Appendix of Inline Tests

Anonymous Author(s)

A SEARCH KEYWORDS

In this section, we show the keywords used for searching Java and Python examples.

First, we started with a pilot study to manually check 165 examples using simple search keywords, out of which we deemed 46 examples (27.9%) worth writing inline tests for. Table 1 shows a breakdown of the programming language features that we explored in our pilot study, the search keyword used, the number of examples checked, and the number and percentage of examples that we deemed as worth writing inline tests for.

Table 1: The search keywords used during the pilot study, the number of examples checked, and the number and percentage of examples that we deemed worth writing inline tests for.

Kind	PL	Keyword	Checked	Worth Writing	
			#Exp	#Exp	%Exp
Regex	Python	`re.match`	24	15	62.5%
	Java	`.matches(`	86	15	17.4%
String	Python	`split`	20	9	45.0%
Stream	Java	`Stream.of(`	35	7	20%

Based on the pilot study, we defined the following criteria of automatically excluding the examples that match the search keyword but are likely not worth writing inline tests for:

(1) exclude if the keyword appears in a comment.
(2) exclude if the keyword matches usage of a third-party package.
(3) exclude if the keyword appears in a statement that is too simple and will likely not warrant an inline test, for example, `x = 1 << 3`.
(4) exclude if the keyword appears in a return statement (so, it can be tested with a unit test).
(5) exclude if the keyword appears in a method that has only one statement (so, again, it can be tested with a unit test).
(6) exclude if the keyword matches code with a different meaning than we intended. For example, `>>` can represent the right shift operator in bit manipulation but it also sometimes matches the closing of a parameterized generic type, such as `< String, Box < Integer >>`.

After these exclusions, we expanded the set of search keywords to cover the five kinds of programming language features that inline tests can be beneficial for, as discussed in Section 2.2 of the paper. For each kind, we list the search keywords used and the number of matches in the top-100 Python and Java projects after applying the exclusion criteria; for example, `re.match` (1237) indicates we found 1,237 examples with `re.match` in the top-100 Python projects that are likely worth writing inline tests for.

A.1 Regular Expression

Python: `re.match` (1237), `re.fullmatch` (69), `re.search` (981), `re.sub` (848), `re.subn` (12), `re.split` (141), `re.findall` (573), `re.finditer` (213), `re.compile` (1311), `re.purge` (5), `re.escape` (366).
Java: `.matches(` (3370), `Pattern.compile` (2414).

A.2 String Operation

Python: `capitalize(` (159), `casefold(` (57), `center(` (298), `count(` (3002), `encode(` (5033), `endswith(` (1869), `expandtabs(` (70), `find(` (1639), `format(` (16360), `format_map(` (33), `index(` (5912), `isalnum(` (67), `isalpha(` (90), `isdecimal(` (29), `isdigit(` (256), `isidentifier(` (68), `islower(` (65), `isnumeric(` (53), `isprintable(` (26), `ispunct(` (118), `istitle(` (34), `isupper(` (95), `join(` (20255), `lower(` (3180), `replace(` (5126), `rfind(` (196), `rindex(` (108), `rsplit(` (224), `split(` (8713), `upper(` (905), `splitlines(` (1049), `startswith(` (4860), `strip(` (5617).
Java: `split(` (7215), `.subString` (48), `.indexOf` (6493), `.format(` (18617), `.replace(` (4668).

A.3 Bit Manipulation

Python: `<<` (565), `>>` (461).
Java: `<<` (6740), `>>` (3468).

A.4 Stream Operation

Java: `Stream.of(` (2065), `Stream.builder(` (17).

A.5 Collections

Python: `list.sort` (42), `for x in` (3251).

B API DESIGN

In this section, we show the API of I-Test via source code.

B.1 Python

```python
class Here:
    def __init__(
        self,
        test_name: str = None,
        parameterized: bool = False,
        repeated: int = 1,
        tag=[],
        disabled = False
    ):
        """
        Initialize Inline object with test name / parameetrized flag

        :param test_name: test
        :param parameterized: whether the test is parameterized
        """

    def given(self, variable, value):
        """
        Set value to a variable.

        :param variable: a variable name
        :param value: a value that will be assigned to the variable
        :returns: Inline object
        """
        return self

    def check_eq(self, actual_value, expected_value):
        """
        Assert whether two values equal

        :param actual_value: the value to check against expected
        :param expected_value: expected value
        :returns: Inline object
        :raises: AssertionError
        """
        return self

    def check_true(self, expr):
        """
        Assert whether a boolean expression is true

        :param expr: a boolean expression
        :returns: Inline object
        :raises: AssertionError
        """
        return self

    def check_false(self, expr):
        """
        Assert whether a boolean expression is false

        :param expr: a boolean expression
        :returns: Inline object
        :raises: AssertionError
        """
        return self


class Group:
    def __init__(self, *arg):
        """
        Initialize Group object with index
        """
```

