Extending Support for Analyzing Eye Tracking Studies on Python Source Code in iTrace

Joshua A.C. Behler
Kent State University
Kent, Ohio, USA
jbehler1@kent.edu

Zachary Kozak
University of Nebraska-Lincoln
Lincoln, Nebraska, USA
zkozak2@unl.edu

Kang-il Park
University of Nebraska-Lincoln
Lincoln, Nebraska, USA
kangil.park@huskers.unl.edu

Bonita Sharif
University of Nebraska-Lincoln
Lincoln, Nebraska, USA
bsharif@unl.edu

Jonathan I. Maletic
Kent State University
Kent, Ohio, USA
jmaletic@kent.edu

ABSTRACT

The work presents a small pilot study of developers reading Python programs in the iTrace eye-tracking infrastructure. The main objective is to understand how adding new languages to srcML impacts the usage of iTrace. iTrace uses srcML to extract syntactic information from the source code being examined. This allows iTrace to automatically determine regions of interest (ROIs) and drastically reduce the amount of time and effort required to process the collected eye tracking data. The srcML infrastructure has a beta version to support Python and which allows iTrace to fully support analysis of eye tracking studies on Python source code. This work demonstrates the viability of supporting new languages in iTrace.

CCS CONCEPTS

• **Software and its engineering**; • **Human-centered computing** → **Empirical studies in HCI**;

KEYWORDS

eye-tracking python

ACM Reference Format:
Joshua A.C. Behler, Zachary Kozak, Kang-il Park, Bonita Sharif, and Jonathan I. Maletic. 2025. Extending Support for Analyzing Eye Tracking Studies on Python Source Code in iTrace. In *2025 Symposium on Eye Tracking Research and Applications (ETRA '25), May 26–29, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3715669.3725867

1 INTRODUCTION

iTrace is a powerful suite of tools for performing eye-tracking studies on software while using an IDE. The iTrace infrastructure [Guarnera et al. 2018] is used by researchers to conduct studies focused on the comprehension of source code [Abbad-Andaloussi et al. 2022; Abid et al. 2019a,b; Bansal et al. 2024; Martins et al. 2024; Park et al. 2023; Peterson et al. 2019; Yoshioka and Uwano 2024; Zyrianov et al. 2022]. As the scope of programming increases, the need to conduct studies on different styles and types of code is critical. Thus, it is inevitable that eye-tracking studies will need to be conducted on different programming languages. To date, many of the above mentioned studies are done mainly in Java.

Currently, iTrace supports eye tracking on any programming language, or even any plain-text file. The iTrace infrastructure can record eye-movement information from an eye-tracking session regardless of what a participant looks at while doing a task. It provides the character or specific token that a user looks at. However, by itself, iTrace cannot provide syntactical information of the tokens and characters viewed. This is not an easy process and requires the implementation of custom abstract syntax trees (ASTs) to identify the syntactical context of every token in the source code. To enable researchers to understand the context of what the participant looks at, iTrace leverages srcML [Collard et al. 2011, 2013], an advanced infrastructure that provides an XML AST that contains the original source code, whitespace, and formatting. By providing a srcML file alongside the eye-tracking data, iTrace's various tools, such as iTrace-Toolkit [Behler et al. 2023b] and iTrace-Visualize [Behler et al. 2023a, 2024], provide syntactical information alongside the eye-movement data.

As of this writing, srcML only officially supports four languages: C, C++, C#, and Java. This means that if a researcher wishes to perform an eye-tracking study in a different language, they must provide their own mechanism to map the eye-movement data to the source code. iTrace enables this as much as possible by saving all processed eye movement data in a SQLite database, allowing for easy processing by anyone who needs custom analysis.

