Using Automatic Persistent Memoization to Facilitate Data Analysis Scripting

Philip J. Guo   Dawson Engler

Computer Systems Laboratory
Stanford University
Stanford, California, USA

ABSTRACT

Programmers across a wide range of disciplines (e.g., bioinformatics, neuroscience, econometrics, finance, data mining, information retrieval, machine learning) write scripts to parse, transform, process, and extract insights from data. To speed up iteration times, they split their analyses into stages and write extra code to save the intermediate results of each stage to files so that those results do not have to be re-computed in every subsequent run. As they explore and refine hypotheses, their scripts often create and process lots of intermediate data files. They need to properly manage the myriad of dependencies between their code and data files, or else their analyses will produce incorrect results.

To enable programmers to iterate quickly without needing to manage intermediate data files, we added a set of dynamic analyses to the programming language interpreter so that it automatically memoizes (caches) the results of long-running pure function calls to disk, manages dependencies between code and on-disk data, and later re-uses memoized results, rather than re-executing those functions, when guaranteed safe to do so. We created an implementation for Python and show how it enables programmers to iterate faster on their data analysis scripts while writing less code and not having to manage dependencies between their code and datasets.

Categories and Subject Descriptors:
D.3.4 [*Processors*]: Interpreters, Run-time environments
H.5.2 [*User Interfaces*]: Prototyping

General Terms: Languages, Human Factors

Keywords: Scientific workflows, dependency management

1. INTRODUCTION

Programmers across a wide range of disciplines (e.g., bioinformatics, neuroscience, econometrics, finance, data mining, information retrieval, machine learning) write scripts to parse, transform, process, and extract insights from data. By some estimates, the number of people who write these *data analysis scripts* is on par with the number of professional software engineers. A study of US labor statistics predicts that by 2012, 13 million American workers will do programming beyond creating spreadsheet macros and database queries (i.e, not simply end-user programming [18]), but amongst those, only 3 million will be professional software engineers [25]. It is reasonable to assume that a non-trivial percentage of these 10 million programmers who are not software engineers will write scripts to analyze data. Three out of the ten most popular languages today [3] — Python, Perl, and Ruby — are often used for data analysis.

Data analysis scripting differs from professional software engineering in fundamental ways: First, the end products of data analysis are insights about a topic [17], whereas the end products of software engineering are (hopefully) robust, well-tested, and maintainable pieces of software. Next, data analysis is often ad-hoc and exploratory in nature, where requirements are ill-defined and constantly changing [6]. Lastly, data analysis scripts are written by people of all levels of programming expertise, ranging from CS veterans to scientists who learn barely enough about programming to write basic scripts. Thus, the creators of data analysis scripts often program using high-level interpreted languages (e.g., Python, Perl, Ruby) because they care more about flexibility, iteration speed, and ease of development than code robustness, maintainability, and run-time performance [5].

To illustrate a common problem that arises during data analysis scripting, we describe the first author's experiences during a summer internship at Microsoft Research. His project was to analyze software bug databases and employee personnel datasets to quantify people-related factors that affect whether bug reports are fixed (published in ICSE 2010 [11]). He wrote all scripts in one language (Python), but his datasets were stored in diverse file formats (e.g., semi-structured plaintext, CSV, SQL database, serialized objects), a typical setup for data analysis workflows [6].

He first wrote a few scripts to process the primary datasets to create output charts and tables. However, since those scripts took a long time to run (tens of minutes to several hours), he split up his analysis into multiple stages and wrote serialization code to output the intermediate results of each stage to disk. Breaking up his scripts into stages, implemented as functions or separate scripts, improved performance and sped up iteration times: When he edited and re-executed later stages, those stages could re-use intermediate results from disk rather than waste time re-executing unchanged earlier stages. However, his hard disk was now filled with dozens of scripts and intermediate data files.

Upon inspecting his output charts and tables, his super-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISSTA '11, July 17-21, 2011, Toronto, ON, Canada
Copyright 2011 ACM 978-1-4503-0562-4/11/05 ...$10.00.

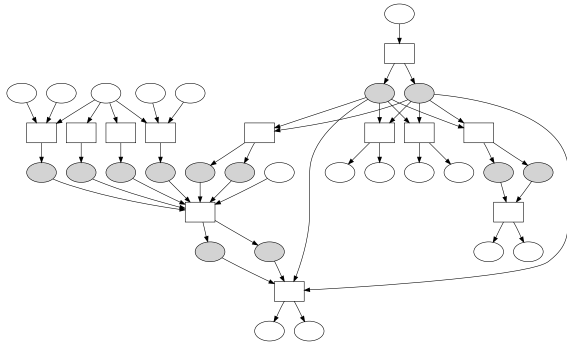

Figure 1: A data analysis workflow [11] comprised of Python scripts (boxes) that process and generate data files (circles). Gray circles are intermediate data files, which IncPy can eliminate.

visor often asked him to adjust his scripts or to fork his analyses to explore multiple alternative hypotheses (e.g., *"Please explore the effects of employee location on bug fix rates by re-running your analysis separately for each country."*). Thus, he parameterized his scripts to generate multiple output datasets based on command-line parameters, which created even more data files.

Having to manually keep track of dozens of scripts and data files led to frustrating bugs. For example, he would update certain datasets but forget to re-run the scripts that analyzed them, which meant that some other data files were now incorrect. Or he would delete scripts but forget to delete the data files they generated, leaving "orphan" data files that might erroneously be processed by subsequent scripts. Or he would forget which scripts and datasets generated which charts, and whether those charts were up-to-date. Since these bugs all manifested as incorrect outputs and *not as crashes*, it was difficult to actually determine when a bug occurred. To be safe, he would periodically re-run all of his scripts, which eliminated the performance benefits of splitting up his workflow into multiple stages in the first place.

One possible solution would be to write all of his code and dataset dependencies in a Makefile [7], so that invoking `make` would only re-run the scripts whose dependent datasets have changed. However, since he rapidly added, edited, and deleted dozens of scripts and datasets as he explored new hypotheses, it was too much of a hassle to also write and update the dependencies in parallel in a Makefile. At the end of his internship, he finally created a Makefile to document his workflow, but some dependencies were so convoluted that the file likely contains errors. Figure 1 shows the dependencies between his Python scripts (boxes) and data files (circles), extracted from his end-of-internship Makefile.

In a 2010 workshop paper [10], we presented anecdotes from colleagues who also experienced similar frustrations throughout the data analysis scripting process. A literature search revealed that even veteran computational scientists acknowledged that having to organize code, data, and their dependencies was an impediment to productivity [17, 21].

Problem: General-purpose programming languages provide no support for managing the myriad of dependencies between code and datasets that arises throughout the data analysis process. Veteran data analysts suggest using disciplined file naming conventions and Makefiles as the "best practices" for coping with these dependencies [17, 21].