B.2 Java

```java
package org.inlinetest;

public class Here {
    public Here() {
        return;
    }

    public Here(String name) {
        return;
    }

    public Here checkEq(Object expected, Object actual) {
        return this;
    }

    public Here given(Object variable, Object value) {
        return this;
    }

    public Here checkTrue(Object value) {
        return this;
    }

    public Here checkFalse(Object value) {
        return this;
    }
}
```

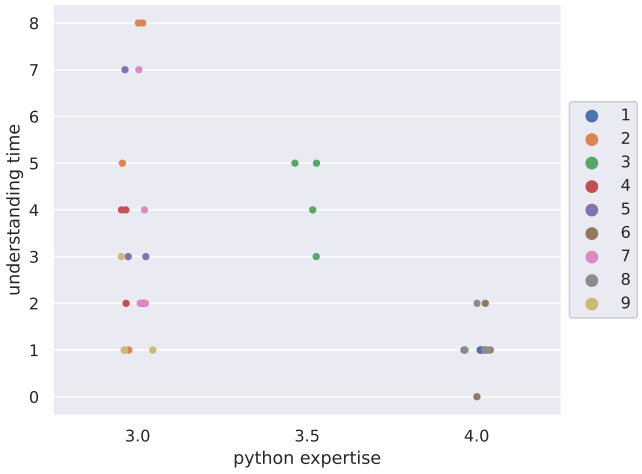
C USER STUDY

In this section, we show the plots of the user study results. A sample non-executable user study is in the folder `userstudy`.

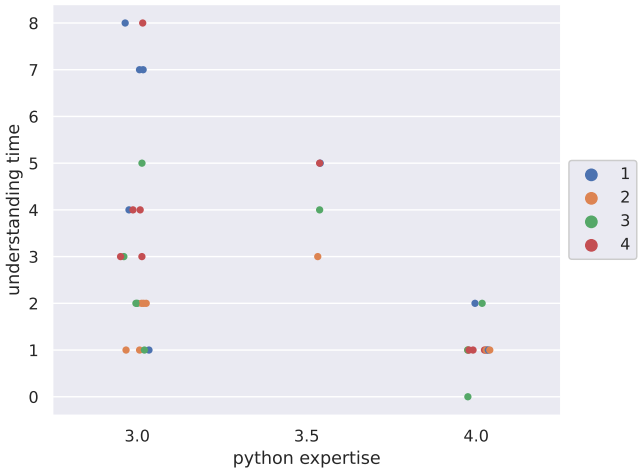
Figure 1 presents the relationship between Python programming expertise and understanding time. Each point is the time spent by a participant to understand the target statement in a task. Figure 1a groups the points by participants (so there are eight colors for eight participants, and four points per color); Figure 1b groups the points by tasks (so there are four colors for four tasks, and eight points per color). One participant answered "between 3 and 4" to the question "How do you rank your Python programming expertise between 1–5", so we consider this participant's answer to be 3.5. Generally, the more expertise the participant had, the less time they spent in understanding the target statement.

Figure 2 presents the relationship between Python programming expertise and test-writing time. Each point is the time spent by a participant to write one inline test test for a task. If the participants write two tests and spend 5 minutes in total, we compute the test writing time as 2.5 minutes. The time for writing each test is low overall, and it takes at most 3 minutes to write an inline test for the participants with higher Python expertise (4.0) except for an outlier.

Figure 3 and Figure 4 present that the relationship between years of general programming expertise and time to understand and time to write tests, respectively. There is a similar but less obvious trend that more skilled programmers spend less time to understand the target statement and to write inline tests.

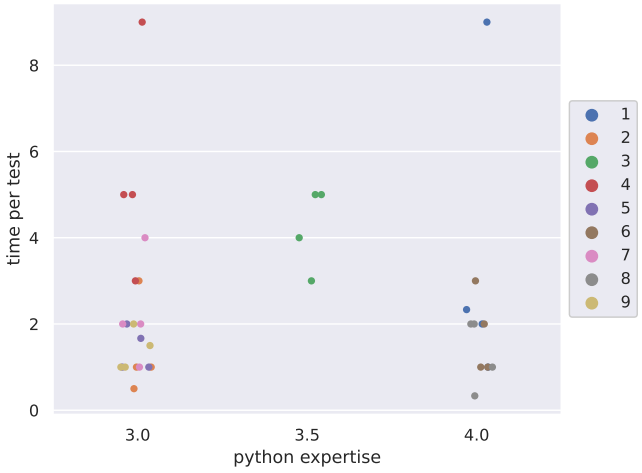


(a) Grouped by participants

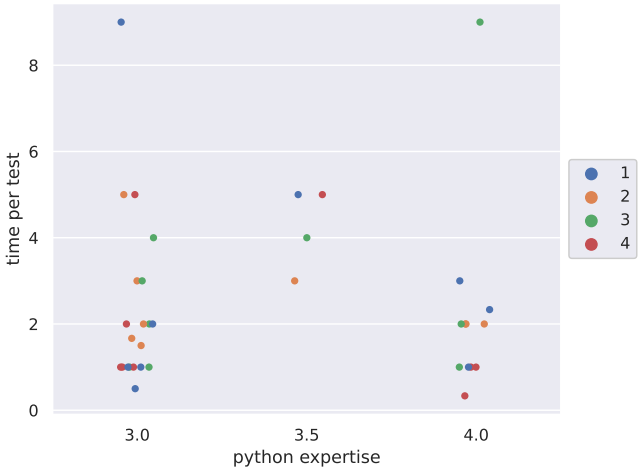


(b) Grouped by tasks

Figure 1: Python expertise to understanding time.