Recently, the srcML development team has begun work on supporting other programming languages. There is currently an in-development branch of the project that provides support for the Python programming language. Python is an very popular programming language, and being able to conduct eye-tracking studies on popular languages seamlessly is invaluable to the software engineering community. Because iTrace already makes extensive use of srcML, we investigate how well the iTrace infrastructure handles changes and additions to srcML, and if iTrace can support advanced processing of Python eye-tracking data now that srcML has begun supporting Python.

ETRA '25, May 26–29, 2025, Tokyo, Japan

ACM ISBN 979-8-4007-1487-0/2025/05
https://doi.org/10.1145/3715669.3725867

2 RELATED WORK

There are a small number of eye-tracking studies that target Python code. Turner et al. compares the comprehension of C++ and Python code [Turner et al. 2014], finding little difference in time spent looking at the code or task response accuracy. However, students fixate on buggy lines at significantly different rates between the languages. The study does not use syntactic information to explore how code structure influence these differences. In addition this study was done on very small Python code snippets where the stimulus was an image.

Roberto et al. uses eye-tracking to explore the effects of Python's style guide on the readability of Python [Roberto et al. 2024]. This study indicates that some PEP 8 style guidelines reduce fixation duration and regressions for novices. They do not mention using syntactic analysis to determine PEP 8 compliance or inform conclusions about code comprehension.

Chauhan explores which Python language features are classified by participants depending on the language paradigm in which it was written [Chauhan 2022]. Using eye tracking, the tokens and participants' reading strategies are analyzed in relation to accuracy and classification of the language paradigm. Although this study utilizes iTrace-Toolkit to find line and column information, the authors wrote their own external script to map the gazes to tokens in the source code.

Segedinac et al. investigate Python code readability in undergraduate programming courses using eye-tracking data [Segedinac et al. 2024]. After programmatically obtaining structural (syntactic), textual (linguistic), and observational (gaze behavior) features of the code, they use a machine learning model to determine that readability can be well-predicted by tracking how an actual person reads it. They use their own method of extracting syntactic information from abstract syntax trees of the code.

All of the above work involves the exploration of Python source code, but they all either do not use source code syntactic information to perform analysis or do but need to implement their own ad-hoc method to extract the syntactical information from the fixations and source code.

3 PRELIMINARY STUDY ON USING ITRACE WITH PYTHON

To explore the efficacy of iTrace and how well it can support new languages that srcML supports, we perform a preliminary study on Python code. Our goal is to confirm that iTrace supports working with new languages that srcML comes to support seamlessly and demonstrate an example of how to perform an eye-tracking study on Python code.

3.1 Hardware Apparatus and Software Environment

To track the eyes of participants, both a Tobii X3-120 eye-tracker running at 120Hz (at one location) and a Tobii Pro Spectrum eye-tracker running at 150Hz (at another location) are used. The study is performed with a height-adjustable table, which is raised or lowered to accommodate each participant's height. The lab setup was similar at both locations collecting data.

To gather eye-tracking data, we use the iTrace-Core[1] program. Additionally, we use the PyCharm IDE part of the recently released iTrace-JetBrains[2] plugin to gather contextual information from the IDE such as font size, current file, and line/column info. We choose PyCharm to facilitate this study due to its specialization as a Python development environment. PyCharm is also a popular IDE among Python developers.

Additionally, we use the Open Broadcaster Software (OBS) and the iTrace-ScreenRecording[3] plugin to record the computer's screen during the session. This provides us with a video of the exact length of the eye-tracking session, which can be used for various visualizations using the iTrace-Visualize[4] tool. Refer to Figure 1 for an example of what the participant sees within the IDE. Because we use iTrace-ScreenRecord to record the screen including participant's faces, we show a picture-in-picture of the webcam output (blurred for anonymity) on the bottom right of the recorded screen.

3.2 Participants

Our preliminary study is conducted at two different locations and includes four total participants. The study is conducted in a neutral closed lab location with no distractions or external stimulation. Our participants are comprised of three graduate students and one undergraduate student. All four participants are familiar with eye-tracking and Python. The participants had not seen the provided Python code before beginning the session. They were only made aware that they would look at various sorting algorithms in Python before starting the session.

