Feedback-Directed Partial Execution

Ishrak Hayet
North Carolina State University
Raleigh, North Carolina, USA
ihayet@ncsu.edu

Adam Scott
North Carolina State University
Raleigh, North Carolina, USA
amscott9@ncsu.edu

Marcelo d'Amorim
North Carolina State University
Raleigh, North Carolina, USA
mdamori@ncsu.edu

Abstract

Partial code execution is the problem of executing code with missing definitions. The problem has gained recent traction as solutions to the problem could enable various downstream analyses. We propose feedback-directed partial execution, a technique supported by a tool, named Incompleter, that uses the error feedback from executions to enable partial code execution. Incompleter builds on the observation that errors observed during the execution of incomplete snippets often follow similar error patterns. Incompleter takes an incomplete snippet as input and applies rules (e.g., add class, add field, add file, etc.) to resolve the successive dynamic errors it encounters during execution of the snippet. Incompleter stops when the snippet successfully executes or when it reaches certain bounds. Our results indicate that Incompleter outperforms LExecutor, the state-of-the-art in partial execution. For example, considering a dataset of 4.7K incomplete StackOverflow snippets, Incompleter enables the execution of 10% more code snippets compared to LExecutor and covers 23% more statements. We also show that Incompleter's type inference significantly improves over LExecutor's type inference, with a 37% higher F1 score.

CCS Concepts

• **Software and its engineering → Software testing and debugging**.

Keywords

Code completion, Debugging, Rule mining

ACM Reference Format:
Ishrak Hayet, Adam Scott, and Marcelo d'Amorim. 2024. Feedback-Directed Partial Execution. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3680320

1 Introduction

Partial (code) execution [23, 32, 33, 58, 70] is the problem of executing a code fragment with missing elements, such as variable and function definitions. It *empowers dynamic analyses* as developers could start the analysis from arbitrary program locations; it *enables debugging in isolation* for similar reason, and it *facilitates the adoption of code snippets from Q&A forums* as prior work has shown that ≈75% of code snippets from those forums cannot be directly executed [26, 27]. Souza and Pradel [58] recently proposed LExecutor, a technique for partial execution that uses neural type prediction [23, 43, 48] to determine the types of unknown identifiers and define them based on those types. LExecutor uses hard-coded values for identifiers whose types it can infer and it uses dummy objects for identifiers whose types it fails to infer. Despite the big step forward that LExecutor made in partial execution, there still remains much room for improvement. For example, LExecutor can successfully execute only 54% of the code snippets from its dataset. Two fundamental reasons explain those results: the inflexibility of dummy objects (e.g., to define a required method or attribute) and the inability to modify the environment where the snippet runs (e.g., to add a required file or a module).

We propose *feedback-directed partial execution*, a technique supported by a tool, named Incompleter, leveraging *execution feedback* to address the problems mentioned above. Execution feedback enables Incompleter to obtain data that an *incomplete code snippet* (i.e., a snippet with missing definitions) needs for running. Incompleter builds on the observation that errors observed during the execution of incomplete snippets often follow a similar pattern. For example, considering the errors manifested when running the snippets from the LExecutor dataset, we observe that 50% of those are `NameError` (e.g., due to an undefined variable) and 40% of those are `ModuleNotFoundError` (e.g., due to a missing module). That observation led us to manually craft a set of rules to address those issues (e.g., add a class, add a field to a class, add an import, remove import, etc.). Incompleter takes an incomplete snippet as input and iteratively applies rules to resolve the successive dynamic errors it encounters during execution of the given snippet. Incompleter stops when it finds a solution (i.e., the snippet runs successfully) or when it cannot make progress or when it reaches a budget on the number of iterations.

Incompleter has two components: the *mocker* and the *unmocker*. The goal of the mocker is to enable the execution of the input snippet whereas the goal of the unmocker is to make the executable mocked code more faithful to the original code. The mocker component follows the feedback-directed approach described above to mock the behavior of the missing elements of the input. In the process, it creates artificial "mocked" types to circumvent some of the errors. Subsequently, the unmocker replaces –or recommends the replacement of– the mocked types with concrete types. The unmocker's type inference uses type deduction and type prediction in tandem. Type deduction relies on a list of type deduction rules to infer user-defined types and certain built-in types. Type deduction, albeit precise, is fundamentally incomplete. For that, Incompleter uses a CodeT5-based [66] neural type predictor [23, 43, 48] to complement type deduction and cover a broader range of types.

ISSTA '24, September 16–20, 2024, Vienna, Austria
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3680320

```
1  def find_nearest(array, values):
2    array = np.asarray(array)
3    values = np.expand_dims(values,axis=−1)
4    indices = np.abs(array−values).argmin(axis=−1)
5    return array[indices]
6  image = plt.imread('example_3_band_image.jpg')
7  print(image.shape)
8  quantiles = np.linspace(0, 255, num=2**2, dtype=np.uint8)
9  quantiled_image = find_nearest(quantiles, image)
10 print(quantiled_image.shape)
```

Figure 1: Snippet 653 from the LExecutor dataset with identifiers `plt` and `np` undefined and a reference to a non-existing file `example_3_band_image.jpg`.

Results show that Incompleter obtains higher executability and coverage compared to LExecutor. For example, considering a dataset of 4.7K incomplete StackOverflow snippets, Incompleter enables the execution of 10% more code snippets compared to LExecutor and covers 23% more statements. We also show that Incompleter's type inference significantly improves over LExecutor's type inference, with a 37% higher F1 score. Aiming to further demonstrate the usefulness and soundness of our approach, we also conduct a case study on a highly-popular open-source project to download videos from YouTube, YouTube-dl [18], and show that Incompleter faithfully reproduces 8 of the 15 YouTube-dl bugs from the BugsInPy [57] dataset of reproducible Python bugs.

This paper makes the following contributions:

Technique. A technique that leverages execution feedback to progressively complete a code snippet with missing definitions. Our technique is rule-based and therefore is complementary to alternative approaches, such as LExecutor. We implement the technique in the publicly-available tool Incompleter [6].

Evaluation. A study involving a large dataset of code snippets from StackOverflow, showing that Incompleter outperforms the SoTA, LExecutor, on executability and coverage. We also demonstrate Incompleter's usefulness through a case study.

2 Example

We illustrate Incompleter on two representative examples.

Terminology. An *incomplete snippet* is a syntactically correct fragment of Python code whose execution raises exceptions because of missing definitions, some of which can be external to the code (e.g., files, directories, and modules). Figure 1 shows an example of an incomplete snippet from the LExecutor dataset [58]. The snippet has the identifier `plt` undefined and a reference to a non-existing file `example_3_band_image.jpg` (Line 6). Execution of this snippet raises a `NameError` exception at Line 6 because `plt` is undefined. A *complete snippet* is the counterpart of an incomplete snippet with missing definitions resolved. The execution of a complete snippet must terminate without raising exceptions (for correctness) and must exercise the same statements that the execution of the corresponding incomplete snippet exercises up to the point it raises an exception (for consistency).

2.1 Example 1: Adding Missing Imports and File

Incompleter repeats the following steps until all errors are resolved or it reaches a time budget: (1) execute snippet, (2) collect error, and (3) choose an action based on previous error. We use the example from Figure 1 to illustrate this process. The figure shows the snippet 653 from the LExecutor dataset [5]. The execution of this snippet produces a `NameError` indicating that identifier `plt` is undefined. Incompleter finds the relationship between the library `matplotlib` and the identifier `plt` and prepends the import statement `import matplotlib.pyplot as plt` to the snippet to resolve the error. Then, Incompleter executes the revised snippet observing the error `FileNotFoundError` at line 6 because the file `example_3_band_image.jpg` does not exist in the environment from where the snippet executes. In response to the error, Incompleter extracts the file name and extension from the error message and adds a file with the same name to the environment where the snippet executes. The content of the file corresponds to the expected extension (in this case, `jpg`) and may be relevant. In this example, it is relevant as the function `imread` expects an image file to return a corresponding array of bytes. To create content, Incompleter maintains example files for common file extensions. It is worth noting that file is an example of a missing definition that is external to the code. Other examples include missing directory and module (Section 3.1.2). After resolving the missing file error, Incompleter executes the code again encountering another `NameError` indicating that the identifier `np` is undefined. Once again, Incompleter resolves the issue by prepending the import statement `import numpy as np`. After that change, the snippet runs successfully, printing `(700, 1050, 3)` and `(700, 1050, 3)`.

LExecutor fails to execute the example from Figure 1 as it does not add import statements and does not create missing resources. LExecutor uses the type `DummyObject` for types of undefined identifiers that it is unable to infer. The `DummyObject` type *contains only one definition*–the generic constructor `def __init__(self, *args, **kwargs): pass`, which admits any sequence of parameters. In this example, LExecutor infers the type `DummyObject` for the undefined identifiers `plt`, `np`, and for the return values of the functions `asarray`, `expand_dims`, and `abs`. In lines 2 and 3, the `DummyObject` values returned from the call to `asarray` and `expand_dims` are assigned to the identifiers `array` and `values`, respectively. When evaluating the expression `array-values` in line 4, the runtime reports a `TypeError` related to the unsupported arithmetic subtraction between two objects of type `DummyObject`. To sum up, LExecutor prematurely decides to mock the types of undefined identifiers in this example. As we showed before with Incompleter, that was not the right direction to resolve the issue.

