Non-Native English Speakers Learning Computer Programming: Barriers, Desires, and Design Opportunities

Philip J. Guo
UC San Diego
La Jolla, CA, USA
pg@ucsd.edu

ABSTRACT

People from nearly every country are now learning computer programming, yet the majority of programming languages, libraries, documentation, and instructional materials are in English. What barriers do non-native English speakers face when learning from English-based resources? What desires do they have for improving instructional materials? We investigate these questions by deploying a survey to a programming education website and analyzing 840 responses spanning 86 countries and 74 native languages. We found that non-native English speakers faced barriers with reading instructional materials, technical communication, reading and writing code, and simultaneously learning English and programming. They wanted instructional materials to use simplified English without culturally-specific slang, to use more visuals and multimedia, to use more culturally-agnostic code examples, and to embed inline dictionaries. Programming also motivated some to learn English better and helped clarify logical thinking about natural languages. Based on these findings, we recommend learner-centered design improvements to programming-related instructional resources and tools to make them more accessible to people around the world.

Author Keywords

non-native English speakers; learning programming

ACM Classification Keywords

K.3.2 Computers and Education: Computer and Information Science Education – *Literacy*

INTRODUCTION

English is the de facto international language of science, technology, and commerce [16, 40, 77]. Becoming digitally literate and professionally competitive in today's global workforce often involves learning technical skills from English-based resources. Yet 95% of the world's population does *not* have English as their native language [2], so it is important to discover barriers faced by non-native English speakers when trying to acquire technological expertise in complex domains. This knowledge can help us to design more inclusive learning technologies that further broaden access to digital literacy and job opportunities for people around the world.

```
con = sqlite3.connect(":memory:")
con.execute("CREATE TABLE person ... <omitted for space>")
with con:
    con.execute("INSERT INTO person ... ", ("Joe",))
# con.rollback() is called after the with block finishes
# with exception, exception still raised and must be caught
try:
    with con:
        con.execute("INSERT INTO person ... ", ("Joe",))
except sqlite3.IntegrityError:
    print("couldn't add Joe twice")
```

Figure 1. English is ubiquitous in source code, as shown in this example adapted from the official Python language docs for SQL database management [55]. English appears in comments, variable names, sqlite3 standard library API identifiers, and in both Python and SQL keywords.

In this paper, we focus on an exemplar form of technical training that is now in high demand: computer programming. Millions of people from over 190 countries are now learning programming online [11, 76]. Yet the most popular programming languages are all designed in English [39]. Their official documentation pages and code examples (e.g., Figure 1) are written in English, their ecosystems of libraries and accompanying API docs are English-based [51], and the most popular textbooks, MOOCs [11, 76], and discussion sites (e.g., Stack Overflow [8]) are primarily in English. Although translations exist, people often perceive them to be of lower quality and less convenient than the original English versions [16, 38]. English is also the official language of many international open-source software communities, as manifested in mailing list conventions and coding style guidelines [6, 39, 54].

What barriers do non-native English speakers face when learning programming from English-based resources? What desires do they have for improving instructional materials? How can we redesign programming resources and tools to be more universally accessible to learners from all language backgrounds? We investigated these questions by deploying a survey to a widely-used programming education website (`pythontutor.com`) and analyzing 840 responses from people spanning 86 countries and 74 native languages. To our knowledge, this is the most diverse population so far for studying the impact of human language on learning programming.

CHI 2018, April 21–26, 2018, Montreal, QC, Canada

ACM 978-1-4503-5620-6/18/04...$15.00
DOI: https://doi.org/10.1145/3173574.3173970

Our study found that non-native English speakers faced barriers with reading instructional materials, technical communication (listening and speaking), reading and writing code, and simultaneously learning English and programming. They wanted instructional materials to use simplified English without culturally-specific slang, to use more visuals and multimedia, to use more code examples that are culturally-agnostic, and to incorporate inline dictionaries. Finally, some reported that programming actually served as a motivating context for them to learn English better and helped clarify their logical thinking about natural languages.

Based on these findings, we adopt a learner-centered design [37] approach to improving programming instructional resources and tools. We suggest ideas such as bilingual pair programming, internationalizing code examples, IDE plugins to help users understand and write better identifier names in code, and browser+IDE extensions to provide contextually-relevant English learning exercises within one's programming workflow. These ideas work toward a form of *universal design* [44]—improving instructional materials for non-native English speakers may actually benefit *all* learners.

This paper takes steps toward a future where the optimistic promise of *Computer Science For All* [53] further broadens its scope to include people around the world from all human language backgrounds. Its main contributions are:

- Barriers and desires of non-native English speakers learning programming, obtained from 840 survey responses spanning 86 countries and 74 native languages.
- A learner-centered design of resources and tools to make programming more accessible to people around the world.

BACKGROUND AND RELATED WORK

English is frequently the language of instruction in university classrooms worldwide, despite many students not growing up with it as their native language [50, 56]. Reasons for pervasive English use in higher education include canonical textbooks often being in English [56], English serving as a common working language when students from up to dozens of native language backgrounds (e.g., in India) attend the same university [56], and students wanting to prepare for jobs at English-speaking multinational organizations [16, 40, 56, 77]. In addition, the majority of MOOCs (Massive Open Online Courses) are in English [3]. Even when versions are available in their native language, some students still report choosing to take English MOOCs because they want to improve their English fluency and perceived career prospects, and to prepare to move to English-speaking countries [75].

Non-native English speakers face well-known challenges in English-language classrooms when learning a wide array of topics including math, science, engineering, medicine, and the humanities [50, 56]. Barriers range from cognitive to affective to social: Needing to mentally translate concepts into one's native language—especially in real time while listening to a lecture—increases extraneous cognitive load [49, 71] and decreases comprehension. The difficulty of formulating verbal questions, along with anxiety about lack of English fluency, makes them less likely to ask clarifying questions [63]. Fear of social ostracism from classmates leads to additional affective barriers [56, 63]. Our study's findings confirm that some of these barriers also exist both in online settings and with respect to learning computer programming.

Beyond the general ESL (English as a Second Language) barriers described above, programming adds an extra layer of complexity since, as Figure 1 shows, English is deeply intertwined into programming language keywords and API naming conventions [16, 38, 39]. Coding style guidelines mandate writing not only comments but also all identifiers (e.g., variable, function, and class names) in English [6, 39, 54]. Non-native speakers report struggling to decipher the meanings of code identifiers, especially when they are abbreviated (e.g., the C function `getch()` stands for "get character") [16].

There have been attempts to create programming languages with non-English keywords [80], but so far none have been widely adopted [77]. Attempts to translate APIs and error messages have faced a similar lack of adoption since programmers cannot as easily search for online help using the localized terms [16, 38]. One promising line of work in this direction is to localize keywords and UI components in blocks-based languages for novices such as Scratch [1] and App Inventor [81]. Dasgupta and Mako Hill analyzed usage logs from five countries and found that learners who used a version of Scratch in their native language could more rapidly build more complex programs (a proxy for progress in learning) than those who used the default English version [19].

To our knowledge, there have been very few studies on the impact of human language on learning programming. As described above, Dasgupta and Mako Hill found that Scratch learners who used localized versions in five mostly-European languages (German, Italian, Norwegian, Portuguese, Brazilian Portuguese) learned faster than those who used the default English version of Scratch [19]. Pal and Iyer translated the audio portions of English lecture [57] and screencast videos [58, 59] into Hindi to present to Indian university students; they found that Hindi students who did not grow up speaking English appeared to learn best from Hindi-dubbed screencast videos. Our work contributes to this nascent literature by bringing in a far broader sample of 74 native languages from 86 countries. Also, instead of quantitatively measuring learning gains like prior studies, our study instead focuses on obtaining qualitative perceptions from learners.

