Towards Inclusive Source Code Readability Based on the Preferences of Programmers with Visual Impairments

Maulishree Pandey*
maupande@umich.edu
University of Michigan School of Information
Ann Arbor, Michigan, USA

Steve Oney
soney@umich.edu
University of Michigan School of Information
Ann Arbor, Michigan, USA

Andrew Begel
abegel@cmu.edu
Carnegie Mellon University Software and Societal Systems Department
Pittsburgh, PA, USA

ABSTRACT

Code readability is crucial for program comprehension, maintenance, and collaboration. However, many of the standards for writing readable code are derived from sighted developers' readability needs. We conducted a qualitative study with 16 blind and visually impaired (BVI) developers to better understand their readability preferences for common code formatting rules such as identifier naming conventions, line length, and the use of indentation. Our findings reveal how BVI developers' preferences contrast with those of sighted developers and how we can expand the existing rules to improve code readability on screen readers. Based on the findings, we contribute an inclusive understanding of code readability and derive implications for programming languages, development environments, and style guides. Our work helps broaden the meaning of readable code in software engineering and accessibility research.

CCS CONCEPTS

• **Human-centered computing → Empirical studies in accessibility**; *Empirical studies in collaborative and social computing*.

KEYWORDS

software developers, blind or visually impaired, accessibility, code readability

ACM Reference Format:
Maulishree Pandey, Steve Oney, and Andrew Begel. 2024. Towards Inclusive Source Code Readability Based on the Preferences of Programmers with Visual Impairments. In *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI '24), May 11–16, 2024, Honolulu, HI, USA*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3613904.3642512

1 INTRODUCTION

Reading code is one of the most fundamental and important activities in software development. *Readability* is a subjective measurement of how easy it is to go through any given code. More readable code is easier to comprehend and maintain in the long term. Typically, software maintenance comprises 70% of any project's life cycle [15], making it the most intensive aspect of software development projects. Elshoff and Marcotty recommended adding another phase to the software lifecycle just to make the source code more readable [23]. They suggested the phase should require developers to apply consistent formatting, leave good comments, and remove unused code blocks. Software companies enforce adherence to coding standards [48] and use code reviews to ensure code quality [22]. Companies like Google and AirBnb have even made their coding standards public to ensure consistent and readable code contributions from the larger programming community [3, 27]. Others have recommended ensuring the readability of documentation to aid developers in making readable edits to codebases [1, 29]. Some also propose teaching students to write readable code as part of standard programming coursework [19].

The focus on readability has led to the development of rich visual design and functionality in code editors. For instance, indentation is long known to improve readability among sighted developers [53]. Code editors like Sublime Text, and IntelliJ display vertical lines to visually match indentation levels. IDEs such as VS Code offer mini-maps, which are zoomed out representations of the code structure. Developers can quickly navigate to different code blocks by identifying their shape and relative position in the map.

However, our current understanding of readability is based on the opinions and preferences of sighted developers [21, 36]. Blind and visually impaired (BVI) programmers use assistive technologies (ATs) such as screen readers, which lack the visual expressiveness and information density of graphical user interfaces (GUIs). The serial and ephemeral nature of screen reader output [10] leads to different browsing [12] and skimming [2] strategies among BVI people in comparison to sighted people. These differences between screen readers and GUIs suggest that BVI developers may have different readability preferences from sighted developers. In this paper, we investigate code readability for BVI developers. Specifically, we pose the following research questions:

(1) RQ1. How and why do the code readability preferences of BVI developers differ from that of sighted developers as identified through literature review (see §2.2)?
(2) RQ2. What implications do these differences have for programming tools such as static analyzers and code editors, code styling guidelines, and programming languages?

We conducted a remote exploratory qualitative study with 16 BVI developers to answer our research questions. During the study, we asked participants to review 15 rules related to code readability (see Table 1). We presented two functionally equivalent but differently

*The author is currently a UX researcher at Google. The research was done when she was an a doctoral student at the University of Michigan.

formatted versions of code snippets for each rule. One version's presentation was informed by PEP8 [63], the official Python style guide, serving as a proxy for sighted developers' preferences; the second version's formatting was informed by accessibility research. We asked participants to select their preferred option for each rule. We asked follow-up questions to understand their preferences and concluded with a short semi-structured interview to elicit their experiences with code styling during collaborative activities such as code reviews.

Our research leads to a more inclusive understanding of *code readability* and makes the following contributions to the fields of HCI, accessibility, and software engineering research:

- A taxonomy for what is good code formatting on screen readers vs. GUIs to support better code readability
- Empirical data to explain how various factors shape code readability on screen readers
- Design recommendations for code editors and programming languages

2 RELATED WORK

Buse and Weimer defined code readability as "a human judgement of how easy a text is to understand" [17]. Readability is known to improve program comprehension but is distinct from overall understandability of code. For instance, readable code may still be difficult to understand due to unfamiliar APIs, poor documentation, and complexity of source code [51]. Sighted developers do not read code linearly. They are far more likely to *skim* the source code to locate regions of interest where they do more *focused reading* [54]. In this section, we first draw on empirical studies at the overlap of accessibility and programming to explain what we know about code reading, comprehension, and navigation on screen readers. Then we summarize the factors that shape code readability for sighted developers.

2.1 Code Reading on Screen Readers

The primary focus of existing HCI and accessibility studies has been on code navigation and comprehension. However, a close review of these papers reveals a few insights about readability.

2.1.1 Linear Navigation. Prior research suggests that BVI developers want to avoid going through the codebase line by line but are forced to do so to get an overview of the code structure [5]. Francioni and Smith developed JavaSpeak to enable BVI developers to acquire details about the code structure and semantics more efficiently [25]. JavaSpeak spoke the code with different intonations to communicate structure. The researchers also suggested using prosodic elements like speaking rate, pitch, or phrasing to communicate semantic characteristics about code [25], a recommendation seconded by Stefik [58]. Screen readers like JAWS [26] and NVDA [35] use prosody to indicate the capitalization, which may come in handy during programming.

Stefik suggested using audio cues to inform BVI developers about the "scoping relationships between pieces of syntax" to communicate the information provided by syntax highlighting [58]. An example of Stefik's suggestion would be the work by Hutchinson and Metatla [30]. They designed 12 audio cues to represent different programming constructs, such as the sound of door opening for `if` blocks and door closing for `else` blocks [30]. The idea was that developers could use the audio cues to skip listening to the entire statement and move through the codebase more efficiently [30]. However, BVI participants in the study reported wanting more practice with the audio cues to map them accurately to the constructs. Evidence suggests that skeuomorphic audio cues can help reduce the learning curve [43].

Studies suggest that BVI developers avoided indenting code altogether unless collaborating with sighted developers [5, 41]. It makes linear code reading very verbose by announcing all the whitespaces. BVI developers are known to develop custom scripts to minimize the indentation announcement [5]. For similar reasons, they prefer to not receive all punctuation announcement [9]. One way to address verbosity is by outputting the semantic meaning of a code statement but that can make editing the syntax challenging in real-world projects [58]. Thus, researchers have used the approach only for making source code more understandable to novice BVI developers [50, 59]. Lastly, recent evidence shows that poor identifier or variable names affect code reading and debugging on screen readers [40, 41] but we lack perspective on their casing, length, and naming choices.

2.1.2 Non-Linear Navigation. Sighted people can use an array of methods for non-linear navigation: scroll, point and click, use keyboard shortcuts, utilize IDE features like tree views and mini maps, and keyword search. BVI developers only have a subset of these options available to them to make sweeping jumps through the code [43]. Keyword search is reportedly one of the most common methods for code navigation [5, 44]. BVI developers have reported maintaining a document to easily look up variable and function names [4]. However, search can be time consuming and frustrating when multiple results pop up for the same keyword [5]. BVI developers have to review code statements multiple times to verify they are on the required line [5]. As a workaround, they may leave comments to bookmark interesting locations in the code [5].

Another common strategy is to jump between function signatures [6, 7]. Audio-based plugins are especially helpful in non-linear navigation. StructJumper provided a hierarchical tree view of the source code's nested structure to facilitate skimming and non-linear navigation [9]. Its evaluation showed that efficient navigation meant BVI developers did not have to remember much of the code during code reading [9]. The success of hierarchical trees was extended to support navigation of larger software projects with several files [42, 56].

2.2 Factors Affecting Code Readability for Sighted Developers

Prior research suggests that readability for sighted developers depends on the following: (1) use of spacing to make blocks visually distinguishable and easily identifiable using indentation, vertical line breaks, and whitespaces (2) identifier names and their naming style (camel case vs. snake case), (3) line length for source code and comments, and (4) text formatting [36]. We discuss these below; Table 1 summarizes the factors and their sub-factors.

2.2.1 Spacing. Indentation is one of the most widely used approaches for modifying code layout. Early evidence suggested that as program complexity increased, indentation improved program comprehension [18, 53]. Subsequent studies investigated the optimal amount of indentation that aided in readability without increasing typing effort. For instance, Miara *et al.* suggested using 2–4 spaces to indent code blocks in Pascal, with 2 spaces offering most readability across developers' experience levels [32]. Furthermore, they found that an overly indented code made scanning difficult. Indentation also had diminishing returns in heavily nested code or when entities were separated by blank lines [18]. While developers' opinions remain undecided between 2 vs. 4 spaces [11, 21], the latter gives the visual appearance of a tab character and may lead to inconsistent use of tabs and spaces during collaboration, causing breakdowns in programming languages such as Python.