2.2 Example 2: Mocking and Unmocking

This example illustrates a case where Incompleter needs to infer types to resolve an issue. For simplicity, we use snippet 251, consisting of the single statement `number += 1`, from the LExecutor dataset [1]. The code contains a use of the undefined variable `number`; hence, execution raises `NameError`.

Incompleter has two main components: the mocker and the unmocker. The goal of the first component is to execute the code; it does that by adding code-related artifacts. The previous example illustrates the mocker, which introduces two import statements and a file. In this example, the mocker will introduce a *mocked type*. In contrast, the goal of the unmocker is to identify the actual types of mocked types. In this case, that enables Incompleter to initialize the variable. It is worth noting that, in addition to *executability*, type inference also improves *readability* of the snippets.

For the snippet `number += 1`, Incompleter lazily defines `number` as a `TBD0` object, i.e., Incompleter introduces the assignment `number = TBD0()`. At this point, the definition of `TBD0` is equivalent to that of LExecutor's `DummyObject`, i.e., the mocked type only declares a generic constructor. The key difference between these types is that TBD types can be refined, hence the need to differentiate TBD types with a counter (e.g., `TBD0`). The execution of the revised version of the original snippet now raises `TypeError` as a result of an attempt to add the numeric literal 1 to an instance of the non-numeric type `TBD0`, which the variable `number` refers to. In response to this error, Incompleter makes two modifications to the `TBD0` class: (1) it adds the built-in function `__iadd__` and (2) it modifies the class to inherit from the built-in type `int`. Although the code executes successfully after those changes, `TBD` classes remain in the code and might be inconvenient to developers (e.g., for debugging). At that stage, the unmocker component of Incompleter kicks in to deduce that `TBD0` denotes the set of integers. As a consequence, the unmocker removes the definition `TBD0` and replaces the initial assignment to the variable `number` with a numeric value. For that, it assigns the value 1, arbitrarily chosen. (Section 7 elaborates on how test data generation techniques, as symbolic execution, could guide the choice of primitive values.) Finally, Incompleter reports the following snippet on output:

```
number = 1
number += 1
```

LExecutor is unable to handle this example because the code instrumentation it uses does not support addition assignment +=. We manually refactored the code to successfully circumvent that engineering limitation, but, then LExecutor raises `TypeError` (on `number = number + 1`) as it predicts `number` to be an instance of `DummyObject`. Recall that `DummyObject` has the same definitions irrespective of the input. In particular, it is not a subclass of any numeric type, so it does not overload the operator `+`.

It is worth noting that this section uses short examples to demonstrate the principle of mocking and unmocking, but the idea generalizes to several other cases. For example, Incompleter can execute code with missing classes and functions. Section 3.1.2 elaborates a list of transformations that Incompleter supports to make the code executable and Section 3.2 elaborates on the method we use to identify the actual types that the mocked types represent.

3 Incompleter

We describe Incompleter, a technique to enable the execution of incomplete Python code snippets by leveraging execution feedback. Incompleter takes as input an incomplete code snippet free of syntactical errors and produces a corresponding complete snippet.

Incompleter is organized as a pipeline of two main processor components: the mocker and the unmocker. The goal of the mocker is to make the code executable whereas the goal of the unmocker is to make the executable code more faithful to the original code by predicting the concrete types that represent the mocked types. Figure 2 shows Incompleter's organization, which we detail in the following sections.

Algorithm 1: Mocker algorithm

```
Input original code snippet s and environment env
Output mocked code snippet and modified environment
Require: s is free from syntax errors

def execute(s, env): ... # executes snippet s inside environment env
def parse(err): ... # parses error string err
def find_action(err_type, err_msg): ... # maps error data to an action

iter ← 0
while iter++ < MAX_ITER do
    err_str ← execute(s, env)
    if len(err_str) == 0 then
        break # no error
    end
    err_type, err_msg, err_line ← parse(err_str)
    err_id ← err_type + err_msg + err_line
    if err_id == prev_err_id then
        break # reached a fixpoint; no progress
    end
    prev_err_id ← err_id
    action ← find_action(err_type, err_msg)
    if action is None then
        break # no action for this error
    end
    s, env ← action.apply(s, env)
end
return s, env
```

3.1 Mocker

Algorithm 1 shows the pseudocode of the mocker component of Incompleter. It takes as input an incomplete snippet and produces a corresponding code snippet that executes with no errors. The algorithm is organized as an iterative procedure that progressively resolves dynamic errors and stops when it reaches a fixpoint or a budget (number of iterations).

Incompleter contains a set of *rules* for mapping error messages to *actions* that can resolve those errors. Typically, actions are code transformations but they can also change the environment where the code is being executed, e.g., adding a file or installing a new package. At each iteration, Algorithm 1 proceeds as follows. Incompleter executes the code snippet *s* within a virtual environment *env*. If no error occurs, execution terminates. Otherwise, the mocker checks if the error is the same as observed in the previous iteration. If that is the case, execution terminates as that condition indicates the previous changes were ineffective in resolving the error. It is worth noting that the preconditions for applying different actions are typically disjoint. Incompleter does have a simple backtracking mechanism that tries all actions that satisfy preconditions, reverting the effects of those actions was ineffective. The pseudocode omits this feature for space and simplicity. If `err_id` and `prev_err_id` are distinct, execution proceeds to select an appropriate action to resolve the current error. The auxiliary function `find_action` looks up the appropriate action. If no such action is found, execution terminates. Otherwise, Incompleter applies the action and proceeds to the next iteration. In the following, we describe the methodology we use to obtain rules.

3.1.1 Rule Mining Methodology. We use the LExecutor dataset to decide which rules to specify. The rationale is that the LExecutor dataset is relatively small –enabling human inspection– yet

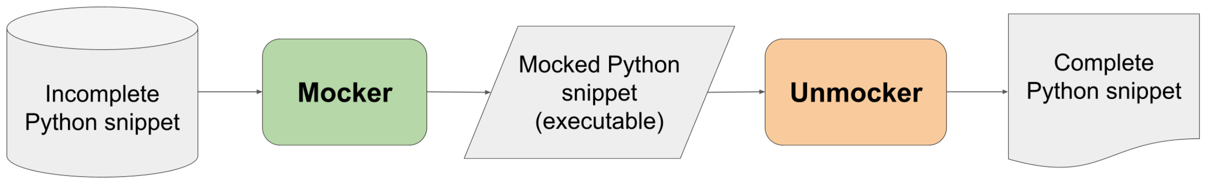


Figure 2: INCOMPLETER organization as a pipeline of two components–the "mocker" and the "unmocker".

Table 1: List of rules with corresponding number of snippets. The rules are grouped by the error type they resolve.

Error Type	Rule	# Snippets
NameError	Add-Import	43
NameError, AttributeError	Define-Func	27
	Define-Var	126
FileNotFoundError	Create-Resource	16
ModuleNotFoundError	Resolve-Module	17
TypeError	Define-Container	42
	Define-Callable	4
	Define-Length	2
	Define-Operator	16
	Define-Literal	17
KeyError	Define-Key	9
ValueError	Convert-Literal	2

representative. To choose which rules to specify, we proceed as follows. Initially, we execute the entire set of incomplete code snippets from the LEXECUTOR dataset. We then group snippets that fail for a similar reason (e.g., missing variable definition). Conceptually, this step partitions the set of code snippets in groups related by the cause of error. Then, we select the group with the highest number of snippets and elaborate an *action* to resolve the issues from that group. An action modifies the code (e.g., adding or removing an import statement) or the execution environment where the code executes (e.g., adding a file). Finally, we apply the action to all eligible snippets, run the modified snippets, and select another group of issues to analyze. This process is repeated until we can no longer formulate a rule to fix two or more snippets. We obtain a set of 12 rules following this process.

Table 1 shows the list of 12 rules that INCOMPLETER uses grouped by error type, denoting a Python's built-in exception type [3] that is raised during the execution of the corresponding group of snippets. The table shows the number of snippets (column "# Snippets") that could resolve a certain error (column "Error Type") by applying the corresponding rule (column "Rule"). INCOMPLETER's ruleset can be expanded with other rules. The following section details each of the mined rules.

3.1.2 Rule Description. This section describes the 12 rules mined with the method from Section 3.1.1.

M1. Add-Import. We built a list of the most popular third-party Python packages [64]. Then we mined from GitHub [2] the first 100 *import statement* usages related to these popular packages. We mapped these *import statement* usages to the module names of the selected popular packages. The mapping also contains the names and *import statement* usages of Python built-in modules [12]. Eventually, in response to a `NameError`, we add the corresponding *import statement* in the code if the undefined identifier is a module's name in our prepared mapping.

