Bolt-on, Compact, and Rapid Program Slicing for Notebooks [Technical Report]

Shreya Shankar[†1], Stephen Macke[†2], Sarah Chasins[1], Andrew Head[3], Aditya Parameswaran[1]
[1]University of California, Berkeley {shreyashankar,schasins,adityagp}@berkeley.edu
[2]Unaffiliated stephen.macke@gmail.com
[3]University of Pennsylvania head@seas.upenn.edu
[†]Equal contribution (order determined by coin flip)

ABSTRACT

Computational notebooks are commonly used for iterative workflows, such as in exploratory data analysis. This process lends itself to the accumulation of old code and hidden state, making it hard for users to reason about the lineage of important results such as plots depicting insights or trained machine learning models. We present NBSLICER, a dynamic slicer optimized for the notebook setting whose instrumentation for resolving dynamic data dependencies is both *bolt-on* (and therefore portable) and *switchable* (allowing it to be selectively disabled in order to reduce instrumentation overhead). We demonstrate NBSLICER's ability to construct *backward* (i.e., historical statement dependencies) and *forward* (i.e., statements affected by the "rerun" of an earlier statement) slices, to aid in notebook reproducibility and reactivity, respectively. Comparing NBSLICER with a static slicer on 374 real notebook sessions, we found that NBSLICER's slices are, on average, 66% and 54% smaller for backward and forward slices, respectively.

1 INTRODUCTION

Computational notebooks, and Project Jupyter [27] in particular, have revolutionized the workflows of data scientists [34, 35]. Notebooks admit a flexible execution model that segments units of computation into so-called "cells" that can easily be back-referenced for editing, duplication, or reordering, with intermediate program state persisted to memory between subsequent cell executions. This iterative cell-based execution modality is ideal for rapid prototyping and testing of hypotheses, a cornerstone of typical data science work, and has led to their extensive usage—as of October 2020, there were nearly 10 million notebooks available on GitHub [18].

Notebooks are Messy. The popularity of notebooks has come with increased scrutiny; as such, notebooks now have a number of well-documented disadvantages, related primarily to their tendency to accumulate cruft in the form of both visible notebook code [21, 25, 38] and invisible in-memory program state [16]. Data science workflows, in particular, tend to be exploratory in nature [25]. These flaws can make execution behavior in notebooks difficult to reason about and cause misleading or incorrect findings, leading to confusion during ad-hoc exploration, prototyping, and iteration.

Organizing Notebook Iteration with Program Slicing. To address notebook shortcomings, recent work has developed approaches based on *backward program slicing* to gather code in messy notebooks [19, 23], thereby making it easier for data scientists to retrace their steps. In brief, a *backward program slice* determines a (typically smaller) subset of program statements that affect some other program statement(s); in the context of notebooks, backward program slicing captures the lineage required to reproduce the outputs of one or more cells; e.g., to "gather" code that was written in an ad-hoc fashion, potentially out-of-order across multiple notebook cells, into a clean script that reliably reproduces the data scientist's analyses. The more compact this slice is, the smaller the lineage, meaning it is more efficient to re-execute for reproducibility while preserving accuracy.

Forward program slicing also has applications in data science workflows in computational notebooks. In brief, a *forward program slice* determines the set of statements that are affected by a given statement or set of statements. Forward slicing is traditionally associated with debugging tools with applicability more widespread than just notebooks, but in the context of notebooks particularly, forward program slicing can be combined with other analysis techniques to automatically (or *reactively*) re-execute all the cells that could be affected (via data dependencies) by some other cell, ensuring that cells do not become stale. A reactivity tool for notebooks can be thought of as performing a form of materialized view maintenance — specifically, "refreshing" the notebook after an earlier cell is rerun by re-executing dependent cells. Such reactivity features help to push the burden of tracking what cells have become stale away from the user and down into the notebook kernel. This feature is particularly helpful for data science workflows, which can involve toggling many values (e.g., hyperparameters) and re-executing what might be dozens of downstream data transformation dependencies. Once again, correctness (i.e., making sure we are rerunning everything affected) while minimizing slice size (i.e.,only rerunning what is needed) is key to ensuring interactivity during exploratory data analysis.

Notebook-Centric Slicing: Challenges. Backward and forward slicing can help automate away some of the messiness inherent in notebook-resident data science workflows and thereby improve downstream reproducibility. Ideally, slices should be as small as possible in order to eliminate extraneous computation, while preserving correctness of the underlying program. However, computation of slices that are simultaneous *small* and *accurate*, and without noticeable degradation of existing notebook behavior and performance, is difficult to achieve in practice. We now outline challenges we faced while developing NBSLICER, a state-of-the-art dynamic slicer optimized specifically for the notebook setting, along with contributions that addressed each challenge.

Challenge 1: Small and accurate program slices. Backward slicing was first explored in the context of code gathering in notebooks by Head et al. [19]. However, it is not difficult to construct cases wherein the static slicing technique used in [19] will yield overly-conservative (and therefore larger than necessary) slices. One approach to reduce slice size while preserving correctness is to leverage *dynamic slicing* [29]. In contrast with static slicing, which

infers dependencies between statements solely from programs' text, dynamic slicing infers such dependencies using the actual runtime state of the program.

We now illustrate such dynamic behavior with an example. Consider an abridged verison of a real example session from the replay dataset described in Section 4.1 wherein NBSLICER, which uses dynamic slicing, yields a more compact slice than a static slicer. In this session, a user explores a dataset and fits a least-squares model to it:

In [1]:
```
housing
= pd.io.parsers.read_csv('housing.txt', sep='\s+')
```

In [2]:
```
corr = housing.corr()
np.abs(corr.sort(columns=['crime', 'residential']))
```

In [3]:
```
def LR_solve(X, y):
    """ Solves a linear regression problem."""
    if len(X.shape) == 2:
        A = np.hstock( (np.ones((X.shape[0],1)), X) )
    else:
        A = np.hstack( (np.ones((X.shape[0],1)),
        np.expand_dims(X,1)) )
    rtn = np.linalg.lstsq(A, y)
    return rtn[0]
```

In [4]:
```
X = housing[:, :(- 1)]
b = housing[:, (- 1)]
LR_solve(X, b)
```

The user runs into an error because there was a typo of `np.hstock` in cell 3, motivating them to rerun cells after a fix:

In [5]:
```
def LR_solve(X, y):
    """ Solves a linear regression problem."""
    if len(X.shape) == 2:
        A = np.hstack( (np.ones((X.shape[0],1)), X) )
    else:
        A = np.hstack( (np.ones((X.shape[0],1)),
        np.expand_dims(X,1)) )
    rtn = np.linalg.lstsq(A, y)
    return rtn[0]
```

In [6]:
```
X = housing[:, :(- 1)]
b = housing[:, (- 1)]
LR_solve(X, b)
```

When running NBSLICER to reconstruct the last cell's output, we see that the dynamic slice contains only cells 1, 5, and 6. However, a static slicer would include *all cells in its slice*. We discuss why several cells are included in the static slicer below:

- Cell 2 calls functions from external libraries `numpy` and `pandas`. A dynamic slicer can figure out that the `np` and `pd` objects reference these external libraries and do not modify state; thus NBSLICER is able to exclude these cells.
- Cells 4 and 5 attempt to fit a least squares model to the data. However, the typo in cell 4 — `np.hstock` (instead of `np.hstack`) — still parses and is included in the static slice. Again, these cells also include calls to external libraries and are therefore included in the static slice even though they do not modify state.