Another way to improve source code navigation is through *segmenting i.e.*, putting blanks lines between code blocks that are functionally not similar [17, 49, 61, 64]. While it has not been found to have a significant effect on program comprehension and recall [31] and developer opinion seems split on the topic [21], coding standards recommend the use of vertical space to delineate code blocks [63]. Furthermore, the approach is an alternative to more explicit form of coding such as marking the beginnings and ends of code blocks with explicit statements or comments, which makes the code longer and difficult to read [60].

Coding standards also recommend using whitespaces around operators to improve legibility at line level [63]. While they have not been reported to significantly improve readability [49], they are considered good coding practice [17].

2.2.2 *Identifiers.*

Meaningful identifier names (e.g., variable names or function names) have been found to improve readability [61] whereas poor naming practices can increase developers' cognitive load [24]. Developers may not follow good naming practices due to differing opinions on what constitutes a good name [61], with novice developers more likely to use poor naming choices [47].

When it comes to identifiers, the word boundary style also matters. Sharif and Maletic investigated the effect of camel case and snake case on identifier names [52]. They found that participants took 13.5% longer to recognize camel case identifiers [52]. On the other hand, Binkley *et al.* [14] found that regardless of developer experience, camel casing led to higher accuracy for source code manipulation in Java and C. Their follow-up study found that beginners recalled camel cased identifiers better whereas experts recalled better with snaked case. However, there was no statistically significant difference in visual effort needed for both styles [13]. Furthermore, regardless of the word boundary, longer names took more time to be recognized [14].

2.2.3 *Line Length.*

Readability also depends on line length. Long lines of code are more difficult to understand, much like long sentences. Most coding standards recommend limiting lines to 79 characters [63]. It allows sighted developers to open multiple editor windows side by side and avoid horizontally scrolling [21]. Some researchers have even recommended that programming languages should favor constructs that allow developers to write shorter lines of code, for example using pre and post increments (*e.g.*, `i++`) instead of addition operations (*e.g.*, `i = i + 1`) [17].

Coding standards such as PEP8 typically recommend a shorter line length of 72 characters for more free flowing text such as comments and docstrings, which are strings used to document functions and classes [63]. Comments are especially useful in large non-modular code [62]. Developers are encouraged to use comments sparingly and write them in simple language [57], while ideally writing code where the intent is apparent without the need for additional explanations [28].

2.2.4 *Text Formatting.*

Readability for sighted people is shaped by *legibility* of the displayed text, which comprises layout (discussed above) and text formatting characteristics. Good legibility is related to readers' spatial visual abilities [65]. Depending on one's visual acuity, one needs to modify formatting attributes such as font type, contrast, font size, etc. to maximize the legibility of readable text [65]. For instance, Baecker applied the principles of graphic design to C programs [8]. He relied on different font types, proportional character spacing, and color contrast to improve the parsing of complex statements and special symbols by 25%, as measured by performance on a comprehension test [8]. Similarly, Raymond explored the use of typography to enhance readability [46]. Code editors set the formatting characteristics to reasonable defaults and these can be personalized by sighted developers to their liking. Among the factors discussed above, visual formatting is least relevant to BVI developers. They do not use text formatting attributes such as font size, font family, and colors on screen readers.

Modern code editors offer syntax highlighting and auto indentation to help sighted developers in identifying areas of interest. Static analysis tools such as code linters flag departures from coding standards such as line length violations or poor indentation without having to run the code. Together, these features facilitate skimming and focused reading for sighted developers. But we know little about how BVI developers identify areas of interest and what helps them in focused reading. Our study attempts to address that gap.

3 STUDY DESIGN

We conducted a remote exploratory qualitative study with 16 BVI developers to understand their preferences and perspectives on factors that impacted code readability. Studies lasted between 58 to 90 minutes.

3.1 Participants

We obtained IRB approval from the university for our study. We recruited our participants through snowball sampling and online forums such as program-l, a mailing list primarily comprising BVI developers [45]. The eligibility criteria for participation were that developers should be 18 years or older, possess at least one year of experience programming with screen readers, and be able to communicate about code in spoken English. Since code styling guidelines vary across programming languages, we selected Python and JavaScript for the study. Our choice was informed by the immense and consistent popularity of both programming languages in the developer community [38, 39].

We circulated a questionnaire to screen participants who met our eligibility criteria. The questionnaire asked respondents to self-report their programming experience in Python and JavaScript on a scale of 1 to 5; 1 meant no experience and 5 meant expertise in

Table 1: Readability factors we considered in our study. #O1 and #O2 indicate the number of participants who chose option 1 and option 2 respectively for any factor/sub-factor combination. #O3 indicates participants who found both options equally readable or proposed a third alternative. Last column is a sum of O1 – O3 and equals the total number of participants in our study

Factor	Sub-Factor	Code Type	Option 1 (O1)	# O1	Option 2 (O2)	# O2	# O3	# Sum
Spacing	Indentation	Nested Data Structures	Separate parentheses and key-value pairs	12	Match key-value pairs and parentheses	4	0	16
		Docstrings	Indent docstring arguments	4	Do not indent docstring arguments	9	3	16
	Segmenting	–	Use two blank lines to separate entities	4	Use single blank lines	12	0	16
	Whitespaces	Mathematical Operators	Surround operators with whitespaces	10	Avoid whitespaces	3	3	16
		Slice Operators	Surround operator with whitespaces	10	Avoid whitespaces	5	1	16
Identifiers	Word Boundaries	–	Use snake case	2	Use camel case	10	4	16
	Length	–	Long variable name	13	Short variable name	0	3	16
	Intent	–	Use consistent prefixes	2	Use consistent suffixes	12	2	16
Line Length	–	Function Calls	Render arguments on separate lines	8	Render arguments on same line	6	2	16
	–	Function Signatures	Render arguments on separate lines	10	Render arguments on same line	5	1	16
	–	Chaining	Treat dot operator as a delimiter	14	Do not treat dot operator as a delimiter	1	1	16
	–	Binary Operations	Place line break before the operator	7	Place line break after the operator	4	5	16
	–	Comments	Split comment across lines	3	Do not split comment	12	1	16
	–	Imports	Place imports on different lines	7	Place imports on the same line	6	3	16
String Quotes	Quote character	–	Use single quote	2	Use double quotes	12	2	16

the language. We selected respondents who reported an experience of 3 or higher. We received a total of 20 responses and conducted the study with 16 respondents. All recruited participants reported either equal experience between Python and JavaScript or more programming experience with Python. Therefore, we conducted the study entirely using the Python stimuli. The questionnaire also collected details about participants' demographics, assistive technology use, and job role (see Table 2).

In our final study sample, 14 participants identified as men and 2 identified as women. Participants were between 18 – 38 years old. They were employed as backend developers, full stack developers, tech lead positions, or were pursuing a higher education degree in computer science or a related field. All participants relied on screen readers to interact with digital devices; three participants reported using braille displays in the screening questionnaire but did not utilize them during the study. Specifically for the study, 14 participants used NVDA and 2 used JAWS (see Table 2).

3.2 Stimuli

The study was conducted remotely on Zoom. During the study, we presented participants with a markdown file that listed 15 code formatting rules based on the factors identified from existing research (see section 2.2). For each rule, we provided two functionally equivalent Python code snippets, inspired by Santos and Gerosa's study design [21]. One version conformed to PEP8 standards [63] and served as a proxy for sighted developers' preferences (*e.g.*, indented code block, snake case for identifiers); the other option was either formatted based on the evidence from accessibility research (e.g., unindented code to minimize verbosity) [5] or the alternative considered in studies with sighted developers (*e.g.*, camel case for identifiers) [52]. We added a rule to understand preferences for quoting strings because we expected it to impact verbosity [63] We randomized the order of rules and the order of options before each study session to mitigate learning effects across participants. Table 1 summarizes the rules and their breakdown across factors that affect readability. Appendix A lists code snippets corresponding to Table 1 whereas appendix B shows a randomized markdown file presented as stimuli to one of the participants.

3.3 Procedure

We asked participants to open the markdown in a code editor of their choice. Table 2 lists the code editor and the screen reader they used during the study. Participants were told to read each rule and its options as they would naturally go through any code. For each rule, we asked them to share which option they preferred and why. The research coordinator asked follow up questions about how the options affected readability, navigation, and verbosity on screen readers. Participants had the choice of creating alternatives if they did not like either of the two options presented in the markdown. The study concluded with a semi-structured interview to elicit their perspectives about differences in code styling preferences with sighted developers and the workflows they followed to improve code readability during collaboration. We compensated each participant with a USD $60 gift card (or its equivalent in local currency) for their participation.

3.4 Analysis

We transcribed the data from each study session. We organized and summed up participants' choices for each rule. We report these in Table 1 (columns 5, 7, and 8). We used the transcripts to highlight quotes that explained participants' choices. We identified emerging as well as missing themes in participants' reasoning for their choices through analytic memos [34] and weekly team check-ins. In round 1, we used descriptive codes [34] to organize the themes into the following categories: (1) readability, (2) ease of navigation, (3) typing effort, (4) collaboration, (5) programming tool and screen reader settings. In round 2, we used inductive coding [34] to develop sub-themes within each of them, followed by merging of certain themes. We finally ended up with three high-level themes that explain participants' choices across all factors and form the findings section of our paper.

4 FINDINGS

This section along with Table 1 answers RQ1: how and why do the code readability preferences of BVI developers differ from that of sighted developers as identified through literature review? We explain how length of code (see §4.1), programming environment (see §4.2), and different levels of navigation (see §4.3) shaped preferences across the factors and sub-factors we considered. Participants' quotes are lightly edited for clarity.

4.1 Impact of Line Length on Readability

We open our findings section by discussing how line length and type of code (e.g., function calls, library imports, comments, etc) shaped participants' code styling preferences.