Our solution: To enable programmers to iterate quickly without the burden of managing code and file dependencies, we added dynamic analyses to the programming language interpreter to perform automatic memoization and dependency management. Our technique works as follows:

1. The programmer's script runs in a custom interpreter.
2. The interpreter automatically memoizes [19] (caches) the inputs, outputs, and dependencies of certain function calls to disk, only doing so when it is safe (pure and deterministic call) and worthwhile (faster to re-use cached results than to re-run) to do the memoization.
3. During subsequent runs of the same script (possibly after the programmer edits it), the interpreter skips all memoized calls and re-uses cached results if the code and data that those results depend on are unchanged.
4. The interpreter automatically deletes on-disk cache entries when their code or data dependencies are altered.

We implemented our technique as a custom open-source Python interpreter called IncPy (**Inc**remental **Py**thon) [1]. However, our technique is not Python-specific; it can be implemented for similar languages like Perl, Ruby, or R.

Benefits: IncPy improves the experience of writing data analysis scripts in three main ways:

- **Less code**: Programmers can write data analysis stages as pure functions that return ordinary program values and connect stages together using value assignments within their scripts. IncPy automatically memoizes function inputs/outputs to a persistent on-disk cache, so programmers do not need to write serialization and deserialization code. Less code means fewer sites for bugs. Although programmers get the most memoization benefits when they write code in a modular and functional style, IncPy does not enforce any particular style. Programmers can use the full Python language and perform impure actions when convenient.
- **Automated data management**: IncPy manages the dependencies between code and datasets so that the proper data can be updated when the code they depend on changes, thus preventing stale data bugs. IncPy tracks datasets that already exist on disk (e.g., CSV files and SQL databases) as well as those that it creates by memoizing function calls. For example, the first author could have eliminated all of the gray circles (intermediate data files) in Figure 1 if he had used IncPy. Each analysis stage (box in Figure 1) could directly operate on the return values from upstream stages, and IncPy would automatically create and manage the intermediate datasets (cache entries).
- **Faster iteration times**: IncPy allows programmers to iterate and explore ideas faster because when they edit and re-run scripts, memoized results from unchanged stages can be loaded from the on-disk cache rather than re-computed. Programmers get these performance benefits without having to write any annotations, caching code, or manage intermediate datasets.

Because IncPy works with ordinary Python scripts, it is well-suited for programmers who want to focus on analyzing their data without needing to invest the effort to learn new language features, domain-specific languages, or other tools.

```
MULTIPLIER = 2.5 # global variable

# Input: name of file containing SQL queries
def stageA(filename):
  lst = [] # initialize empty list
  for line in open(filename, 'r'):
    lst.append(stageB(line))
  transformedLst = stageC(lst)
  return sum(transformedLst) # returns a number

# Input: an SQL query string
def stageB(queryStr):
  db = open_database('masterDatabase.db')
  q = db.query(queryStr)
  res = ... # run for 1 minute processing q
  return (res * MULTIPLIER) # returns a number

# Input: a list of numerical values
def stageC(lst):
  res = ...  # run for 1 minute processing lst
  return res # returns a list of numbers

print stageA("queries.txt") # top-level call
```

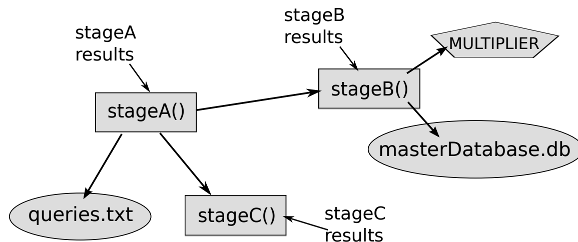

Figure 2: Example Python data analysis script (top) and dependencies generated during execution (bottom). Boxes are code, circles are file reads, and the pentagon is a global variable read.

2. EXAMPLE

We use an example Python script to illustrate the basic features of IncPy. Figure 2 shows a script that processes a `queries.txt` file using three functions. Assume that the script takes 1 hour to process a 59-line `queries.txt` file, where each line contains an SQL query. `stageA` makes 59 calls to `stageB` (one call for each line in `queries.txt`) followed by 1 call to `stageC`, where each of those calls lasts for 1 minute. The rest of `stageA` terminates instantaneously.

When we run this script for the first time, IncPy tracks the names and values of the global variables and files that each function reads and the code that each function calls. It dynamically generates the dependency graph shown on the bottom half of Figure 2, which contains three types of dependencies: For example, the function `stageA` has a *code dependency* on `stageB`; `stageB` has a *global variable dependency* on `MULTIPLIER`; `stageB` also has a *file read dependency* on the `masterDatabase.db` file. As each function call finishes, IncPy memoizes the arguments, return values, and dependencies of each call to a persistent on-disk cache.

Now when we edit some code or data and run the same script again, IncPy can consult the memoized dependencies to determine the minimum number of functions that need to be re-executed. When IncPy is about to execute a function whose dependencies have not changed since the previous execution, it will skip the call and directly return the memoized return value to its caller. Here are some ways in which a subsequent run can be faster than the initial 1-hour run:

- If we edit the end of `stageA` to return, say, the product of `transformedLst` elements rather than the sum, then re-executing is nearly instantaneous since IncPy can re-use all the memoized results for `stageB` and `stageC`.
- If we fix a bug in `stageC`, then re-executing only takes 1 minute: We must re-run `stageA` and `stageC` but can re-use memoized results from all 59 calls to `stageB`.
- If we modify a line in `queries.txt`, then re-executing takes 2 minutes since `stageB` only needs to re-run on the SQL query string specified by the modified line, and `stageC` must also re-run since its input is different.
- If we append a new line to `queries.txt`, then re-executing takes 2 minutes since `stageB` only needs to run on the new line, and `stageC` must also re-run.

For comparison, if we ran this script in the regular Python interpreter, then *every subsequent run would take 1 hour*, regardless of how little we edited the code or the `queries.txt` file. If we wanted to have our script run faster after minor edits, we would need to write our own serialization and deserialization code to save and load intermediate data files, respectively. Then we would need to remember to regenerate particular data files after their dependent functions are edited, or else risk getting incorrect results. IncPy automatically performs all of this caching and dependency management so that we can iterate quickly without needing to write extra code or to manually manage dependencies.

3. DESIGN AND IMPLEMENTATION

IncPy consists of dynamic analyses that perform dependency tracking and function memoization (§3.1), persistent cache management (§3.2), function profiling and impurity detection (§3.3), and object reachability detection (§3.4).

We created IncPy by adding ∼4000 lines of C code to the official Python 2.6.3 interpreter. IncPy is fully compatible with existing Python scripts and 3rd-party extension modules already installed on the user's machine.

3.1 Memoizing Function Calls

IncPy's main job is to automatically memoize certain function calls to a persistent on-disk cache when the target program is about to exit the call. Each cache entry represents one memoized call and contains the fields in Table 1.