3.3 Tasks and Stimuli

Each participant is provided a series of instructions on how to set up the eye-tracking session, what to do during the session, and how to process the data collected. Additionally, each participant is provided with four Python files. Each file contains a different implementation of a common sorting algorithm. We chose to use files from the "TheAlgorithms" Python repository[5] due to its collection of various algorithms and the code's extensive documentation. Specifically, we chose the bubble sort, heap sort, insertion sort, and quick sort implementations.

During the eye-tracking session, each participant is asked to open one of the sorting algorithm files, read through the code, and document what the worst-case runtime is for the current algorithm. If the participants already knew the runtime of a particular sort, they were asked to verify that the code implemented the algorithm correctly. No time limit is imposed on the participants; however, each participant finished the session in under 12 minutes.

3.4 Data Processing

After all the participants finish their tracking sessions, we begin to process the data using the iTrace infrastructure pipeline as described next.

First, we need the srcML representation of the four Python files so that we can map the eye-movement data to specific Python

[1] https://github.com/iTrace-Dev/iTrace-Core
[2] https://github.com/iTrace-Dev/iTrace-JetBrains/
[3] https://github.com/iTrace-Dev/iTrace-ScreenRecording
[4] https://github.com/iTrace-Dev/iTrace-Visualize
[5] https://github.com/TheAlgorithms/Python