M2. Define-Func. For a `NameError`, if we cannot find the undefined identifier in the mapping from rule M1, we statically analyze the source code to check whether the undefined identifier is a function call. If yes, we try to infer the function's signature and then we define the function in the code. In the case of an `AttributeError`, we similarly try to analyze whether the undefined attribute of an object is a method call. If yes, we define the method in the scope of the corresponding class as either an instance or a class method, depending on the information from the error message.

M3. Define-Var. For a `NameError`, if the undefined identifier is neither the name of a module (rule. M1) nor the name of a function (rule M2), we lazily assign a value to the identifier. The value is initially an object from a generic *(to-be-defined)* `TBD` class. We refrain from committing to a value of a specific type since for an undefined variable, it is difficult to know about its type information without knowing more about its runtime behavior. So, we delay binding an identifier to the value of a certain type as long as we don't have sufficient information from the error messages about that identifier's expected runtime behavior.

M4. Create-Resource. Mocker creates mock file and directory resources within the execution environment *(env)* in response to `FileNotFoundError` and `NotADirectoryError`. Within a code snippet, when a certain API reads from a file, the API may expect a particular encoding for that file, commonly determined by the file's extension. So, the mocker extracts the missing file's extension and creates a mock file from an example file with the same extension. INCOMPLETER stores a set of example files with common extensions to facilitate mock file creation. If no example file with a certain extension is stored, an example *.txt* file is used by default.

M5. Resolve-Module. If a module used within the code snippet, is missing, we try to install the module using the Python package manager. After trying to install the module in response to a `ModuleNotFoundError`, if the module is still not found, we remove the corresponding import statement so that the module can be mocked using the rule M3. Additionally, suppose an attribute of a module is undefined. In that case, we apply rule M2 or M3 to mock the attribute as a function or as a variable respectively depending on whether the attribute is used as a function call in the code.

M6. Define-Container. After lazily defining an undefined identifier as a generic `TBD` object in rule M3, the execution might encounter follow-up `TypeErrors` since a `TBD` class is very generic. Based on the error message, if a certain TBD class is expected to be an iterable or a subscriptable, we convert that TBD class into a specialized TBDContainer class which can behave like an iterable or a subscriptable. The TBDContainer class has an internal dictionary to store data and the class implements built-in functions like `__getitem__`, `__setitem__`, `__iter__`, and `__next__` to comply with an iterable or a subscriptable interface.

M7. Define-Callable. Another TypeError occurs when a TBD object is supposed to be callable. In this case, the mocker first checks if that TBD object is an attribute of another class. If yes, the mocker uses rule M2 to define a function foo within that class, where the identifier foo was originally undefined and later defined as the TBD object. If foo i.e. the TBD object is not an attribute of another class, the mocker implements the built-in function `__call__` inside the TBD class to make it callable.

M8. Define-Length. If the runtime needs to compute the length of a certain TBD object, the mocker will implement the built-in `__len__` method within that TBD class.

M9. Define-Operator. Some TypeErrors occur when arithmetic/assignment/comparison/logical/bitwise operations are carried out on objects of incompatible type(s). In the case of a unary operator, if the operand is of TBD type, the mocker implements within the TBD class, the built-in function corresponding to the unary operator. In the case of a binary operator, if one of the operands is of TBD type, the mocker similarly implements the built-in function corresponding to the binary operator, within that TBD class. For example, `__le__(self, other)` is the built-in function for the < operator. Based on the error message, for a binary operator, if the other operand is str or int type, the mocker makes the TBD class to inherit from str or int respectively.

M10. Define-Literal. Based on the error message, if a certain TBD object is expected to be a string or an integer, the mocker subclasses that particular TBD class from the built-in str or int classes respectively.

M11. Define-Key. When a KeyError occurs, the error message states only the key that is not found with no reference to which dictionary the key was not found in. To overcome the challenge of locating the target dictionary, we pre-process all lists and dictionaries in the original snippet as objects of the TBDContainer class (ref. M6). Any item-level access of a TBDContainer object occurs through its overridden `__getitem__` built-in function. In the case of a KeyError, the mocker checks the stack trace to find the exact location of the `__getitem__` method where the KeyError had occurred. By comparing this location with the location of the TBDContainer class definitions, we can find out the exact TBDContainer class where the key is missing. Then, the mocker inserts the missing key into the TBDContainer's internal dictionary.

M12. Convert-Literal. When a value error indicates a failure to convert a string to an integer or float, the mocker converts the relevant TBD class, originally inheriting from the str class, to newly inherit from int or float classes respectively.

3.2 Unmocker

This section describes Incompleter's unmocker component, whose goal is to infer the actual types for mocked types and simplify the code accordingly. Figure 3 illustrates one unmocking step. After

```
1  class TBD3:
2     def __init__ ( self, *args, **kwargs ):
3        pass
4  class TBD2:
5     def __init__ ( self, *args, **kwargs ):
6        pass
7  class TBD1:
8     def __init__ ( self ):
9        self.container={ 0: TBD2(), 1: TBD3() }
10    ... ... ...
11 class TBD0:
12    def __init__ ( self, *args, **kwargs ):
13       self.shape = TBD1()
14 df = TBD0 ( )
15 count_row = df.shape[0]
16 count_col = df.shape[1]
17 ( r, c ) = df.shape
```

(a) Mocked snippet.

```
1  class TBD3():
2     def __init__ ( self, *args, **kwargs ):
3        pass
4  class TBD2():
5     def __init__ ( self, *args, **kwargs ):
6        pass
7
8
9
10
11 class TBD0():
12    def __init__ ( self, *args, **kwargs ):
13       self.shape = [ TBD2(), TBD3() ]
14 df = TBD0()  # pandas.DataFrame
15 count_row = df.shape[0]
16 count_col = df.shape[1]
17 ( r, c ) = df.shape
```

(b) Unmocked TBD1

Figure 3: Example of unmocking step using "Deduce-List". Snippet #92 from LExecutor dataset [58].

mocking, the type TBD1 is found to be a container class holding two instances of other mocked types. The unmocker component leverages that observation to infer that TBD1 is a list. Incompleter uses that information to refactor the code snippet by inlining TBD1. Note the change in the initialization of the instance variable shape in class TBD0 at line 13.

Type inference consists of two complementary steps: deduction (Section 3.2.1) and prediction (Section 3.2.2). Deduction reflects on the mock premises expressed in code to determine the types of identifiers [53]. For example, Incompleter used deduction to "unmock" TBD1 in the example above. Prediction uses a machine learning model to predict the types of identifiers. Incompleter uses type prediction as a fallback for cases where deduction fails to determine the type of a mocked type.

Algorithm 2 shows the pseudocode of unmocker. It takes as input a code snippet s and virtual environment env and returns as output a refactored code snippet s', with some mocked types in s resolved. It is worth noting that Incompleter only replaces types that it infers to be built-in type (e.g., list, dictionary). Replacing non-built-in types (i.e., library types and user-defined types) requires domain knowledge. For non-built-in types, Incompleter annotates the mocked type tbd with the type that it predicts using a neural model, delegating the rest of the resolution process to the developer. For example, Incompleter adds a comment to the mocked type TBD0 in Figure 3 to indicate that its type corresponds to a pandas.DataFrame.

The outer loop of Algorithm 2 (Lines 7-24) iterates through the set of mocked types in s. A loop iteration tries to infer the type of the mocked type tbd. Initially, Incompleter uses deduction to infer the type of tbd (Line 8). Function *deduce* evaluates the deduction rules that Incompleter proposes (Section 3.2.1). If the deduced type is a built-in type (Line 9), the unmocker calls the function *replace_and_execute* to replace the instance of the mocked type, *tbd_instance*, with the corresponding deduced value, val, and executes the modified snippet s'. Considering the example from Figure 3, tbd corresponds to the type TBD1, *tbd_instance* corresponds to the instantiation of that type in the assignment `self.shape = TBD1()`, and val corresponds to the list `[TBD2(),TBD3()]`. If the execution of s' is successful, Incompleter proceeds to the next iteration.