Though dynamic slicing can reduce slice size while preserving correctness, it has traditionally required deep modification of the underlying interpreter or virtual machine responsible for running code in the target programming language [10, 44] and furthermore introduces nontrivial overhead. This presents a number of problems from both performance and portability standpoints, which we describe next.

Challenge 2: Balancing performance with portability. Dynamic slicing for smaller and more accurate slices is itself not a new idea, but it comes with the drawback of significant overhead, since it requires visibility into executing code in order to infer runtime data dependencies. For example, Python ≥ 3.8 supports tracing of individual bytecode operations via builtin system tracing utilities [3] — functionality upon which runtime data dependency tracking could be implemented — but we found that leveraging this approach leads to slowdowns of 100× or more. Such overhead is clearly infeasible for typical data science workloads, which are interactive in nature.

An alternative approach is to instrument the interpreter itself in a lower-level language such as C, as in prior work [10]. However, if a dynamic slicer were to require a custom-compiled Python interpreter, data scientists wishing to use said slicer would need to re-download a new interpreter every time a new version of Python is released — a process with considerably more friction than the typical workflow of grabbing a library from PyPI or Anaconda. From a maintainer standpoint, supporting multiple versions of Python is also less than ideal. Indeed, previous Python dynamic slicers [10] instrument much older versions of the Python interpreter and seem to no longer be publicly available. Overall, implementing a dynamic slicer that can be installed portably as a simple Python library, while simultaneously introducing negligible overhead and retaining interactive latencies, is entirely nontrivial.

NBSLICER: A Hybrid Static-Dynamic Slicer. To make dynamic slicing work for the notebook setting, wherein interactivity and portability are key, we developed *NBSLICER*, a novel hybrid static-dynamic slicing tool optimized for the notebook setting. The key technical contribution of NBSLICER is to instrument data scientists' code at the level of the *abstract syntax tree* (AST), rather than at the level of Python bytecode, allowing for portable and *switchable* instrumentation that can be selectively enabled and disabled in order to bound the amount of overhead induced by such instrumentation, all while retaining most of its benefits regarding compactness and accuracy of slices.

When instrumentation is enabled, nodes in a program's AST are transformed in a way that injects additional observability into the data referenced by the program, but without changing the program's behavior. The aforementioned "additional observability" takes the form of a *data dependency resolver* that decorates portions of the *uninstrumented* AST with the data dependencies that would be difficult to infer purely statically. From here, a static analyzer operates on this enriched AST and short-circuits whenever it encounters subtrees that have been marked as having dynamically-resolved data dependencies. For example, NBSLICER will attempt to dynamically resolve data dependencies involved in `Call` nodes; e.g., in our earlier example, it will resolve the dependencies involved with the statement

```
A = np.hstack( (np.ones((X.shape[0],1)), X) )
```

as edges from `np` and `X` to `A`, and it will *not* include an edge from `np` to `np` as it is aware that calls to `numpy` functions such as `hstack` and `ones` are non-mutating. If the AST were not enriched with this dynamically-resolved information, the fallback static slicer might conclude that the call to `np.hstack` could modify some internal

Statement	Read Set	Write Set
`import numpy as np`	{}	$\{np_1\}$
`import pandas as pd`	{}	$\{pd_2\}$
`df = pd.read_csv("housing.txt")`	$\{pd_2\}$	$\{df_3\}$
`X, y = df[:, :-1], df[:, -1]`	$\{df_3\}$	$\{X_4, y_4\}$
`X = np.expand_dims(X, 1)`	$\{X_4, np_1\}$	$\{X_5\}$
`A = np.hstack((np.ones((X.shape[0], 1)), X))`	$\{X_5, np_1\}$	$\{A_6\}$
`rtn = np.linalg.lstsq(A, y)`	$\{A_6, y_4\}$	$\{rtn_7\}$

Figure 1: Example of how statements' read and write sets induce dependencies between them, which are then used for slice construction. The "sliced" statement is highlighted. Orange arrows represent backward slice construction. Blue arrows represent forward slice construction.

state to the `np` object, and include that statement in slices that involve subsequent usages of `np`, even if they do not involve `A`.

Switchable Instrumentation. The key property of our hybrid approach to slicing that admits interactive latencies in spite of instrumentation is that this instrumentation can be disabled when performance would suffer (e.g., in tight loops or repeated function calls) and re-enabled when the additional overhead would no longer adversely impact the user experience (e.g., for start-to-finish execution of top-level / outermost "basic block" statements). In order to facilitate this *switchable instrumentation*, it is vital to maintain a correspondence between executing instructions and the program's AST (which is where the static component of slicing operates). It is for this reason that our instrumentation operates at the level of the program's AST rather than at the level of bytecode, since an instrumented AST can more easily reference its uninstrumented counterpart (as such back-pointers can be inserted when the original, uninstrumented AST is transformed). In contrast, bytecode instrumentation operates over streams of Python opcodes, which lack metadata that would allow for constructing a correspondence with the program AST. Although it is possible to get some information such as the current line in the program text, it is not easily possible to deduce, e.g., to which attribute reference a `LOAD_ATTR` instruction corresponds to in a given line, should multiple be present.

Outline. The rest of this paper is organized as follows. Section 2 gives relevant background context on program slicing along with our formal problem definition. Section 3 presents our hybrid static-dynamic dependency resolution techniques for portable and low-overhead program slicing in computational notebooks, which we evaluate in terms of speed, slice size, and correctness in Section 4. We compare and contrast NBSLICER with related work in Section 5 before concluding in Section 6.

2 PROBLEM

In this section, we define requisite terminology and introduce the backward and forward slicing problems in the context of computational notebooks. Figure 1 will serve as a running example on which we anchor the discussion.

2.1 Preliminaries

We begin by defining *forward* and *backward slices*, and show how they are constructed from *data dependencies*.

Definition 1 [Forward and Backward Slices]. *A* backward slice *for a statement* `s` *consists of the transitive closure over statements that may affect any of the variables read by* `s`. *A* forward slice *for* `s` *consists of the transitive closure over statements that reference variables that are affected by* `s`.

In Figure 1, the sliced statement, s_4, is highlighted. The backward slice is constructed by transitively following the arrows pointing from s_4 (colored in orange), and the forward slice is constructed by transitively following the arrows pointing *to* s_4 (colored in blue).