Listing 1: Options presented to participants for call chains

4.1.1 Line Length. Participants preferred lengthy function calls, signatures, and chained statements to be split across multiple lines instead of single line (e.g., Option #1 in Listing 1). PEP8 recommends limiting line length to 79 characters unless teams prefer otherwise [63]. The character limit enables sighted developers to open files side by side without horizontally scrolling to read the overflowing text. While our finding is in agreement with PEP8's guideline, our participants' choices were driven by reasons of code comprehension. Screen readers are programmed to read out all the content on the line when the cursor reaches it. Participants shared a long and complex line of code, such as a function chain (e.g., Option #2 in Listing 1), was difficult to process when read out in one go. To avoid the continuous audio stream, they used the control-right and control-left arrows to read one word at a time. However, that proved to be too slow a reading pace. On the contrary, when code was split across multiple lines, the screen reader read smaller chunks. These were not only easier to process but also gave more control to participants. They could choose which chunk to pause on or skim past without listening to it entirely:

> *"So like if they are in the same line like, my mental process cannot process anything. So like this one, if it is split into multiple lines, I just read a part of the content bit by bit [reads Option #1 of Listing 1]. So this line is not too long so after reading it [...] And after processing, I can just move to the next line."* — P15

If the line included complex variable names, participants had to navigate through each character to verify the contents. Here again chunking helped! Participants could get to complex-sounding arguments quickly by first down-arrowing to the chunk they were interested in and then using right and left arrows to verify the characters:

> *"I want to read this character by character. Probably I'll be a little bit more faster because I'm right in the starting of the line, and I don't need to find that word. Immediately I can start reading, right? From the first character. "* — P11

We noted that participants' preferences were mediated by the likelihood of code reuse. A few participants pointed out that function signatures could be kept on one line despite its length since one is unlikely to change it. P15 mentioned that keeping function definition on one line enabled him to *"just copy the line and paste it"*, which he could then populate with the arguments to invoke the function. Typing or copy-pasting the function call in multiple places helped memorize the function definition. The ability to easily recall the code meant they could skip past the signature, which in turn made them prioritize formatting choices that facilitated efficient navigation.

A few participants said that in addition to splitting a lengthy line, they also preferred using named arguments. P8 described his work as a game developer involved function overloads that had up to 15 similar sounding arguments, such as the X, Y, and Z coordinates to map the three dimensional sound. In such cases, splitting the code across lines was not enough to remember the order of arguments. Furthermore, IntelliSense, the code editor feature that displays documentation upon mouse hovers, was not fully accessible to BVI developers:

Table 2: Participants' Demographic Details and Environment (code editor and screen reader) they used during the study. The first column lists gender and age in brackets (e.g., P1 is 32 years old and identifies as a man)

#	Job Role	Prog. Experience	Region	Screen Reader	Punctuation Setting	Indent Reporting	Code Editor
P1 (32M)	Backend Developer	10–14 years	Europe	NVDA	All	None	Notepad++
P2 (18M)	Student	5–9 years	Canada	NVDA	All	Speech	Notepad2
P3 (20M)	Student	1–4 years	India	NVDA	Most	Speech	Notepad
P4 (23M)	Game Developer	1–4 years	Pakistan	NVDA	All	Speech + Tones	VS Code
P5 (32M)	Backend Developer	20–24 years	Europe	NVDA	All	None	VS Code
P6 (26M)	Backend Developer	1–4 years	India	NVDA	Most	Tones	Notepad
P7 (34M)	Backend Developer	10–14 years	South Africa	NVDA	Most	Speech	Notepad++
P8 (38M)	Game Developer	20–24 years	USA	NVDA	Some	Tones	VS Code
P9 (28M)	Full Stack Developer	10–14 years	India	NVDA	All	Speech	VS Code
P10 (18M)	Student	5–9 years	India	NVDA	Some	Speech	VS Code
P11 (24M)	Backend Developer	5–9 years	India	JAWS	Most	N/A	VS Code
P12 (29M)	Tech Lead	10–14 years	Canada	NVDA	Some	None	Notepad++
P13 (25F)	Student	1–4 years	Europe	NVDA	All	Tones	Notepad++
P14 (31F)	Data Scientist	10–14 years	USA	JAWS	Most	N/A	VS Code
P15 (37M)	Researcher	15–19 years	China	NVDA	All	Speech	Notepad++
P16 (21M)	Student	1–4 years	Europe	NVDA	All	Tones	Notepad

"You [sighted developers] all have a lot of cool stuff where you can highlight something with a mouse [...] That's not something we get as blind programmers. I think it's getting better now 'cause you can do it in VS Code. You can kind of highlight an argument and I think you can press F12, and it will tell you what it goes with. But still it's not the most intuitive thing [...] But I love named arguments, I really adore them!" — P8

We also found tension between participants' desire to reduce navigation and splitting the code. For instance, a few participants proposed a third option of keeping 2—3 arguments per line instead of one argument on each line. It meant less typing effort compared to the formatted option as well as fewer down arrow presses. P12 shared that the Eclipse IDE offered a way to wrap lines in a manner which is accessible to both screen reader users and GUI users:

"So in Eclipse, sometimes I've seen [...] a few of the function names, which have a lot of arguments, so they get intended in a way that they fit on the screen. So you might be having one argument in front of the function name, and then here we'll have a couple of arguments in the second line, then another three arguments in the third line that way. So yes, it provides better readability and better scalability." — P12

A couple arguments on each line were short enough to process while one navigated downwards without adding vertical length to the code. Others shared that they would prefer a single argument on each line despite it requiring more arrow presses. This not only ensured consistency but also reduced the burden of having to remember that some lines could have multiple arguments, ultimately preventing the loss of information if one skimmed the code too fast. Participants felt that longer but consistent formatting positively shaped code comprehension when they revisited the code after a hiatus.

A few participants also recommended refactoring the code and making it more modular instead of longer function chains, emphasizing participants' desire for non-linear navigation. A more modular code enables developers to jump across functions, also reported by Albusays *et al.* [5]

4.1.2 *Type of Code.*

Length of line interacted with type of code in determining participants' preferences. We included examples to account for different types of code statements: (1) function signatures or definitions (2) function calls (3) function chains (4) comments (5) import statements. Majority of the participants preferred separation for the former three (as discussed in the previous section) whereas the preferences were more divided for the latter two, shaped by the need for consistency, efficient navigation, and less typing.

Participants mentioned that comments were typically written in English without special syntax or characters. They were easier to comprehend even when they exceeded the recommended character length, with our example being 109 characters long (see Rule #3.0.5 Option 2 in Appendix A):

"It's [comments] not that much sensitive that I need to read character by character. Whereas, if it is a code,

> *syntax, right? That I need to read character by character. So that makes sense to logically break."* — P11

The preference is in contrast with PEP8's recommendation, which suggests limiting comments to 72 characters for ease of visual consumption [63]. Participants also mentioned that ideally comments should be written in plain English because its purpose is to explain the code. However, if a comment was fairly descriptive and listed *"2 or 3 different steps"* (P2), they would consider breaking them down.

Although we did not include an example, we followed up with participants about their views on inline comments. PEP8 recommends using inline comments sparingly as they can distract from code reading [63]. Only select participants said they relied on inline comments and limited them to *"two to five words"* (P6). Most participants preferred comments to be on their own line because it tended to interfere with code reading in two ways. First, when participants tried to jump to the end of the code, their screen reader focus got placed at the end of the comment instead. They had to use control-left arrow to go backwards from the comment until they reached the code to *"edit the line"* (P1), wasting time in inline navigation. Second, they might completely miss the comment when down arrowing *"fast through the lines"* (P1).

Figure 1: Options presented to participants for import statements

Much like comments, we noted difference in opinions with regard to import statements due to three reasons (see Listing 1). First, the participants made a distinction between standard libraries and third-party libraries. Our example included standard Python libraries and a few participants said they were likely to *"group them together"* (P8) to *"get over them quicker with the down arrow"* (P1). On the other hand, third party libraries needed to be placed on their own individual lines because one was likely to import a submodule or rename the module:

> *"They can just import a specific module into the namespace. So then, you do* `from this import this` *, or, you know,* `import this as this`*"* — P2

Second, editing concerns affected choices. A few participants felt that import statements were only typed once, mostly read once at the beginning of the code, and were unlikely to be modified again. Therefore, one could place multiple imports on a single line without compromising readability. Others felt that because imports were typed precisely once, they should in fact be separated out, ultimately affording more convenience if any library had to be removed or replaced:

> *"if it is one library per line, so that if you just want to remove one of the library, it is more easier."* — P15

Third, participants' programming experience with other languages had a bearing on their opinions. For instance, P12 recalled that JAVA only permitted placing imports on separate lines. He chose the same option to stay consistent in our study despite describing the practice as a *"headache"* (P12).

4.2 Impact of Programming Environment on Readability

We now elaborate on the effect of screen reader settings such as punctuation settings and synthesizer choice on code readability. We also describe how these settings interacted with the code editor features.