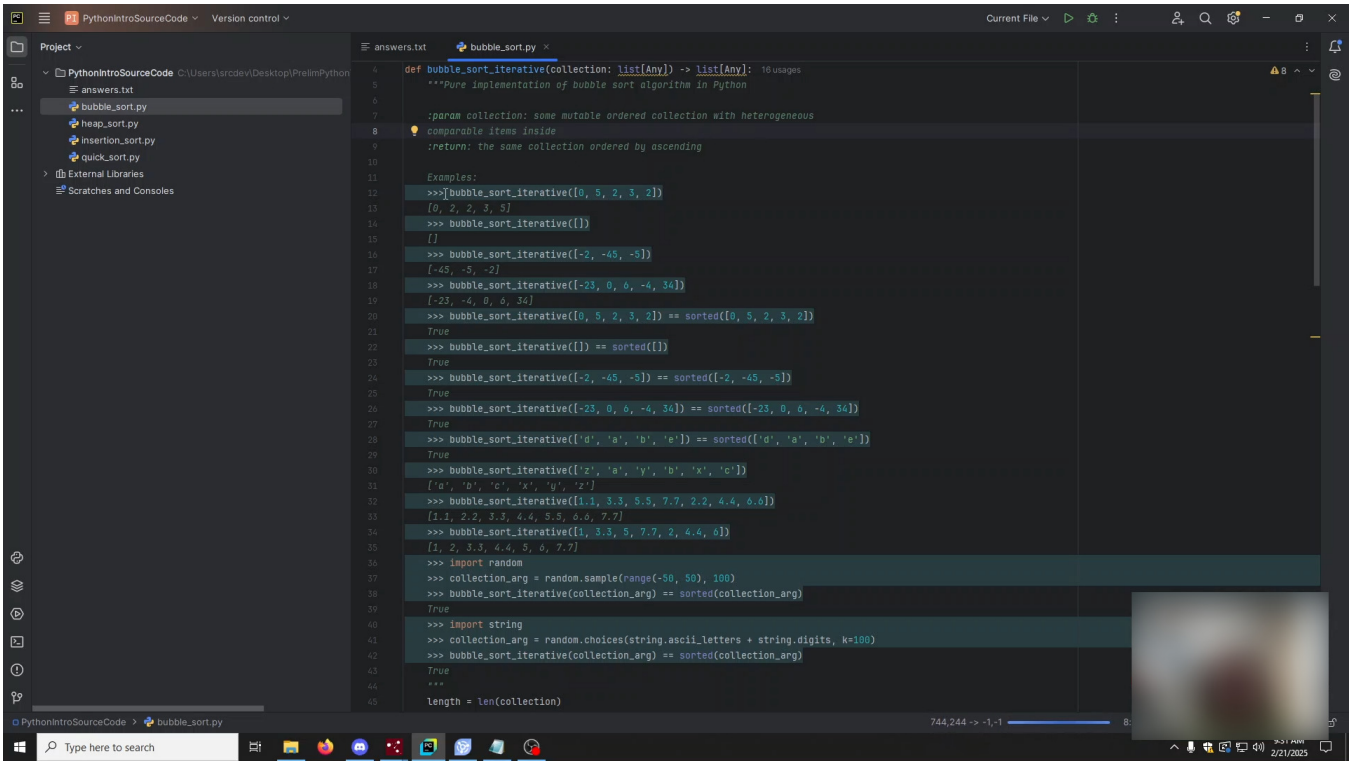

Figure 1: The JetBrains PyCharm IDE as seen by the participants. The bottom status bar shows the x,y pixel -> line,col towards the right. The blurred picture to the bottom right is the webcam output of the participant's face captured by OBS (optional and not visible to participant during recording).

tokens. To acquire this, we download and build the beta branch of srcML[6] which has support for Python. Using this version of srcML, we process the four Python files into a single srcML archive file. We then manually inspect the srcML to ensure that there are no errors in its output, such as incorrect nesting or any unclosed tags, as we do not want any bugs in srcML to affect our result. This process is only done due to the beta nature of srcML's Python support, and will be unnecessary when Python support in srcML is fully tested and released. srcML did not parse any of the Python files incorrectly, and we had nothing to manually adjust. Figure 2 demonstrates a small snippet of Python srcML.

We then load the four eye-tracking sessions into iTrace-Toolkit. Using the srcML archive, we map the gaze data to tokens and syntactic info from the source code using the iTrace-Toolkit UI. We then generate fixations using the IDT algorithm [Salvucci and Goldberg 2000] and default settings listed – duration window of 100*ms*, dispersion of 125, and maximum gaze span of 1000*ms*.

Additionally, we run the collected data through iTrace-Visualize. Using iTrace-Visualize, we generate a marked-up video of the eye-tracking session, tokenized heat-maps of fixations on each file, and Region of Interest (ROI) scarf plots detailing the timeline of when participants view each section. We manually create the ROIs for each file by grouping together lines with a similar purpose - doc-string documentation, importing, function implementation, etc. If this was an actual study, the researchers would determine what their ROIs would be based on their research questions and hypotheses. All of the generated data and visualizations are available in our online artifact [Behler et al. 2025]..

4 RESULTS

This section describes the use of PyCharm, and the two post-processing tools used after the data is collected.

4.1 Using iTrace-JetBrains – PyCharm

iTrace-JetBrains is one of iTrace's newest plugins, and it supports the entire suite of the JetBrains family of IDEs. The GitHub repository for iTrace-JetBrains says it was tested on IntelliJ, PyCharm, WebStorm, and Rust Rover - the IDEs which JetBrains offers a community edition of. PyCharm is a natural fit for Python eye-tracking studies due to its innate focus on Python development.

Some participants made use of PyCharm's built-in linting features while performing the study. Despite reading the small pop-up that would appear when hovering over code which has an issue, there were no difficulties in collecting the IDE contextual data.