Our work also contributes to the literature on barriers faced by novice programmers. Foundational studies in computing education showed that novices face significant barriers when trying to form mental models of a *notional machine*—a working representation of how the computer operates on code and data [24, 69]. Soloway and Spohrer provide a broader discussion of such novice programmer barriers as observed in classroom settings [67]. From an HCI perspective, Green's cognitive dimensions framework casts programming languages and environments as user interfaces and applies human factors principles to frame barriers [31]. Pane and Myers also viewed programming languages as user interfaces and classified dozens of novice usability barriers [60] based on heuristic evaluation [52]. Many barriers arose because colloquial En-

glish interpretations of keywords (e.g., "and", "or", "while") did not match their formal semantics in programming languages [62, 72]. Myers and Stylos applied a similar methodology to find usability problems in APIs, which uncovered many badly-named classes and methods in popular APIs [51]. Ko et al. observed novices learning VB.NET and distilled six kinds of learning barriers [41] ranging from not knowing what they wanted the computer to do all the way to not understanding the computer's output. Our study extends this rich lineage of barriers research by focusing on lack of English language fluency as a barrier for aspiring programmers.

Lastly, our work contributes to the HCI and computing education literature on *broadening participation in computing* [29], which strives to bring programming to more diverse populations. Representative initiatives here include increasing participation amongst female [5, 12, 66] and underrepresented minority [20, 21, 27, 45] students at both the K-12 and college levels. Beyond traditional school settings, researchers have sought to engage adult learners such as graphic and web designers [22], coding bootcamp attendees [73], aspiring computer science teachers [28], technology company employees from non-computing backgrounds [15, 79], and older adults [35]. This paper extends the scope of broadening participation in computing to potentially include the 95% of the world's population who are not native English speakers [2].

METHODOLOGY: INTERNATIONAL ONLINE SURVEY

For this study, we wanted to reach a broad global population of learners, so we deployed an online survey to the programming education website `pythontutor.com` (Python Tutor [33]). Learners use this website to practice writing code and to visually debug their errors using a step-by-step run-time data structure visualizer. Despite its legacy name, Python Tutor visualizes popular languages such as Python, Java, C, C++, Ruby, and JavaScript. It has been operating for the past eight years and has had over 3.5 million total users.

We deployed our survey to the Python Tutor website because it is a free and widely-used resource for learners who come to the site from a diverse variety of online learning channels such as: Massive Open Online Courses (MOOCs) by all three major providers (Coursera, edX, Udacity), Khan Academy, Codecademy, Stack Overflow, digital textbooks, and many coding tutorials, blogs, and discussion forums. Thus, even though a survey on any single website will reach only a limited population, deploying to Python Tutor likely reaches a broader population than say, deploying to a single MOOC, since users come to Python Tutor from a wide array of referring websites. To our knowledge, Python Tutor has been used as a site to deploy surveys to international learner populations for two prior (but unrelated) HCI studies [23, 35], so we adopt a similar methodology for embedding our survey into the site.

Although we do not consider demographic factors other than language background for this study, the Python Tutor user base is diverse in terms of age and geography: 13% of users are younger than 18 years old, 29% are 18–24 years old, 20% are 25–34, 12% are 35–44, 11% are 45–54, 8% are 55–64, and 7% are 65 or older. Users come from over 180 countries. 25% are female, which is slightly higher than the average percentage of female CS majors at U.S. universities (∼18%) [7].

The Survey Instrument

We added this sentence to the bottom of the Python Tutor website with a link to a survey on Google Forms: "If you would like to help our research on how one's native language affects learning programming, please fill out this survey [link URL]." We made the survey link legible but unobtrusive so as not to unnecessarily distract learners on the site. Participation was voluntary; we did not pay survey respondents. We opened this survey in Dec 2016 and closed it in August 2017.

We made our survey as short as possible to collect respondents' language backgrounds and their self-reflections about barriers and desires. It contained eight questions:

1. What is the first language that you learned to speak?
2. What language(s) are you most comfortable speaking now?
3. In what language(s) do you read programming-related instructional materials (e.g., websites, books, forums)?
4. How would you rate your fluency in reading English? {Beginner, Intermediate, Proficient, Fully Fluent}
5. If you *are* a native English speaker, what difficulties (if any) have you noticed when watching people who are not native English speakers learn programming? In what ways (if any) have you helped them out?
6. If you are *not* a native English speaker, in what ways (if any) has your level of English fluency made it harder for you to learn programming? How have you overcome those challenges?
7. If you are *not* a native English speaker, how can instructional materials for programming (e.g., websites, books, forums) be improved to help you learn better?
8. If you are *not* a native English speaker, in what ways (if any) has your level of English fluency improved by learning or reading about programming?

Q1 and Q2 gauge one's native and conversational spoken languages, respectively. Q3 and Q4 gauge self-reported reading literacy. To give respondents maximum flexibility, all questions except for Q4 were open-ended, and all were optional.

Data Overview and Analysis

We received 840 responses that contained at least one non-blank, non-spam response for any of Questions 5–8. Out of 184 self-reported native English speakers, 97% answered Q5. Out of 656 non-native English speakers, 59% answered Q6, 71% answered Q7, and 73% answered Q8.

To standardize the names of languages reported for Questions 1–3, we used the international ISO 639-2 standard for classifying human languages [9] and also canonicalized dialects accordingly. For instance, both Mandarin and Cantonese were canonicalized to "Chinese," all variants of English (e.g., Australian, British, U.S.) were canonicalized to "English," and endonyms were canonicalized to their ISO 639-2 names (e.g., Kodava→Kannada, Farsi→Persian).

Native language (Question 1)		Most comfortable (Question 2)		Read materials in (Question 3)	
English	22%	English	37%	English	96%
Spanish	9%	Spanish	8%	Chinese	3%
Chinese	8%	Chinese	7%	French	2%
French	7%	French	6%	Russian	2%
Russian	6%	Russian	5%	German	1%
Portuguese	5%	Portuguese	5%	Portuguese	1%
German	5%	German	5%	Spanish	1%
Hindi	4%	Hindi	3%	Polish	0.5%
Dutch	3%	Dutch	3%	Dutch	0.5%
Italian	2%	Italian	3%	Italian	0.5%

Table 1. Percent of respondents (N=840) that reported top 10 most common languages for Questions 1–3: native language, the one they are most comfortable speaking, and the one in which they read programming-related instructional materials, respectively. Columns can sum to more than 100% since respondents can report more than one language.

To classify open-ended Questions 5–8, the research team (professor + research assistant) iteratively developed a set of codes based on an inductive analysis approach [18]. The team read all responses together, incrementally wrote up emergent themes in a bottom-up inductive manner, then grouped those into the smallest set of distinct categories related to possible effects of English on learning programming. Once we finalized the categories, we made another full pass through the responses together to assign one or more labels to each. 14 respondents wrote non-English responses (6 Portuguese, 5 Chinese, 1 Afrikaans, 1 French, 1 Russian); we pre-processed those using Google Translate (see Limitations subsection).

Study Design Limitations

The goal of our survey was to capture a broad overview of barriers faced by non-native English speakers learning programming, along with their desires for improving instructional materials. Ideally we would be able to uniformly sample from *all people* around the world who are learning programming. But since we deployed this survey to a U.S.-made English website (Python Tutor)—albeit one that has been used in over 180 countries and is linked from many non-English educational websites—our respondents will still likely have higher levels of English literacy than the general population. Thus, our study captures the sentiments of those who learn more from English-language resources, which is consistent with prior reports of how English is often the primary language of instruction for programming and other technical topics [50, 77]. However, it may not fully reflect challenges faced by people with very low levels of English literacy or those who exclusively use non-English resources. Also, since we rely on self-reported survey data, we may be missing latent barriers that people fail to self-report.

Our study focuses only on respondents' language backgrounds (Q1–3) but does not consider the complex intertwined roles [17] of culture, socioeconomic status, education levels, age, gender, race, ethnicity, and other factors. We also do not unpack the possible effects of specific natural languages or language families; we simply split respondents into native and non-native English speakers based on Q1.

Native English speakers		Non-native English speakers	
United States	42%	United States	16%
United Kingdom	9%	India	7%
India	6%	France	5%
Canada	5%	Brazil	5%
Germany	4%	Germany	4%
Australia	3%	Canada	3%
Singapore	3%	United Kingdom	3%
South Korea	2%	Ukraine	3%
Kenya	2%	Netherlands	3%
Japan	1%	China	3%

Table 2. Percent of respondents from the top 10 most common countries, as determined by geolocating their IP addresses. This table is split by whether they are native or non-native English speakers, as determined by Question 1. In total, respondents came from 86 different countries.