Figure 2: Options presented to participants for identifier length

The perceived *verbosity* of code had a bearing on participants' styling preferences. For instance, certain naming choices required listening to more audio output and slowed down participants. Consider the options we presented to evaluate preferences for identifier length (see Listing 2). Sighted developers are likely to read both options as "radio button height". On the contrary, for our participants, the second option was announced as "radio B T N H T" – a more verbose output despite being fewer characters to type:

> *"So would you believe that even though option 2 is shorter, it's actually longer on the screen reader. Yeah, it's more syllables. Ain't that crazy! Because 'radioButtonHeight' is 5. But it's more characters, whereas 'radioBtnHt' [...] is actually 8."* — P8

Participants shared that a verbose name was harder to remember. Furthermore, they may confuse the output with similar sounding alphabets while skimming. The name may also be mispronounced by differences in capitalization or due to synthesizer choice:

> *"*`API` *is capital A, capital P, capital I. It's not a word but people try to use it as a word, so what they do is 'capital A, small P, small I' (Api). Then it will not read as API, that's when I get confused."* — P11

> *"I had one or two instances, where it will just call out something else. For example, my screen reader will often call out 'capital A, capital S' (AS) as American Samoa."* — P12

A funny instance of screen reader mispronunciation was when function signature arguments were rendered on separate lines (see Rule #3.0.2 Option 1 in Appendix A). The signature's closing parentheses and colon '`):`' ended up on a separate line. P12 chuckled when it was announced as "sad face". Such differences made the seemingly shorter option more verbose, harder to remember, and could introduce errors in the code. To avoid these issues, participants had to slow down their navigation and clarify the spelling by reading the variable *"character by character"* (P11).

We had included examples of names that encoded the context of use in either the suffix (e.g, `foregroundColorMenu`) or the prefix (`menuForegroundColor`) of the identifier. Majority of the participants preferred context to be announced first (*e.g.*, `menuForegroundColor`, `footerForegroundColor`) to reduce verbosity associated with long names during code skimming. A few participants pointed out that they would prefer `foregroundColorMenu` only if the code also contained counterparts such as `backgroundColorMenu`. They felt it would be more useful to glean the global relationship between identifier categories before learning about the specific UI elements they were responsible for. Participants' comments are reminiscent of Hungarian notation [55] and suggest a preference for quick navigation with lower verbosity.

Verbosity was also determined by the screen reader's punctuation setting. As shown in Table 2, 8 participants had set their punctuation to *all*, 5 had set it to *most*, and 3 had set it to *some*. *All* announced every punctuation character but meant greater verbosity, which interfered with reading and processing. On the other hand, *most* or *some* was likely to skip over important characters. The setting had a strong bearing on whether to use camel case or snake case. PEP8 recommends snake case *i.e.*, underscores to separate words in variable names (*e.g.*, `primary_address_apartment`) [63]. However, if participants' screen reader punctuation setting was set to *most* or *all*, it was announced as "primary line address line apartment" on NVDA (JAWS announces underscore "underline"). On the other hand, when punctuation was set to *some*, the announcements were same for both options (`primary address apartment`) but participants had to go through the identifier to ensure the presence of the underscore character. The verification once again meant the slower character-level navigation that interrupted skimming. Therefore, participants spoke of using camel case even if their colleagues preferred snake case:

> *"I prefer Option 1 (camel case) because, A, it's shorter and it reads fine [...] I like CamelCase for my Python variables. I've convinced my colleagues not to judge me for it"* — P7

Figure 3: Options presented to participants to understand line break preferences

The choice of punctuation setting could also skip information relevant for code comprehension. For instance, we asked participants where they would like to insert line breaks in long lines — split the line after the operator or before the operator (see Listing 3). Operators placed at the beginning of the line were announced regardless of one's punctuation setting. However, a less granular punctuation setting did not announce operators placed at the end of the line:

> *"If you put the dot at the end, it will not announce,* `filter dot`*. It will just announce* `filter`*. Because for JAWS, it's a full stop."* — P11 (punctuation set to *most*)
>
> *"It doesn't read the dash on* `dividends - qualified_dividends -` *So it's not reading the dashes but that's my punctuation settings. That's my own fault."* — P8 (punctuation set to *some*)

The above quotes reveal how the dot and the subtraction (announced as dash) operators are treated as if they are being used in a text document and not in a coding environment. P10 reasoned that characters like dot and dash are *"used for many purposes"*. For instance, he shared that not putting whitespaces around the dash operator also mutes its announcement, possibly because it implies a range (e.g., 15-10). Taking into account all of these scenarios is difficult and screen reader developers might have felt that *"not announcing them would make sense"* (P10) in *some* and *most* settings.

Lastly, single quote ('tick' on NVDA; 'apostrophe' on JAWS) was only announced when punctuation was set to *all*; double quote ('quote' on both NVDA and JAWS) was announced for *most* and *all* settings. We noted a strong preference for double quotes among participants because it required less disambiguation and was more likely to be announced. For instance, in the docstring example (Rule #1.1.2 in Appendix A), the screen reader did not announce the single quotes to participants who had not set their punctuation to *all*. They had to do character-level navigation to verify whether the line was indeed blank or it had characters relevant to code reading:

> *"I was sure something is there, but I couldn't read and I tried to go back. Then I understood there is an apostrophe, like single quotes [...] If it is not saying blank, there is something but it is not readable [to the screen reader]"* — P11

4.3 Impact of Navigation on Readability

We identified 5 kinds of navigation that participants used to skim and read the code in detail: (1) character-level, (2) word-level, (3) line-level, (4) entity-level, (5) editor's search feature. Prior work has investigated the latter two [9, 42, 56]. We are the first study to describe how the first three shaped readability.

4.3.1 Character-level navigation. The previous sections discussed how the lack of punctuation information or too much verbosity meant participants had to parse each character of a line to verify details such as spelling and use of special characters. Presence of whitespaces further slowed down participants by increasing the total characters they had to navigate. For the very reason, majority of our participants preferred tabs over spaces to indent code blocks in Python. They could *"go over a tab with just one press"* (P1) while spaces were four characters. Besides, whitespaces introduced verbosity at character-level navigation:

> *"I usually don't put spaces, because I think that kind of makes it more time consuming. It's going to keep saying 'space space and space' whatever."* — P2

However, participants agreed that whitespaces facilitated better word-level navigation (discussed next). A few participants mentioned that presence of whitespaces prevented over-editing or accidentally deleting characters by acting like buffer. Furthermore, whitespaces around mathematical operators improved the readability for their sighted colleagues, which they prioritized by either using whitespaces while authoring code or reformatting the code using a code formatter according to coding standards.

Figure 4: Options presented to understand use of whitespaces

4.3.2 *Word-level navigation.*

Participants shared that presence of whitespaces tended to improve word navigation by acting as *"word boundaries"* (P1). Some participants also shared that statements comprising slice operations were *"read slowly because of the spaces"* (P12) (see Listing 4). However, whitespaces could cause tensions with one's punctuation setting. For instance, with whitespaces present and the punctuation set to *some*, the screen reader did not announce the colon character. Thus, the operation performed in the statement was not communicated. But without the whitespaces, the colon ended up acting as the word boundary and was output by the screen reader, enabling participants to understand the operation without having to resort to character-level navigation:

> *"With my [punctuation] setting, if there's no space between the colon and the words, it is reading the colon as well the plus sign. So it's reading the entire thing properly. But in the first one, it's not announcing the colon symbol in 'some' setting, and it's treating it as a pause. 'Lower', then a pause, then 'upper'."* — P10

We noticed similar tension when snake case was used for variable names. Underscores acted as word boundaries and allowed participants to jump across individual words despite increasing typing effort and verbosity (when punctuation was set to 'most' or 'all'). P1 shared how he had started preferring camel case once he discovered an NVDA addon that enabled navigation just like snake case did:

> *"3–4 years ago someone made an NVDA addon called WordNav, which stops control arrows even in camel case. So in a word, it's not like you navigate with control-right/left arrows. With this addon, it stops after the first and second word even though they are not separated by anything"* — P1

In conclusion, while whitespaces made character-level navigation and typing slower, they improved word-level navigation by acting as boundaries between words. Punctuation and special characters could also act as boundaries but it depended on one's punctuation setting.

4.3.3 *Line-level navigation.*

We have already discussed how participants were able to pause at will when lengthy code lines were split. By down arrowing through code chunks, they were able to process the code better and avoid word-level or even character-level navigation. In this section, we discuss how the use of indentation and line breaks shaped overall navigation and code skimming.

NVDA allows four options for indentation reporting: (1) none, (2) tones where higher pitch implies greater indentation (3) speech (e.g., "twelve space" or "four tab"), (4) both speech and tones[1]. 5 participants did not use any indent reporting whereas 13 participants had it turned on (see Table 2). We noted that the choice of setting influenced participants' presentation choices for nested dictionaries but not so much for docstrings. Typically, participants used indentation reporting to *"visualize where the things are, how far in they are"* (P7). Thus, option 1 was more preferable for Rule #1.1.1. They could down arrow to key-value pairs at the same nested levels and navigate past heavily nested items using the audio cues:

> *"If the indentation is consistent, I could just skip past, like let's say there's a list in here. If I don't need that, I can just skip past that to the next block."* — P2

Participants who did not use indentation reporting were divided in their preferences. They compared the effort it took to write well-indented code with the improvements to readability. Usually, they wrote the code without indentation and formatted it later for the benefit of sighted developers. They felt the lack of announcements led to *"a lot of confusion when dealing with more nested structures"* (P14) but keeping it turned on interfered with other aspects of their work such as emails, document writing, etc. However, even without indent announcement, a few people preferred option 1 for nested data structures. The placement of parentheses on its own line clearly indicated the beginnings and ends of a nested level. P14 said she left small inline comments after each closing brace to serve as checkpoints. These helped her keep track of nested structures and helped her skim faster. Furthermore, the key bindings in code editors helped participants to jump quickly to opening or closing parentheses. Some participants used addons like IndentNav, which allowed skipping to statements that shared the same nesting level. In Python, it could be used to jump across entities, conditionals, and loops:

> *"NVDA has this add on called IndentNav, which basically just lets me navigate past code blocks. So sometimes when I'm skimming and if a block does something and I know what it does, I don't need to go in there, I'll just skip past the indentation. Go to the next block or whatever, skip past the loop and stuff like that. "*

Majority of the participants preferred no indentation in multiline docstrings irrespective of indent reporting. Since docstrings were similar in nature to comments, they were likely to be read only a handful of times. They preferred going through them quickly to get to the main body of the code. Even while writing docstrings, participants preferred spending as little time as possible in formatting the text compared to other aspects of source code. P14 mentioned using the autoDocstring plugin for VS Code, which provided placeholders for populating details about a class or function. The plugin not only ensured correct formatting but also saved her writing time.