A statement `s` affects a variable `v` whenever there exists a *data dependency* from variables that are read by `s` to `v`:

Definition 2 [Data Dependency]. *A variable* `y` *is* data-dependent *on another variable* `x` *if the same program statement both reads from* `x` *and writes to* `y`.

In Figure 1, for the highlighted statement s_4, there is a data dependency from df_3 to each of X_4 and y_4, and in general every statement has data dependencies from each variable in its read set to each variable in its write set.

Our notion of data dependency deviates from the traditional definition in that it is *data-centric* (i.e., the dependency exists between pieces of data), whereas the traditional definition is *statement-centric* (i.e., the dependency exists between two statements in a read-after-write / RAW scenario). The two notions are equivalent in that our definition just swap the roles of nodes and edges, so that data take the role of nodes and statements take the role of edges (i.e., the opposite of what is depicted in Figure 1). Our definition focuses on data because data is the component that is more difficult to determine statically, while statements are always known ahead-of-time. Section 3.2 is devoted to discussing how to pin down the data referred to by program statements.

For traditional program slicing, one has to consider both data dependencies and *control dependencies* to determine which statements actually get executed; however, in NBSLICER, we only consider data dependencies. This particularity stems from the design decision to use top-level statements for inclusion in or exclusion from slices.

Definition 3 [Top-Level Statement]. *A* top-level *statement is a program statement (and corresponding* `stmt` *AST node) that appears at the top-level of the abstract syntax tree of some Python program; i.e., it is not nested inside any other AST* `stmt` *node.*

Top-level statements are so-called in reference to their appearance in the Python grammar specification [2] on the right-hand-side of the `file` rule, which is the only non-statement non-terminal to generate statement non-terminals. i.e., top-level statements are not nested inside of any other statements, as for all the statements in Figure 1. Ignoring the pathological case of exceptions, top-level statements cannot have control dependencies, as we will see next.

Top-Level Slicing. To simplify the design of our slicer, we include program statements at the granularity of top-level statements in the abstract syntax tree. By doing so, NBSLICER does not need to consider whether individual statements in, e.g., the body of a loop or `if` statement need to be included — they are rather included or excluded in an all-or-nothing fashion along with the top-level statement. For example, consider this program:

```
In []:
    lst = list(range(1, 10))   # top-level
    if reduction == "sum":  # top-level
        base = 0
        func = lambda x, y: x + y
    elif reduction == "power":
        base = 1
        func = lambda x, y: x * y
    else:
        raise ValueError("invalid reduction")
    for val in lst:        # top-level
        base = func(base, val)
```

This example only has three top-level statements: the assignment to `lst`, the entire `if` statement (including all sub-statements), and the entire `for` loop (including the loop body) – indented statements that appear inside of `if` blocks or loop bodies are not considered for slice inclusion independently.

This design choice has the advantage that NBSLICER needs only track data dependencies across top-level basic blocks, and does not need to consider control dependencies within blocks. Although this design choice could be considered conservative from a slice size standpoint, we found that is fitting for the notebook use case, wherein the majority of the program is in the form of these top-level statements. Furthermore, backward slicing in a notebook setting is typically applied to code gathering, , while forward slicing is typically used for reactivity (bringing the notebook state in harmony with the written code) — both use cases wherein this coarser granularity is more justified when compared with traditional applications of slicing , such as debugging, for which the finest granularity slices possible are desired in order to see exactly what executed.

Timestamps. To disambiguate between different versions of the same variable appearing in some data dependency, each variable is associated with a *timestamp* corresponding to the top-level statement that most recently modified it:

Definition 4 [Timestamp]. *A statement* $\mathtt{s}$ *has a* timestamp *of* k *if it appears at or within the* k*th top-level statement that has executed. A variable* $\mathtt{v}$ *has timestamp* k *if the most recent statement to write to* $\mathtt{v}$ *has timestamp* k*.*

By our definition of timestamp, there is exactly one top-level statement with timestamp k, which we denote by $\mathtt{s}_k$. If $\mathtt{s}_k$ writes variable $\mathtt{v}$, we similarly use $\mathtt{v}_k$ to denote its resultant value, when the extra disambiguation is needed (as for the read / write sets of Figure 1).

2.2 Problem Statements

In this work, we focus on two applications of slices in the notebook context: backward and forward. Suppose we have a notebook session S whose execution log is composed of top-level statements $\{\mathtt{s}_1,\mathtt{s}_2,\mathtt{s}_3,\ldots,\mathtt{s}_n\}$, and we are interested in computing slice T.

Problem 1 (Backward Slicing). *For any* $\mathtt{s}_k \in S$*, construct a minimal (in terms of the number of top-level statements) slice* $T \subseteq S$ *such that* T *includes every statement upon which* $\mathtt{s}_k$ *(transitively) depends.*

In a notebook setting, backward slicing can be used to gather messy code into a coherent script, to aid in later reproducibility. *Forward slicing*, in contrast, can be used to enable *reactive semantics*, wherein dependent statements are automatically re-executed when upstream statements are rerun (also aiding reproducibility, by keeping the notebook state consistent with the program text):

Problem 2 (Forward Slicing). *For any* $\mathtt{s}_k \in S$*, construct a minimal (in terms of the number of top-level statements) slice* $T \subseteq S$ *such that* T *includes every statement that (transitively) depends on* $\mathtt{s}_k$*.*

In order to solve these problems, NBSLICER keeps the metadata presented in Figure 1 materialized in memory and updates it as users execute statements in the notebook. When users want to construct a slice with a given statement `s`, we perform a "lineage query" in the appropriate direction, either forward or backward, to compute the statements reachable from `s`(or statements from which `s` is reachable, respectively).

The success of a slicing algorithm depends on the ability to rapidly and precisely identify data dependencies. Despite prior work for program slicing in Python [10, 19], there are no approaches for doing so *dynamically* in a notebook context, wherein portability and interactivity are vital. In Section 3, we contribute such an approach for dynamically identifying data dependencies. Our approach leverages instrumentation that is both *bolt-on* (not requiring a reimplementation of the Python interpreter) as well as *switchable* (allowing it to be low-overhead).

3 HYBRID SLICING

In this section, we present the key components underlying NBSLICER; namely, its traditional AST-level static data dependency resolver, as well as its dynamic data dependency resolver, and how these two components work together in the context of NBSLICER's data model to implement a dynamic slicer with static fallback. We show how the dependency resolver, which uses AST-level instrumentation, facilitates portable and *switchable* instrumentation that can be disabled in order to keep overhead bounded, thereby allowing data scientists to use NBSLICER without adjusting their interactive workflows.

3.1 Static Data Dependency Resolution

Before discussing how NBSLICER statically captures data dependencies, we begin by introducing its model used to represent such dependencies.