Algorithm 2: Unmocker algorithm

```
1  Input mocked code snippet $s$ and environment $env$
2  Output unmocked code snippet $s'$
3  Require: $s$ is free from errors
4  Ensure: coverage(s) = coverage($s'$)
5
6  $s' \leftarrow s$
7  for each tbd in tbds(s) do
8      type, tbd_instance, val ← deduce(tbd, $s'$)            // Section 3.2.1
9      if builtin(type) then
10         $s'$, err_str, _ ← replace_and_execute(tbd_instance, val, $s'$, env)
11         if len(err_str) > 0 then
12             $s' \leftarrow$ restore_snippet()
13             $s'$, type ← predict_and_check(tbd, type, $s'$, env)
14         end
15     end
16     else
17         if not user_defined(type) then
18             $s'$, type ← predict_and_check(tbd, type, $s'$, env)
19         end
20     end
21     if not builtin(type) then
22         $s' \leftarrow$ annotate(tbd, $s'$)
23     end
24 end
25 return remove_unused_tbd_classes($s'$)
26
27 def replace_and_execute (tbd_instance, val, $s'$, env):
28     $s' \leftarrow$ replace(tbd_instance, val, $s'$)
29     return $s'$, execute($s'$, env)     // execute returns (err_str, cov)
30
31 def predict_and_check (tbd, deduced_type, $s'$, env):
32     type, tbd_instance, val ← predict(tbd, $s'$)           // Section 3.2.2
33     if is_builtin(type) then
34         $s'$, _, cov ← replace_and_execute(tbd_instance, val, $s'$, env)
35         if coverage_changed(cov) then
36             return restore_snippet(), deduced_type
37         end
38     end
39     return $s'$, type
```

Otherwise, Incompleter reverts to the previous version of the snippet and calls the function *predict_and_check* to predict the type of *tbd* (Line 13). If the deduced type from line 8 is a library type (i.e., not a built-in type or a user-defined type), unmocker calls the function *predict_and_check* to predict the actual type of *tbd* (Line 18). Finally, the unmocker annotates the definitions of mocked types with inferred types, if they are not built-in types (Line 22), and discards no longer used class definitions (Line 25).

The function *predict_and_check* calls the function *predict* (Section 3.2.2) to predict the type of *tbd* in s'. If the predicted type is a built-in type, the unmocker attempts to execute the modified snippet s' (Line 34), including the corresponding change. If the coverage profile is preserved, the function returns a tuple with the revised snippet and predicted type. Otherwise, it returns the restored snippet and the originally deduced type (Line 36).

3.2.1 *Type Deduction (IncTD).*

Type deduction uses properties from the code to deterministically deduce the type of identifiers. Algorithm 2 (Line 8) calls the function *deduce* for type deduction. As explained in the previous section, the function *deduce* takes a mocked type *tbd*, whose type we want to infer, and code snippet *s*, and returns a tripe including the deduced concrete type and additional meta information. Type deduction supports built-in types (e.g., list, set, dictionary, int, etc.) and user-defined types. Section 3.2.2 discusses type prediction, which supports library types.

For a given pair $\langle tbd, s\rangle$ of mocked type and snippet, the function *deduce* scans a list of *deduction rules*, checking if any rule matches the input pair. A deduction rule checks if a property $R(tbd, s)$, relating *tbd* and *s*, holds. Table 2 lists the deduction rules that Incompleter uses. These rules cover user-defined types (D1), integer and string literals (D2 and D3), and collections (D4-D6). For illustration, let us consider the example from Figure 3. It features a mocked class `TBD1`, with an internal dictionary whose keys are 0 and 1. Incompleter uses the rule D4 to infer that `TBD1` denotes the type `list`. More precisely, the pattern of keys in `self.container`, being sequential, leads the unmocker to interpret the related object (self.shape in `TBD0`) as an instance of a list containing the elements associated with the keys. With these rules, the system can effectively deduce objects into built-in types and user-defined types.

3.2.2 *Type Prediction (IncTP).*

The type predictor and the type deducer have the same goal–type inference. Similar to the type deducer, the type predictor handles built-in types (e.g., `string`, `dict`, `set`), providing a fallback alternative to the deducer for the cases uncovered by the deduction rules (Algorithm 2, Line 13). In contrast to the type deducer, the type predictor handles library types.

Incompleter's type predictor *fine-tunes* an existing SoTA neural type prediction model [58]. Fine-tuning is a popular technique to transfer the knowledge learned during pretraining to target a certain downstream task; a pretrained model (e.g., CodeT5 [66] and UniXCoder [29]) is further trained for the downstream task on some amount of supervised data [68]. Incompleter's type predictor is modeled as a sequence prediction task that is a fine-tuned version of LExecutor's CodeT5 model [58], which itself is a fine-tuned CodeT5 model. CodeT5 is a neural transformer model that takes as input a sequence of tokens and outputs the generated token sequence. Since CodeT5 is based on the T5 architecture [49] that supports multi-task learning, LExecutor and Incompleter both fine-tune a CodeT5 model for type inference.

We chose to fine-tune LExecutor's CodeT5 model because, in terms of Top-1 type prediction accuracy, their CodeT5 model outperforms other baselines, including CodeBERT [25] and Type4Py [43, 58]. LExecutor's CodeT5 predictor is competent in predicting built-in types but does not support library type prediction. We have observed over 30% snippets containing usage of library types. Moreover, Islam et al. report a high usage of library types in Stack Overflow code snippets [34]. So we further fine-tune LExecutor's CodeT5 model to predict library types on top of built-in type prediction and integrate our fine-tuned CodeT5 model as part of our comprehensive unmocking strategy. It is worth noting that IncTP does not use LExecutor's instrumentation or runtime engine, only the neural model.

Fine-tuning consists of training a pre-trained model on a set of input-output pairs. Input to the model begins with a classification instruction followed by the name of the mocked type *tbd* whose actual type we want to predict and the mocked snippet, provided as the context. Incompleter uses the executable mocked snippet instead of the original snippet as the input context as the

Table 2: Description of Deduction Rules. INCOMPLETER's type deducer takes as input the pair $\langle tbd, s \rangle$ of mocked type and snippet, and deduces the type under column "Type Deduced" if the property relating tbd and s, described under column "Property", holds.

Id	Type Deduced	Property
D1	x	*tbd* and user-defined class x (i.e., a class defined in snippet s) have overlapping function names. See M2.
D2	`int`	*tbd* inherits from type `int`. See M9/M10.
D3	`string`	*tbd* inherits from type `string` or implements string-specific methods (e.g., `upper, lower, strip, etc.`). See M9/M10.
D4	`list`	*tbd* has a container attribute and the container's keys are in sequential order, or it implements list-specific methods (e.g., `append, extend, insert, etc.`). This is also the default collection type if a container is present but no other specific type information exists. See M6/M11.
D5	`dict`	*tbd* declares the attribute `container` and implements dict-specific methods (e.g., `items, values, keys, etc.`). See M2/M6/M11.
D6	`set`	*tbd* declares the attribute `container` and implements set-specific methods (e.g., `add, union, difference, etc.`). See M2/M6/M11.

Table 3: Datasets.

Dataset	Source	Size	Use	Kind
Dataset 1	Stack Overflow [58]	241	RQ1	Incomplete
Dataset 2	Stack Overflow [11]	4.7K		Incomplete
Dataset 3	Kaggle [35]	3K	RQ2	Complete

mocked snippet contains rich semantic information which is gradually added to the snippet based on execution feedback and the type deduction step. The input and output format is provided below:

Input. *classify type: <tbd#>: <snippet>*

Output. *<predicted_type>*

In the input, the string literal "classify type" denotes the task directive for the CodeT5 model, *<tbd#>* represents the mocked *tbd* type for which we need to predict the actual type, and *<snippet>* represents the executable mocked snippet. The output *<predicted_type>* represents either a built-in or a library type. In algorithm 2, line 32 shows the call to the function *predict* passing a mocked type, *tbd*, and snippet s'. The *predict* function queries INCOMPLETER's fine-tuned CodeT5 model to predict the actual type of the mocked type *tbd*. Considering the example from Figure 3b, INCOMPLETER uses its type predictor to infer that the mocked type `TBD0` corresponds to the library type `pandas.DataFrame`.

4 Evaluation

This section reports on the evaluation of INCOMPLETER. We pose the following research questions:

- **RQ1:** How does INCOMPLETER perform to successfully execute incomplete code snippets?
- **RQ2:** How does INCOMPLETER perform to infer types?

The first question compares the performance of INCOMPLETER's first component, *mocker*, against the SoTA technique, LEXECUTOR, in terms of executability and coverage. It also evaluates the contribution of each rule towards execution. The second question evaluates the ability of INCOMPLETER's second component, *unmocker*, to infer the types of undefined identifiers and improve the readability of the mocked code snippets.

4.1 Experimental Setup

Dataset. Table 3 shows the datasets we use for evaluating INCOMPLETER. Dataset 1 consists of 823 snippets that Souza and Pradel [58] mined from StackOverflow. We found that that dataset contains snippets with syntax errors (357), indentation errors (13), and snippets that run without raising any errors (212). To avoid confounding effects, we filter out snippets with out-of-scope problems (e.g., indentation and syntax errors) and snippets whose executions do not raise errors. After filtering out those cases, the dataset contains 241 snippets. Given the relatively small size of Dataset 1 and the fact that INCOMPLETER uses that dataset to mine rules (Section 3.1.1), we mined 4.7K incomplete snippets from StackOverflow [11] to create Dataset 2. RQ1 uses datasets 1 and 2, which contain only *incomplete snippets* from StackOverflow. RQ2 requires a dataset of *complete snippets* to enable comparison between the types of the undefined identifiers that INCOMPLETER infers and the ground truth present in the complete snippets from the dataset. So, we collected a third dataset from Kaggle [35] consisting of various tutorial snippets from different sources [4, 7–10].

