that way. Another common mistake is to think that "free software" refers only to software licensed under copyleft licenses, since that is how the FSF typically releases software, while "open source" refers to software released under so-called permissive (i.e., non-copyleft) licenses. In fact, *both* terms refer to software released under *both* kinds of license.

Neither term binds exclusively to one set of associations or another, however; it is always question of context and intended audience. When you sense a potential misunderstanding, you may wish to reassure your audience that the terms are essentially interchangeable, except when being used specifically to discuss the history or connotations of the terminological difference itself. Some people also prefer to use the term "free and open source software" for this reason.

See also our history page for more information about the history and usage of the term "open source".

What is "copyleft"? Is it the same as "open source"?

"Copyleft" refers to licenses that allow derivative works but require them to use the same license as the original work. For example, if you write some software and release it under the GNU General Public License (a widely-used copyleft license), and then someone else modifies that software and distributes their modified version, the modified version must be licensed under the GNU GPL too — including any new code written specifically to go into the modified version. Both the original and the new work are Open Source; the copyleft license simply ensures that property is perpetuated to all downstream derivatives. (There is at least one copyleft license, the Affero GPL, that even requires you to offer the source code, under the AGPL, to anyone to whom you make the software's functionality available as a network service — however, most copyleft licenses activate their share-and-share-alike requirement on distribution of a copy of the software itself. You should read the license to understand its requirements for source code distribution.)

Most copyleft licenses are Open Source, but not all Open Source licenses are copyleft. When an Open Source license is *not*copyleft, that means software released under that license can be used as part of programs distributed under other licenses, including proprietary (non-open-source) licenses. For example, the BSD license is a non-copyleft Open Source license. Such licenses are usually called either "non-copyleft" or "permissive" open source licenses

Copyleft provisions apply only to actual derivatives, that is, cases where an existing copylefted work was modified. Merely distributing a copyleft work alongside a non-copyleft work does not cause the latter to fall under the copyleft terms.

For more information, look at the text of the specific copyleft license you're thinking of using, or see the Wikipedia entry on copyleft. C.f. "permissive" licensing.

What is a "permissive" Open Source license?

A "permissive" license is simply a non-copyleft open source license — one that guarantees the freedoms to use, modify, and redistribute, but that permits proprietary derivative works. See the copyleft entry for more information.

Is <*SOME PROGRAM*> Open Source?

Only if it uses one of the approved licenses, and releases appropriate software.

Can I call my program "Open Source" even if I don't use an approved license?

> Please don't do that. If you call it "Open Source" without using an approved license, you will confuse people. This is not merely a theoretical concern — we have seen this confusion happen in the past, and it's part of the reason we have a formallicense approval process. See also our page on license proliferation for why this is a problem.

Is <*SOME LICENSE*> an Open Source license, even if it is not listed on your web site?

> In general, no. We run all licenses through an approval process to provide an accepted standard on which licenses are Open Source, and we list the approved ones. Be dubious of claimed Open Source-ness for licenses that haven't gone through the process. See also the license proliferation page for why this matters so much.

What about software in the "public domain"? Is that Open Source?

> For most practical purposes, it is — sort of. This is a complicated question, so please read on.
>
> *"Public domain"* is a technical term in copyright law that refers to works not under copyright — either because they were never in copyright to begin with (for example, works authored by U.S. government employees, on government time and as part of their job, are automatically in the public domain), or because their copyright term has finally lapsed and they have "fallen into" the public domain.
>
> Not all jurisdictions have a public domain, and it doesn't always mean exactly the same thing in the jurisdictions that do have it. Furthermore, even where it is clear what it means, it's still not a license. To be subject to a license, a work must still be in copyright. That means there is no way for the "public domain", as a concept, to go through the OSI evaluation and approval process. We wouldn't be evaluating a license text. Instead, we would have to somehow evaluate the laws themselves, in different jurisdictions, and say which jurisdictions have a public domain that meets the Open Source Definition and does not create problems for software authors and users. This would be very difficult, because it would mean evaluating not just the statutes but various bodies of case law (for example, open source licenses usually have a strong disclaimer of liability for the copyright holder — but we don't know how or whether the author would be protected from liability for software released into the public domain in various jurisdictions). This approach would not be useful to the OSI's mission, because open source is an international phenomenon and we only want to approve licenses that meet the Open Source Definition everywhere.
>
> Thus we recommend that you always apply an approved Open Source license to software you are releasing, rather than try to waive copyright altogether. Using a clear, recognized Open Source license actually makes it *easier* for others to know that your software meets the Open Source Definition. It also enables the protection of attribution, and various other non-restrictive rights, that cannot be reliably enforced when there is no license.
>
> There are certain circumstances, such as with U.S. government works as described above, where it is not easy to apply a license, and the software must be released into the public domain. In these cases, while it would be inaccurate to display the OSI logo or say that the license is OSI-approved (since there is no license), nevertheless we think it is accurate to say that such software is effectively open source, or open source for most practical purposes, even though it is not officially released under an open source license. (This is assuming, of course, that in the laws of releasing jurisdiction the meaning of "public domain" is compatible with the Open Source Definition.) After all, the freedoms guaranteed by open source licenses are still present, and it is possible for the familiar dynamics of open source collaboration to arise around the software.
>
> For a detailed discussion of the complexities of the public domain and open source, search for the words "public domain" and "PD" in the subject headers of
> the January 2012, February 2012, and March 2012 archives of the OSI License Review mailing

> list. And if the thought of reading all those conversations is daunting, please take that as more evidence that it's just better to use an approved Open Source License if you can!

Source: http://opensource.org/faq

Exhibit 8: Examples of Open Source License Requirements

Comparison of the Open Source Licences

The bullets mark if the the licence explicitly states the item in question. Implicit items are not marked by this chart

	Must distribute license with binray or source	Cannot use contributors name to endorse	There has to be a notification for changed files	Any change must distributed in source form	Lets you provide warrenty if you want to, normally no	Lets you explicitly charge for providing warrenty or gurantee or transfer of code	All derivative work must be under the same license	Must show License when Run from command line	Non derivative works can have different license	May exclude countries where there is a contradiction with patent in that ocuntry	Must describe any deviation due to regulation
Apache License 2.0	●	●	●	●	●						
Common Development and Distribution License	●		●				●				
GNU General Public License (GPL)	●		●	●		●	●	●	●	●	
GNU Library General Public License (LGPL)	●		●	●		●			●	●	
Microsoft Public License (Ms-PL)	●	●									
Microsoft Reciprocal License (Ms-RL)	●	●							●		
Mozilla Public License 1.1 (MPL)	●		●	●							●
New BSD License	●	●									
The MIT License	●										

Source: http://programmers.stackexchange.com/questions/105344/is-there-a-chart-for-helping-me-decide-between-open-source-licenses