IncPy updates the fields of Table 1 in each function's stack frame as it is executing. It does so by interposing on the interpreter's handlers for the corresponding program events. For example, we inserted code in the interpreter's handler for file I/O to add a file read/write dependency to all functions on the stack whenever the target program performs a file read/write. Thus, if a function `foo` calls `bar`, and some code in `bar` reads from a file `data.txt`, then *both* `foo` and `bar` now have a *file read dependency* on `data.txt`. Similarly, IncPy adds a global variable dependency to all functions on the stack whenever the program reads a value that is reachable from a global variable (see §3.4 for details).

Although interposing on every file access, value access, and function call might seem slow, performance is reasonable since the bookkeeping code we inserted is small compared to what the Python interpreter already executes for these program events. For example, the interpreter executes a few hundred lines of C code to initiate a Python function call, so the few extra lines we inserted to fetch its argument values

Full name	Function's name and enclosing filename (for methods, also add the enclosing class's full name)
Arguments	Argument values for this call
Return value	Return value for this call
Terminal output	Contents of text printed to `stdout` and `stderr` buffers during this call
Global var. dependencies	Names and values of all global variables, variables in enclosing lexical scopes, and static class fields read during this call
File read dependencies	Names and last modified times of files read during this call
File write dependencies	Names and last modified times of files written during this call
Code dependencies	Full names and bytecodes of this function and of all functions that it transitively called

Table 1: Contents of a persistent on-disk cache entry, which represents one memoized function call.

and update code dependencies has a minimal performance impact. We evaluate performance in Section 4.1.

When the target program finishes executing a function call, if IncPy determines that the call should be memoized (see §3.3 for criteria), it uses the Python standard library `cPickle` module to serialize the fields in Table 1 to a binary file. For arguments, global variables, and function return values, IncPy serializes the entire object that each value refers to, which includes all objects transitively reachable from it. Since IncPy saves these objects to disk at this time, it does not matter if they are mutated later during execution. All cached arguments and global variables have the same values as they did when the function call began; otherwise, the call would be impure and not memoized (§3.3.1).

IncPy stores each serialized cache entry on disk as a separate file, named by an MD5 hash of the serialized argument values (collisions handled by chaining like in a hash table). Each file is written atomically by first writing to a temp. file and then doing an atomic rename, which allows multiple processes to share a common cache. All cache entries for a function are grouped together into a sub-directory, named by an MD5 hash of the function's *full name* (see Table 1).

IncPy does not limit the size of the on-disk cache; it will keep writing new entries to the filesystem as long as there is sufficient space. IncPy automatically deletes cache entries when their dependencies are altered (§3.2). Also, since each cache entry is a separate file, a user (or script) can manually delete individual entries by simply deleting their files.

3.2 Skipping Function Calls

When the target program calls a function:

1. **Cache look-up**: IncPy first looks for an on-disk cache entry that matches its full name and the values of its current arguments and global variable dependencies, checking for equality using Python's built-in object equality mechanism. If there is no matching cache entry, then IncPy simply executes the function.
2. **Checking dependencies and invalidating cache**: If there is a match, then IncPy checks the file and code dependencies in the matching cache entry. If a dependent file has been updated or deleted, then the cache entry is deleted. If the bytecode for that function or any function that it called has been changed or deleted, then all cache entries for that function are deleted. When these dependencies are altered, we cannot safely re-use cached results, since they might be incorrect. For example, if a function `foo` calls `bar` and returns `42`, then if someone modifies the code of `bar` and re-executes `foo`, it might no longer return `42`.
3. **Skipping the call**: If there is a matching cache entry and all dependencies are unchanged, then IncPy will skip the function call, print out the cached `stdout` and `stderr` contents, and return the cached return value to the function's caller. This precisely emulates the original call, except that it can be much faster.

One practical benefit of IncPy atomically saving each cache entry to a file as soon as the function exits, rather than doing so at the end of execution, is that if the interpreter crashes in the middle of a long-running script, those cache entries are already on disk. The programmer can fix the bug and re-execute, and IncPy will skip all memoized calls up to the site of the bug fix. For example, some of our users have encountered annoying bugs where their scripts successfully processed data for several hours but then crashed at the very end on a trivial bug in the output printing code. IncPy was able to memoize intermediate results throughout execution, so when they fixed those printing bugs and re-executed, their scripts ran much faster, since IncPy could skip unchanged function calls and load their results from the cache.

3.3 Which Calls Should Be Memoized?

IncPy automatically determines which function calls to memoize without requiring any programmer annotations. However, a programmer can force IncPy to always or never memoize particular functions by inserting annotations.

3.3.1 Which Calls Are Safe To Memoize?

It is only safe to memoize function calls that are pure and deterministic, since only for those calls will the program execute identically if they are later skipped and replaced with their cached return values.

Pure calls: Following the definition of Salcianu and Rinard, we consider a function call *pure* if it never mutates a value that existed prior to its invocation [24]. In Python (and similar languages), all objects reachable from global variables[1] and a function's arguments might exist prior to its invocation. Thus, when a program mutates a globally-reachable object, IncPy marks *all functions* on the stack as impure. For example, if a function `foo` calls `bar`, and some code in `bar` mutates a global variable, then IncPy will mark *both* `foo` and `bar` as impure, since both functions were on the stack when an impure action occurred. When a program mutates an object reachable from a function's arguments, IncPy marks that function as impure (see §3.4 for details).

IncPy does *not* mark a function as impure if it writes text to the terminal; instead, it separately captures `stdout` and `stderr` outputs in the cache (see Table 1) and prints those cached strings to the terminal when the function is skipped.

[1] includes variables in enclosing scopes and static class fields

Unlike static analysis [24], IncPy dynamically detects impurity of individual execution paths. This is sufficient for determining whether a particular call is safe to memoize; a subsequent call of a pure deterministic function with the same inputs will execute down the same path, so it is safe to skip the call and re-use memoized results. If a function is pure on some paths but impure on others, then calls that execute the pure paths can still be memoized.

Deterministic calls: We consider a function call *deterministic* if it does not access resources like a random number generator or the system clock. It is difficult to automatically detect all sources of non-determinism, so we have annotated a small number of standard library functions as non-deterministic (e.g., those related to randomness, time, or `stdin`). IncPy marks all functions on the stack as impure when the target program calls one of these functions.

In theory, memory allocation is a source of non-determinism if a program makes control flow decisions based on the addresses of dynamically-allocated objects. For the sake of practicality, IncPy does not treat memory allocation as non-deterministic, since if it did, then almost all functions would be impure. In our experience, it is rare for programs written in memory-safe languages like Python to branch based on memory addresses, since pointers are not directly exposed. For example, none of the scripts in our benchmark suite (see §4.1.1) branch execution based on memory addresses.