[6]https://github.com/srcML/srcML/tree/python

```
if len(collection) < 2:
    return collection
pivot_index = randrange(len(collection))
#------------------------------------------------------------
```

```
<if_stmt><if>if <condition><expr><call><name>len</name><argument_list>(<argument><expr><name>collection</name></expr
    ></argument>)</argument_list></call> <operator>&lt;</operator> <literal type="number">2</literal></expr></condition
    ><block>:<block_content>
    <return>return <expr><name>collection</name></expr></return>
</block_content></block></if></if_stmt>
<expr_stmt><expr><name>pivot_index</name> <operator>=</operator> <call><name>randrange</name><argument_list>(<argument
    ><expr><call><name>len</name><argument_list>(<argument><expr><name>collection</name></expr></argument>)</argument_list
    ></call></expr></argument>)</argument_list></call></expr></expr_stmt>
```

Figure 2: An example of Python code (taken from quick_sort.py) (top) and its corresponding srcML representation (bottom).

4.2 Using iTrace-Toolkit

The next step is to load in the data collected into iTrace-Toolkit. Loading in PyCharm contextual/iTrace-Core data and Python srcML archive data into iTrace-Toolkit was seamless. We did not encounter any errors or issues. Across the four sessions, 305,001 gaze points were recorded. Of those, 286,779 (94%) gaze points had corresponding IDE contextual data recorded. Of the 286,779 gazes with IDE context, 94,397 were located off of the source code tokens, either in PyCharm's project explorer, toolbar, or locations in the code editor with no tokens or whitespace. Additionally, there were no difficulties in calculating fixations in iTrace-Toolkit either. As mentioned previously, iTrace already supports fixation calculation over any kind of stimulus. However, because iTrace can now load in Python srcML into iTrace-Toolkit, our generated fixations now also contain syntactical information from the source code. For this work, we used the I-DT algorithm provided by iTrace-Toolkit with default settings. In total we had 5,374 fixations from the four eye-tracking sessions. 2,975 (55%) of these fixations occurred on a token within the source code. There were 214 unique syntactic categories that the participants viewed from the code. Table 1 details some of the more significant categories viewed

Note that in this work, we are not trying to make any claims about how proficient the participants are at reading and comprehending Python code. We are simply analyzing our small dataset to highlight the usability of iTrace to conduct eye-tracking studies on Python code.

By far the most viewed individual category are the docstrings present within the files. This is expected, as these code stimuli were explicitly chosen for their large docstrings. After docstrings, comments are the next most viewed. The combination of these two imply that the participants spent the majority of their time reading the function's documentation of the code. Additionally, the participants looked within top-level for-loops over 1000 times. This highlights that they did evaluate the actual functionality of the sorting algorithms, and did not rely solely on documentation to answer the questions (although the quick sort implementation did not use a traditional for-loop).

Figure 3: A segment of the combined heatmap of heap_sort.py across all four participants

4.3 Using iTrace-Visualize

In order to test iTrace-Visualize [Behler et al. 2023a, 2024], we use all of the various visualization options the tool offers such as heatmaps and scarfplots. From these visualizations, we can make a couple of observations about the data, which previously would require in-depth analysis to notice.

Our first observation is that the participants did not need to reread large parts of the docstrings and comments in the code. While they on average read the documentation thoroughly, there was little need to go back and reread. The heatmaps show that the tokens in the documentation rarely saw more than four fixations across all four participants. Additionally, the participants also seemed to rarely look at any of the type annotations included on the sorting functions. Function names and parameters see some level of fixation activity, but their annotations seldom do. The participants also did not spend much time reading the "main" section at the bottom of the file. This is likely because the main section's purpose is to the