Data Model. NBSLICER uses `nbsafety`'s [30] abstraction of a "symbol" to capture the basic unit of data capable of participating in a data dependency. In this model, symbols are used to capture data at the resolution of unqualified variable names (e.g., `lst`), as well as data nested as subscripts or attributes (e.g., `lst[42]` or `foo.bar`, respectively). We will use the following code snippet to illustrate:

```
In []:  lst = [1, 2, 3, 4, 5]   # line 1
        lst[3] = 42             # line 2
        y = lst[2] + 7          # line 3
        lst = [6, 7, 8, 9, 10]  # line 4
        y = lst[2] + 7          # line 5
```

Each symbol is associated with up to two namespaces, for representing data nested as either attributes or subscripts. For example, the symbol `lst` is associated with a namespace containing nested symbols for `lst[2]` and `lst[3]`. Symbols nested inside of another symbol's namespace are implicitly data-dependent on that symbol. For example, `lst[2]` is dependent on `lst`.

Slicing with Field-Granular Data Dependencies. In the above example, line 3 has data dependency from the symbol `lst[2]` to `y`. These *record-* or *field-*granular dependencies allow NBSLICER to compute the backward slice for line 3 as just the first and third lines. Field-granular data dependencies allow NBSLICER to capture the property that a write to `lst[3]` will not affect something that depends on `lst[2]`, thereby allowing it to compute smaller slices.

Note that the implicit data dependencies of nested symbols on their parents ensures correctness of slices that involve both nested and non-nested data. For example, the backward slice for line 5 contains lines 4 and 5, since `y` depends on `lst[2]` and `lst[2]` depends on `lst`.

Static Resolution. For simple cases, data dependencies can be inferred solely via statically analyzing code. Consider, for example, the snippet `x = lst[42]`. In this case, NBSLICER's static dependency resolver will operate on an AST `Assign` node, whose target consists of a `Name` in a store context, `x`, and whose right hand side consists of

a subscripted `Name` node in a load context, `lst[42]`. For such a case, NBSLICER will draw a data dependency between `x` and `lst[42]`.

However, static analysis can get us only so far in the case of additional dynamic behavior. Suppose the snippet instead uses a variable for the list index: `x = lst[y]`. Here it is still possible to draw the data dependency just by statically inspecting the code, if we have the additional knowledge that `y` is equal to **42** at runtime. However, for the case `x = lst[f()]`, wherein `f` is a function that is called and which eventually returns **42**, we quickly reach the limits of what static code inspection can give us, since in the general case there is no way to know exactly what `f` will do without actually running it.

3.2 Dynamic Data Dependency Resolution

From the example in Section 3.1, we see that NBSLICER must rely on additional instrumentation in order to capture data dependencies from code with dynamic behavior, such as function calls. Providing additional instrumentation without modifying the interpreter (in order to satisfy our portability requirement) is challenging, however. As mentioned in Section 1, one approach built into Python ≥ 3.8 is to attach a tracing handler that runs for each `opcode` event; i.e., for every single bytecode instruction, but this approach introduces unacceptable overhead. In fact, *any* kind of instrumentation introduces significant overhead. The key innovation of NBSLICER for coping with this issue is to make it possible to toggle instrumentation on and off, in order to bound the resultant overhead. When instrumentation is active, NBSLICER uses a dynamic dependency resolver to handle cases such as function calls; when it is inactive, NBSLICER falls back to a static dependency resolver.

We considered trying to develop such "switchable" instrumentation for tracing `opcode` events, but found it difficult practically to connect the stream of running Python bytecode back to the program source code. Such functionality is necessary before we can attach resolved dynamic dependencies to the program's AST, which in turn is needed in order to facilitate dynamic slicing with static fallback. Furthermore, Python opcodes change between different versions of Python, complicating the portability aspect. Though we suspect it is possible in theory to leverage opcode tracing, we found that a much simpler and straightforward approach was to rewrite the program's AST in order to provide switchable instrumentation for dynamic dependency resolution, which we describe next.

Embedded Instrumentation via AST Transformations. The solution we take for NBSLICER is to embed tracing instrumentation directly into notebook programs' source code. Suppose we have two functions, `f` and `g`, which return a list `lst` and the constant **5**, respectively. At a high level, code such as `x = f()[g()]` is rewritten as

```
x = resolve(f())[resolve(g())]
```

where the `resolve(...)` calls handle non-statically-resolvable subtrees of the abstract syntax tree and decorate their roots with symbol information. This symbol information then allows the static dependency resolver to function as a de-facto *dynamic* dependency resolver by short-circuiting when it reaches these non-statically-resolvable subtrees and drawing edges for the dynamically-resolved symbols. This process is depicted in Figure 2.

3.3 Switchable Instrumentation

While instrumentation is enabled, overhead is high, even for the AST-level instrumentation described in Section 3.2. The key advantage of AST-level instrumentation is that it can be temporarily disabled without requiring changes to the behavior of the data dependency resolver, in order to avoid paying the additional overhead from instrumentation. When enabled for some portion of the program, the corresponding AST is decorated with inferred dynamic dependencies; when disabled, this decoration is absent, and the resolver falls back to conservative static behavior.

The question then becomes: when should we disable instrumentation in order to bring performance to a level on par with uninstrumented notebook code, and without affecting the quality of program slices? We developed two heuristics for this decision that work well in practice. These heuristics disable instrumentation (i) for program statements that have run before (e.g., due to their appearance in a loop body or a function that is called repeatedly), and (ii) for calls to external libraries, whose code is defined outside the notebook context. Thanks to these heuristics, the additional overhead from instrumentation is bounded by an amount proportional to the size of the notebook program text. We now discuss these heuristics in more detail, with particular attention to how they interact with data dependency resolution.

Trace-Once Semantics. To reduce instrumentation overhead, NBSLICER only runs a statement with instrumentation the first time the statement in which it is nested runs. This is realized by rewriting, e.g., **for** loops according to the below example:

```
In []:
for ... in ...:
    if first_loop_iteration:
        ... # instrumented loop body
    else:
        ... # uninstrumented loop body
```

If the loop body contains conditionals with multiple branches, any branches not initially taken will have corresponding AST subtrees that lack dynamically-resolved dependencies; for such cases, the dependency resolution will fall back to static behavior.

What about for statements that did run during the first iteration (and whose AST subtrees therefore have resolved dynamic dependencies attached)? The attached dependency information, while accurate for the first loop iteration, may be incomplete. Two choices for handling this incompleteness are to (i) leverage both the incomplete resolved dynamic dependencies in conjunction with static dependencies, or (ii) to just use the incomplete dynamic dependencies in the hope that they are "good enough".

During experimentation on a real corpus of notebook sessions, we found that, empirically, it virtually always sufficed to take option (ii) in practice. For example, consider the below code snippet:

```
In []:
for i in range(len(10)):
    lst[i] += x[i]()
print(", ".join(lst[i] for i in range(10)))
```

In the above example, the loop body will only resolve dynamic dependencies for `x[0]()`, and not for other entries of `x`. While it is theoretically possible that subsequent entries would resolve to additional dependencies needed for slice correctness, we never observed such corner cases in our experiments (Section 4), for which NBSLICER always yielded correct slices despite using the incomplete dependency information yielded by option (ii). Data scientists who require more conservative behavior can configure NBSLICER to take option (i) instead and leverage both static and dynamic dependency resolution for statements that execute multiple times, if desired.