5 DISCUSSION

Prior accessibility research has focused on communicating the information encoded in visual markup such as syntax highlighting, code structure, etc. Our research detaches the source code text from its visual appearance. We find that while it is vital to translate the information available in visual markup, the source code itself is not entirely available on screen readers. We answer RQ1 and show that the attributes of source code such as line length, spacing, etc. are warped in screen reader navigation and the programming environment, thereby shaping readability preferences. In this section, we update Table 1 from related work to move towards an inclusive taxonomy for code readability (see Table 3). We also make recommendations for programming tools and code standards, thereby answering RQ2.

[1]Only P11 and P14 used JAWS in our study. Both did not use indent reporting. They and a few other participants mentioned that JAWS does not offer indent reporting.

Factor	**Sub-Factor**	**Code Type**	**GUI**			**Screen Readers**			
			Recommended Practice	Skimming	Focused Reading	Recommended Practice	Non-Linear Skimming	Linear Skimming	Focused Reading
Spacing	Indentation	Nested Data Structures	Follow consistent indenting	Yes	No	Follow consistent indenting	Yes (with addons)	Yes (with indent reporting)	No
		Docstrings	Indent docstring arguments	Yes	No	Do not indent docstring arguments	N/A	Yes	No
	Separate Closing Parentheses	-	Separate parentheses and key-value pairs	N/A	N/A	Separate parentheses and key-value pairs	Yes	Yes	Yes
	Segmenting	-	Use 2 blank lines to separate entities	Yes	No	Use single blank line to separate entities	N/A	Yes	No
	Whitespaces	Slice and Math Operators	Surround operators with whitespaces	N/A	Yes	Surround operators with whitespaces	N/A	Yes	Yes
Identifiers	Word Boundaries	-	Use snake case	N/A	Same effect	Use camel case	N/A	N/A	Yes
	Length	-	Short variable names	N/A	Yes	Consider syllable count	N/A	Yes	Yes
	Intent of Use	-	Convey intent in either prefix or suffix	N/A	Yes	Use consistent suffixes	N/A	Yes	Yes
Line Length	-	Function Calls, Signatures, Chains	Render arguments on separate lines	Yes	Yes	Render arguments on separate lines	N/A	Yes	Yes
	-	Binary Operations	Place line break before the operator	Yes	Yes	Place line break before the operator or same line	N/A	Yes	Yes
	-	Comments	Wrap comments	Yes	Yes	Do not split comments	N/A	Yes	Yes
	-	Imports	Place imports on different lines	Yes	Yes	Either is fine	N/A	Yes	Yes
String Quotes	Quote Character	-	Use quotes consistently	N/A	N/A	Use double quotes	N/A	Yes	Yes

Table 3: Taxonomy for Code Readability on GUIs and Screen Readers

5.1 Moving Towards an Inclusive Taxonomy for Code Readability

5.1.1 Line Length. Splitting long lines (*e.g.*, function chains, function signatures, etc.) helps both sighted and BVI developers. Sighted developers do not need to horizontally scroll; BVI developers have to process smaller chunks as they read the code. It also improves their navigation experience. They need not listen to the entire line before moving to the following line. It is worth pointing out that sighted developers can toggle on word wrapping, which prevents horizontal scrolling. However, word wrapping produces no effect on BVI developers. In fact, the feature is disabled in IDEs like VS Code if it detects the screen reader [33]. Either IDEs should enable an equivalent audio wrapping for screen readers, or they should offer settings to enforce code splitting consistently.

Syntax highlighting helps sighted developers identify regions of interest [54]. Researchers have attempted to use audio cues to communicate the visual cues available to sighted developers in code editors. However, audio cues take time to memorize [30]. We find that audio cues for indent reporting also interfere with tasks of emailing, document editing, etc for BVI developers. Our findings also show that the type of code matters. Everyone preferred splitting call chains, the opinion was divided on breaking function signatures, and long imports and comments were least likely to affect readability. These findings help us decide what programming constructs should be highlighted using audio.

5.1.2 Programming Environment and Screen Reader Settings. The manner in which code is written interacts with screen reader settings and affects output. Consider the example where we asked participants whether they prefer inserting line breaks before or after the binary operator. We found that developers were likely to miss operators at the end of lines when skimming too fast or if the punctuation setting was set to *most* or *some*. Similarly, screen readers did not announce single quotes in less granular punctuation settings; using double quotes to quote string variables and docstrings was better. Lastly, collaborators may capitalize names differently (e.g., `API` vs. `Api`), changing the pronunciation entirely on screen readers. These differences do not affect sighted developers – a quote character is read and interpreted as a quote, missing operators are easy to catch, and `API` and `Api` are visually processed the same way. While BVI developers pick up on the code styling preferences of sighted developers easily, sighted developers do not reciprocate similar awareness. We recommend incorporating the readability preferences of BVI developers in code styling guidelines, such as PEP8 [63]. For instance, the above examples can be used to educate the larger programming community about how screen readers may announce different code snippets.

We find that code editor plugins and screen reader addons can greatly reduce typing effort while improving readability. For instance, P14 was among the few participants who did not mind indenting docstrings because she used the autoDocstring plugin. Similarly, participants who used addons like IndentNav achieved more efficient non-linear navigation. IDEs like VS Code were more popular because of their accessibility features and ability to apply consistent indentation, parentheses, and quoting. Such plugins and features improved readability *as* one wrote the code and *not* after the fact by requiring the use of code formatters. We recommend that teams designing code editors should explore ways to extend the programming environment. The work can be abstracted out at several levels. For examples, JAWS currently does not support indent reporting but IDEs could offer indent reporting through their plugins. IDEs could also provide quick toggles between coding standards suitable for collaboration as well as for personal readability.

Prior research is divided on the use of snake case and camel case for sighted developers [14, 52]. PEP8 recommends snake case for variables [63]. But an overwhelming majority of our participants preferred camel case over snake case for verbosity reasons. We also show that developers ought to consider the syllable count when shortening variable names (*e.g.*, `button` instead of `btn`; `checkbox` instead of `chkBx`) to avoid verbose output. Lastly, developers are encouraged to create meaningful variable names by encoding the intent. In such cases, sighted developers should consider where to place the word representing the intent (e.g, `menuColorForeground` vs. `foregroundColorMenu`) to ensure ease of remembrance and code skimming. The decisions should be informed by categories of variables (e.g., foreground colors, background colors, etc), the total number of variables in the code, and the number of words composing the identifier.

5.1.3 *Granularity of Navigation.*

We add to the prior empirical studies on code navigation with screen readers [5, 9, 44]. We find that high verbosity and ambiguous announcement of special characters forces people to perform word-level and character-level navigation. These are slower forms of navigation, which impede code reading. Ideally, the lexicon and the layout of the code should be such that it can be understood by line-level navigation. Lines should be chunked such that they are easy to process while reading and easy to recall while navigating backwards. The findings have implications for programming language design. For instance, using complex and verbose keywords can force people to stop skimming and look at the line more closely.

Our finding contradicts past finding on nested code structures [5]. We find that participants used indentation for navigation, which was further improved through the use of screen reader addons. We further find that separating parentheses (Option 1 of Rule # 1.1.1) instead of grouping multiple parentheses together (Option 2 of Rule # 1.1.1) is more useful in jumping nested code blocks. Lastly, we find that tabs are better than spaces for indentation because they improve character-level navigation.

When it comes to vertical spacing or segmentation, PEP8 recommends keeping 2 blank lines between entities [63]. While a few participants preferred 2 blank lines to delineate between code blocks, most preferred a single line to reduce linear navigation. We believe the choice of vertical spacing can be left up to BVI developers. Code editors could provide shortcuts to reduce blank lines if they detect screen readers to facilitate efficient line-level navigation.

Lack of whitespaces around operators makes the code less readable for sighted developers. For BVI developers, the statement may get read without discernible pauses and may affect word-level navigation due to poor separation of words. Thus, surrounding operators with whitespaces is useful for both groups. Code editors could provide mechanisms to reformat selected group of statements to reduce the typing effort for BVI developers, which they described as the primary reason that deters them from using whitespaces while authoring code.

5.2 Limitations and Future Work

We studied participants' preferences for one programming language (Python). Because readability preferences must exist within languages' syntactic rules, some of our findings might not generalize to other programming languages. For example, in Python, indentation is an enforced part of the syntax that carries semantic meaning. This is in contrast with most other languages, where indentation is a stylistic choice that can be customized according to developers' preferences. Python is also closer to English, with some calling it executable pseudocode [20]. However, most of our findings relate to syntactic elements that are common across many widely-used programming languages and thus could be generally applicable. However, future work could contrast our results with languages closer to C-style syntax that have been reported to present barriers to novice programmers [59].

Participants actual coding practices may differ and are likely to be shaped by existing coding standards and collaboration. Thus, there may be differences in our findings and preferences one may observe in mixed-ability teams.

The remote nature of our study prevented us from observing code reading on braille displays. Only 3 participants reported using braille displays but they did not use them during the study. In future work, we would examine the factors that constitute code readability on braille displays and pin-matrix tactile displays that even display 2D graphics [16]. Furthermore, visual impairments exist on a spectrum. We did not analyze how the nature of visual impairment and its onset correlates with participants' preferences. Consistent with prior accessibility research, instead of focusing on the visual impairment we have derived our recommendations for assistive technologies and programming languages [40, 43].