Self-contained file writes: Writing to a file might seem like an impure action, since it mutates the filesystem. While that is technically true, we make an exception for a kind of idempotent file write that we call a *self-contained write*. A function performs a self-contained write if it was on the stack when the file was opened in pure-write (not append) mode, written to, and then closed. We observed that data analysis scripts often perform self-contained writes: An analysis function usually processes input data, opens an output file, writes data to it, and then closes it. For example, although only one set of scripts in our benchmark suite performed file writes, all of its 17 writes were self-contained (see Section 4.2.3 for a case study of that benchmark).

For example, if a function `foo` does a self-contained write to `data.txt` (open → write → close), then each call to `foo` creates a new and complete copy of `data.txt`. Thus, IncPy still considers `foo` to be pure and records `data.txt` as a *file write dependency*. As long as `foo` and all its dependencies remain unchanged, then there is no need to re-run `foo` since it will always re-generate the same contents for `data.txt`.

If a function writes to a file in a non-self-contained way (e.g., by opening in append mode or not closing it), then IncPy marks the call as impure and does not memoize it.

OOP support: For object-oriented programs, IncPy memoizes pure deterministic method calls just like ordinary function calls: The receiver (`this` object) is memoized as the first argument, and static class fields are memoized as global variables. Since IncPy tracks methods and objects at run time, it correctly handles inheritance and polymorphism.

3.3.2 Which Calls Are Worthwhile To Memoize?

It is only worthwhile to memoize a function call if loading and deserializing the cached results from disk is faster than re-executing the function. A simple heuristic that works well in practice is to have IncPy track how long each function call takes and only memoize calls that run for more than 1 second. The vast majority of calls (especially in library code) run for far less than 1 second, so it is faster to re-execute them rather than to save and load their results from disk.

There are pathological cases where this heuristic fails. For example, if a function runs for 10 seconds but returns a 1 GB data structure, then it might take more than 10 seconds to load and deserialize the 1 GB data structure from the on-disk cache. Our experiments show that it takes 20 seconds to load and deserialize a 1 GB Python list from disk (§4.1.2). If IncPy memoized the results of that function, then skipping future calls would actually be slower than re-executing (20 seconds vs. 10 seconds). We use a second heuristic to handle these pathological cases: IncPy tracks the time it takes to serialize and save a cache entry to disk, and if that is longer than the function's original running time, then it issues a warning to the programmer and does not memoize future calls to that function (unless its code changes). Our experiments in Section 4.1.2 indicate that it always takes more time to save a cache entry than to load it, so the cache save time is a conservative approximation for cache load time.

3.4 Dynamic Reachability Detection

When a program is about to read from or write to an arbitrary Python object in memory, how does IncPy determine whether that object is reachable from a global variable or a function's argument? IncPy needs this information to determine when to add a global variable dependency (Table 1) and when to mark function calls as impure (§3.3.1).

The Python interpreter represents every run-time value in memory as an object, so IncPy augments every object with two fields: the name of a global variable that reaches this object (`globalName`), and the starting "time" of the outermost function call on the stack whose arguments reach this object (`funcStart`). IncPy measures "time" by the number of function calls that the interpreter has executed thus far. These fields are null for objects that are not reachable from a global or a function argument.

When the program loads a global variable, IncPy sets the `globalName` field of its value to the variable's name. When the program calls a function, IncPy sets the `funcStart` field of all its argument values to the current "time" (number of executed function calls), only for values whose `funcStart` has not already been set by another function currently on the stack. When the program executes an object field access (e.g., `my_obj.field`) or element access (e.g., `my_list[5]`), IncPy copies the `globalName` and `funcStart` fields from the parent to the child object. For example, if a program executes a read of `foo.bar[5]` where `foo` is a global variable, then the objects referred to by `foo`, `foo.bar`, and `foo.bar[5]` would all have the name "`foo`" in their `globalName` fields. When the program is about to read from or write to an object, IncPy can do a fast lookup of its `globalName` and `funcStart` fields to determine whether it is reachable from a global variable or a function argument, respectively.

The `funcStart` field enables IncPy to efficiently determine which functions to mark as impure when an argument is mutated (see §3.3.1). For example, if a function `foo` accepts an argument `x` and passes it into `bar`, then `x` is an argument of both `foo` and `bar`. Assume that the call to `foo` started at time 5 and `bar` started at time 6. The `funcStart` field of `x` is 5, since that is the start time of `foo`, the outermost function call where `x` is an argument. If code within `bar` mutates any component of `x`, then IncPy sees that its `funcStart` field, 5, is less than or equal to the start time of both `foo` and `bar`, so it marks both functions as impure.

```
def stage1():
  infile = open('input.dat', 'r')
  ... # parse and process infile
  outfile = open('stage1.out', 'w')
  outfile.write( ... ) # write output to file
  outfile.close()

def stage2():
  infile = open('stage1.out', 'r')
  ... # parse and process infile
  outfile = open('stage2.out', 'w')
  outfile.write( ... ) # write output to file
  outfile.close()

# top-level script code:
stage1()
stage2()
```

Figure 3: Example Python script that implements a file-based workflow, and accompanying dataflow graph where boxes are code and circles are data files.

Implementation: We initially implemented reachability detection by directly adding two extra fields to the Python object datatype: `globalName` and `funcStart`. This worked fine in development, but when we started getting users, they complained that IncPy did not work with 3rd-party Python extension modules already installed on their machines. Extension modules consist of compiled C/C++ code that rely on the Python object datatype to be of a certain size. Since IncPy augmented that datatype with two extra fields, extension module code no longer worked. To use IncPy with their extensions, users would need to re-compile extension code with the IncPy headers, which can be difficult due to compile-time dependencies.

Thus, to make IncPy work with users' already-installed extensions, we re-implemented using a shadow memory approach [20]. We left the Python object datatype unchanged and instead maintain `globalName` and `funcStart` fields for each object in a sparse table. To do a table look-up, IncPy breaks up an object's memory address into 16-bit chunks and uses them to index into a multi-level table similar to an OS page table. We use a two-level table for 32-bit architectures and a four-level table for 64-bit. For example, to look up an object at address `0xdeadbeef`, IncPy first looks up index `0xdead` in the first-level table. If that entry exists, it is a pointer to a second-level table, so IncPy looks up index `0xbeef` in that second-level table. If that entry exists, then it holds the `globalName` and `funcStart` fields for our target object. This mapping works because Python objects never change memory locations. When an object is deallocated, IncPy clears its corresponding table entry. IncPy conserves memory by lazily allocating tables. However, memory usage is still greater than if we had inlined the fields within Python objects, but that is the trade-off we made to achieve binary compatibility with already-installed extension modules.