Table 4: Comparison of executability and coverage metrics between LEXECUTOR and INCOMPLETER.

Technique	Executability	Coverage	
		Statement	Branch
LEXECUTOR	54%	56%	32%
INCOMPLETER	67%	91%	54%

(a) Dataset 1

Technique	Executability	Coverage	
		Statement	Branch
LEXECUTOR	44%	60%	29%
INCOMPLETER	54%	83%	48%

(b) Dataset 2

Metrics. For RQ1, we use the metrics that LEXECUTOR uses: executability and statement coverage [58]. Executability measures the ability to execute a snippet end-to-end without raising exceptions. Statement coverage measures the percentage of the statements from the original snippet that are executed. Additionally, for the snippets that contain branches, we measure the percentage of branches exercised during code execution. Both the statement and branch coverage are averaged across all snippets. We use `coverage.py` [14] to measure coverage. For RQ2, we use standard metrics to evaluate classification problems (e.g., accuracy, F1, etc.).

Type Prediction Training. We use the dataset 3 of complete code snippets to fine-tune LEXECUTOR's CodeT5 model. The data preparation consists of the following steps. For each snippet in the dataset, we remove the first assignment statement for an identifier so the identifier becomes undefined. Prior to removing the assignment statement, we dynamically evaluate the expression assigned to the identifier (i.e., the right-hand side of the assignment) to obtain the actual type t of the identifier (i.e., the ground truth label). This data preparation results in 3K incomplete snippets where one of the identifiers is undefined and we know the ground truth type of that identifier. Such data preparation step is similar to the one that Souza and Pradel use in LExecutor [58]. They execute the training code to record the types like we do. But, they don't remove identifiers to introduce incompleteness. They use lambda wrappers to capture the predicted value with or without incompleteness. For Incompleter, however, we need those incompleteness to enable mocks to be created so that we can train the dataset with the mocks and eventually evaluate our performance. For training the model, we use the mocked snippet as it contains rich semantic information derived from execution which is beneficial for type prediction (Section 3.1). More precisely, we use Incompleter to mock the 3K incomplete snippets and use the pair of mocked type and mocked snippets $\langle tbd, s \rangle$ to define the model input and use the ground truth type t to define the expected model output. We split the dataset into 80% for training and 20% for testing. We use the same model configuration as LExecutor's CodeT5 and train the model on an NVIDIA RTX4060Ti platform. Considering the example from Figure 3b, the model predicts variable `df` as a `pandas.DataFrame`.

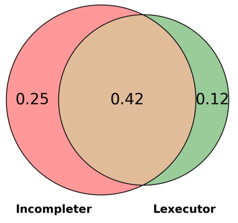
4.2 Answering RQ1: How does Incompleter perform to successfully execute incomplete code snippets?

This question evaluates the ability of the mocker component of Incompleter to modify a given incomplete code snippet to enable successful execution.

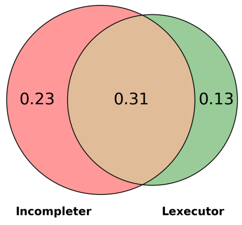
4.2.1 Answering RQ1.1: How Incompleter and LExecutor compare on executability and coverage? Table 4 shows executability and coverage measurements for Incompleter and LExecutor across datasets 1 and 2. On dataset 1, with 241 incomplete snippets of Python code, Incompleter executes 13% more snippets than LExecutor and covers 35% more statements and 22% more branches (Table 4a). On the larger dataset. with 4.7K samples, Incompleter executes 10% more snippets than LExecutor and covers 23% more statements and 19% more branches (Table 4b).

Figure 4 shows Venn diagrams representing the differences and similarities between LExecutor and Incompleter related to executability. Each set represents the snippets that a technique can successfully execute. Although LExecutor and Incompleter commonly resolve a relatively high percentage of code snippets (40.24% and 23.61% on datasets 1 and 2, respectively), no technique subsumes the other and the differences are significant. Incompleter fails when a rule is not present or it lacks the dynamic behavioral cues to deduce a type, whereas it succeeds when such cues are available and it has a rule to cover the error. In contrast, LExecutor fails either when it mispredicts the type or when it predicts a dummy object and cannot continue execution. LExecutor succeeds when it accurately predicts the correct type, which highlights the static nature of LExecutor's approach compared to the dynamic, behavior-driven approach of Incompleter.

Figure 5 shows the distributions of coverage obtained with the execution of the code snippets that each technique produces. Considering both datasets, the distributions of coverage metrics are not normal (Shapiro-Wilk's *p-value*< 0.05) and the differences in statement and branch coverage between Incompleter and LExecutor are statistically significant (Mann-Whitney U's *p-value*< 0.05). Figures 5a and 5b show p-values and effect sizes.

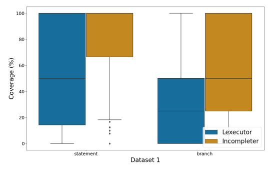


(a) Dataset 1

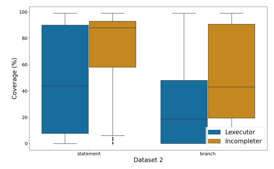


(b) Dataset 2

Figure 4: Venn diagrams showing differences and commonalities between Incompleter and LExecutor's w.r.t. their ability to execute incomplete Python code snippets.



(a) Dataset 1. *p-value*< 0.05, Cliff's δ [65]: stmt (0.49, large), br (0.30, medium)



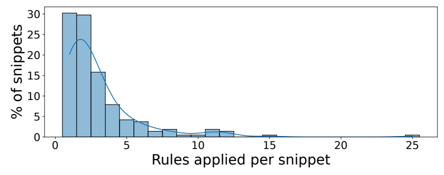
(b) Dataset 2. *p-value*< 0.05 Cliff's δ [65]: stmt(0.46, large), br (0.28, medium)

Figure 5: . Distributions of coverage of LExecutor and Incompleter across datasets 1 and 2.

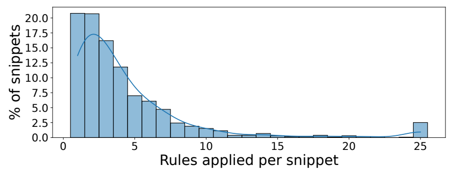
Table 5: Impact of rules on the executability of snippets. *p-exec* and *f-exec* denote partial and full executability, respectively. More (or less) intense shade of gray indicates higher (respectively, lower) discrepancy in rank between datasets 1 and 2.

Rule	Dataset 1		Dataset 2	
	p-exec	f-exec	p-exec	f-exec
Define-Var	221	104	5186	1566
Define-Func	109	53	3205	1252
Define-Container	60	34	842	431
Add-Import	66	28	1135	342
Resolve-Module	54	26	1581	644
Create-Resource	26	16	490	140
Define-Operator	21	13	428	105
Define-Literal	20	12	315	134
Define-Key	44	8	948	109
Define-Callable	5	3	444	230
Define-Length	5	2	79	37
Convert-Literal	2	1	62	19

4.2.2 Answering RQ1.2: What is the contribution of each rule to execution? Table 5 summarizes the contribution of each rule to executability. The metric *p-exec* indicates the number of times a rule fixed an error that emerged with the execution of the snippet, irrespective of whether or not the fix led to a successful (i.e., complete) execution of the snippet. *f-exec* indicates the number of times a rule contributed towards the successful execution of a snippet. Consequently, note that the value of p-exec cannot be inferior to that of f-exec for a given rule. The "p" in "p-exec" refers to partial and the "f" in "f-exec" refers to full. Rules appear sorted in descending order of f-exec values on dataset 1. Shades of gray indicate the rank difference of a rule in the two datasets. We use the



(a) Dataset 1



(b) Dataset 2

Figure 6: Distributions of the number of rules applied per snippet across datasets 1 and 2. The distributions include snippets that Incompleter can fully execute and snippets that are partially executed.

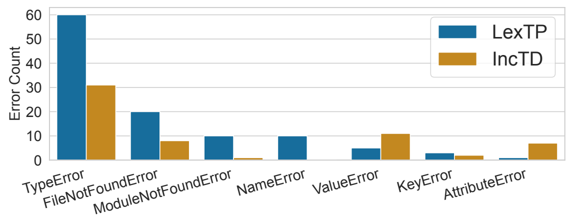


Figure 7: Residual error distribution on *dataset 1* (sorted by LExecutor's error type frequency.)

Table 6: Performance of type inference on *dataset 3*.

Baseline	Acc.	Prec.	Rec.	F1
LexTP	0.17	0.31	0.17	0.21
IncTD	0.23	0.31	0.23	0.27
IncTD+LexTP	0.31	0.29	0.33	0.31
IncTD+IncTP	0.55	0.57	0.60	0.58

following correspondence of color and rank difference: white=0 (i.e., no difference); light gray=1; medium-light gray=2-3; dark gray=>3.