Despite our efforts, our study sample was heavily skewed towards men and fell within a narrow age range, likely due to the lack of equitable gender and age representation in the software engineering field [37, 38]. Its effect is amplified for BVI women and non-binary developers, who are also marginalized due to ableism and accessibility barriers.

6 CONCLUSION

Code editors and IDEs provide features such as syntax highlighting, vertical rulers, etc., to support code skimming and focused reading among sighted developers. However, we do not know what constitutes good code readability for BVI developers. We conducted an exploratory qualitative study with 16 BVI developers. We presented them with two differently formatted options for 15 functionally equivalent Python code snippets and asked them to choose the option that improved code readability for them. The snippets were created to investigate the effect of indentation, line length, identifier names, and quotation characters. We found similarities and differences in how these factors shaped the readability of BVI and sighted developers. Based on the findings, we contribute an inclusive taxonomy for code readability that considers code reading on GUIs and screen readers.

ACKNOWLEDGMENTS

This study would not have been possible without our participants. We are grateful to them for sharing their experiences and insights with us. We also thank Sile O'Modhrain and Hrishikesh Rao for their time and feedback at various points in this research. The work was supported by a gift from Google.

REFERENCES

[1] K.K. Aggarwal, Y. Singh, and J.K. Chhabra. 2002. An integrated measure of software maintainability. In *Annual Reliability and Maintainability Symposium. 2002 Proceedings (Cat. No.02CH37318)*. IEEE Press, Seattle, USA, 235–241. https://doi.org/10.1109/RAMS.2002.981648

[2] Faisal Ahmed, Yevgen Borodin, Andrii Soviak, Muhammad Islam, I.V. Ramakrishnan, and Terri Hedgpeth. 2012. Accessible skimming: faster screen reading of web pages. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology* (Cambridge, Massachusetts, USA) *(UIST '12)*. Association for Computing Machinery, New York, NY, USA, 367–378. https://doi.org/10.1145/2380116.2380164

[3] AirBnb. 2022. Airbnb React/JSX Style Guide. https://airbnb.io/javascript/react/

[4] Khaled Albusays and Stephanie Ludi. 2016. Eliciting programming challenges faced by developers with visual impairments: exploratory study. In *Proceedings of the 9th International Workshop on Cooperative and Human Aspects of Software Engineering* (Austin, Texas) *(CHASE '16)*. Association for Computing Machinery, New York, NY, USA, 82–85. https://doi.org/10.1145/2897586.2897616

[5] Khaled Albusays, Stephanie Ludi, and Matt Huenerfauth. 2017. Interviews and observation of blind software developers at work to understand code navigation challenges. In *Proceedings of the 19th International ACM SIGACCESS Conference on Computers and Accessibility*. Association for Computing Machinery, New York, NY, USA, 91–100.

[6] Ameer Armaly and Collin McMillan. 2016. An empirical study of blindness and program comprehension. In *Proceedings of the 38th International Conference on Software Engineering Companion* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 683–685. https://doi.org/10.1145/2889160.2891041

[7] Ameer Armaly, Paige Rodeghero, and Collin McMillan. 2018. A comparison of program comprehension strategies by blind and sighted programmers. In *Proceedings of the 40th International Conference on Software Engineering*. Association for Computing Machinery, New York, NY, USA, 788–788.

[8] R. Baecker. 1988. Enhancing program readability and comprehensibility with tools for program visualization. In *Proceedings of the 10th International Conference on Software Engineering* (Singapore) *(ICSE '88)*. IEEE Computer Society Press, Washington, DC, USA, 356–366.

[9] Catherine M. Baker, Lauren R. Milne, and Richard E. Ladner. 2015. StructJumper: A Tool to Help Blind Programmers Navigate and Understand the Structure of Code. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems* (Seoul, Republic of Korea) *(CHI '15)*. Association for Computing Machinery, New York, NY, USA, 3043–3052. https://doi.org/10.1145/2702123.2702589

[10] Mark S Baldwin, Jennifer Mankoff, Bonnie Nardi, and Gillian Hayes. 2020. An activity centered approach to nonvisual computer interaction. *ACM Transactions on Computer-Human Interaction (TOCHI)* 27, 2 (2020), 1–27.

[11] Jennifer Bauer, Janet Siegmund, Norman Peitek, Johannes C. Hofmeister, and Sven Apel. 2019. Indentation: simply a matter of style or support for program comprehension?. In *Proceedings of the 27th International Conference on Program Comprehension (ICPC '19)*. IEEE Press, Montreal, Quebec, Canada, 154–164. https://doi.org/10.1109/ICPC.2019.00033

[12] Jeffrey P. Bigham, Irene Lin, and Saiph Savage. 2017. The Effects of "Not Knowing What You Don't Know" on Web Accessibility for Blind Web Users. In *Proceedings of the 19th International ACM SIGACCESS Conference on Computers and Accessibility* (Baltimore, Maryland, USA) *(ASSETS '17)*. Association for Computing Machinery, New York, NY, USA, 101–109. https://doi.org/10.1145/3132525.3132533

[13] Dave Binkley, Marcia Davis, Dawn Lawrie, Jonathan I Maletic, Christopher Morrell, and Bonita Sharif. 2013. The impact of identifier style on effort and comprehension. *Empirical software engineering* 18 (2013), 219–276.

[14] Dave Binkley, Marcia Davis, Dawn Lawrie, and Christopher Morrell. 2009. To camelcase or under_score. In *2009 IEEE 17th International Conference on Program Comprehension*. IEEE, Vancouver, BC, Canada, 158–167.

[15] Barry Boehm and Victor R Basili. 2001. Defect reduction top 10 list. *Computer* 34, 1 (2001), 135–137.

[16] Jens Bornschein, Denise Bornschein, and Gerhard Weber. 2018. Blind Pictionary: Drawing Application for Blind Users. In *Extended Abstracts of the 2018 CHI Conference on Human Factors in Computing Systems* (Montreal, QC, Canada) *(CHI EA '18)*. Association for Computing Machinery, New York, NY, USA, 1–4. https://doi.org/10.1145/3170427.3186487

[17] Raymond PL Buse and Westley R Weimer. 2009. Learning a metric for code readability. *IEEE Transactions on software engineering* 36, 4 (2009), 546–558.

[18] Mitchell H Clifton. 1978. A technique for making structured programs more readable. *ACM Sigplan Notices* 13, 4 (1978), 58–63.

[19] Lionel E Deimel Jr. 1985. The uses of program reading. *ACM SIGCSE Bulletin* 17, 2 (1985), 5–14.

[20] Charles Dierbach. 2014. Python as a first programming language. *Journal of Computing Sciences in Colleges* 29, 3 (2014), 73–73.

[21] Rodrigo Magalhães dos Santos and Marco Aurélio Gerosa. 2018. Impacts of coding practices on readability. In *Proceedings of the 26th Conference on Program Comprehension* (Gothenburg, Sweden) *(ICPC '18)*. Association for Computing Machinery, New York, NY, USA, 277–285. https://doi.org/10.1145/3196321.3196342

[22] Carolyn D Egelman, Emerson Murphy-Hill, Elizabeth Kammer, Maggie Morrow Hodges, Collin Green, Ciera Jaspan, and James Lin. 2020. Pushback: Characterizing and Detecting Negative Interpersonal Interactions in Code Review. In *Proceedings of the 42nd International Conference on Software Engineering*. IEEE/ACM, 174–185.

[23] James L Elshoff and Michael Marcotty. 1982. Improving computer program readability to aid modification. *Commun. ACM* 25, 8 (1982), 512–521.

[24] Sarah Fakhoury, Yuzhan Ma, Venera Arnaoudova, and Olusola Adesope. 2018. The effect of poor source code lexicon and readability on developers' cognitive load. In *Proceedings of the 26th Conference on Program Comprehension* (Gothenburg, Sweden) *(ICPC '18)*. Association for Computing Machinery, New York, NY, USA, 286–296. https://doi.org/10.1145/3196321.3196347

[25] Joan M. Francioni and Ann C. Smith. 2002. Computer science accessibility for students with visual disabilities. *SIGCSE Bull.* 34, 1 (Feb 2002), 91–95. https://doi.org/10.1145/563517.563372

[26] Freedom Scientific. 2022. *JAWS for Windows*. Vispero. https://www.freedomscientific.com/products/software/jaws/

[27] Google. 2022. ESLint shareable config for the Google JavaScript style guide. https://github.com/google/eslint-config-google

[28] Robert Green and Henry Ledgard. 2011. Coding guidelines: Finding the art in the science. *Commun. ACM* 54, 12 (2011), 57–63.

[29] Nuzhat J Haneef. 1998. Software documentation and readability: a proposed process improvement. *ACM SIGSOFT Software Engineering Notes* 23, 3 (1998), 75–77.

[30] Joe Hutchinson and Oussama Metatla. 2018. An Initial Investigation into Non-visual Code Structure Overview Through Speech, Non-speech and Spearcons. In *Extended Abstracts of the 2018 CHI Conference on Human Factors in Computing Systems* (Montreal, QC, Canada) *(CHI EA '18)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/3170427.3188696

[31] Tom Love. 1977. An experimental investigation of the effect of program structure on program understanding. *ACM SIGSOFT Software Engineering Notes* 2, 2 (1977), 105–113.

[32] Richard J Miara, Joyce A Musselman, Juan A Navarro, and Ben Shneiderman. 1983. Program indentation and comprehensibility. *Commun. ACM* 26, 11 (1983), 861–867.