We ran Google Translate on 1.6% of responses. Machine translation has errors, so that may add ~1.6% more error to our hand-classified category counts. Note that all quotes used in this paper were originally in English and not translated.

Finally, our respondent population likely comes from the more autodidactic, technology-literate, self-motivated, and self-reflective end of the general population [61], since those people are more willing to take the initiative to pursue online learning options. Thus, our findings may not reflect the challenges faced by those who are solely taking in-person courses.

LANGUAGE BACKGROUNDS OF RESPONDENTS

Before detailing the barriers and desires reported by survey respondents, we first summarize their language backgrounds. Respondents came from a diverse array of backgrounds, collectively mentioning 74 total languages in Questions 1–3, which span languages from all six populated continents.

Table 1 summarizes the top 10 most frequently-reported languages for Questions 1–3. Only 22% of respondents (184 out of 840) were native English speakers. Of the remaining 78% of non-native English speakers, 3% self-rated their level of English reading fluency as Beginner, 14% as Intermediate, 44% as Proficient, and 39% as Fully Fluent (Question 4). 37% of respondents are now most comfortable speaking English, and 7% reported being most comfortable speaking multiple languages (i.e., are multilingual). The three most common multilingual responses were English+Hindi (0.7%), English+French (0.6%), and English+German (0.6%).

Almost all respondents (96%) read programming-related instructional materials in English. However, 8% reported reading in more than one language, which is why the rightmost column of Table 1 sums to more than 100%. The three most common multilingual responses here were English+French (1%), English+Chinese (1%), and English+Russian (1%).

Respondents also came from a diverse array of locations around the world. Using MaxMind GeoLite2 [46] to geolocate their IP addresses, we found that respondents came from 86 total countries. Table 2 shows location distributions for native and non-native English speakers, who came from 40 and 82 different countries, respectively. As expected, relatively

Learning Barrier	NN†	Native
Problems w/ Reading Instructional Materials	35%	29%
Problems with Technical Communication	24%	49%
Problems with Reading Code	16%	38%
Problems with Writing Code	11%	28%
Hard to Learn English+Programming	17%	8%

Table 3. The five kinds of barriers that non-native English speakers faced when learning programming, reported both by non-native (NN†) speakers (Question 6, N=384) and by native English speakers observing non-native peers (Question 5, N=178). Percentages add up to more than 100% since each response can mention more than one kind of barrier.

more native English speakers are located in the United States and in other primarily-English-speaking countries.

The purpose of this section is to establish the language and location diversity of our respondent population. The barriers and desires we present in the following sections were reported by people from diverse backgrounds—74 native languages from 86 countries—in contrast to prior studies that saw only up to five languages [19, 57, 58, 59]. Even though most respondents (78%) were non-native English speakers, they self-reported relatively high levels of English fluency and English materials usage, which is unsurprising since the survey was administered in English on an American website. However, despite our sample likely being more English-literate than their non-English-speaking peers, respondents *still* reported many learning barriers due to being non-native speakers.

WHAT BARRIERS DO NON-NATIVE ENGLISH SPEAKERS FACE WHEN LEARNING PROGRAMMING?

We discovered barriers faced by non-native English speakers via two complementary survey questions: a) directly asking non-native English speakers what language-related challenges they faced when learning programming (Question 6) and b) asking *native* English speakers what difficulties they observed their non-native counterparts facing (Question 5). Table 3 summarizes the five major classes of barriers.

Problems with Reading Instructional Materials

The most commonly-reported barrier (34% of all responses) was problems reading English instructional materials such as textbooks, online tutorials, discussion forums, and API docs.

First off, non-native speakers often mentioned how they had to rely on English instructional materials since it was hard to find good materials in their native language. A native Spanish speaker summarized this sentiment: *"Almost all good code documentation is in English. There aren't many options in choosing the human language for a programming language. English is the official language of all programs."* Similarly, a Portuguese speaker wrote: *"There are not many materials about programming or even not many computer-related material in portuguese. The ones we can find usually are very out of date."* And an Arabic speaker wrote: *"Most of the resources I use and the ones that are available are in English (online courses, videos, notes, books, etc.). Options in Arabic would be quite limited and not as good (translated books, university classes if you're a computer sciences student,..)"*

Even when native-language materials are available, they are often suboptimal translations. e.g.,: *"I find it harder to read [programming] books and such in Romanian because they sometimes translate the terms and I get confused."* A Polish speaker mentioned how, even though they normally browse the web in Polish, they still rely on English for programming materials: *"Honestly, I would never leave the Polish internet and learn much, but everything [about programming] is pretty outdated, covers only the basics or is funky translated."*

Respondents reported the usual barriers with reading in a non-native language, such as slower reading comprehension. For example, a native Chinese speaker wrote, *"It makes me to learn less effectively. The language of programming teaching materials I learn is English, so I have to spend more time in reading, understanding and listening lectures [sic]."*

Technical jargon associated with programming likely made comprehension more challenging than casual prose reading. A French speaker wrote: *"The technical vocabulary linked to programming can be complicated to assimilate, especially in the middle of explanatory sentences if you don't know the equivalent word in your native language. Sometime a single technical word can blur the meaning of an entire sentence."*

Native English speakers made similar observations about what challenges they saw their non-native peers facing. Some mentioned that colloquial language may make text harder to read: *"Biggest problem seems to be in understanding specifications for software. When writing them in English, we [native speakers] often miss that non-native English speakers won't understand nuance or less formal descriptions."*

Problems with Technical Communication

The next most common kind of barrier was technical communication via listening and speaking. As a representative quote in this category, a native Czech speaker mentioned the challenges of listening comprehension: *"I can't imagine the exact meanings of words used in programming in my mind so it takes a lot more time to understand if it's explained in English."* A native English speaker echoed those sentiments: *"The thing that makes it difficult to help them is that my own explanations have to be spoken in a way that a non-native speaker can process them. (They must be shorter, spoken more slowly, focus on strict subject-verb-object grammar.)"*

Native English speakers also mentioned how a lack of comfort with speaking can impede non-native peers, e.g.,: *"The only real difficulty I have seen [amongst non-native English speakers] is that it is more difficult for them to TALK about programming, which in turn makes it more difficult to learn."*

Finally, non-native communication problems also extend to "speaking" with machines, not just with humans. A German speaker described the challenges of putting their thoughts into English to enter into Google: *"Especially difficult I'd say is finding the right way to enter your questions into Google to find the answer you're looking for when you're probing a special problem. Not only do I need to be able to describe the problem, I'll also have to guess the most common way to express the problem in English and, if that doesn't work, find some mostly synonymic ways to describe the problem."*

Problems with Reading Code

In addition to problems reading prose, non-native English speakers also reported trouble understanding source code. A common root cause is that programming language keywords (such as "while") are in English. A Portuguese speaker wrote: *"The mainly [sic] challenge for me was understanding exactly what each word means for the code and what it doesn't mean (for example: the translation of "while" in portuguese has more meanings than what is used in a code)."* More generally, programming jargon for similar concepts are often synonyms that are hard to differentiate. For instance, a native Spanish speaker described: *"It is difficult for me to understand the differences between objects, classes and instances."*

Beyond built-in keywords, programmers often use abbreviations and idiomatic naming conventions for identifiers (e.g., function, class, and variable names), which can be hard for non-native speakers to understand. A native English speaker reported: *"For example variable names, we name them the way we do for a reason, for example if a non native English speaker saw num_list they might not figure that it's a list of numbers."* Another added, *"Some [non-native speakers] do not intuitively understand the syntax of some functions or methods because they do not understand what exactly the word means."* In addition, libraries use English naming conventions: *"Libraries are written using english words. i imagine there is a world out there with libraries in other languages but i haven't encountered it yet."* For instance, Apple's Cocoa guidelines for library API naming [6] recommends long function names that resemble abbreviated sentences (e.g., `(void)setAcceptsGlyphInfo:(BOOL)flag`), which may be harder for non-native speakers to decipher.