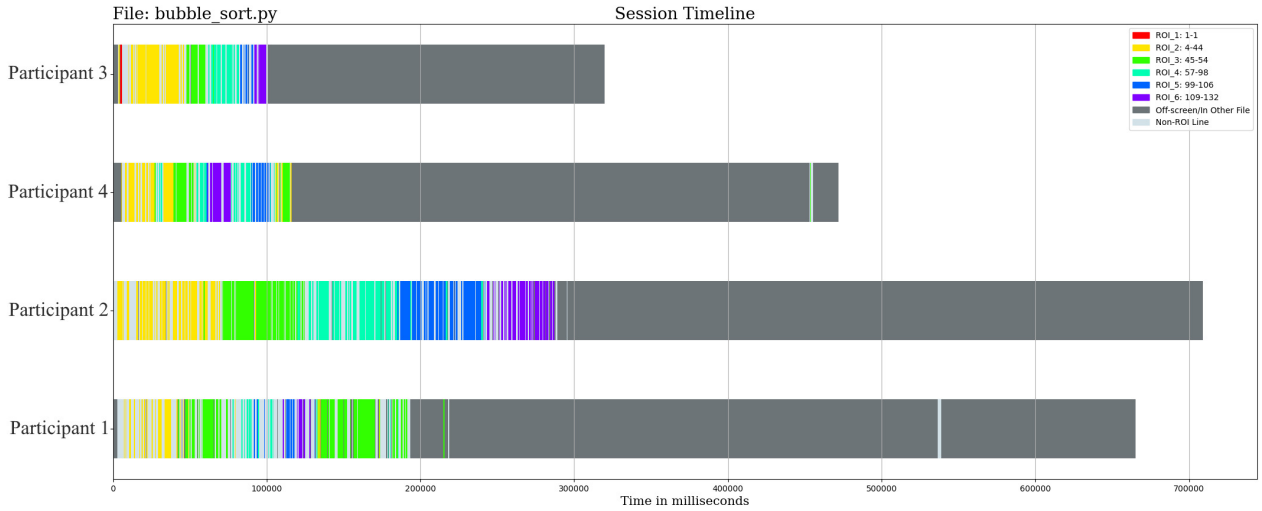

Figure 4: A region of interest scarf plot of the bubble_sort.py file. Of particular note is ROI_1, which is a single line and contains a single import statement. Only participant 3 looked at the region, while everyone else started by reading the next region

Table 1: Some of the viewed categories across the four eye-tracking sessions and their counts.

Full Syntactic Catgeory	Semantic Meaning	Number of Fixations
`unit->function->block->block_content->expr_stmt->expr->literal`	Docstring in function	785
`unit->function->block->block_content->comment`	Comment within function	147
`unit->expr_stmt->expr->literal`	Docstring at top of file	93
`unit->function->name`	A function's name	36
`unit->function->block->block_content->for->...`	Within a for-loop in a function	1006

Table 2: Total number of minutes spent looking at each file across the participants

File	Time (min)
bubble_sort.py	11.6 min
heap_sort.py	4.7 min
insertion_sort.py	7.2 min
quick_sort.py	5.9 min

test the sorting algorithm, and is thus inconsequential to answering the question we asked.

The participants spend the majority of their time examining the various conditions and facets of the sorting algorithms. For example, in the bubble sort iterative implementation, the participants fixated on the `swapped` variable being set to false more often than any other token. In the heap sort implementation as shown in Figure 3, the most viewed token is the `unsorted` argument to the `heapify` call in the first for loop.

Looking at the ROI graphs, the first thing we notice is the time spent per-file. The bubble sort implementation was looked at the longest, while heap sort was the shortest. Table 2 details the times spent per file. The increase in time on bubble_sort.py is likely due to a couple factors: the fact that bubble_sort.py was invariably the first file the participants viewed, meaning they needed to spend the most time getting used to the format of the code, and the fact that bubble_sort.py had two implementations - both a recursive and iterative version of bubble sort.

Additionally, we notice that the participants rarely viewed any small ROIs which contain an import statement. All files except heap_sort.py contain at least one import statement, which were grouped into their own ROI. These ROIs are rarely viewed by the participants. Figure 4 showcases this on the bubble_sort.py file.

From the marked-up videos, we notice an issue with one of the participants eye-tracking setups. This participant, who was the lone participant to use the Tobii X3-120 tracker at a differing location from the other three, had a much wider spread on their gazes. This suggests either an issue with the lower-speed eye-tracker, or a fundamental issue with the eye-tracking setup. Visualizing the quality of the data collected is greatly helpful for finding issues in experiments, as issues like this can be caught early during the pilot phase before more participants are recruited.