Note that *all rules are relevant but they participate in the resolution of an issue at different proportions.* For example, considering the group of 54% snippets from dataset 2 that Incompleter resolved, two rules contribute to fix over 50% of the cases, four rules contribute to fix over 30% of the cases, and six rules contribute to fix over 11% of the cases. The fact that rule usage is non-uniform does not come with surprise. For example, Define-Var and Define-Func are more commonly applied as variables and functions are the primary missing elements in incomplete code snippets. Also, note that *often multiple rules are needed to resolve an issue.* For example, for 80% of the cases in dataset 2, more than a rule is necessary to resolve an issue. Figure 6 shows histograms for the number of applied rules to resolve issues on datasets 1 and 2 to illustrate that.

Summary of RQ1. Incompleter improves over the SoTA on partial execution on standard evaluation metrics: executability and coverage. Results also show that all 12 transformation rules the mocker proposes are impactful, with those related to defining function and variable appearing more impactful than others.

4.3 Answering RQ2: How does Incompleter perform to infer types?

This question evaluates the ability of the unmocker component of Incompleter to infer types. We evaluate the following four alternative approaches to type inference:

- **LexTP**: LExecutor's CodeT5 model [58];
- **IncTD**: Incompleter's type deduction (Section 3.2.1);
- **IncTD+LexTP**: Integration of IncTD and LexTP;
- **IncTD+IncTP:** Incompleter's type deduction and prediction (Section 3.2).

The suffixes in the names above indicate whether an approach uses deduction (TD) or prediction (TP) for type inference. It also indicates what part of the unmocker (Algorithm 2) is modified. The suffixes TD and TP indicate that the functions *deduce* and *predict* are respectively replaced with the corresponding approach. For example, IncTD+LexTP replaces the function *deduce* with Incompleter's deducer and replaces the function *predict* with LExecutor's predictor. It is worth noting that IncTP is obtained by fine-tuning LExecutor's neural model. Therefore, assuming no loss in predictive power from transfer learning (Section 3.2.2), the combination IncTD+IncTP should subsume the combination IncTD+LexTP.

Figure 7 shows the distribution of *unresolved* errors observed in dataset 1 by kind (of error message) when using LexTP or IncTD alone. The lower the bar the better. Results show that IncTD produces fewer type errors compared to LexTP suggesting that type information of many identifiers cannot be predicted from static context alone. Recall that IncTD deduces type information based on *dynamic* execution behavior while LexTP predicts type information by providing *static* code context to a language modeling task. Results also demonstrate that deduction and prediction can complement each other. For example, note that the number of observed `ValueError` is less in LexTP compared to IncTD.

Table 6 shows the performance of the various type inference approaches considering metrics commonly used in prior work on neural type inference [19, 58, 67], namely accuracy (Acc.), precision (Prec.), recall (Rec.), and the harmonic mean of precision and recall (F1). In this case, the problem is to determine whether or not a technique can correctly identify the type of an identifier, i.e., if the predicted type and the ground truth type are an exact match. Recall that Dataset 3 consists of complete code snippets (Table 3), so it is possible to obtain –for comparison– the concrete types of identifiers by monitoring the execution of the code snippets. Results show that Incompleter's type inference (i.e., the combination IncTD+IncTP) significantly outperforms all other alternative and that each type inference component is relevant. Note the gradual improvement in F1 as new features are incorporated LexTP<IncTD<IncTD+LexTP<IncTD+IncTP.

Summary of RQ2. Results show that (1) Incompleter's type inference performs significantly better than the comparison baselines and that (2) type deduction and type prediction are both relevant.

5 Discussion

5.1 Threats to Validity

An external threat to the validity of our approach is the generalizability of the small number of rules that are mined from the smaller LExecutor dataset. To mitigate this threat, we evaluate Incompleter on a much larger independently mined dataset and demonstrate that the same set of rules can execute and cover more code compared to the SoTA on both the smaller and larger datasets. An internal threat to validity is that Incompleter takes approximately 1.2x longer than the SoTA to successfully execute a snippet. We mitigate this threat by ensuring that Incompleter can achieve higher executability and coverage at the cost of slightly longer runtime. A threat to Incompleter's construct validity arises when a TBD class fails to reveal an object's intended behavior. We note a similar threat in the SoTA where empty DummyObject classes are used. However, we mitigate the threat by progressively adding attributes and functions to a TBD class and by integrating a comprehensive type deduction and prediction strategy to get as close as possible to a concrete built-in or library type.

5.2 Limitations

While Incompleter shows promising results, several limitations warrant discussion. First, the semantic validity of the fixes generated by Incompleter is not guaranteed. Fixes might require inspection to ensure they adhere to the intent of the code. Second, creating and mining the rules that Incompleter uses to handle incomplete snippets is a manual process, which can be time-consuming. Third, Incompleter relies on LExecutor's neural model for type prediction, which is constrained by the distribution of the training data. Consequently, this limitation affects the range of custom (library) types Incompleter is able to infer. Finally, Incompleter currently focuses on fixing one file at a time, which may not be sufficient for projects with cross-file dependencies.

6 Case Study: Youtube-dl

This section reports on a case study that contributes to demonstrate Incompleter's usefulness and soundness. The study uses the project Youtube-dl, a highly-popular open-source project to download YouTube videos [18]. The use case is debugging. More precisely, the focused task is to reproduce a bug manifested through a failing test on a partially-defined application code. We obtained test cases of the Youtube-dl project from BugsInPy [57], a public repository of real-world Python bugs with documented test failures. We chose BugsInPy for its comprehensive collection of bugs with a ground truth to evaluate the soundness of Incompleter's output.

Methodology. Incompleter currently operates in a single Python file, so our first step is to aggregate all dependent files in one, merging the code under test with its corresponding test to obtain self-contained test files. Following the consolidation step above, we identify the buggy function and add incompleteness. For that, we systematically remove one assignment at a time starting from the top of the function in the execution path of the bug. Because of the introduced incompleteness, the execution would encounter runtime errors before even catching the actual bug. Then, we request Incompleter to resolve those runtime issues and to reproduce the original bug by advancing the code execution. We selected the test cases sequentially from bug 1 to bug 28, discarding those that tested the same logic as previously selected cases or didn't have any unique assignments to remove. In this process, we specifically targeted unique assignment statements for removal.

```
def test_prepare_filename():   # test function
 def fname(templ):
  ydl=YoutubeDL({'outtmpl': templ})
  return ydl.prepare_filename(info)
 assertEqual(fname('Hello %(title1)s'),'Hello $PATH')
class YoutubeDL:
 def prepare_filename(self, info_dict):  # code under test
  ...
  autonumber_size = self.params.get('autonumber_size')
  if autonumber_size is None: autonumber_size = 5 ...
  field_size_compat_map={'playlist_index': ...,'autonumber':
        autonumber_size}
  FIELD_SIZE_COMPAT_RE=r'...'
  if re.search(FIELD_SIZE_COMPAT_RE, ...)
    ... ... ...
  filename = expand_path(...)
  return sanitize_path(filename)
if __name__=='__main__': test_prepare_filename()
```

Figure 8: YouTube-dl bug(#19 from BugsInPy dataset [57]).

Results. Out of 10 bugs selected for our analysis, corresponding to 10 test cases, we explored Incompleter's capability by removing then testing a total of 15 unique assignments, one at a time. For some test cases, this process was repeated up to three times, with a different assignment removed each time. The rationale is to assess the tool's effectiveness across various complexities of removed assignments. Incompleter successfully restored six of these modifications to their original assertion errors. Two other instances resulted in assertion errors similar to the original but with slight content differences, and the remaining 7 modifications led to unresolved errors. On average, two rules were applied to each assignment removal, with M2 being the most frequently used. LExecutor had nearly identical performance in terms of total successful and unsuccessful restorations, though the content of errors varied slightly between the two. LExecutor successfully restored five modifications to their original assertion errors, three resulted in the original error with different content, and seven led to unresolved errors. For passing cases, LExecutor identified `None` and dictionary three times each. For failing cases, LExecutor identified dictionary the most for three times.

Analysis. Figure 8 shows an excerpt of one of the tests from YouTube-dl that we analyze. When testing the original code (i.e., the code without incompleteness added), execution raises an `AssertionError`. We analyze two different variants of this code; each removing one of the following highlighted assignments from `prepare_filename`. After removing `autonumber_size`'s definition, execution raises `NameError` preventing the actual bug from surfacing. After running the incomplete consolidated code through Incompleter, the execution of the mocked snippet reaches the end of the execution path resulting the same assertion error as that of the original bug. Removal of the `FIELD_SIZE_COMPAT_RE`'s definition also leads to a `NameError`. After processing this code through Incompleter, execution encounters a `TypeError` caused by the regular expression library expecting a compiled string. Since the error does not specify which TBD object requires correction, Incompleter fails to resolve the issue, preventing execution from completing.

In this case study, both Incompleter and LExecutor were able to reproduce a considerable number of real-world bugs from the YouTube-DL project. In a few cases, both techniques could trigger a bug with a similar `AssertionError` message as that of the originally reported bug. In some cases, neither of the techniques could reproduce the bug. However, the majority of the cases suggest the benefit of using both Incompleter and LExecutor to enable partial execution of incomplete buggy code snippets for detecting bugs that would otherwise remain elusive had the code stayed incomplete.