Internationalization and localization issues also frustrate non-native speakers when reading English-centric source code. A native Slovak speaker summarized these challenges:

> *"For example almost in every course within few first lesson are stated that string can be compared (in dictionary order), but in my primary language string/words correctly ordered acidofil, agat, achat differ from stated point and even worst string achat is in my language only 4 characters long! and there many other false assumptions even if I thinking in terms that words can contains only A..Za..z0..9 which is another lie for my language. So it is hard to follow when premises are 'wrong'."*

Problems with Writing Code

Native English speakers also reported how some non-native speakers had trouble writing comprehensible code. For instance, they mentioned how their non-native peers sometimes wrote unclear code due to unusual variable naming choices: *"Often their choice of names are not very specific or don't match my intuition."* Another summarized how naming conventions are where writing code turns into writing natural language, which can be harder for non-native speakers:

> *"I've seen folks struggle with the naming of variables – opting for simple names like 'a' or meaningless (or worse misleading names) like 'table'. Otherwise the folks I've seen who are not native English speakers but are coders tend to do quite well with the structures and grammar of coding. It is where the natural human language meets the code that there are issues... :-)"*

Similarly, code that prints text outputs and error messages are other parts of the code base where non-native speakers must write English prose. A native speaker observed the following issues from their non-native peers: *"I've noticed errors mainly in string output in terms of grammar and spelling."*

Native English speakers also had trouble comprehending code written by non-native peers due to them using non-English words in identifier names and comments. e.g.,: *"If the programmer is bilingual, I've seen him/her comment their code in more than one language"*

Non-native English speakers also mentioned similar barriers, albeit less frequently. For instance, some pointed out the difficulty of choosing good code identifier names: *"Definitely the naming. When you write code, actually you are trying to understand the concepts and choosing appropriate words to express."* One mentioned how a lack of English fluency may make them less productive by missing out on the most relevant library functions: *"By not knowing some words, you might miss some easy pre-programmed functions."*

Hard to Simultaneously Learn English and Programming

Since non-native speakers must often rely on English-language instructional materials, some reported trouble with learning enough English to comprehend those materials while simultaneously learning the given programming concepts. For instance, a native Arabic speaker wrote: *"It's really hard I expect 90% will quit because I have to learn English and coding at the same time, and it's disappointing because I'm making slow progress in both topics."* A Bulgarian speaker wrote: *"I started learning to program without knowing English. In order to progress I had to learn both in parallel."* And a Zulu speaker wrote: *"My low level of English fluency badly affected my learning of a programming language at the beginning. I had to put an extra effort to improve my English in order to comfortably learn programming languages."*

WHAT HELPS NON-NATIVE ENGLISH SPEAKERS OVERCOME THESE BARRIERS?

In addition to identifying barriers, Question 5 also asked native English speakers about how they helped their non-native peers overcome barriers, and Question 6 similarly asked non-native English speakers how they tried overcoming barriers themselves. Of all the respondents who answered Questions 5 and 6, respectively, 25% of native speakers mentioned ways to overcome barriers, and 58% of non-native speakers mentioned it. Here we summarize the most common responses.

How Native English Speakers Have Tried to Help

Native English speakers who are bilingual can serve as translators for their peers. For instance, one wrote: *"I live in Germany now, and the biggest problem is not knowing instantly what concept the keywords connect to (such as 'if' or 'while loop', etc). I've had to translate a lot into German for them."*

Use Simplified English without Culture-Specific Slang	36%
Use More Visuals and Multimedia	23%
Use More Examples that are Culturally-Agnostic	21%
Incorporate Inline Dictionaries	21%
Lost in Translation: Prefer English Over Translations	7%

Table 4. The five kinds of desires for improving programming-related instructional materials, as reported by non-native English speakers (Question 7, N=464). Percentages add up to more than 100% since each response can mention more than one kind of desire.

Some also recalled having to more clearly define programming-related jargon for non-native speakers: *"Mainly, it's understanding the English-based shorthanded words that I've noticed that can be a little difficult for non-native English speakers. It's important as a teacher to frontload the key words and language before teaching a concept."* Another mentioned non-native English speakers *"not knowing what some of the more obscure words used (like concatenate) mean in regular English so they don't know how to translate that knowledge to programming. Usually a description or replacement word fixes it."*

More generally, some brought up the importance of empathy and patience when teaching non-native English speakers: *"Just need to accept they need simpler steps and will take longer because they first need to convert to native language in head and then understand programming concepts"*

How Non-Native English Speakers Overcome Barriers

Many non-native speakers mentioned the critical importance of improving their own English skills as the main way to overcome their current learning barriers. A native Portuguese speaker wrote: *"The main words used in programming don't have translation to my language, so it is absolutely necessary to have at least intermadiate [sic] english, otherwise you will have many problems trying to understand the way programming works and what words are used."*

People also used dictionaries, translators, or Google searches to translate unfamiliar terms. A German speaker wrote: *"I sometimes need to translate specific words when they are the name of a function and I didn't know them already like for example the word 'append'. I just googled the meaning in my native language and was right back on course."*

WHAT DO NON-NATIVE ENGLISH SPEAKERS DESIRE IN PROGRAMMING INSTRUCTIONAL MATERIALS?

464 out of 656 non-native English speakers in our respondent population (71%) answered Question 7, which asked them how instructional materials for programming could be improved to help them learn better. Table 4 summarizes the five kinds of learner desires for improving instructional materials.

Use Simplified English without Culturally-Specific Slang

The most common desire was for instructional materials to use simplified English: smaller words, simpler sentences, fewer colloquialisms, and less culturally-specific references. For instance, a Hindi speaker was confused by the term "nickel and dimed" when learning algorithms: *"Some blogs have cultural references which are hard to understand. For e.g. I had to look up the meaning of a nickel, dime etc. while reading about dynamic programming on a blog."* Others mentioned avoiding slang in general: *"Try to avoid local expressions only meaningful in your area of the world."*

For programming-specific terms, some suggested defining concepts more clearly upfront, especially because the meanings of English words in a programming context may differ from their everyday usage. A native Zulu speaker suggested *"explaining the important terms that have a different meaning when used in a computer programming field than in ordinary English."* A Spanish speaker wrote: *"It's essential to put a big emphasis in the meaning of the most basic concepts. The concept, for example, of what means an 'assignment statement' could be more intuitive to an English speaker, but this doesn't necessarily translate into the rest of speakers."*

Others proposed user testing to discover points of confusion in materials: *"Test the books with non native english speaker in order to detect possible overcomplicated sentences."* Another respondent suggested: *"Field-test them to discover expressions that are too cryptic for non-native speakers."*

Use More Visuals and Multimedia

Respondents wanted instructional materials to have more visual imagery and multimedia (e.g., videos) rather than plain text, presumably because visuals can more easily transcend languages. An Arabic speaker observed: *"Figures and illustrations are easier to be understood than written english."* A French speaker wanted to *"include drawings, sketches, and if there is some dynamics, include an animation in blogs/websites."* A native Hindi speaker summarized this kind of desire: *"Diagrams & interactive tutorials beat all text."*

Some preferred watching videos over reading. A Chinese speaker wrote: *"Video helps me the most for learning programming because i can see what is happening step by step, and Video is usually simplified. Books are slower because usually it is thick, i feel i have to take ages to finish it."*

Use More Examples that are Culturally-Agnostic

Non-native English speakers also wanted to see more code examples. A native Nepali speaker elegantly summarized this desire: *"Less text, more examples."*

Just like in their requests for simplified English, they wanted to have those examples be as culturally-agnostic as possible. A native Chinese speaker wrote: *"Please don't assume all the cultural and political jokes and celebrity stories used in the programming examples can be understood by all non-native speakers."* It is also important for the target learner population to personally relate to the examples, as mentioned by a Kannada speaker: *"Use examples that are familiar to that audience. Eg - I know nothing (and am least interested in) what happens in the United States of America. Would have helped if we had something we could relate to (across subjects)."*

Finally, respondents wanted code examples to be aware of internationalization and localization issues: *"Offering more examples on how to deal with the kind of issues that arise when you are programming an application that is not in English (character encoding, need to support multiple languages)."*