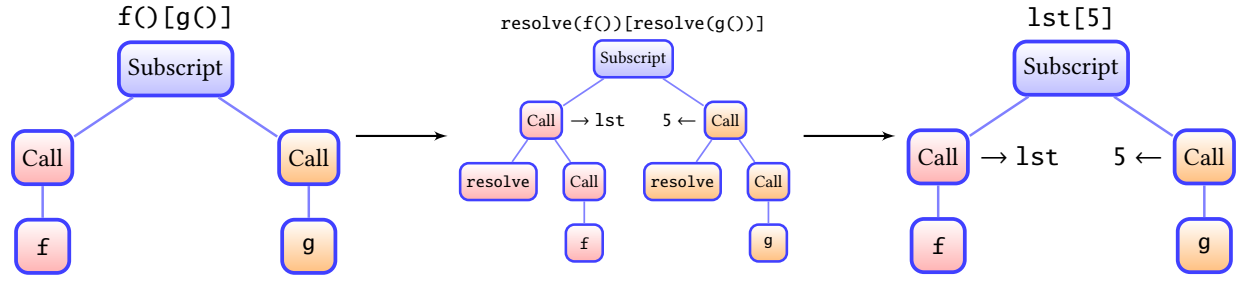


Figure 2: AST transformation and symbol resolution example

Rule-Based Dependency Inference for Libraries. The final performance optimization NBSLICER leverages is to only trace code written in the notebook, so that API calls to libraries such as, e.g., `numpy` or `Pandas` are uninstrumented. However, calls to external libraries (or methods on objects instantiated from external libraries) can still introduce data dependencies. NBSLICER uses a simple rule-based algorithm to cope with this reality. We defer the full details to the technical report, but we give one example below. Suppose we are running code involving a dataframe `df` with column `A`:

```
In []:  df.A.dropna(inplace=inplace)
        print(df.A.avg())
```

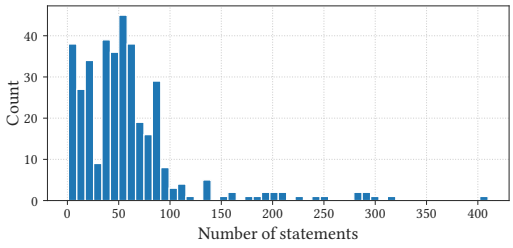
Depending on the value of `inplace`, the call to `dropna(...)` will either return a value, or `None`. The default behavior of NBSLICER is to assume that external library calls that return a value do not introduce data dependencies, while calls that return `None` will introduce edges from the old object value (i.e. previous value of `df.A` in this case) and any arguments (none in this case) to the new object value. In this way, when `inplace` is `False` above, the backward slice for the last statement is just the last statement, but when it is `True`, the backward slice for the last statement includes the previous call the `dropna(...)`, which is the correct behavior. Note that without a dynamic dependency resolver capable of checking return values of functions, we would not know which case holds.

4 EMPIRICAL STUDY

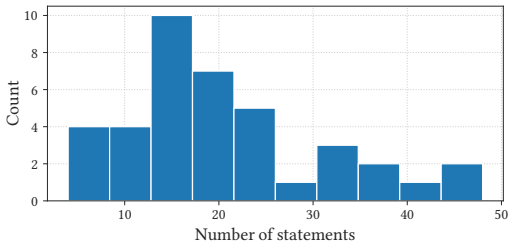
We evaluated NBSLICER's effectiveness in two dimensions: slice size and computational overhead. We ran two sets of experiments: one to evaluate slice size by comparing NBSLICER to `nbgather` [19], a static slicer for computational notebooks in Python, and another to evaluate NBSLICER's computational overhead compared to a regular `IPython` kernel with no extensions. Our experiments are based off the version of the static slicer used in `nbgather` at the time of the CHI 2019 paper [19]. Since then, Microsoft has made some improvements to the `nbgather` notebook extension; however, we were unable to run the latest version of `nbgather` due to incompatibility issues with newer Jupyter versions. For all experiments, we programmatically executed code cells in a Python script instead of manually executing them in the Jupyter interface to eliminate human-created delays between running cells.

4.1 Notebook Data

For each evaluation dimension—slice size and computational overhead—we collected a dataset of notebooks. The first dataset consists of notebook *sessions*, or ordered lists of code cells run by a user while their `IPython` kernels were active. We found that most of these notebooks executed in under a tenth of a second, motivating us to collect a second dataset of more computationally-intensive notebooks to measure NBSLICER's runtime overhead on.



(a) Session Replay Dataset



(b) D2L Dataset

Figure 3: Histograms of statement counts in each dataset.

Session Replay Data. We obtained notebook sessions from the repository collected in the `nbsafety` paper [30]. This repository consists of sessions corresponding to `IPython` notebooks scraped from public GitHub repositories. Macke et al. implemented a cleaning process to ensure the notebooks did not contain malicicious code or calls to external services (e.g., AWS, Spark, Postgres, MySQL), removing about $\frac{1}{6}$ of the notebooks[30]. The notebook sessions included in our experiments did not require filesystem use (e.g., reads or writes to external files) .

A session begins when the notebook kernel starts, and ends when the kernel shuts down. Session data was extracted from `history.sqlite` files produced by `IPython` and includes information about individual cell executions, including the source code and execution counter for every cell execution. We filtered the repository to include only 374 sessions we could deterministically

run without errors. Many of these sessions contain cells with typos, syntax errors, or runtime errors, reflecting programming errors and exceptions encountered as the notebooks were created. Figure 3(a) represents a histogram of the number of statements in these notebook sessions.

Dive into Deep Learning Data. Many of the notebooks collected in Section 4.3 analyze only small amounts of data and have short runtimes. To conduct the computational overhead analysis, we obtained a set of deep learning tutorial notebooks from the Dive Into Deep Learning (D2L) initiative that have longer runtimes. These notebooks do not contain cell replay data; each notebook is only represented by an `.ipynb` file, or a `.json` containing an ordered list of source code cells visible in the notebook environment.

We downloaded the PyTorch notebooks from the D2L repository and successfully ran 71 notebooks locally. Our dataset for the computational overhead experiments consists of these 71 notebooks. Figure 3(b) represents a histogram of the number of statements in the notebooks.

4.2 Metrics

We now describe metrics used to evaluate our dynamic slicer.

Slice size. The slice size is defined as the number of statements in that slice divided by the number of statements in that full program. Assuming equivalent execution results, smaller slice sizes are preferred because they require less time to execute and are more concise for human readers.