7 Related Work

Partial code execution. The problem of analyzing and partially executing incomplete code has been studied in different programming languages [21, 22, 28, 45, 46, 59]. In Java, Dong et al. solve type constraints against a knowledge base to resolve unknown import statements from incomplete snippets [23] and Terragni et al. developed CSNIPPEX, which automates the synthesis of compilable Java code from incomplete snippets by resolving external dependencies and generating necessary import statements [61]. In Python, LExecutor infers built-in types of undefined variables, attributes, and return values of functions. The inferred types are later mapped to a fixed number of concrete values [58]. We show that (1) Incompleter is complementary to LExecutor (Figure 4) and (2) can leverage learning for improved type prediction (Section 3.2.2).

Code completion. Code completion is the task of using prior context to complete code [38, 39, 50]. Zhang et al. proposed a retrieval-augmented code completion while accounting for repository-level information [69]. Liu et al. introduced a multi-task learning model to complete code identifiers and types [40]. Subramanian et al. presented Baker, a tool that links source code examples to API documentation, thereby enhancing traditional documentation with practical examples and facilitating code completion [60]. Incompleter generates code similar to code completion models but the code is generated not as a next statement prediction task but instead to remove incompleteness scattered throughout the code.

Code repair. Both rule-based and neural-based approaches have been applied in code repair, which automatically detects and fixes bugs [30, 36, 54]. While Incompleter does not perform code repair itself, it provides an environment conducive to detecting and facilitating the repair of bugs through partial execution by defining missing identifiers and creating necessary resources.

Code reuse. NLP2Code [16] and Blueprint [15] are tools proposed to facilitate code reuse. Reid et al. [51] proposed Node Code Query (NCQ), a tool that integrates JavaScript package search, snippet search, and linting-based fixes within a Read-Eval-Print Loop environment to facilitate code search. Terragni and Salza introduced APIzator which uses static analysis-based rules to convert a Java code snippet from Stack Overflow into a reusable wrapper method with automatically identified arguments and return values [62]. Incompleter also uses rules to mock snippets and deduce types but does so with dynamic feedback from executions. Partial code execution and code search are different problems. It remains to evaluate how they can complement each other.

Mock generation. In software testing, different objects, dependencies, and services are implemented as mock components for faster deterministic, and isolated testing [24, 37, 47, 63]. Recently, Salva and Blot showed an approach to develop mocks using model learning [55]. We remain to assess how Incompleter can integrate with and benefit from such approaches.

Symbolic execution. Symbolic execution is a technique for test input generation that uses constraint solvers to generate inputs from constraints denoting program paths. Ruaro et al.'s [52] proposed SyML, a technique that uses supervised learning for prioritizing execution paths in dynamic symbolic execution for vulnerability discovery in binary programs. Sapra et al. [56] proposed CutiePy, a dynamic symbolic execution tool tailored for Python, which manages the language's dynamic typing and semantics by balancing concrete and symbolic execution strategies. Similar to symbolic execution we create symbols in the form of generic types to mock undefined identifiers in the code. We then use a combination of rule-based deduction and neural prediction of generic types defined in the mocked snippet. We remain to evaluate the integration of symbolic execution and Incompleter for the determination of concrete values.

Other Related Works. Neural techniques have been applied for inferring unknown types of identifiers in code [13, 17, 20, 44, 48]. Inline testing has been recently proposed to test specific code statements [41, 42]. Unlike our approach, inline testing ensures the correctness of fully defined code blocks by validating against predefined oracles. Inline testing and partial code execution are complementary problems.

8 Conclusion

Partial execution is the problem of executing a code fragment with missing elements, such as variable and function definitions. Partial execution enables several applications, including dynamic analyses and debugging. We propose Incompleter, a technique for partial execution that leverages execution feedback to progressively complete a code snippet. Results show that Incompleter obtains higher executability and coverage compared to LExecutor [58].

Considering a dataset of 4.7K incomplete StackOverflow snippets, Incompleter enables the execution of 10% more code snippets compared to LExecutor and covers 23% more statements. We also show that Incompleter's type inference significantly improves over LExecutor's type inference, with a 37% higher F1 score. We also conduct a case study to demonstrate Incompleter's usefulness. Results show that Incompleter faithfully reproduces 8 of the 15 bugs of the program under test.

Data Availability

The source code and data are publicly available [6, 31].

Acknowledgments

We would like to thank Uchswas Paul for his contribution to the preparation of the second dataset and the reviewers for their time and feedback. This work was supported and funded by the National Science Foundation grant numbers 2319472 and 2349961.

References

[1] 2021. Python integer incrementing with ++ — stackoverflow.com. https://stackoverflow.com/questions/2632677/python-integer-incrementing-with. [Accessed 12-01-2023].

[2] 2022. GitHub REST API documentation - GitHub Docs — docs.github.com. https://docs.github.com/en/rest?apiVersion=2022-11-28. [Accessed 11-01-2023].

[3] 2024. Built-in Exceptions — docs.python.org. https://docs.python.org/3/library/exceptions.html. [Accessed 21-07-2024].

[4] 2024. Comprehensive Compilation of Programming Problems and Solutions in C, Java, C++, and Python — csinfo360.com. https://www.csinfo360.com/p/ccjavapython-practice-questions-with.html. [Accessed 03-01-2024].

[5] 2024. Find nearest value in numpy array — stackoverflow.com. https://stackoverflow.com/questions/2566412/find-nearest-value-in-numpy-array. [Accessed 12-01-2023].

[6] 2024. Incompleter's GitHub repository. https://github.com/ncsu-swat/incompleter. [Accessed 07-03-2024].

[7] 2024. Python Exercise with Practice Questions and Solutions - GeeksforGeeks — geeksforgeeks.org. https://www.geeksforgeeks.org/python-exercises-practice-questions-and-solutions/. [Accessed 03-01-2024].

[8] 2024. Python Exercises, Practice, Solution - w3resource — w3resource.com. https://www.w3resource.com/python-exercises/. [Accessed 03-01-2024].

[9] 2024. Python Programming Examples | Python Programs - Sanfoundry — sanfoundry.com. https://www.sanfoundry.com/python-problems-solutions/. [Accessed 03-01-2024].

[10] 2024. Python-programming-exercises/100+ Python challenging programming exercises.txt at master · zhiwehu/Python-programming-exercises — github.com. https://github.com/zhiwehu/Python-programming-exercises/blob/master/100+%20Python%20challenging%20programming%20exercises.txt. [Accessed 03-01-2024].

[11] 2024. Stack Overflow - Where Developers Learn, Share, & Build Careers — stackoverflow.com. https://stackoverflow.com/. [Accessed 01-02-2024].

[12] 2024. The Python Standard Library — docs.python.org. https://docs.python.org/3/library/index.html. [Accessed 10-29-2023].

[13] Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: neural type hints. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 91–105. https://doi.org/10.1145/3385412.3385997

[14] Ned et al. Batchelder. 2023. Coverage.py &x2014; Coverage.py 7.4.0 documentation — coverage.readthedocs.io. https://coverage.readthedocs.io/en/7.4.0/. [Accessed 28-03-2024].

[15] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R Klemmer. 2010. Example-centric programming: integrating web search into the development environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 513–522.

[16] Brock Angus Campbell and Christoph Treude. 2017. NLP2Code: Code snippet content assist via natural language tasks. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 628–632.

[17] Yiu Wai Chow, Luca Di Grazia, and Michael Pradel. 2024. PyTy: Repairing Static Type Errors in Python. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. ACM. https://doi.org/10.1145/3597503.3639184

[18] +786 contributors. 2024. Youtube-dl. https://github.com/ytdl-org/youtube-dl. [Accessed 04-10-2024].

[19] Siwei Cui, Gang Zhao, Zeyu Dai, Luochao Wang, Ruihong Huang, and Jeff Huang. 2021. Pyinfer: Deep learning semantic type inference for python variables. *arXiv preprint arXiv:2106.14316* (2021).

[20] Siwei Cui, Gang Zhao, Zeyu Dai, Luochao Wang, Ruihong Huang, and Jeff Huang. 2021. PYInfer: Deep Learning Semantic Type Inference for Python Variables. *CoRR* abs/2106.14316 (2021). arXiv:2106.14316 https://arxiv.org/abs/2106.14316

[21] Barthélémy Dagenais and Laurie Hendren. 2008. Enabling Static Analysis for Partial Java Programs. *Sigplan Notices - SIGPLAN* 43, 313–328. https://doi.org/10.1145/1449955.1449790

[22] Barthélémy Dagenais and Martin P. Robillard. 2012. Recovering traceability links between an API and its learning resources. In *2012 34th International Conference on Software Engineering (ICSE)*. 47–57. https://doi.org/10.1109/ICSE.2012.6227207

[23] Yiwen Dong, Tianxiao Gu, Yongqiang Tian, and Chengnian Sun. 2022. SnR: constraint-based type inference for incomplete Java code snippets. In *Proceedings of the 44th International Conference on Software Engineering*. 1982–1993.