[33] Microsoft. 2020. Accessibility in Visual Studio Code. https://code.visualstudio.com/docs/editor/accessibility#_screen-readers

[34] Matthew Miles, A. Michael Huberman, and Michael Saldaña. 2013. *Qualitative Data Analysis: A Methods Sourcebook*. Sage Publications, Thousand Oaks, CA.

[35] NV Access. 2022. *Nonvisual Desktop Access*. NV Access. https://www.nvaccess.org/

[36] Delano Oliveira, Reydne Santos, Fernanda Madeiral, Hidehiko Masuhara, and Fernando Castor. 2023. A systematic literature review on the impact of formatting elements on code legibility. *Journal of Systems and Software* 203 (2023), 111728.

[37] Stack Overflow. 2021. Stack overflow developer survey 2021. https://insights.stackoverflow.com/survey/2021

[38] Stack Overflow. 2022. Stack Overflow Developer Survey 2022. https://survey.stackoverflow.co/2022/#technology

[39] Stack Overflow. 2023. Stack Overflow Developer Survey 2023. https://survey.stackoverflow.co/2023/#technology

[40] Maulishree Pandey, Sharvari Bondre, Sile O'Modhrain, and Steve Oney. 2022. Accessibility of UI Frameworks and Libraries for Programmers with Visual Impairments. In *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE Press, Rome, Italy, 1–10.

[41] Maulishree Pandey, Vaishnav Kameswaran, Hrishikesh V. Rao, Sile O'Modhrain, and Steve Oney. 2021. Understanding Accessibility and Collaboration in Programming for People with Visual Impairments. *Proc. ACM Hum.-Comput. Interact.* 5, CSCW1, Article 129 (apr 2021), 30 pages. https://doi.org/10.1145/3449203

[42] Vanessa Petrausch and Claudia Loitsch. 2017. Accessibility analysis of the eclipse ide for users with visual impairment. In *Harnessing the Power of Technology to Improve Lives*. IOS Press, 922–929.

[43] Venkatesh Potluri, Maulishree Pandey, Andrew Begel, Michael Barnett, and Scott Reitherman. 2022. CodeWalk: Facilitating Shared Awareness in Mixed-Ability Collaborative Software Development. In *Proceedings of the 24th International ACM SIGACCESS Conference on Computers and Accessibility* (Athens, Greece) *(ASSETS '22)*. Association for Computing Machinery, New York, NY, USA, Article 20, 16 pages. https://doi.org/10.1145/3517428.3544812

[44] Venkatesh Potluri, Priyan Vaithilingam, Suresh Iyengar, Y. Vidya, Manohar Swaminathan, and Gopal Srinivasa. 2018. CodeTalk: Improving Programming Environment Accessibility for Visually Impaired Developers. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (Montreal QC, Canada) *(CHI '18)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/3173574.3174192

[45] Program-l. 2023. Program-l: V.I. Programmers Discussion List. https://www.freelists.org/archive/program-l/

[46] Darrell R Raymond. 1991. Reading source code.. In *CASCON*, Vol. 91. 3–16.

[47] Phillip A Relf. 2005. Tool assisted identifier naming for improved software readability: an empirical study. In *2005 International Symposium on Empirical Software Engineering, 2005.* IEEE, Noosa Heads, QLD, Australia, 10–pp.

[48] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. 2018. Modern code review: a case study at google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice* (Gothenburg, Sweden) *(ICSE-SEIP '18)*. Association for Computing Machinery, New York, NY, USA, 181–190. https://doi.org/10.1145/3183519.3183525

[49] Isabel Braga Sampaio and Luís Barbosa. 2016. Software readability practices and the importance of their teaching. In *2016 7th International Conference on Information and Communication Systems (ICICS)*. IEEE Press, Irbid, Jordan, 304–309.

[50] Jaime Sánchez and Fernando Aguayo. 2005. Blind learners programming through audio. In *CHI '05 Extended Abstracts on Human Factors in Computing Systems* (Portland, OR, USA) *(CHI EA '05)*. Association for Computing Machinery, New York, NY, USA, 1769–1772. https://doi.org/10.1145/1056808.1057018

[51] Simone Scalabrino, Gabriele Bavota, Christopher Vendome, Mario Linares-Vásquez, Denys Poshyvanyk, and Rocco Oliveto. 2017. Automatically assessing code understandability: How far are we?. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Urbana, IL, USA, 417–427.

[52] Bonita Sharif and Jonathan I Maletic. 2010. An eye tracking study on camelcase and under_score identifier styles. In *2010 IEEE 18th International Conference on Program Comprehension*. IEEE Press, Braga, Portugal, 196–205.

[53] Ben Shneiderman and Don McKay. 1976. Experimental Investigations of Computer Program Debugging and Modification. *Proceedings of the Human Factors Society Annual Meeting* 20, 24 (1976), 557–563. https://doi.org/10.1177/154193127602002401 arXiv:https://doi.org/10.1177/154193127602002401

[54] Janet Siegmund, Norman Peitek, Chris Parnin, Sven Apel, Johannes Hofmeister, Christian Kästner, Andrew Begel, Anja Bethmann, and André Brechmann. 2017. Measuring neural efficiency of program comprehension. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 140–150. https://doi.org/10.1145/3106237.3106268

[55] Charles Simonyi. 1999. Hungarian notation.

[56] Ann C. Smith, Justin S. Cook, Joan M. Francioni, Asif Hossain, Mohd Anwar, and M. Fayezur Rahman. 2003. Nonvisual tool for navigating hierarchical structures. In *Proceedings of the 6th International ACM SIGACCESS Conference on Computers and Accessibility* (Atlanta, GA, USA) *(Assets '04)*. Association for Computing Machinery, New York, NY, USA, 133–139. https://doi.org/10.1145/1028630.1028654

[57] Diomidis Spinellis. 2003. Reading, Writing, and Code: The key to writing readable code is developing good coding style. *Queue* 1, 7 (2003), 84–89.

[58] Andreas Stefik. 2008. On the design of program execution environments for non-sighted computer programmers. *Washington State University* (2008).

[59] Andreas Stefik and Susanna Siebert. 2013. An empirical investigation into programming language syntax. *ACM Transactions on Computing Education (TOCE)* 13, 4 (2013), 1–40.

[60] Floyd Sykes, Raymond T Tillman, and Ben Shneiderman. 1983. The effect of scope delimiters on program comprehension. *Software: Practice and Experience* 13, 9 (1983), 817–824.

[61] Yahya Tashtoush, Zeinab Odat, Izzat M Alsmadi, and Maryan Yatim. 2013. Impact of programming features on code readability. *International Journal of Software Engineering and its Applications* 7 (2013), 441–458.

[62] Ted Tenny. 1988. Program readability: Procedures versus comments. *IEEE Transactions on Software Engineering* 14, 9 (1988), 1271–1279.

[63] Guido van Rossum, Nick Coghlan, and Barry Warsaw. 2001. PEP 8 – Style Guide for Python Code. https://peps.python.org/pep-0008/

[64] Xiaoran Wang, Lori Pollock, and K Vijay-Shanker. 2011. Automatic segmentation of method code into meaningful blocks to improve readability. In *2011 18th Working Conference on Reverse Engineering*. IEEE, Limerick, Ireland, 35–44.

[65] Silvia Zuffi, Carla Brambilla, Giordano Beretta, and Paolo Scala. 2007. Human computer interaction: Legibility and contrast. In *14th International Conference on Image Analysis and Processing (ICIAP 2007)*. IEEE, Modena, Italy, 241–246.

A READABILITY RULES

This section contains code rules and snippets correspond to the factors detailed in Table 1. Next section shows a sample markdown presented to one of the participants.

~~~~
# Code Formatting Rules

## 1. Spacing

### 1.1 Indentation

#### 1.1.1 Nested Data Structures

Option 1: Keep parentheses and key-value pairs on separate lines
```
{
  "menu": {
    "id": "file",
    "value": "File",
    "popup": {
      "menuitem": [
        {
          "value": "New",
          "onclick": "CreateNewDoc()"
        },
        {
          "value": "Open",
          "onclick": "OpenDoc()"
        },
        {
          "value": "Close",
          "onclick": "CloseDoc()"
        }
      ]
    }
  }
}
```
Option 2: Match key-value pairs and parentheses
```
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]}
}}
```

#### 1.1.2 Multiline docstrings

Option 1: Doctring is not indented
```
def add_binary(a, b):
    '''
    Returns the sum of two decimal numbers in binary digits.

    Parameters:
    a (int): A decimal integer
    b (int): Another decimal integer

    Returns: binary_sum (str): Binary string of the
 sum of a and b
    '''
    binary_sum = bin(a+b)[2:]
    return binary_sum
```

Option 2: Doctring is indented
```
def add_binary(a, b):
    '''
    Returns the sum of two decimal numbers in binary digits.