Latency. The latency is defined as the notebook execution time under the `nbsafety` kernel with NBSLICER enabled divided by the notebook execution time under the regular `IPython` kernel. Smaller latencies are better (i.e. smaller runtimes).

4.3 Static vs Dynamic Slicer Results

Both slicers constructed a list of statement dependencies. For each slice, we concatenate the statements to compute slice size.

Slice correctness. For each of the 374 notebook sessions, we verified that NBSLICER's slices correctly reproduced any printed output of the last statement in the original program. One benefit of our dynamic slicer is that it will not include statements that result in runtime errors in a slice, while static slices might include such statements (for example, if the statement can be successfully parsed). Many statements in `.ipynb` sessions can have runtime errors, especially when the notebook author is running a cell for the first time. Only 103 of the statically-generated slices *did not* contain statements with runtime errors, demonstrating the nature of notebook history sessions to contain cells with broken code.

Slice sizes. Figure 4 shows histograms of the slice sizes from the 374 notebook sessions. Overall, slices from NBSLICER are significantly smaller (Wilcoxon test statistic 317.0, $p = 2.0 \times 10^{-39}$), as shown in Table 1. We observe that the distribution of slice size differences is long-tailed, with the minimum absolute difference being -12 statements and the maximum absolute difference being 131 statements.

4.4 Computational Overhead Results

We ran notebooks from the D2L dataset described in Section 4.1 under the `nbsafety` and regular `IPython` kernels. The median latency is approximately 2.59.

Latencies. Table 2 and Figure 5 indicate that the latency is mainly 1:1 between a regular `IPython` kernel and NBSLICER for varying

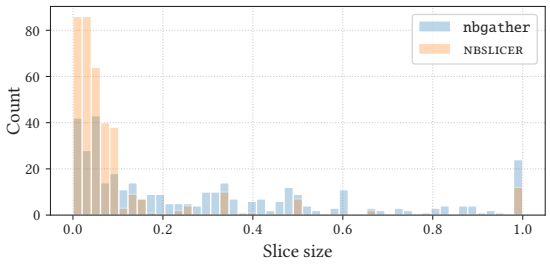


Figure 4: Histogram of slice sizes captured in all 374 sessions in the session replay dataset. There are 50 evenly spaced buckets.

	`nbgather`	NBSLICER	difference
min	0.005319	0.003521	-0.292683
mean	0.297776	0.102191	0.195585
50%	0.185800	0.043478	0.070423
75%	0.475000	0.083333	0.333333
max	1.0	1.0	0.975610

Table 1: Statistics for raw slice sizes and differences in slice sizes for the session replay dataset. Wilcoxon test statistic 317.0 and p-value 2.0×10^{-39}.

	`IPython` kernel	NBSLICER	Latency
mean	45.79	47.48	3.53
50%	0.39	1.22	2.59
75%	4.72	8.75	3.08
max	1012	993.0	83.50

Table 2: Statistics for runtimes (in seconds) under different methods and the latency for the D2L dataset. We note that the maximum runtime corresponds to a notebook that downloads some data from the internet; networking delays may account for the difference.

runtimes. There are a few notebooks that execute quickly where the tracing overhead is large, possibly due to networking delays in downloading datasets. Most of our slowdowns probably will not disrupt a user's experience in an interactive notebook setting; less than 100ms is a common heuristic for how much time a UI has to respond for its response to have imperceivable lag [32]. Based on the assumption that end-to-end program execution time is not as important as the ease of experimentation offered by notebook tools, we believe such multipliers are acceptable to users . Additionally, although our experiments ran the code cells in a notebook programatically, in practice, users mostly manually run cells an interface such as Jupyter notebook, with some time passing in between cell runs. This "think time" [41] would increase the end to end notebook session runtime but not necessarily the NBSLICER overhead, contributing to a smaller observed latency in practice than what our experiments demonstrate .

Slice construction times. Since statement dependency graphs are constructed at runtime, computing a slice only requires a depth-first search that can be performed on the fly . We observed small slice construction times, as described in Table 3. The median slice construction time was approximately 0.3 ms. For notebooks with hundreds of thousands of cells , it is possible for the slice construction to take several seconds. However, in practice, such notebooks are unlikely to exist.

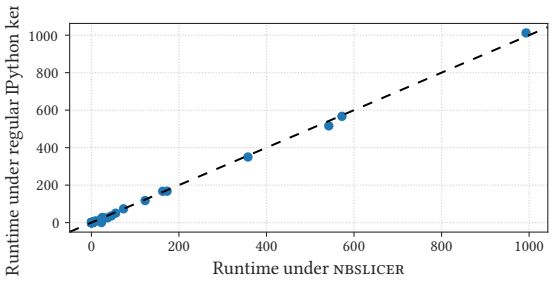


Figure 5: Regular `IPython` vs NBSLICER runtimes for the D2L dataset. The relationship is close to linear, meaning that the NBSLICER runtimes are roughly equivalent to the normal runtimes across the board.

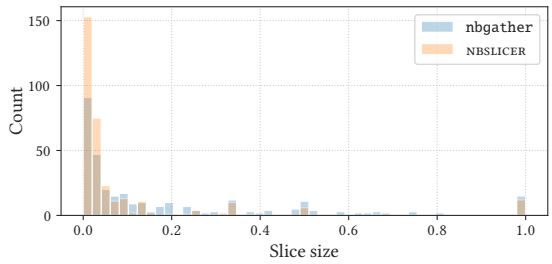


Figure 6: Histogram of forward slice sizes. There are 50 evenly-spaced buckets.

4.5 Extension: Notebook Reactivity

Reactive notebooks have increased in popularity for data exploration because of their ability to stay up-to-date, like a spreadsheet, at all times [8, 15]. We can use forward slicing to support reactivity: if a user reruns cell c, what cells have statements depending on any of the top-level statements appearing in c and thus need to be run again? Here, we show that a dynamic slicer gives considerable gains over a static slicer because the dynamic slicer requires fewer statements of code to rerun.

Methodology. Given data dependencies, NBSLICER computes the forward slice as described in Section 2.2. For our static slicing baseline, we use `nbgather`'s static slicer to get backwards edges in the dependency graph, then reverse directions of the edges. For each notebook in the session replay dataset, we pick a random cell to rerun, compute static and dynamic forward slices, and compare these slice sizes.

Results. Figure 6 shows a histogram of forward slice sizes for NBSLICER and the static slicer used in `nbgather`. We observe that NBSLICER produces smaller slice sizes, with the median NBSLICER slice size being 0.02 and the median `nbgather` slice size being 0.07 (Wilcoxon test statistic 282.0, p-value 8.8×10^{-24}). Detailed results are shown in Table 4.