Incorporate Inline Dictionaries

Even though most respondents used English-language resources (Table 1), non-native speakers wanted lightweight dictionaries to help them learn unfamiliar terms while reading the original English source materials. A native Polish speaker mentioned an analogy to e-book readers: *"For myself the option to translate words 'inline' from the text (without opening a separate browser window or application) I have in my ebook reader is giving me a lot: so enriched, context relevant content to the original is IMHO a great accelerator."* A Spanish speaker similarly suggested *"maybe by including a section on some kind of 'etymology of the terms', or a section on history of the term. That has proven very helpful in other disciplines, like medicine."* An Albanian speaker mentioned captioning: *"I am thinking of something similar to subtitles for movies: one can always make use of them and compare translated concepts against the original English source."*

Lost in Translation: Prefer English Text Over Translations

The final class of responses was not direct desires but rather was how non-native speakers in our respondent population did *not* want translated versions of English materials. For example, a Korean speaker wrote: *"I actually searched blogs and information in both Korean and English, but I felt translated Korean version of all the explanations were more confusing since syntax is in English anyway."* Another added: *"Because the syntax and terminologies are written in English, it's too much complication for one to learn it in [our] own language, because lot of phrases are misinterpreted."*

Other non-native speakers mentioned how English is already the de facto official language of programming, so it is better not to fragment the ecosystem with too many translations:

> *"It would be a massive waste of resources if every non-native english speaking programmer would start to translate documentation, let alone keep it up to date. Since the dawn of the current power of open-source, english – whether someone likes it or not – has become our lingua franca to describe programming languages and tools. That fosters the ability of the programming community to communicate across borders. Likewise with science, we have to find a common ground and pool resources instead of disparaging them in linguistic balcanization [sic]."* (native German speaker)

A native French speaker also corroborated these sentiments: *"For now, English seems to be universally accessible for an international user/programmer. I believe it's more practical for everyone to be able to share their work and their knowledge in one language that almost everyone understands."*

CAN PROGRAMMING HELP PEOPLE LEARN ENGLISH?

Our final survey question was more exploratory and probed the inverse of our main research questions: Instead of asking about how English skills affect non-native speakers learning programming, we wanted to find out whether learning programming might actually affect their English skills. Table 5 summarizes the responses to Question 8, which asked non-native speakers in what ways (if any) has their level of English fluency improved by learning or reading about programming.

Did not respond to question	27%
Responded but reported no noticeable improvement	14%
Programming provides context for improving English	34%
Programming clarifies logical thinking about language	14%
Reported English improved but gave no explanation	11%

Table 5. Did non-native English speakers feel that learning programming helped improve their level of English fluency? (Question 8, N=656)

Programming Provides Context for Improving English

The most common kind of affirmative response was that programming provided a concrete context for people to improve their English skills. A native Polish speaker summarized this sentiment: *"Well, the desire to learn programming was the driving force behind my desire to learn English."*

Programming was a forcing function to motivate people to improve their English reading skills out of necessity. A native Italian speaker wrote: *"Reading about programming has helped my level of English fluency in the same way any other reading has helped me. Learning programming has helped me to learn English as a side effect – because it 'forced' me to read English books/websites."*

Since English is the lingua franca of software engineering, it also provides a social context for practicing speaking. A native Punjabi speaker wrote: *"Programming provided an environment in which it is expected to be good in English. So, not only programming related books improved my english but also conversing with other people, clients, managers, interviewers etc on a regular basis helped in improving fluency."*

Not only did respondents mention how programming forced them to read and speak more in English, but some also reported how it forced them to improve their English writing since they needed to ask technical questions on discussion forums. A Spanish speaker reported their English improving *"A LOT, I mean this week I just improve what I haven't in years... the [discussion forums] are great for this because in order to get a more direct help you are forced (in the good way) to write in english whatever doubt you have (I mean is the fastest way to get an answer) None of my english classes has made me improve what I have improved here."* A Portuguese speaker reported similar benefits from writing online: *"the exposition to discussion groups, wikis and tech communities has pretty importance on this improvement [sic]."*

Programming Clarifies Logical Thinking About Language

Aside from motivating people to improve their English reading, writing, and speaking, some non-native speakers also reported how they felt programming makes them think more logically, which might also contribute to improving their English skills. A Central Khmer speaker summarized this idea: *"[Programming] helps me think the logic of what I'm trying to say, which means it helps speak or think in English less ambiguous."* A Russian speaker mentioned that programming can *"shed new light on the logical processes that go on in statements and functions, applying this to spoken/written texts helps analyze them more thoroughly and therefore allows for a better fluency in the language itself."*

DISCUSSION

In light of our findings, how can we redesign programming instructional resources and tools to better serve the 95% of the world's population who are not native English speakers?

We theoretically ground our idea-generation process using the *learner-centered design* methodology proposed by Guzdial et al. [37], which posits that we must use the learner's own motivations and frustrations as the starting points for design. In a similar spirit as user-centered design, Guzdial advocates performing targeted studies on specific learner populations and then using empirical findings—rather than instructor intuitions—as the basis for formulating design suggestions. In our case, survey respondents reported their frustrations (Table 3) and desires (Table 4), so we use those as the basis for following the seven parts of the proposed framework [37]. Note that some parts are not as directly relevant to our study findings but we still include them for completeness.

1. Understand where learners are starting from and what they want to do: Although our study did not explicitly investigate motivations for learning, based on responses from both native and non-native speakers regarding technical communication barriers in workplace settings, one class of learners we revealed is starting from not having English as their native language and wanting to acquire enough proficiency in both English and programming to obtain gainful employment. They might aspire to join a *community of practice* [43] of professionals who use English as the lingua franca of technical communication in both in-person and online contexts (e.g., Stack Overflow, IRC, mailing lists). Thus, our proposed design ideas all operate under this utilitarian assumption.

2. Understand where learners have trouble: Table 3 summarizes the barriers our respondents faced. For improving technical communication skills (reported by 31% of all respondents as a barrier), one idea is to match non-native and native English speakers to do *pair programming* in a class or MOOC, thereby forcing non-native speakers to speak and listen in an immersive one-on-one setting with rapid feedback. Pair programming is known to be effective for technical mentorship [47, 48], but to our knowledge, people have not explored its potential for also improving ESL communication. One challenge here is finding enough native English speakers to serve as programming+language tutors. A potential solution is to adopt affordances from real-time extensions to Python Tutor [34, 36], which have shown that people from different countries are able and willing to remotely pair up.

3. Scaffolding: How can we extend traditional forms of instructional scaffolding to help non-native English speakers with both code reading and writing (reported by 23% and 17% overall, respectively)? One idea is to add bilingual labels to worked examples [74], which could help learners see terminology in both English and their native language. This design meshes with our respondents' reported desires for more examples and for inline dictionaries (both 21% in Table 4).

To help learners comprehend class, function, and variable names in a code base, an IDE plugin could parse identifier names and heuristically decipher their possible meanings based on common English abbreviations and idiomatic conventions (e.g., deciphering `getch()` [16]). To help people *write* better-named identifiers in their own code, a similar IDE-based tool could analyze code to suggest better names based on what their run-time values are used for. This idea was loosely inspired by using *variable roles* [10, 13] to talk about code semantics in introductory programming materials.

4. Use terminology that learners understand: Many respondents reported trouble understanding English idioms, slang, and cultural references when reading both instructional materials and code examples. They requested for both prose and code to use simplified English without culturally-specific slang (Table 4). This is not merely an issue with reading comprehension alone; seeing culturally-unfamiliar materials might also negatively impact learners' sense of belonging and make people from different cultures feel like "programming is not for me" since it seems so Anglocentric. To remedy, creators of instructional materials could be inspired by projects like Simple English Wikipedia [4] that use a subset of the English language with simpler grammar and vocabulary.

5. Contextualization: Learning resources are more compelling when put into the context of domains that learners personally relate to and care about [37]. Thus, even if programming instructional materials still remain in English, it would be helpful for code examples to demonstrate localized conventions that some survey respondents have suggested, such as showing instances of more robust Unicode string handling and locale-specific currency, date, and number formats.