5 CONCLUSIONS

The iTrace infrastructure offers a simple yet powerful solution to running eye-tracking studies on software. iTrace offers high-level processing of four programming languages - C, C++, C#, and Java - all languages which the srcML project officially supports. With srcML testing support for Python, iTrace is also able to support eye-tracking studies on Python source code, including advanced analysis like token mapping and source-code aware visualizations. iTrace is capable of supporting new languages that are added to srcML, meaning the eye-tracking community can conduct studies on these languages with ease. The results of this preliminary feasibility study show the value of having such tools built into the iTrace eye tracking infrastructure to support the ease of analysis.

REFERENCES

Amine Abbad-Andaloussi, Thierry Sorg, and Barbara Weber. 2022. Estimating developers' cognitive load at a fine-grained level using eye-tracking measures. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension* (Virtual Event) *(ICPC '22)*. Association for Computing Machinery, New York, NY, USA, 111–121. https://doi.org/10.1145/3524610.3527890

Nahla J. Abid, Bonita Sharif, Natalia Dragan, Hend Alrasheed, and Jonathan I. Maletic. 2019a. Developer Reading Behavior While Summarizing Java Methods: Size and Context Matters. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 384–395. https://doi.org/10.1109/ICSE.2019.00052

Nahla J. Abid, Bonita Sharif, Natalia Dragan, Hend Alrasheed, and Jonathan I. Maletic. 2019b. Developer Reading Behavior While Summarizing Java Methods: Size and Context Matters. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 384–395. https://doi.org/10.1109/ICSE.2019.00052

Aakash Bansal, Robert Wallace, Zachary Karas, Ningzhi Tang, Yu Huang, Toby Jia-Jun Li, and Collin McMillan. 2024. Programmer visual attention during context-aware code summarization. *arXiv preprint arXiv:2405.18573* (2024).

Joshua Behler, Gino Chiudioni, Alex Ely, Julia Pangonis, Bonita Sharif, and Jonathan I. Maletic. 2023a. iTrace-Visualize: Visualizing Eye-Tracking Data for Software Engineering Studies. In *2023 IEEE Working Conference on Software Visualization (VISSOFT)*. 100–104. https://doi.org/10.1109/VISSOFT60811.2023.00021

Joshua Behler, Praxis Weston, Drew T. Guarnera, Bonita Sharif, and Jonathan I. Maletic. 2023b. iTrace-Toolkit: A Pipeline for Analyzing Eye-Tracking Data of Software Engineering Studies. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 46–50. https://doi.org/10.1109/ICSE-Companion58688.2023.00022

Joshua A.C. Behler, Zachary Kozak, Kang il Park, Bonita Sharif, and Jonathan I. Maletic. 2025. Extending Support for Analyzing Eye Tracking Studies on Python Source Code in iTrace - Artifact. https://osf.io/4cnqk/

Joshua A. C. Behler, Giovanni Villalobos, Julia Pangonis, Bonita Sharif, and Jonathan I. Maletic. 2024. Extending iTrace-Visualize to Support Token-based Heatmaps and Region of Interest Scarf Plots for Source Code. In *2024 IEEE Working Conference on Software Visualization (VISSOFT)*. 139–143. https://doi.org/10.1109/VISSOFT64034.2024.00027

Jigyasa Chauhan. 2022. *An Empirical Study on the Classification of Python Language Features Using Eye-TrackingFeatures Using Eye-Tracking*. Master's thesis. University of Nebraska - Lincoln.