3.5 Supporting File-Based Workflows

Figure 3 shows a script that implements a two-stage data analysis workflow. The programmer wrote extra code to save the results of `stage1` to an intermediate data file `stage1.out`, so that when `stage2` is edited, the code for `stage1` does not have to re-run. However, if the programmer changes the code for `stage1`, then it must be re-run to generate a new `stage1.out`, or else the input to `stage2` will be incorrect.

```
def stage1():
  infile = open('input.dat', 'r')
  out = ... # parse and process infile
  return out

def stage2(dat):
  out = ... # process dat argument
  return out

# top-level script code:
stage1_out = stage1()
stage2_out = stage2(stage1_out)
```

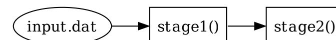

Figure 4: The Python script of Figure 3 refactored to take advantage of IncPy's automatic memoization. Data flows directly between the two stages without an intermediate data file.

By using IncPy, the programmer can simplify that script into the one shown in Figure 4. There is no more need to write code to save and load data files, or to manually manage their dependencies. The on-disk cache entries that IncPy creates after it memoizes `stage1` and `stage2` are the substitutes for the `stage1.out` and `stage2.out` files, respectively.

Despite the benefits of a pure-Python workflow (Figure 4), some of our users still choose to create intermediate data files (Figure 3) for performance reasons. IncPy serializes entire Python data structures, which is convenient but can be slow, especially for data larger than a few gigabytes. Scripts often run faster when accessing data files in a format ranging from a specialized binary format to a database.

IncPy supports these file-based workflows by recording file read and write dependencies. After it executes the script in Figure 3, `stage1` will have a read dependency on `input.dat` and a write dependency on `stage1.out`; `stage2` will have a read dependency on `stage1.out` and a write dependency on `stage2.out`. IncPy can use this dependency graph to skip calls and issue warnings. For example, if only `stage2` is edited, IncPy can skip the call to `stage1`. If `stage1.out` is updated but `stage1` did not change, then IncPy will issue a warning that some external entity modified `stage1.out`, which could indicate a mistake. In contrast, the regular Python interpreter has none of these features.

3.6 Limitations

IncPy's main limitation is that it cannot track impurity, dependencies, or non-determinism in non-Python code such as C/C++ extension modules or external executables. Similarly, IncPy does not handle non-determinism or remote dependencies arising from network accesses. Users can make annotations to manually specify impurity and dependencies in external code. Fortunately, most C functions in the Python standard library are pure and self-contained, but we have annotated a few dozen as impure (e.g., list `append` method) and non-deterministic (e.g., time, randomness).

Another limitation is that IncPy is not designed to track fine-grained changes in code or data. If even one line in a function or dataset changes, then IncPy deletes cache entries that depend on that code or data, respectively. However, implementing finer-grained dependency tracking would increase IncPy's baseline run-time slowdown.

3.7 Discussion

The technique underlying IncPy can be implemented in a straightforward manner for other interpreted languages commonly used for data analysis scripting, such as Perl, Ruby, R, or MATLAB. An implementation for these languages could interpose on the interpreter's handlers for function calls, run-time value accesses, and file I/O in the exact same way as IncPy does. Implementing for a compiled language (e.g., Java, C++) is less straightforward but still feasible: One could create a source-to-source translator or static bytecode rewriter that augments the target program with the appropriate interposition callbacks prior to execution. However, performance now becomes a concern. IncPy's run-time overhead is reasonable (mean of 16% on our benchmarks) because interpreters are already slow compared to executing compiled code. Static purity and escape analyses [24], and other compile-time optimizations, might be needed to achieve reasonable overheads on compiled code.

One could further generalize the ideas in this paper by implementing an IncPy-like tool that works across programs written in multiple languages. Some data analysis workflows consist of separate programs that coordinate with one another using intermediate data files [17, 21]. A tool could use system call tracing (e.g., `ptrace` on Linux) to dynamically discover dependencies between executed programs and the data files that they read/write in order to eliminate unnecessary program executions.

Our technique can also be useful in domains beyond data analysis. For instance, it could be used to speed up regression testing by ensuring that after the programmer makes a code edit, only the regression tests affected by that edit are re-run. This might make it feasible for tests to run continuously during development, which has been shown to reduce wasted time and allow bugs to be caught faster [23]. Continuous testing is most effective for short-running unit tests tightly coupled with individual modules (e.g., one test per class). However, many projects only contain system-wide regression tests, each covering code in multiple modules. Manually determining which tests need to re-run after a code edit is difficult, and waiting for several minutes for *all* tests to re-run precludes interactive feedback. Instead, an IncPy-like tool could use function-level dependency tracking and memoization to automatically incrementalize regression testing.

4. EVALUATION

Our evaluation addresses two main questions: What are the performance impacts of dynamic dependency tracking and automatic memoization (Section 4.1)? How can IncPy speed up iteration times on real data analysis tasks without requiring programmers to write caching code (Section 4.2)?

We ran all experiments on a Mac Pro with four 3GHz CPUs and 4 GB of RAM, with Python 2.6.3 and IncPy both compiled as 32-bit binaries for Mac OS X 10.6 using default optimization flags. For faster performance, IncPy did not track dependencies within Python standard library code, because we assume users do not ever change that code.

4.1 Performance Evaluation

Running a script with IncPy when the cache is empty will be slower than running with the regular Python interpreter (by ~16% on our benchmarks) for two reasons: First, IncPy needs to dynamically track dependencies, global reachability, and function impurity to determine when it is safe and worthwhile to memoize. Second, IncPy must save and later load memoized data from the on-disk cache.

Script name	# lines of code	Running time		Peak RAM (MB)	
		Python	IncPy	Python	IncPy
mmouse	230	0:51	1:00	50	101
tags-3	1200	2:29	3:09	371	374
tags-1	1200	3:00	3:17	440	444
linux	200	4:58	5:10	8.6	15
tags-2	1200	6:07	7:57	486	488
vr-log	700	24:42	28:30	323	694
sys-log	200	33:13	37:56	67	73
biology	250	8:11:27	8:54:46	1884	1966

Table 2: Running times and peak memory usage of data analysis scripts executed with Python and IncPy, averaged over 5 runs (variances negligible). Figure 5 shows run-time slowdown percentages.

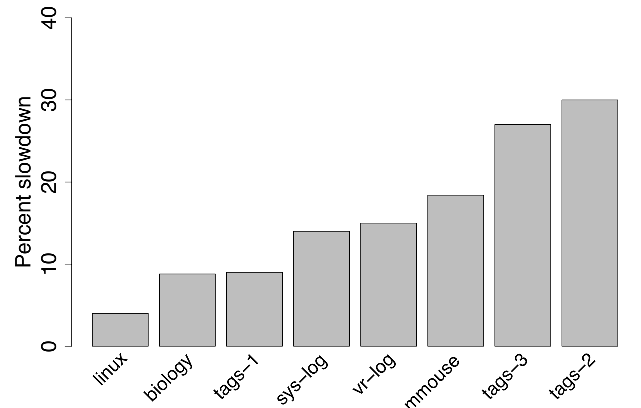

Figure 5: Percent slowdowns relative to regular Python when running data analysis scripts with IncPy. The mean slowdown is 16%.