Looking at contextualization from another angle, our findings from Table 5 showed that programming provides a motivating context for some people to learn English better due in part to possible future job opportunities [40, 77, 75]. Prior HCI projects have used instant messaging [14], movie watching [42], and mobile apps [25, 26] as real-world contexts for learning human languages. Following in this lineage of work, how can we use learners' interests in programming to give them opportunities to simultaneously improve their English skills? One idea along these lines is to create a web browser extension that analyzes the text of existing API docs, technical blog posts, and online tutorials. The tool can then inject contextually-relevant English comprehension exercises into webpages so that ESL readers can perform active learning [30] while reading online. This extension could also be linked to an IDE so that it can pop up in-IDE English exercises relevant to the user's current code and documentation context while their code is compiling or tests are running.

6. Learners change as they acquire expertise and
7. Acknowledge differences among learners: For this study, we considered all non-native English speakers together as a group, but in reality there is enormous variation in human languages along with even more nuances in their underlying cultures. Furthermore, we did not control for learner expertise levels. Thus, it is important not to overgeneralize these design ideas or to implicitly claim that there is some sort of optimal design for everyone around the world. These ideas should be viewed as starting points to provide impetus for future work.

Toward Universal Design of Programming Technologies

Many of the aforementioned ideas work toward a form of *universal design* [44] where thinking about a particular user population (in this case, non-native English speakers) can result in technologies that help everyone. For instance, everyone can benefit from IDE tools to help them comprehend and create better identifier names in code; using simplified English helps even native speakers since they do not need to worry about local dialects or regional idioms; and internationalizing code examples helps everyone learn to write more robust code. For our proposed ideas that explicitly deal with ESL issues, we could "flip the language bit" and use those ideas to instead help native English speakers learn a foreign language within the context of their normal programming activities.

Generality of Findings and Implications for Future Work

Some reported barriers about reading code are not specific to non-native English speakers. Prior studies found that even learners who are proficient in English have trouble disambiguating between colloquial definitions of words (e.g., "or", "while") and their precise semantic meanings in code [70]. Appendix A of Sorva's dissertation lists a research-backed misconception catalogue [68] with cases of novices conflating linguistically-similar terms such as "instance" and "object." Future work here could aim to tease apart which terms disproportionately impede non-native English speakers.

Similarly, some reported barriers about reading instructional materials and ESL technical communication are not specific to programming; learners in other fields struggle with similar comprehension issues [49, 56, 63]. That said, one salient feature of programming that our study highlighted was the extent to which code is closely intertwined with English (Figure 1). Although the platonic ideal of a programming language is pure mathematical logic independent of messy human languages, in reality, programming languages—along with their surrounding ecosystems of APIs, frameworks, IDEs, and online help communities—are strongly tied to English.

What implications might this reality have on the usability and robustness of software, since nowadays worldwide technological (and increasingly, physical) infrastructure runs on software powered by English-centric programming tools? Prior work found that Western-designed user interfaces are not ideal for everyone around the world [64, 65], so does that also apply to other types of software as well since their underlying implementation languages are so English-centric?

To address these questions, future work could empirically measure the differences in how native and non-native English speakers choose to implement similar pieces of software, and the ensuing impacts on maintainability, robustness, and usability. From the comprehension side, one could measure differences in cognitive load faced by native and non-native English speakers when reading a code base to isolate which features cause the most trouble for non-native speakers.

More broadly, future work could compare how hard it is for non-native English speakers to learn to code versus learning other complex technical topics such as mathematics, physics, or 3D graphics. Would programming be harder to learn since its "language" is textual and thus more tied to English rather than being symbolic or graphical? Relatedly, could blocks-based or graphical languages help make programming more agnostic to human languages and thus more globally accessible? If so, how can we help learners manage the eventual transition they need to make to English-centric text-based languages if they want to join the global workforce?

Broadening Further to Look Beyond Utilitarian Goals

This paper has taken a highly utilitarian point-of-view of programming as a marketable skill that adults want to acquire to participate in a global workforce. But what about other oft-cited reasons for learning to code, such as using it as a conduit to achieve computational literacy, to develop computational thinking skills, or to learn about the world [37]?

If we expand the scope of programming beyond utilitarian job-related goals, then we will not be bound by its legacy English-centric origins and potential cultural homogeneity. If we can free ourselves from needing to interoperate with existing production-grade APIs and ecosystems, what future opportunities might there be for designing new programming languages and tools that mesh with the native languages and cultures of learners from diverse backgrounds? These would not merely be rote translations of existing English-based tools but would rather be designed from the ground up to be intimately intertwined with one's native language and culture. Speculating even further, do people from different human language backgrounds express their computational thoughts differently and thus prefer entirely different constructs from their programming languages and tools? That is, might there be a Sapir-Whorf hypothesis [32] for programming?

CONCLUSION

This paper contributes, to our knowledge, the largest-scale qualitative investigation so far of human language on learning programming. By analyzing 840 survey responses spanning 86 countries and 74 native languages, we discovered a set of barriers faced by non-native English speakers (as reported by *both* native and non-native speakers), along with their desires for improving instructional materials. We then proposed learner-centered design ideas [37] for making instructional resources and tools more accessible to non-native English speakers. This work brings us closer to a future where the hopeful promise of *Computer Science For All* [53] broadens its scope to include people from all language backgrounds.

More generally, programmers are a form of *lead users* [78] who are "ahead of the curve" in terms of willingness to both adopt and adapt existing resources to suit their needs. Thus, designing for novice programmers could pave the way for making other creative technical skills more accessible to people from all sorts of language and cultural backgrounds. It is hard to predict exactly what new creative skills will be in demand in the coming decades, but making sure that they will be maximally accessible to a global audience will become even more important as the world grows more interconnected. By empowering more future creators from diverse backgrounds, the artifacts that end up being created will hopefully in turn become more representative of the needs of the world.

Acknowledgments: Thanks to Hsien-che (Charles) Chen for assisting with data analysis and to Xiong Zhang and the UC San Diego Design Lab for their feedback on this project.

REFERENCES

1. 2016. About Scratch. https://scratch.mit.edu/about. (2016). Accessed: 2016-09-19.
2. 2017. Ethnologue: Languages of the World – English. https://www.ethnologue.com/language/eng. (2017). Accessed: 2017-09-05.
3. 2017. MOOC List – Find MOOCs By Languages. https://www.mooc-list.com/languages?static=true. (2017). Accessed: 2017-09-05.
4. 2017. Simple English Wikipedia. https://simple.wikipedia.org/wiki/Main_Page. (2017). Accessed: 2017-09-14.
5. Christine Alvarado and Zachary Dodds. 2010. Women in CS: An Evaluation of Three Promising Practices. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education (SIGCSE '10)*. ACM, New York, NY, USA, 57–61. DOI: http://dx.doi.org/10.1145/1734263.1734281
6. Apple. 2017. Introduction to Coding Guidelines for Cocoa. https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/CodingGuidelines/CodingGuidelines.html. (2017). Accessed: 2017-09-14.
7. Computing Research Association. 2016. The Taulbee Survey. http://cra.org/resources/taulbee-survey/. (2016). Accessed: 2017-09-14.
8. Jeff Atwood. 2009. Stack Overflow Blog: Non-English Question Policy. https://stackoverflow.blog/2009/07/23/non-english-question-policy/. (2009). Accessed: 2017-09-14.
9. ISO 639-2 Registration Authority. 2017. Codes for the Representation of Names of Languages. https://www.loc.gov/standards/iso639-2/. (2017). Accessed: 2017-09-14.
10. Mordechai Ben-Ari and Jorma Sajaniemi. 2004. Roles of Variables As Seen by CS Educators. In *Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '04)*. ACM, New York, NY, USA, 52–56. DOI: http://dx.doi.org/10.1145/1007996.1008013
11. Coursera blog. 2012. Coursera hits 1 million students across 196 countries. https://blog.coursera.org/coursera-hits-1-million-students-across-196/. (2012). Accessed: 2017-09-14.
12. Bo Brinkman and Amanda Diekman. 2016. Applying the Communal Goal Congruity Perspective to Enhance Diversity and Inclusion in Undergraduate Computing Degrees. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16)*. ACM, New York, NY, USA, 102–107. DOI: http://dx.doi.org/10.1145/2839509.2844562
13. Pauli Byckling, Petri Gerdt, and Jorma Sajaniemi. 2005. Roles of Variables in Object-oriented Programming. In *Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. ACM, New York, NY, USA, 350–355. DOI: http://dx.doi.org/10.1145/1094855.1094972
14. Carrie J. Cai, Philip J. Guo, James R. Glass, and Robert C. Miller. 2015. Wait-Learning: Leveraging Wait Time for Second Language Education. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*. ACM, New York, NY, USA, 3701–3710. DOI: http://dx.doi.org/10.1145/2702123.2702267
15. Parmit K. Chilana, Rishabh Singh, and Philip J. Guo. 2016. Understanding Conversational Programmers: A Perspective from the Software Industry. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16)*. ACM, New York, NY, USA, 1462–1472. DOI: http://dx.doi.org/10.1145/2858036.2858323
16. Artem Chistyakov. 2017. The language of programming. https://temochka.com/blog/posts/2017/06/28/the-language-of-programming.html. (Jun 2017).
17. Patricia H. Collins. 2015. Intersectionality's Definitional Dilemmas. *Annual Review of Sociology* 41, 1 (2015), 1–20.
18. Juliet M. Corbin and Anselm L. Strauss. 2008. *Basics of qualitative research: techniques and procedures for developing grounded theory*. SAGE Publications, Inc.
19. Sayamindu Dasgupta and Benjamin Mako Hill. 2017. Learning to Code in Localized Programming Languages. In *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale (L@S '17)*. ACM, New York, NY, USA, 33–39. DOI: http://dx.doi.org/10.1145/3051457.3051464
20. Betsy DiSalvo, Mark Guzdial, Charles Meadows, Ken Perry, Tom McKlin, and Amy Bruckman. 2013. Workifying Games: Successfully Engaging African American Gamers with Computer Science. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13)*. ACM, New York, NY, USA, 317–322. DOI: http://dx.doi.org/10.1145/2445196.2445292
21. Betsy DiSalvo, Sarita Yardi, Mark Guzdial, Tom McKlin, Charles Meadows, Kenneth Perry, and Amy Bruckman. 2011. African American Men Constructing Computing Identity. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '11)*. ACM, New York, NY, USA, 2967–2970. DOI: http://dx.doi.org/10.1145/1978942.1979381