M. L. Collard, M. J. Decker, and J. I. Maletic. 2011. Lightweight Transformation and Fact Extraction with the srcML Toolkit. In *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*. 173–184. https://doi.org/10.1109/SCAM.2011.19

M. L. Collard, M. J. Decker, and J. I. Maletic. 2013. srcML: An Infrastructure for the Exploration, Analysis, and Manipulation of Source Code: A Tool Demonstration. In *2013 IEEE International Conference on Software Maintenance*. 516–519. https://doi.org/10.1109/ICSM.2013.85

Drew T. Guarnera, Corey A. Bryant, Ashwin Mishra, Jonathan I. Maletic, and Bonita Sharif. 2018. iTrace: eye tracking infrastructure for development environments. In *10th ACM Symposium on Eye tracking Research and Applications*. Warsaw, Poland, 3. https://doi.org/10.1145/3204493.3208343

Vinícius Martins, Pedro Lopes Verardo Ramos, Breno Braga Neves, Maria Vitoria Lima, Johny Arriel, João Victor Godinho, Joanne Ribeiro, Alessandro Garcia, and Juliana Alves Pereira. 2024. Eyes on Code Smells: Analyzing Developers' Responses During Code Snippet Analysis. In *Simpósio Brasileiro de Engenharia de Software (SBES)*. SBC, 302–312.

Kang-il Park, Pierre Weill-Tessier, Neil C. C. Brown, Bonita Sharif, Nikolaj Jensen, and Michael Kölling. 2023. An eye tracking study assessing the impact of background styling in code editors on novice programmers' code understanding. In *Proceedings of the 2023 ACM Conference on International Computing Education Research - Volume 1* (Chicago, IL, USA) *(ICER '23)*. Association for Computing Machinery, New York, NY, USA, 444–463. https://doi.org/10.1145/3568813.3600133

Cole S. Peterson, Nahla J. Abid, Corey A. Bryant, Jonathan I. Maletic, and Bonita Sharif. 2019. Factors influencing dwell time during source code reading: a large-scale replication experiment. In *Proceedings of the 11th ACM Symposium on Eye Tracking Research & Applications* (Denver, Colorado) *(ETRA '19)*. Association for Computing Machinery, New York, NY, USA, Article 38, 4 pages. https://doi.org/10.1145/3314111.3319833

Pablo Roberto, Rohit Gheyi, José Aldo Silva da Costa, and Márcio Ribeiro. 2024. Assessing Python Style Guides: An Eye-Tracking Study with Novice Developers. arXiv:2408.14566 [cs.SE] https://arxiv.org/abs/2408.14566

Dario D. Salvucci and Joseph H. Goldberg. 2000. Identifying Fixations and Saccades in Eye-tracking Protocols. In *Proceedings of the 2000 Symposium on Eye Tracking Research & Applications* (Palm Beach Gardens, Florida, USA) *(ETRA '00)*. ACM, New York, NY, USA, 71–78. https://doi.org/10.1145/355017.355028

Milan Segedinac, Goran Savić, Ivana Zeljković, Jelena Slivka, and Zora Konjović. 2024. Assessing code readability in Python programming courses using eye-tracking. *Computer Applications in Engineering Education* 32, 1 (2024), e22685.

Rachel Turner, Michael Falcone, Bonita Sharif, and Alina Lazar. 2014. An eye-tracking study assessing the comprehension of c++ and Python source code. In *Proceedings of the Symposium on Eye Tracking Research and Applications* (Safety Harbor, Florida) *(ETRA '14)*. Association for Computing Machinery, New York, NY, USA, 231–234. https://doi.org/10.1145/2578153.2578218

Haruhiko Yoshioka and Hidetake Uwano. 2024. An Analysis of Program Comprehension Process by Eye Movement Mapping to Syntax Trees. In *Networking and Parallel/Distributed Computing Systems: Volume 18*. Springer, 137–152.

Vlas Zyrianov, Cole S Peterson, Drew T Guarnera, Joshua Behler, Praxis Weston, Bonita Sharif, and Jonathan I Maletic. 2022. Deja Vu: semantics-aware recording and replay of high-speed eye tracking and interaction data to support cognitive studies of software engineering tasks—methodology and analyses. *Empirical software engineering* 27, 7 (2022), 168.