4.1.1 Overall Slowdown

To quantify IncPy's overall slowdown on typical data analysis scripts, we compared running times and memory usage when executing six scripts with regular Python and IncPy. We obtained the following scripts from researchers who had written them to analyze data for research published in peer-reviewed papers (or currently under submission):

- **linux** — A script we wrote in 2007–2008 to mine data about the Linux kernel project's revision control history for an empirical software engineering paper [9]. We present a case study of this script in Section 4.2.1.
- **tags** — A set of information retrieval scripts for a paper contrasting crowdsourced tags with expert annotations for keywords describing books [15]. It consists of three stages, named *tags-1*, *tags-2*, and *tags-3*, respectively. We present a case study in Section 4.2.2.
- **vr-log** — A set of scripts that process event logs from a virtual world for a distributed systems paper [16]. We present a case study in Section 4.2.3.
- **sys-log** — A script that processes a 2.5 GB supercomputer error log for an anomaly detection paper [22].
- **mmouse** — A script that post-processes and graphs synchronized mouse input events for an HCI paper [13].

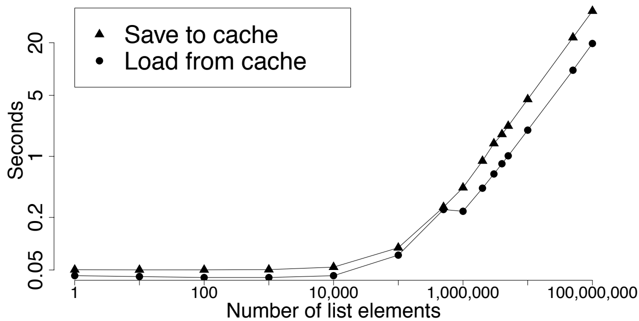

Figure 6: Number of seconds it takes for IncPy to save/load a Python list of N integers to/from cache.

- **biology** — A bioinformatics script that uses a hidden Markov model to analyze human genome data.

Table 2 and Figure 5 show IncPy's run-time and memory overheads. Figure 5 shows a mean slowdown of 16% relative to regular Python, most of which is due to IncPy dynamically tracking dependencies, global reachability, and impurity. IncPy only memoizes the few long-running function calls corresponding to the data analysis stages in each script (§3.3.2), and memoization only takes a few seconds (§4.1.2). Total script running times range from 1 minute to 8 hours (Table 2). Memory overheads range from negligible to 2X, mainly due to maintaining object metadata (§3.4).

We could not find data analysis scripts larger than ∼1000 lines of code, so to demonstrate IncPy's scalability, we used it to run the test suite of the Django project. Django is a popular framework for building web applications and is one of the largest Python projects, with 59,111 lines of Python code [2]. To estimate worst-case behavior, we measured IncPy's run-time slowdown on the Django test suite. All 151 tests passed when running with IncPy. The mean running time for an individual test case was 1.1 seconds (median was 0.57 sec). The mean slowdown relative to regular Python was 88% (maximum slowdown was 118%). These short runs elicit IncPy's worst-case behavior because IncPy has a longer start-up time than the regular Python interpreter (it must set up data structures for dependency tracking). In reality, it would be impractical to use IncPy on short-running applications like Django, since its start-up time nullifies any potential speed-ups. However, for its intended use on long-running data analysis scripts, IncPy has a reasonable slowdown of ∼16%.

4.1.2 Automatic Memoization Running Times

To measure cache save and load times in isolation, we created a microbenchmark consisting of one Python function that allocates a list of N random integers and returns that list to its caller. We annotated that function to force IncPy to memoize it; otherwise it would not be memoized since it is both non-deterministic and short-running. We ran the function once and measured the time it takes for IncPy to save its memoized results to disk. Then we ran it again and measured the time it takes for IncPy to load the matching cache entry from disk and skip the original function call.

Figure 6 shows cache save and load times for lists ranging from 1 to 100 million elements. Running times are instantaneous for lists of less than 100,000 elements and then scale up linearly. At the upper extreme, a list of 100 million integers is 1 gigabyte in size and takes 20 seconds to load from the cache. Cache load times are end-to-end, taking into account the time to start up IncPy, find a matching on-disk cache entry, load it from disk, deserialize it into a Python object, and finally skip the original function call.

In sum, memoization is worthwhile as long as cache load time is faster than re-execution time (see §3.3.2), so Figure 6 shows that IncPy can almost always speed up functions that originally take longer than ∼20 seconds to run.

4.2 Case Studies on Data Analysis Scripts

To demonstrate how IncPy can provide the benefits of less code, automated data management, and faster iteration times when writing data analysis scripts, we present brief case studies of three scripts from Table 2: **linux** (§4.2.1), **tags** (§4.2.2), and **vr-log** (§4.2.3). IncPy provides similar benefits for the other three scripts used in our experiments.

4.2.1 Speeding Up Exploration of Code Variants

To show how IncPy enables faster iteration when exploring alternative hypotheses in a data analysis task, we studied a Python script we wrote in 2007–2008 to mine Linux data for a paper [9]. Our script computes the chances that an arbitrary file in the Linux kernel gets modified in a given week, by querying an SQLite database containing data from the Linux project's revision control history. It retrieves the sets of files present in the Linux code base ("alive") and modified during a given week, using the `get_all_alive_files(week)` and `get_all_modified_files(week)` functions, respectively, and computes probabilities based on those sets. Both functions perform an SQL query followed by post-processing in Python (this mix of declarative SQL and imperative code is common in data analysis scripts). Each call takes 1 to 4 seconds, depending on the queried week.

Our script computes probabilities for 100 weeks and aggregates the results. It runs for 4 minutes, 58 seconds (4:58) with regular Python and 5:10 with IncPy (4% slower). IncPy memoizes all 100 calls to those two functions and records a file dependency on the SQLite database file.

To see how we edited this script throughout its development process, we checked out and inspected all historical versions from its revision control repository. The first version only did the original computation, but as we delved deeper into our research questions, we augmented subsequent versions to do related computations. The final version of our script (dated Feb. 2008) contains code for all 6 variants; a command-line flag controls which gets run. Here are the variants and their running times with regular Python:

1. Original computation: Chances that a Linux source code file gets modified in a given week (4:58)
2. Chances that a Linux file gets modified, conditioned on the modification type (e.g., bug fix, refactoring) (5:25)
3. Chances that a Linux file gets modified, conditioned on the time of day (e.g., morning vs. evening) (5:25)
4. Chances that a Linux file gets modified by the person who has modified it the most in the past (5:15)
5. Chances that a Linux file gets modified by someone above 90th percentile for num. edits to any file (5:25)
6. Chances that a Linux file gets modified by a person with a `.com` email address (5:25)

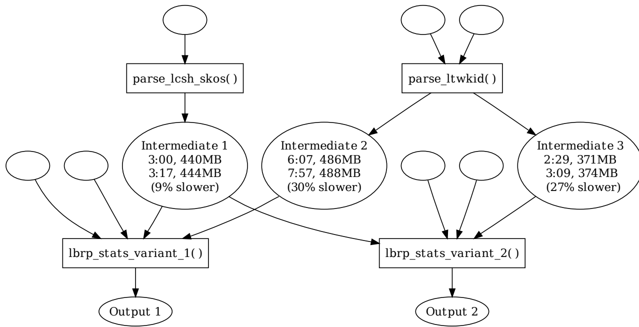

Figure 7: Datasets (circles) and Python functions (boxes) from Paul's information retrieval scripts [15]. In each "Intermediate" circle, the 1st row is run time and memory usage for generating that data with Python, and the 2nd row for IncPy.