22. Brian Dorn and Mark Guzdial. 2010. Learning on the Job: Characterizing the Programming Knowledge and Learning Strategies of Web Designers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. ACM, New York, NY, USA, 703–712. DOI: http://dx.doi.org/10.1145/1753326.1753430

23. Ian Drosos, Philip J. Guo, and Chris Parnin. 2017. HappyFace: Identifying and Predicting Frustrating Obstacles for Learning Programming at Scale. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC) (VL/HCC '17)*.

24. Benedict Du Boulay. 1986. Some Difficulties of Learning to Program. *Journal of Educational Computing Research* 2, 1 (1986), 57–73. http://www.tandfonline.com/doi/abs/10.1207/S15327809JLS0904_3

25. Darren Edge, Kai-Yin Cheng, Michael Whitney, Yao Qian, Zhijie Yan, and Frank Soong. 2012. Tip Tap Tones: Mobile Microtraining of Mandarin Sounds. In *Proceedings of the 14th International Conference on Human-computer Interaction with Mobile Devices and Services Companion (MobileHCI '12)*. ACM, New York, NY, USA, 215–216. DOI: http://dx.doi.org/10.1145/2371664.2371715

26. Darren Edge, Elly Searle, Kevin Chiu, Jing Zhao, and James A. Landay. 2011. MicroMandarin: Mobile Language Learning in Context. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '11)*. ACM, New York, NY, USA, 3169–3178. DOI: http://dx.doi.org/10.1145/1978942.1979413

27. Barbara Ericson, Shelly Engelman, Tom McKlin, and Ja'Quan Taylor. 2014. Project Rise Up 4 CS: Increasing the Number of Black Students Who Pass Advanced Placement CS A. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14)*. ACM, New York, NY, USA, 439–444. DOI: http://dx.doi.org/10.1145/2538862.2538937

28. Barbara J. Ericson, Kantwon Rogers, Miranda Parker, Briana Morrison, and Mark Guzdial. 2016. Identifying Design Principles for CS Teacher Ebooks Through Design-Based Research. In *Proceedings of the 2016 ACM Conference on International Computing Education Research (ICER '16)*. ACM, New York, NY, USA, 191–200. DOI: http://dx.doi.org/10.1145/2960310.2960335

29. National Science Foundation. 2017. Broadening Participation in Computing (BPC). https://www.nsf.gov/cise/bpc/. (2017). Accessed: 2017-09-14.

30. Scott Freeman, Sarah L. Eddy, Miles McDonough, Michelle K. Smith, Nnadozie Okoroafor, Hannah Jordt, and Mary Pat Wenderoth. 2014. Active learning increases student performance in science, engineering, and mathematics. *Proceedings of the National Academy of Sciences* 111, 23 (2014), 8410–8415. DOI: http://dx.doi.org/10.1073/pnas.1319030111

31. T. R. G. Green. 1989. Cognitive dimensions of notations. In *People and Computers V*. University Press, 443–460.

32. J.J. Gumperz and S.C. Levinson. 1996. *Rethinking Linguistic Relativity*. Cambridge University Press. https://books.google.com/books?id=dPXvxgL2t1oC

33. Philip J. Guo. 2013. Online Python Tutor: Embeddable Web-based Program Visualization for CS Education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13)*. ACM, New York, NY, USA, 579–584. DOI: http://dx.doi.org/10.1145/2445196.2445368

34. Philip J. Guo. 2015. Codeopticon: Real-Time, One-To-Many Human Tutoring for Computer Programming. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software and Technology (UIST '15)*. ACM, New York, NY, USA, 599–608. DOI: http://dx.doi.org/10.1145/2807442.2807469

35. Philip J. Guo. 2017. Older Adults Learning Computer Programming: Motivations, Frustrations, and Design Opportunities. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. ACM, New York, NY, USA, 7070–7083. DOI: http://dx.doi.org/10.1145/3025453.3025945

36. Philip J. Guo, Jeffery White, and Renan Zanelatto. 2015. Codechella: Multi-user program visualizations for real-time tutoring and collaborative learning. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC) (VL/HCC '15)*. 79–87. DOI: http://dx.doi.org/10.1109/VLHCC.2015.7357201

37. Mark Guzdial. 2015. Learner-Centered Design of Computing Education: Research on Computing for Everyone. *Synthesis Lectures on Human-Centered Informatics* 8, 6 (2015), 1–165.

38. Scott Hanselman. 2008. Do you have to know English to be a Programmer? http://www.hanselman.com/blog/DoYouHaveToKnowEnglishToBeAProgrammer.aspx. (Nov 2008).

39. Masayuki Igawa, Dong Ma, and Samuel de Medeiros Queiroz. 2017. Non-native English speakers in Open Source communities: A True Story. (talk at Linux.conf.au). https://youtu.be/fsn6buk-BtE. (Jan 2017).

40. Andy Kirkpatrick. 2011. Internationalization or Englishization: Medium of Instruction in Today's Universities. (01 2011).

41. Andrew J. Ko, Brad A. Myers, and Htet Htet Aung. 2004. Six Learning Barriers in End-User Programming Systems. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC) (VL/HCC '04)*.

42. Geza Kovacs and Robert C. Miller. 2014. Smart Subtitles for Vocabulary Learning. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14)*. ACM, New York, NY, USA, 853–862. DOI: http://dx.doi.org/10.1145/2556288.2557256

43. J. Lave and E. Wenger. 1991. *Situated Learning: Legitimate Peripheral Participation*. Cambridge University Press.

44. Ron Mace. 1997. What is universal design. *The Center for Universal Design at North Carolina State University* http://www.udinstitute.org/principles.php (1997).

45. John H. Maloney, Kylie Peppler, Yasmin Kafai, Mitchel Resnick, and Natalie Rusk. 2008. Programming by Choice: Urban Youth Learning Programming with Scratch. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '08)*. ACM, New York, NY, USA, 367–371. DOI: http://dx.doi.org/10.1145/1352135.1352260

46. MaxMind. 2017. GeoLite2 Free Downloadable Databases. https://dev.maxmind.com/geoip/geoip2/geolite2/. (2017). Accessed: 2017-09-14.