        Parameters:
                a (int): A decimal integer
~~~~

~~~
                b (int): Another decimal integer

        Returns:
                binary_sum (str): Binary string of the
sum of a and b
    '''
    binary_sum = bin(a+b)[2:]
    return binary_sum
```


### 1.2 Segmenting

#### 1.2.1 Line breaks in source code

Option 1: Use double empty lines to separate
functions, conditionals, and classes
```
def factorial(num):
    fact = 1
    for i in range(1, num+1):
        fact = fact * i
    return fact


if condition:
    print("This condition was TRUE")


class Point:
    x: int
    y: int
```

Option 2: Use single empty lines between functions, conditionals,
and classes
```
def factorial(num):
    fact = 1
    for i in range(1, num+1):
        fact = fact * i
    return fact

if condition:
    print("This condition was TRUE")

class Point:
    x: int
    y: int
```


### 1.3 Whitespaces

#### 1.3.1 Whitespaces in operators

Option 1: Avoid whitespaces before and after operators
```
b = config.base**5.2
submitted+=1
hypot2 = x*x+y*y
```

Option 2: Surround operators with whitespaces
```
b = config.base ** 5.2
submitted += 1
hypot2 = x*x + y*y
```

#### 1.3.2 Whitespace in slice operators

Option 1: Use whitespaces
```
ham[lower : upper + offset]
```

Option 2: Avoid whitespaces
```
ham[lower:upper+offset]
```


## 2. Identifiers

### 2.1 Naming style for variables

Option 1: Use snake case
```
primary_address_apartment = ""
```

Option 2: Use camel case
```
primaryAddressApartment = ""
```


### 2.2 Length preference for variable names

Option 1: Long names
```
radioButtonHeight = "20"
```

Option 2: Short names
```
radioBtnHt = "20"
```


### 2.3 Consistency in variable names

Option 1: Use consistent prefixes
```
foregroundColorMenu = ""
foregroundColorBody = ""
foregroundColorFooter = ""
```

Option 2: Use consistent suffixes
```
menuForegroundColor = ""
bodyForegroundColor = ""
footerForegroundColor = ""
```


## 3. Line Length

### 3.0.1 Formatting function calls

Option 1: Render arguments on the same line
```
ImportantClass.important_method(exc, limit, lookup_lines, capture_locals,
extra_argument)
```

Option 2: Render arguments on separate lines
```
ImportantClass.important_method(
    exc,
    limit,
    lookup_lines,
    capture_locals,
    extra_argument
)
```


### 3.0.2 Formatting function signatures

Option 1: Render arguments on separate lines
```
# Applies `variables` to the `template` and writes to `file`
def very_important_function(
~~~

~~~
    template: str,
    *variables,
    file: os.PathLike,
    engine: str,
    header: bool = True,
    debug: bool = False,
):
    with open(file, 'w') as f:
    ...
```

Option 2: Render arguments on the same line
```
# Applies `variables` to the `template` and writes to `file`
def very_important_function(template: str, *variables, file: os.PathLike,
engine: str, header: bool = True, debug: bool = False):
    with open(file, 'w') as f:
    ...
```


### 3.0.3 Call chains

Option 1: Treat dot operator as a delimiter
```
def example(session):
    result = (
        session.query(models.Customer.id)
        .filter(models.Customer.account_id == account_id)
        .order_by(models.Customer.id.asc())
        .all()
    )
```

Option 2: Do not treat dot operator as a delimiter
```
def example(session):
   result = (session.query(models.Customer.id).filter(models.Customer.
   account_id == account_id).order_by(models.Customer.id.asc()).all())
```


### 3.0.4 Line breaks with binary operators

Option 1: Place line break after the operator
```
income = (gross_wages +
          taxable_interest +
          (dividends - qualified_dividends) -
          ira_deduction -
          student_loan_interest)
```

Option 2: Place operator after the line break
```
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)
```


### 3.0.5 Comments

Option 1: Wrap comments across lines
```
from collections import defaultdict

def get_top_cities(prices):
    top_cities = defaultdict(int)

    # Count number of times the city was searched for each price range,
    # get the top 3 cities, and add to dictionary
    return dict(top_cities)
```

Option 2: Do not wrap comments
```
from collections import defaultdict

def get_top_cities(prices):
    top_cities = defaultdict(int)

   # Count number of times the city was searched for each price range, get
    the top 3 cities, and add to dictionary
    return dict(top_cities)
```


### 3.0.6 Imports

Option 1: Place imports on different lines
```
import os
import sys
import random
import json
```

Option 2: Place imports on the same line
```
import os, sys, random, json
```


## 4. String Quotes

### 4.1 Use of quotation marks in docstrings

Option 1: Use single quotation marks
```
def square(n):
    '''Takes in a number n, returns the square of n'''
    return n**2
```

Option 2: Use double quotation marks
```
def square(n):
    """Takes in a number n, returns the square of n"""
    return n**2
```
~~~

B STUDY STIMULUS

This shows a markdown with order of rules and options randomized. The markdown was given to one of the participants as part of the study.

~~~
# Code Formatting Rules

## 1. Length preference for variable names

Option 1: Long names
```
radioButtonHeight = "20"
```

Option 2: Short names
```
radioBtnHt = "20"
```

## 2. Consistency in variable names

Option 1: Use consistent prefixes
```
foregroundColorMenu = ""
foregroundColorBody = ""
foregroundColorFooter = ""
```

Option 2: Use consistent suffixes
```
menuForegroundColor = ""
bodyForegroundColor = ""
footerForegroundColor = ""
~~~

~~~
```

## 3. Whitespace in slice operators

Option 1: Use whitespaces
```
ham[lower : upper + offset]
```

Option 2: Avoid whitespaces
```
ham[lower:upper+offset]
```

## 4. Line breaks with binary operators

Option 1: Place operator after the line break
```
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)
```

Option 2: Place line break after the operator
```
income = (gross_wages +
          taxable_interest +
          (dividends - qualified_dividends) -
          ira_deduction -
          student_loan_interest)
```

## 5. Formatting function signatures

Option 1: Render arguments on the same line
```
# Applies `variables` to the `template` and writes to `file`
def very_important_function(template: str, *variables,
 file: os.PathLike,
engine: str, header: bool = True, debug: bool = False):
    with open(file, 'w') as f:
    ...
```

Option 2: Render arguments on separate lines
```
# Applies `variables` to the `template` and writes to `file`
def very_important_function(
    template: str,
    *variables,
    file: os.PathLike,
    engine: str,
    header: bool = True,
    debug: bool = False,
):
    with open(file, 'w') as f:
    ...
```

## 6. Naming style for variables

Option 1: Use snake case
```
primary_address_apartment = ""
```

Option 2: Use camel case
```
primaryAddressApartment = ""
```

## 7. Imports

Option 1: Place imports on the same line
```
import os, sys, random, json
```

Option 2: Place imports on different lines
```
import os
import sys
import random
import json
```

## 8. Use of quotation marks in docstrings

Option 1: Use double quotation marks
```
def square(n):
    """Takes in a number n, returns the square of n"""
    return n**2
```

Option 2: Use single quotation marks
```
def square(n):
    '''Takes in a number n, returns the square of n'''
    return n**2
```

## 9. Line breaks in source code

Option 1: Use double empty lines to separate functions, conditionals,
and classes
```
def factorial(num):
    fact = 1
    for i in range(1, num+1):
        fact = fact * i
    return fact


if condition:
    print("This condition was TRUE")


class Point:
    x: int
    y: int
```

Option 2: Use single empty lines between functions, conditionals, and
classes
```
def factorial(num):
    fact = 1
    for i in range(1, num+1):
        fact = fact * i
    return fact

if condition:
    print("This condition was TRUE")

class Point:
    x: int
    y: int
```

## 10. Formatting function calls

Option 1: Render arguments on the same line
```
ImportantClass.important_method(exc, limit, lookup_lines, capture_locals,
extra_argument)
```

Option 2: Render arguments on separate lines
```
ImportantClass.important_method(
    exc,
    limit,
    lookup_lines,
~~~

~~~
    capture_locals,
    extra_argument
)
```

## 11. Splitting parentheses

Option 1: Keep parentheses and key-value pairs on separate lines
```
{
  "menu": {
    "id": "file",
    "value": "File",
    "popup": {
      "menuitem": [
        {
          "value": "New",
          "onclick": "CreateNewDoc()"
        },
        {
          "value": "Open",
          "onclick": "OpenDoc()"
        },
        {
          "value": "Close",
          "onclick": "CloseDoc()"
        }
      ]
    }
  }
}
```

Option 2: Match key-value pairs and parentheses
```
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]}
}}
```

## 12. Multiline docstrings

Option 1: Doctring is indented
```
def add_binary(a, b):
    '''
    Returns the sum of two decimal numbers in binary digits.

        Parameters:
                a (int): A decimal integer
                b (int): Another decimal integer

        Returns:
                binary_sum (str): Binary string of the sum of a and b
    '''
    binary_sum = bin(a+b)[2:]
    return binary_sum
```

Option 2: Doctring is not indented
```
def add_binary(a, b):
    '''
    Returns the sum of two decimal numbers in binary digits.

    Parameters:
    a (int): A decimal integer
    b (int): Another decimal integer

    Returns: binary_sum (str): Binary string of the sum of a and b
    '''
    binary_sum = bin(a+b)[2:]
    return binary_sum
```

## 13. Comments

Option 1: Do not wrap comments
```
from collections import defaultdict

def get_top_cities(prices):
    top_cities = defaultdict(int)

    # Count number of times the city was searched for each price range,
    get the top 3 cities, and add to dictionary
    return dict(top_cities)
```

Option 2: Wrap comments across lines
```
from collections import defaultdict

def get_top_cities(prices):
    top_cities = defaultdict(int)

    # Count number of times the city was searched for each price range,
    # get the top 3 cities, and add to dictionary
    return dict(top_cities)
```

## 14. Call chains

Option 1: Treat dot operator as a delimiter
```
def example(session):
    result = (
        session.query(models.Customer.id)
        .filter(models.Customer.account_id == account_id)
        .order_by(models.Customer.id.asc())
        .all()
    )
```

Option 2: Do not treat dot operator as a delimiter
```
def example(session):
   result = (session.query(models.Customer.id).filter(models.Customer.
account_id == account_id).order_by(models.Customer.id.asc()).all())
```

## 15. Whitespaces in operators

Option 1: Surround operators with whitespaces
```
b = config.base ** 5.2
submitted += 1
hypot2 = x*x + y*y
```

Option 2: Avoid whitespaces before and after operators
```
b = config.base**5.2
submitted+=1
hypot2 = x*x+y*y
```
~~~