5 RELATED WORK

Messy Notebooks. Computational notebooks, being widely used for exploratory programming and data analysis, lend themselves to "messy" programming [19]. Users employ all sorts of poor coding practices while using notebooks, such as storing old versions of code as comments [46] and frequently reordering code cells [26]. Messes accumulate as users iterate on their notebooks, forcing them to refactor their notebooks by deleting and consolidating cells prior to sharing their work with others [39]. This process can be tedious and yield nonreproducible results [37].

	construction time
mean	0.583
50%	0.329
75%	0.568
max	6.337

Table 3: NBSLICER slice construction time statistics (ms) for the D2L dataset.

	`nbgather`	NBSLICER	difference
min	0.003205	0.003205	-0.238636
mean	0.201735	0.093295	0.108440
50%	0.071429	0.021739	0.000000
75%	0.296474	0.052632	0.133333
max	1.0	1.0	0.950000

Table 4: Statistics for raw forward slice sizes and differences in forward slice sizes. Wilcoxon test statistic 282.0 and p-value 8.8×10^{-24}.

Users of live programming environments, such as notebooks, complain that these environments lack adequate support for storing and retrieving historical versions of code [14, 38]. Studies show that data scientists often rely on informal versioning techniques such as copy-pasting code or commenting out code, and these techniques are useful because they support fast retrieval of old versions [22]. Many approaches to provide information about code history are challenging to extend due to the complex nature of dependencies between code, data, and analysis intents in notebooks [24]. Prior works consider automatically versioning code cells and distinct abstract syntax tree (AST) representations at execution time [24, 37]. `nbgather` attempts to clean up versions by performing static analysis on code cells [19]. However, static analysis does not leverage information that could be collected at runtime, such as data provenance, motivating NBSLICER.

Data Management for Notebooks. Previous work has proposed treating notebooks as dataflow computation graphs [1, 28, 48]. Such work requires restricts flexibility because users are forced to explicitly annotate cells with their ordering or succumb to a forced temporal ordering of cells. `nbsafety` extends these dataflow-based approaches preserves existing notebook semantics (e.g., execution in any order) to keep track of staleness for all symbols in a notebook [30]. `nbsafety` leverages data lineage information to identify stale cells and alert users with potential staleness errors.

The database community has a rich history of work in data versioning and provenance. Coarse-grained provenance is typically explored in data management for workflow systems [5, 9, 13]. For example, Burrito tracks file provenance [17] and noWorkflow analyzes provenance of scripts [31]. While these solutions analyze worklows post-hoc, `nbsafety` and Vizier leverage lineage to track information "online," or at runtime. Computing provenance in an online fashion is beneficial in notebook environments, where users frequently iterate on their code and data rather than write a script once from start to finish. NBSLICER leverages this provenance information to slice notebooks in a cleaner fashion.

Program Slicing. Program slicing techniques have been extensively studied in the programming languages (PL) community to assist with debugging, understanding, and maintaining code [42]. Several studies have shown than conservative static slices are imprecise,

and dynamic slices tend to be smaller than static slices [7, 33, 43]. Many existing approaches to dynamic slicing operate at the compiler, bytecode, or VM level [4, 12, 47], which may require a different environment setup. However, Sen et al. argue that "portable" slicing and applicability across different versions of a programming language are desirable for popular programming languages [40]. It is impractical to expect data scientists to compile a specific version of Python instrumented with slicing, link this version to their PYTHONPATH, and configure their development environment appropriately , motivating a “bolt-on" dynamic slicing tool for Python.

Program Slicing for Python. Many popular Python tools leverage static analysis, such as Pylint and flake8. However, dynamic features are widely used in Python [20], and static analysis techniques can fail to produce tight, or short, slices. This can pose problems when users rerun slices that contain code that they did not intend to rerun [6]. Much of the existing work in static and dynamic slicing for Python programs focuses on scripts, not computational notebooks [11, 36, 45]. noWorkflow leverages dynamic program slicing to capture fine-grained provenance in Python scripts but is unable to track dependencies on complex data structures such as lists, collections, or objects [36], which are frequently used in computational notebooks and supported by NBSLICER. We were unable to find any open-source dynamic slicer for Python.

6 CONCLUSION

We introduced NBSLICER, a portable and performant dynamic slicer for computational notebooks. We described its implementation via bolt-on AST instrumentations and discussed how it achieves interactive performance with its hybrid static-dynamic dependency resolver. Finally, for both backward and forward slicing problems, we demonstrated NBSLICER’s low runtime overhead as well as its ability to produce smaller slices than a static slicer in a corpus of real notebook sessions.

BIBLIOGRAPHY

[1] 2018 (accessed December 1, 2020). *Datalore.* https://datalore.jetbrains.com/.

[2] 2022. Python Docs: Full Grammar Specification. https://docs.python.org/3/reference/grammar.html. Date accessed: 2022-02-25.

[3] 2022. sys: System-specific parameters and functions. https://docs.python.org/3/library/sys.html#sys.settrace. Date accessed: 2022-02-28.

[4] Gagan Agrawal and Liang Guo. 2001. Evaluating Explicitly Context-Sensitive Program Slicing. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (Snowbird, Utah, USA) *(PASTE ’01)*. Association for Computing Machinery, New York, NY, USA, 6–12. https://doi.org/10.1145/379605.379630

[5] Manish Kumar Anand, Shawn Bowers, Timothy Mcphillips, and Bertram Ludäscher. 2009. Exploring scientific workflow provenance using hybrid queries over nested data and lineage graphs. In *Scientific and Statistical Database Management*. Springer, 237–254.

[6] David Binkley, Nicolas Gold, Mark Harman, Syed Islam, Jens Krinke, and Shin Yoo. 2015. ORBS and the limits of static slicing. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 1–10. https://doi.org/10.1109/SCAM.2015.7335396

[7] David W Binkley and Mark Harman. 2004. A survey of empirical results on program slicing. *Adv. Comput.* 62, 105178 (2004), 105–178.

[8] Mike Bostock. 2020 (accessed March 1, 2020). *Observable: The magic notebook for exploring data.* https://observablehq.com/.

[9] Shawn Bowers. 2012. Scientific workflow, provenance, and data modeling challenges and approaches. *Journal on Data Semantics* 1, 1 (2012), 19–30.

[10] Zhifei Chen, Lin Chen, Yuming Zhou, Zhaogui Xu, William C Chu, and Baowen Xu. 2014. Dynamic slicing of Python programs. In *2014 IEEE 38th Annual Computer Software and Applications Conference*. IEEE, 219–228.

[11] Zhifei Chen, Lin Chen, Yuming Zhou, Zhaogui Xu, William C. Chu, and Baowen Xu. 2014. Dynamic Slicing of Python Programs. In *Proceedings of the 2014 IEEE 38th Annual Computer Software and Applications Conference (COMPSAC ’14)*. IEEE Computer Society, USA, 219–228. https://doi.org/10.1109/COMPSAC.2014.30

[12] Jim Chow, Dominic Lucchetti, Tal Garfinkel, Geoffrey Lefebvre, Ryan Gardner, Joshua Mason, Sam Small, and Peter M. Chen. 2010. Multi-Stage Replay with Crosscut. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Pittsburgh, Pennsylvania, USA) *(VEE ’10)*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/1735997.1736002

[13] Susan B Davidson and Juliana Freire. 2008. Provenance and scientific workflows: challenges and opportunities. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, 1345–1350.