47. Charlie McDowell, Linda Werner, Heather Bullock, and Julian Fernald. 2002. The Effects of Pair-programming on Performance in an Introductory Programming Course. In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education (SIGCSE '02)*. ACM, New York, NY, USA, 38–42. DOI: http://dx.doi.org/10.1145/563340.563353

48. Charlie McDowell, Linda Werner, Heather E. Bullock, and Julian Fernald. 2006. Pair Programming Improves Student Retention, Confidence, and Program Quality. *Commun. ACM* 49, 8 (Aug. 2006), 90–95. DOI: http://dx.doi.org/10.1145/1145287.1145293

49. Peeter Mehisto. 2012. Criteria for producing CLI learning material. *Encuentro Journal* (2012), 15–33.

50. Subrata Kumar Mitra. 2010. Internationalization of Education in India: Emerging Trends and Strategies. *Asian Social Science* 6, 6 (2010).

51. Brad A. Myers and Jeffrey Stylos. 2016. Improving API Usability. *Commun. ACM* 59, 6 (May 2016), 62–69. DOI: http://dx.doi.org/10.1145/2896587

52. Jakob Nielsen. 1994. Usability Inspection Methods. John Wiley & Sons, Inc., New York, NY, USA, Chapter Heuristic Evaluation, 25–62. http://dl.acm.org/citation.cfm?id=189200.189209

53. The White House: Office of the Press Secretary. 2016. FACT SHEET: President Obama Announces Computer Science For All Initiative. (Jan 2016).

54. Python official documentation. 2017a. PEP 8 – Style Guide for Python Code. https://www.python.org/dev/peps/pep-0008/. (2017). Accessed: 2017-09-14.

55. Python official documentation. 2017b. sqlite3 DB-API 2.0 interface for SQLite databases. https://docs.python.org/3/library/sqlite3.html. (2017). Accessed: 2017-09-14.

56. Yogendra Pal. 2016. *A Framework for Scaffolding to Teach Programming to Vernacular Medium Learners*. Ph.D. Dissertation. Indian Institute of Technology Bombay.

57. Yogendra Pal and Sridhar Iyer. 2012. Comparison of English Versus Hindi Medium Students for Programming Abilities Acquired Through Video-Based Instruction. In *Proceedings of the 2012 IEEE Fourth International Conference on Technology for Education (T4E '12)*. IEEE Computer Society, Washington, DC, USA, 26–30. DOI: http://dx.doi.org/10.1109/T4E.2012.30

58. Yogendra Pal and Sridhar Iyer. 2015a. Classroom Versus Screencast for Native Language Learners: Effect of Medium of Instruction on Knowledge of Programming. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '15)*. ACM, New York, NY, USA, 290–295. DOI: http://dx.doi.org/10.1145/2729094.2742618

59. Yogendra Pal and Sridhar Iyer. 2015b. Effect of Medium of Instruction on Programming Ability Acquired through Screencast. In *2015 International Conference on Learning and Teaching in Computing and Engineering*. 17–21. DOI: http://dx.doi.org/10.1109/LaTiCE.2015.38

60. John F. Pane and Brad A. Myers. 1996. *Usability Issues in the Design of Novice Programming Systems*. Technical Report CMU-CS-96-132. Carnegie Mellon University.

61. Annie Murphy Paul. 2014. Bill Gates Is an Autodidact. You're Probably Not. Ed tech promoters need to understand how most of us learn. *Slate* (July 2014).

62. Roy D. Pea. 1986. Language-Independent Conceptual "Bugs" in Novice Programming. *Journal of Educational Computing Research* 2, 1 (1986), 25–36. DOI: http://dx.doi.org/10.2190/689T-1R2A-X4W4-29J2

63. Margaret Probyn. 2001. Teachers Voices: Teachers Reflections on Learning and Teaching through the Medium of English as an Additional Language in South Africa. *International Journal of Bilingual Education and Bilingualism* 4, 4 (2001), 249–266. DOI: http://dx.doi.org/10.1080/13670050108667731

64. Katharina Reinecke and Abraham Bernstein. 2011. Improving Performance, Perceived Usability, and Aesthetics with Culturally Adaptive User Interfaces. *ACM Trans. Comput.-Hum. Interact.* 18, 2, Article 8 (July 2011), 29 pages. DOI: http://dx.doi.org/10.1145/1970378.1970382

65. Katharina Reinecke and Krzysztof Z. Gajos. 2014. Quantifying Visual Preferences Around the World. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14)*. ACM, New York, NY, USA, 11–20. DOI: http://dx.doi.org/10.1145/2556288.2557052

66. Gabriela T. Richard, Yasmin B. Kafai, Barrie Adleberg, and Orkan Telhan. 2015. StitchFest: Diversifying a College Hackathon to Broaden Participation and Perceptions in Computing. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15)*. ACM, New York, NY, USA, 114–119. DOI: http://dx.doi.org/10.1145/2676723.2677310

67. E. Soloway and James C. Spohrer. 1988. *Studying the Novice Programmer*. L. Erlbaum Associates Inc., Hillsdale, NJ, USA.

68. Juha Sorva. 2012. Visual program simulation in introductory programming education (PhD dissertation). (2012). http://urn.fi/URN:ISBN:978-952-60-4626-6

69. Juha Sorva. 2013. Notional Machines and Introductory Programming Education. *Trans. Comput. Educ.* 13, 2, Article 8 (July 2013), 31 pages. DOI: http://dx.doi.org/10.1145/2483710.2483713

70. Andreas Stefik and Susanna Siebert. 2013. An Empirical Investigation into Programming Language Syntax. *Trans. Comput. Educ.* 13, 4, Article 19 (Nov. 2013), 40 pages. DOI: http://dx.doi.org/10.1145/2534973

71. John Sweller. 1994. Cognitive load theory, learning difficulty, and instructional design. *Learning and Instruction* 4, 4 (1994), 295 – 312. DOI: http://dx.doi.org/10.1016/0959-4752(94)90003-5

72. Josie Taylor. 1990. Analysing novices analysing Prolog: what stories do novices tell themselves about Prolog? *Instructional Science* 19, 4 (01 Jul 1990), 283–309. DOI: http://dx.doi.org/10.1007/BF00116442

73. Kyle Thayer and Andrew J. Ko. 2017. Barriers Faced by Coding Bootcamp Students. In *Proceedings of the 2017 ACM Conference on International Computing Education Research (ICER '17)*. ACM, New York, NY, USA.

74. J. Gregory Trafton and Brian J. Reiser. 1993. The Contributions of Studying Examples and Solving Problems to Skill Acquisition. Lawrence Erlbaum Associates, Inc., 1017–1022.

75. Judith Uchidiuno, Amy Ogan, Evelyn Yarzebinski, and Jessica Hammer. 2016. Understanding ESL Students' Motivations to Increase MOOC Accessibility. In *Proceedings of the Third (2016) ACM Conference on Learning @ Scale (L@S '16)*. ACM, New York, NY, USA, 169–172. DOI: http://dx.doi.org/10.1145/2876034.2893398

76. Udacity. 2017. About Us. https://www.udacity.com/us. (2017). Accessed: 2017-09-14.

77. Ashok Kumar Veerasamy and Anna Shillabeer. 2014. Teaching English Based Programming Courses to English Language Learners/Non-Native Speakers of English. *International Proceedings of Economics Development and Research* 70, 4 (2014), 17–22.

78. Eric von Hippel. 1986. Lead Users: A Source of Novel Product Concepts. *Manage. Sci.* 32, 7 (July 1986), 791–805. DOI: http://dx.doi.org/10.1287/mnsc.32.7.791

79. April Wang, Ryan Mitts, Philip J. Guo, and Parmit K. Chilana. 2018. Mismatch of Expectations: How Modern Learning Resources Fail Conversational Programmers. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. ACM, New York, NY, USA.

80. Wikipedia. 2017. Non-English-based programming languages. https://en.wikipedia.org/wiki/Non-English-based_programming_languages. (2017). Accessed: 2017-09-05.

81. David Wolber, Hal Abelson, Ellen Spertus, and Liz Looney. 2011. *App Inventor*. O'Reilly Media, Inc.