After taking 5:10 to make an initial run of the original computation with IncPy, running *any variant* only takes 30 seconds, a **10X speed-up** over regular Python. This speed-up occurs because each variant calls the same 2 functions:

```
alive_set = get_all_alive_files(week)
mod_set   = get_all_modified_files(week)
# <do fast-running computations on these sets>
```

The computations that differ between variants take much less time to run than the 2 functions that they share in common; IncPy memoizes those calls, so it can provide a 10X speed-up regardless of which variant runs. This idiom of slow-running (but rarely-changing) code followed by fast-running (but frequently-changing) code exemplifies how data analysts explore variations when prototyping scripts. IncPy automatically caches the results of the slow-running functions, so that subsequent runs of any variants of the fast-running parts can be much faster.

4.2.2 Removing Existing Ad-Hoc Caching Code

To show how we can remove manually-written caching code and maintain comparable performance, we studied information retrieval scripts written by our colleague Paul [15]. Figure 7 shows his workflow, consisting of 4 Python functions (boxes) that process data from 7 input datasets (empty circles). His functions are arranged in 2 script variants:

```
# Script variant 1:
x = parse_lcsh_skos()
y1 = parse_ltwkid('dataset1')
print lbrp_stats_variant_1(x, y1)

# Script variant 2:
x = parse_lcsh_skos()
y2 = parse_ltwkid('dataset2')
print lbrp_stats_variant_2(x, y2)
```

Paul frequently edited the final function in each script: `lbrp_stats_variant_1` and `lbrp_stats_variant_2`. Since he rarely edited `parse_lcsh_skos` and `parse_ltwkid`, he grew tired of waiting for them to re-run on each execution and simply produce the same results, so he wrote extra code to save their intermediate results to disk and skip subsequent calls. Doing so cluttered up his script with code to serialize and deserialize results but sped up his iteration times.

We refactored Paul's script to remove his caching code and ran it with IncPy, which performed the same caching

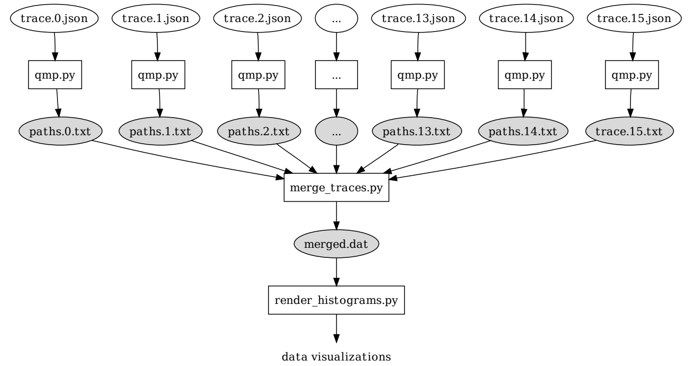

Figure 8: Python scripts (boxes) and data files (circles) from Ewen's event log analysis workflow [16]. Gray circles are intermediate data files, which are eliminated by the refactoring shown in Figure 9.

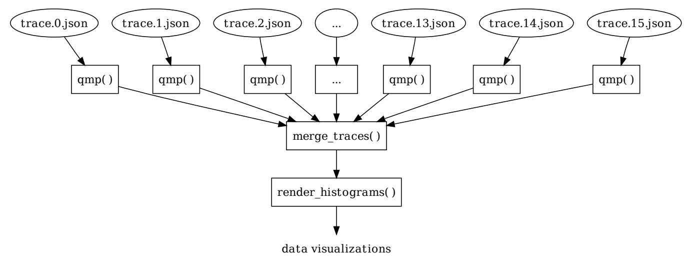

Figure 9: Refactored version of Ewen's event log analysis workflow [16], containing only functions (boxes) and input data files (circles)

automatically. The numbers in the "Intermediate" circles in Figure 7 compare performance of running the original functions to generate that data with Python versus the refactored versions with IncPy (they correspond to the tags-1, tags-2, and tags-3 rows in Table 2). Both versions took less than a second to load cached results on a subsequent run.

Besides eliminating the need to write boilerplate caching code, IncPy also automatically tracks dependencies between Paul's code and his 7 input datasets (empty circles in Figure 7). This can prevent errors like manually updating a dataset and forgetting to re-run the dependent functions.

4.2.3 From File-Based to Pure-Python Workflows

To show how IncPy can benefit existing file-based workflows (§3.5) and provide an easy transition to a pure-Python workflow, we studied Python code written by our colleague Ewen to process event logs from a virtual world system [16]. Figure 8 shows his 3 Python scripts as boxes: The `qmp.py` script takes two filenames as command-line arguments, reads data from the first, processes it, and writes output to the second. The `merge_traces.py` script first invokes `qmp.py` 16 times (once for each input `trace.X.json` file), then aggregates all resulting `paths.X.txt` files into a `merged.dat` file. Finally, `render_histograms.py` creates histograms and other data visualizations from the contents of `merged.dat`.

IncPy can benefit Ewen's current scripts by dynamically recording all code and file read/write dependencies to create the dependency graph of Figure 8. IncPy is 15% slower than regular Python on the initial empty-cache run (28:30 vs. 24:42). But IncPy's caching allows some subsequent runs to

be faster, because only script invocations whose dependent code or data have changed need to be re-run:

- If no input `trace.X.json` files change (or some are deleted), then none need to be re-processed by `qmp.py`
- If new `trace.X.json` files are added, then only those files need to be processed by `qmp.py`
- If an individual `trace.X.json` file changes, then only that file needs to be re-processed by `qmp.py`

Even though IncPy provides these speed-ups, Ewen's code is still cluttered with boilerplate to save and load intermediate files. For example, each line in `paths.X.txt` is a record with 6 fields (4 floats, 1 integer, 1 string) separated by a mix of colons and spaces, for human-readability. `qmp.py` contains code to serialize Python data structures into this ad-hoc textual format, and `merge_traces.py` contains code to deserialize each line back into Python data.