[14] Robert DeLine and Danyel Fisher. 2015. Supporting exploratory data analysis with live programming. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 111–119. https://doi.org/10.1109/VLHCC.2015.7357205

[15] Robert DeLine, Danyel Fisher, Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, James F. Terwilliger, and John Robert Wernsing. 2015. Tempe: Live scripting for live data. *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (2015), 137–141.

[16] Joel Grus. 2018 (accessed June 26, 2020). *I Don’t Like Notebooks (JupyterCon 2018 Talk).* https://t.ly/Wt3S.

[17] Philip J Guo and Margo I Seltzer. 2012. Burrito: Wrapping your lab notebook in computational infrastructure. (2012).

[18] Alena Guzharina. 2020. We Downloaded 10,000,000 Jupyter Notebooks From Github – This Is What We Learned. https://blog.jetbrains.com/datalore/2020/12/17/we-downloaded-10-000-000-jupyter-notebooks-from-github-this-is-what-we-learned. Date accessed: 2022-02-28.

[19] Andrew Head, Fred Hohman, Titus Barik, Steven M Drucker, and Robert DeLine. 2019. Managing messes in computational notebooks. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. 1–12.

[20] Alex Holkner and James Harland. 2009. Evaluating the Dynamic Behaviour of Python Applications. In *Proceedings of the Thirty-Second Australasian Conference on Computer Science - Volume 91* (Wellington, New Zealand) *(ACSC ’09)*. Australian Computer Society, Inc., AUS, 19–28.

[21] Sean Kandel, Andreas Paepcke, Joseph M Hellerstein, and Jeffrey Heer. 2012. Enterprise data analysis and visualization: An interview study. *IEEE Transactions on Visualization and Computer Graphics* 18, 12 (2012), 2917–2926.

[22] Mary Beth Kery, Amber Horvath, and Brad A Myers. 2017. Variolite: Supporting Exploratory Programming by Data Scientists.. In *CHI*, Vol. 10. 3025453–3025626.

[23] Mary Beth Kery and Brad A Myers. 2017. Exploring exploratory programming. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 25–29.

[24] Mary Beth Kery and Brad A. Myers. 2018. Interactions for Untangling Messy History in a Computational Notebook. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 147–155. https://doi.org/10.1109/VLHCC.2018.8506576

[25] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E John, and Brad A Myers. 2018. The story in the notebook: Exploratory data science using a literate programming tool. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. 1–11.

[26] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E. John, and Brad A. Myers. 2018. *The Story in the Notebook: Exploratory Data Science Using a Literate Programming Tool.* Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/3173574.3173748

[27] Thomas Kluyver et al. 2016. Jupyter Notebooks-a publishing format for reproducible computational workflows.. In *ELPUB*. 87–90.

[28] David Koop and Jay Patel. 2017. Dataflow notebooks: encoding and tracking dependencies of cells. In *9th {USENIX} Workshop on the Theory and Practice of Provenance (TaPP 2017)*.

[29] Bogdan Korel and Janusz Laski. 1988. Dynamic program slicing. *Information processing letters* 29, 3 (1988), 155–163.

[30] Stephen Macke, Hongpu Gong, Doris Jung-Lin Lee, Andrew Head, Doris Xin, and Aditya Parameswaran. 2021. Fine-grained lineage for safer notebook interactions. *Proceedings of the VLDB Endowment* 14, 6 (2021), 1093–1101.

[31] Leonardo Murta, Vanessa Braganholo, Fernando Chirigati, David Koop, and Juliana Freire. 2014. noWorkflow: capturing and analyzing provenance of scripts. In *International Provenance and Annotation Workshop*. Springer, 71–83.

[32] Jakob Nielsen. 1994. *Usability Engineering*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[33] Akira Nishimatsu, Minoru Jihira, Shinji Kusumoto, and Katsuro Inoue. 1999. Call-mark slicing: an efficient and economical way of reducing slice. *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)* (1999), 422–431.

[34] Jim Ormond. 2018 (accessed June 26, 2020). *ACM Recognizes Innovators Who Have Shaped the Digital Revolution.* https://awards.acm.org/binaries/content/assets/press-releases/2018/may/technical-awards-2017.pdf.

[35] Jeffrey M Perkel. 2018. Why Jupyter is data scientists’ computational notebook of choice. *Nature* 563, 7732 (2018), 145–147.

[36] Joao Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. 2017. noWorkflow: a tool for collecting, analyzing, and managing provenance from python scripts. *Proceedings of the VLDB Endowment* 10, 12 (2017).

[37] Adam Rule, Ian Drosos, Aurélien Tabard, and James D. Hollan. 2018. Aiding Collaborative Reuse of Computational Notebooks with Annotated Cell Folding.

Proc. ACM Hum.-Comput. Interact. 2, CSCW, Article 150 (Nov. 2018), 12 pages. https://doi.org/10.1145/3274419

[38] Adam Rule, Aurélien Tabard, and James D Hollan. 2018. Exploration and explanation in computational notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. 1–12.

[39] Adam Rule, Aurélien Tabard, and James D. Hollan. 2018. *Exploration and Explanation in Computational Notebooks.* Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3173574.3173606

[40] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 488–498. https://doi.org/10.1145/2491411.2491447

[41] Ben Shneiderman. 1984. Response Time and Display Rate in Human Performance with Computers. *ACM Comput. Surv.* 16, 3 (sep 1984), 265–285. https://doi.org/10.1145/2514.2517

[42] Frank Tip. 1995. A survey of program slicing techniques. *J. Program. Lang.* 3 (1995).

[43] G. Venkatesh. 1995. Experimental results from dynamic slicing of C programs. *ACM Trans. Program. Lang. Syst.* 17 (1995), 197–216.

[44] Tao Wang and Abhik Roychoudhury. 2008. Dynamic slicing on Java bytecode traces. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 30, 2 (2008), 1–49.

[45] Zhaogui Xu, Ju Qian, Lin Chen, Zhifei Chen, and Baowen Xu. 2013. Static Slicing for Python First-Class Objects. *2013 13th International Conference on Quality Software* (2013), 117–124.

[46] YoungSeok Yoon and B. Myers. 2012. An exploratory study of backtracking strategies used by developers. *2012 5th International Workshop on Co-operative and Human Aspects of Software Engineering (CHASE)* (2012), 138–144.

[47] Xiangyu Zhang and Rajiv Gupta. 2004. Cost effective dynamic program slicing. *ACM SIGPLAN Notices* 39, 6 (2004), 94–106.

[48] Kevin Zielnicki. 2017 (accessed July 5, 2020). *Nodebook.* https://multithreaded.stitchfix.com/blog/2017/07/26/nodebook/.