[24] Mattia Fazzini, Alessandra Gorla, and Alessandro Orso. 2020. A framework for automated test mocking of mobile apps. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 1204–1208.

[25] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. 1536–1547.

[26] Ehsan Firouzi, Ashkan Sami, Foutse Khomh, and Gias Uddin. 2020. On the use of C Unsafe Code Context: An Empirical Study of Stack Overflow. In *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (Bari, Italy) *(ESEM '20)*. Association for Computing Machinery, New York, NY, USA, Article 39, 6 pages. https://doi.org/10.1145/3382494.3422165

[27] Akalanka Galappaththi, Sarah Nadi, and Christoph Treude. 2022. Does this apply to me? an empirical study of technical context in stack overflow. In *Proceedings of the 19th International Conference on Mining Software Repositories* (Pittsburgh, Pennsylvania) *(MSR '22)*. Association for Computing Machinery, New York, NY, USA, 23–34. https://doi.org/10.1145/3524842.3528435

[28] Patrice Godefroid. 2014. Micro execution. In *Proceedings of the 36th International Conference on Software Engineering*. 539–549.

[29] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. arXiv:2203.03850 [cs.CL]

[30] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. *Proceedings of the AAAI Conference on Artificial Intelligence* 31, 1 (Feb. 2017). https://doi.org/10.1609/aaai.v31i1.10742

[31] Ishrak Hayet. 2024. *Incompleter (source and dataset)*. https://doi.org/10.5281/zenodo.13147364

[32] Md Monir Hossain, Nima Mahmoudi, Changyuan Lin, Hamzeh Khazaei, and Abram Hindle. 2019. Executability of python snippets in stack overflow. *arXiv preprint arXiv:1907.04908* (2019).

[33] Qing Huang, Jiahui Zhu, Zhilong Li, Zhenchang Xing, Changjing Wang, and Xiwei Xu. 2023. Pcr-chain: Partial code reuse assisted by hierarchical chaining of prompts on frozen copilot. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 1–5.

[34] Md Johirul Islam, Hoan Anh Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. What do developers ask about ml libraries? a large-scale study using stack overflow. *arXiv preprint arXiv:1906.11940* (2019).

[35] Link An Jarad. 2022. Natural Language to Python Code. https://doi.org/10.34740/KAGGLE/DSV/3512475

[36] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72. https://doi.org/10.1109/TSE.2011.104

[37] Boyang Li, Christopher Vendome, Mario Linares-Vásquez, Denys Poshyvanyk, and Nicholas Kraft. 2016. Automatically Documenting Unit Test Cases. 341–352. https://doi.org/10.1109/ICST.2016.30

[38] Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. 2018. Code Completion with Neural Attention and Pointer Networks. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI-2018)*. International Joint Conferences on Artificial Intelligence Organization. https://doi.org/10.24963/ijcai.2018/578

[39] Fang Liu, Ge Li, Bolin Wei, Xin Xia, Zhiyi Fu, and Zhi Jin. 2020. A Self-Attentional Neural Architecture for Code Completion with Multi-Task Learning. arXiv:1909.06983 [cs.SE]

[40] Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. 2020. Multi-task learning based pre-trained language model for code completion. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 473–485.

[41] Yu Liu, Pengyu Nie, Owolabi Legunsen, and Milos Gligoric. 2022. Inline Tests. arXiv:2209.06315 [cs.SE] https://arxiv.org/abs/2209.06315

[42] Yu Liu, Zachary Thurston, Alan Han, Pengyu Nie, Milos Gligoric, and Owolabi Legunsen. 2023. pytest-Inline: An Inline Testing Tool for Python. In *Proceedings of the 45th International Conference on Software Engineering: Companion Proceedings* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 161–164. https://doi.org/10.1109/ICSE-Companion58688.2023.00046

[43] Amir M Mir, Evaldas Latoškinas, Sebastian Proksch, and Georgios Gousios. 2022. Type4py: Practical deep similarity learning-based type inference for python. In *Proceedings of the 44th International Conference on Software Engineering*. 2241–2252.

[44] Amir M. Mir, Evaldas Latoškinas, Sebastian Proksch, and Georgios Gousios. 2022. Type4Py: practical deep similarity learning-based type inference for python. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2241–2252. https://doi.org/10.1145/3510003.3510124

[45] Alon Mishne, Sharon Shoham, and Eran Yahav. 2012. Typestate-based semantic code search over partial programs. *ACM SIGPLAN Notices* 47, 997–1016. https://doi.org/10.1145/2384616.2384689

[46] Joel Ossher, Sushil Bajracharya, and Cristina Lopes. 2010. Automated dependency resolution for open source software. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. 130–140. https://doi.org/10.1109/MSR.2010.5463346

[47] Gustavo Pereira and Andre Hora. 2020. Assessing mock classes: An empirical study. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 453–463.

[48] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2020. TypeWriter: neural type prediction with search-based validation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and*

Symposium on the Foundations of Software Engineering (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 209–220. https://doi.org/10.1145/3368089.3409715

[49] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research* 21, 140 (2020), 1–67.

[50] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. *SIGPLAN Not.* 49, 6 (jun 2014), 419–428. https://doi.org/10.1145/2666356.2594321

[51] Brittany Reid, Marcelo d'Amorim, Markus Wagner, and Christoph Treude. 2023. NCQ: code reuse support for Node. js developers. *IEEE Transactions on Software Engineering* 49, 5 (2023), 3205–3225.

[52] Nicola Ruaro, Kyle Zeng, Lukas Dresel, Mario Polino, Tiffany Bao, Andrea Continella, Stefano Zanero, Christopher Kruegel, and Giovanni Vigna. 2021. SyML: Guiding Symbolic Execution Toward Vulnerable States Through Pattern Learning. In *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses* (San Sebastian, Spain) *(RAID '21)*. Association for Computing Machinery, New York, NY, USA, 456–468. https://doi.org/10.1145/3471621.3471865

[53] Ian Rumfitt. 2011. Inference, deduction, logic. *Knowing how: Essays on knowledge, mind, and action* (2011), 334–60.

[54] Ripon K. Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R. Prasad. 2017. ELIXIR: effective object oriented program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE '17)*. IEEE Press, 648–659.

[55] Sébastien Salva and Elliott Blot. 2020. Using Model Learning for the Generation of Mock Components. In *Testing Software and Systems: 32nd IFIP WG 6.1 International Conference, ICTSS 2020, Naples, Italy, December 9–11, 2020, Proceedings 32*. Springer, 3–19.

[56] Samir Sapra, Marius Minea, Sagar Chaki, Arie Gurfinkel, and Edmund M. Clarke. 2013. Finding Errors in Python Programs Using Dynamic Symbolic Execution. In *Testing Software and Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 283–289.

[57] SOAR Lab. 2023. BugsInPy: A Database of Existing Bugs in Python Programs to Enable Controlled Testing and Debugging Studies. https://github.com/soarsmu/BugsInPy/tree/master.

[58] Beatriz Souza and Michael Pradel. 2023. LExecutor: Learning-Guided Execution. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (, San Francisco, CA, USA,) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1522–1534. https://doi.org/10.1145/3611643.3616254

[59] Siddharth Subramanian and Reid Holmes. 2013. Making sense of online code snippets. *IEEE International Working Conference on Mining Software Repositories*, 85–88. https://doi.org/10.1109/MSR.2013.6624012

[60] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. 2014. Live API documentation. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 643–652. https://doi.org/10.1145/2568225.2568313

[61] Valerio Terragni, Yepang Liu, and Shing-Chi Cheung. 2016. CSNIPPEX: automated synthesis of compilable code snippets from QA sites. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 118–129. https://doi.org/10.1145/2931037.2931058

[62] Valerio Terragni and Pasquale Salza. 2021. APIzation: Generating reusable APIs from StackOverflow code snippets. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 542–554.

[63] N. Tillmann and W. Schulte. 2006. Mock-object generation with behavior. In *Proceedings. 21st IEEE International Conference on Automated Software Engineering*. IEEE Computer Society, Los Alamitos, CA, USA, 365–368. https://doi.org/10.1109/ASE.2006.51

[64] Hugo van Kemenade. 2024. Top PyPI Packages — hugovk.github.io. https://hugovk.github.io/top-pypi-packages/. [Accessed 11-01-2023].

[65] András Vargha and Harold D Delaney. 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132.

[66] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).

[67] Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. 2016. Python probabilistic type inference with natural language support. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*. 607–618.

[68] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. 2023. Dive into Deep Learning. arXiv:2106.11342 [cs.LG]

[69] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. Repocoder: Repository-level code completion through iterative retrieval and generation. *arXiv preprint arXiv:2303.12570* (2023).

[70] Hao Zhong and Xiaoyin Wang. 2017. Boosting complete-code tool for partial program. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 671–681.