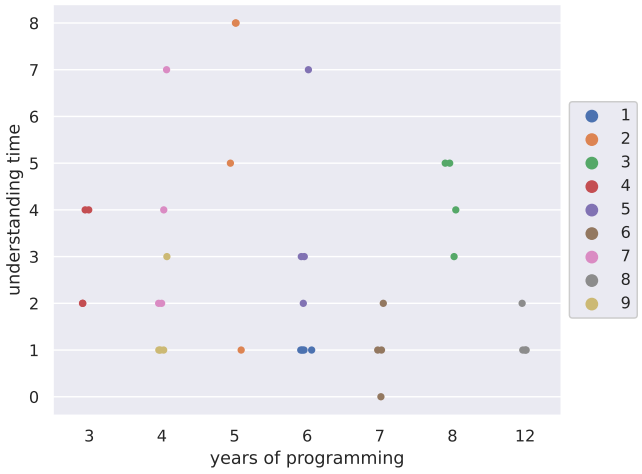


(a) Grouped by participants

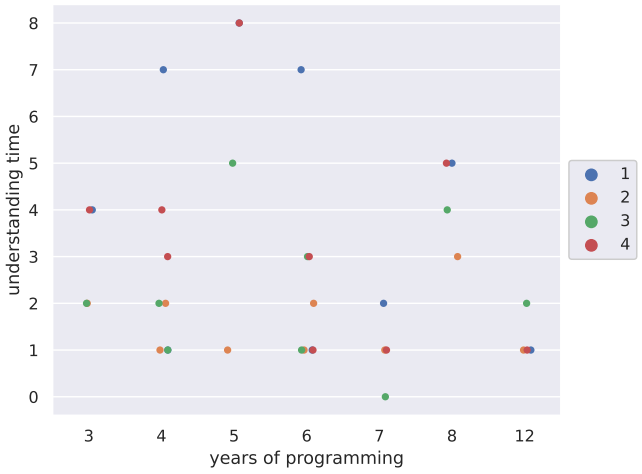


(b) Grouped by tasks

Figure 2: Python expertise to writing time per test.

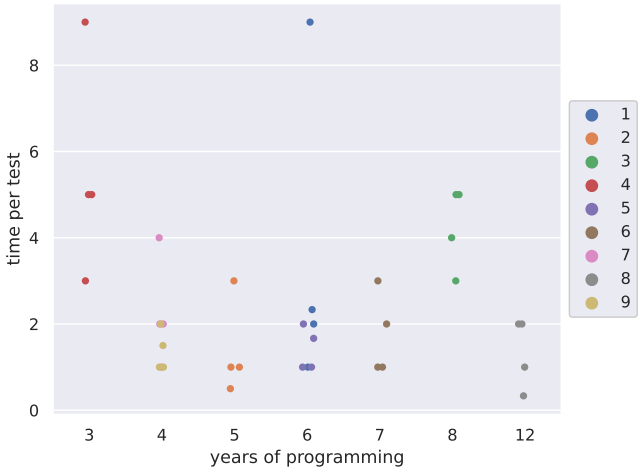


(a) Grouped by participants

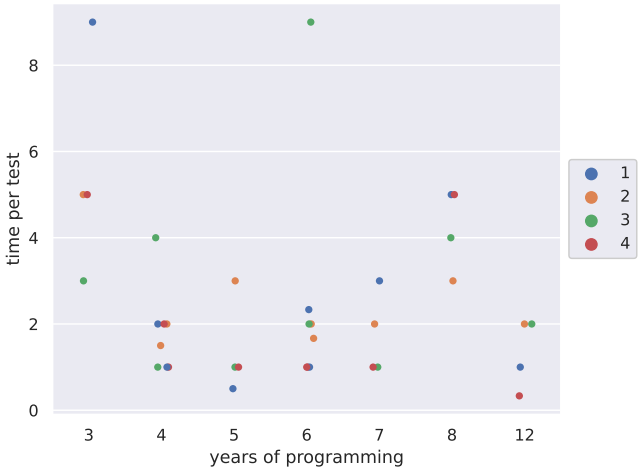


(b) Grouped by tasks

Figure 3: Programming year to understanding time.



(a) Grouped by participants



(b) Grouped by tasks

Figure 4: Programming year to writing time per test.