It took us less than an hour to refactor Ewen's code into the workflow of Figure 9, where each script is now a function that passes Python data into the next function via its return value *without explicitly creating any intermediate data files*. IncPy still provides the same speed-up benefits as it did for Ewen's original scripts, but now the code is much simpler, only expressing the intended computation without boilerplate serialization/deserialization code. For an initial empty-cache run, IncPy is 23% slower than regular Python (29:36 vs. 24:02). A subsequent run takes 0.6 seconds if no dependencies change.

This is a smaller version of the workflow described in the introduction (Figure 1). We could have refactored that workflow in the same way, but we no longer have access to its code or datasets since that work was done within Microsoft.

5. RELATED WORK

To the best of our knowledge, IncPy is the first attempt to integrate automatic memoization and persistent dependency management into a general-purpose programming language. The design of IncPy was inspired by and extends two classic ideas in software development: memoization and `make`.

Memoization is an optimization first introduced in a 1968 *Nature* paper [19]: It involves manually rewriting a function to save its inputs and outputs to a cache, so that subsequent calls with previously-seen inputs can be skipped. IncPy extends memoization by making it fully automatic and persistent, which involves detecting when it is safe and worthwhile to memoize, and invalidating on-disk cache entries when their dependencies are altered.

`make` is a ubiquitous UNIX tool that allows users to declaratively specify dependencies between commands and files, so that the minimum set of commands need to be re-run when dependent files are altered [7]. `make` has spawned dozens of descendent tools that all operate on the same basic premise. In particular, SCons (`www.scons.org`) and Ruffus (`www.ruffus.org.uk`) are modern variants of `make` implemented in Python. IncPy extends the ideas embodied by `make` by automatically extracting these dependencies using dynamic program analysis.

The Vesta software configuration management system [14] provides a pure functional domain-specific language for writing software build scripts. Its interpreter performs automatic memoization and dependency tracking in a similar way as IncPy, but since it is a pure functional language, it does not need to do impurity detection. Also, since it is a domain-specific build scripting language, it has never been used for general data analysis, to the best of our knowledge.

Scientific workflow systems such as VisTrails [26] are graphical development environments for designing and executing scientific computations. Scientists create workflows by using a GUI to visually connect together blocks of pre-made data processing functionality into a data-flow graph. In these domain-specific visual languages, each block is a pure function whose results are automatically memoized and persist across executions. Due to their specialized nature, these systems are not nearly as popular for data analysis as general-purpose languages like Python or Perl. The creators of VisTrails admit, "While significant progress has been made in unifying computations under the workflow umbrella, workflow systems are notoriously hard to use. They require a steep learning curve: users need to learn programming languages, programming environments, specialized libraries, and best practices for constructing workflows [26]."

Self-adjusting computation is a related technique that enables algorithms to run faster in response to small changes in input data, only re-computing outputs for portions that have changed [4, 12]. Self-adjusting computation tracks fine-grained dependencies between executed basic blocks and the objects that they mutate, which can provide large speed-ups but requires programmers to annotate exactly which objects to track. Its creator mentions, "Although self-adjusting computation can be applied without having to change existing code by tracking all data and all dependences between code and data, this is prohibitively expensive in practice." [4]. Even with annotations, there is at least a 500% slowdown on the initial (empty cache) run [12]. In contrast, IncPy tracks dependencies at a coarser level between function calls and entire data structures and files, but it works automatically without user annotations and with small (∼16%) slowdowns.

Just-in-time compilers for dynamic languages (e.g., Unladen Swallow for Python, Rubinius for Ruby, TraceMonkey [8] for JavaScript) can speed up script execution times without requiring programmers to make any annotations, which is similar in spirit to IncPy's goals. However, JIT compilers focus on micro-optimizations of CPU-bound code like hot inner loops, whereas IncPy uses memoization to avoid long-running computations regardless of their source. For example, no JIT compiler optimizations could speed up I/O or network-bound scripts, which is what often consumes time in data analysis scripts. IncPy could be coupled with a JIT compiler to get the benefits of both techniques.

We introduced an early version of IncPy in a 2010 workshop paper [10], whose contents differ from this paper in significant ways: First, that paper focused on defining the problem space and establishing motivation using anecdotes from interviews with scientists. Also, it presented a preliminary version of IncPy (without a full evaluation) that did not support file write dependencies, file-based workflows, print buffering, fine-grained cache invalidation, or adaptive time limits. These new features enable more opportunities for memoization in real-world data analysis settings.

6. CONCLUSION

We have presented a novel and practical technique that uses a set of dynamic analyses to integrate automatic memo-

ization and persistent dependency management into a general-purpose programming language. We implemented our technique as an open-source Python interpreter named IncPy [1]. IncPy works transparently on regular Python scripts with a modest (∼16%) run-time slowdown, which improves both usability and reliability: IncPy is easy to use, especially by novice programmers, since it does not require programmers to insert any annotations or to learn new language features or domain-specific languages. Also, IncPy improves reliability by eliminating human errors in determining what is safe to cache, writing the caching code, and managing code and dataset dependencies. In sum, IncPy allows programmers across a wide range of disciplines to iterate faster on their data analysis scripts while writing less code.

Acknowledgments: Thanks to Christian Bird, Peter Boonstoppel, Suhabe Bugrara, Ewen Cheslack-Postava, Isil Dillig, Imran Haque, Paul Heymann, Robert Ikeda, Cory McLean, Adam Oliner, Fernando Perez, David Ramos, Marc Schaub, and Tom Zimmermann for their valuable feedback on this project since its inception in July 2009. This research was supported by the NSF Graduate Research Fellowship and the United States Air Force Research Laboratory (AFRL) through Contract FA8650-10-C-7024. Opinions, findings, conclusions and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the US Government, or the Air Force.

7. REFERENCES

[1] IncPy home: `http://www.pgbovine.net/incpy.html`.

[2] Official Python wiki: Large Python Projects `http://wiki.python.org/moin/LargePythonProjects`.

[3] TIOBE programming community index `www.tiobe.com/index.php/content/paperinfo/tpci/index.html`. 2011.

[4] U. A. Acar. Self-Adjusting Computation (An Overview). In *Plenary Talk at ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM)*, 2009.

[5] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer. Opportunistic programming: Writing code to prototype, ideate, and discover. *IEEE Software*, 26(5):18–24, 2009.

[6] J. C. Carver, R. P. Kendall, S. E. Squires, and D. E. Post. Software development environments for scientific and engineering software: A series of case studies. In *ICSE '07*, pages 550–559, 2007.

[7] S. I. Feldman. Make — a program for maintaining computer programs. *Software: Practice and Experience*, 9(4):255–265, 1979.

[8] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *PLDI '09*. ACM, 2009.

[9] P. J. Guo and D. Engler. Linux kernel developer responses to static analysis bug reports. In *USENIX Annual Technical Conference*, pages 285–292, 2009.

[10] P. J. Guo and D. Engler. Towards practical incremental recomputation for scientists: An implementation for the Python language. In *TaPP '10: Proceedings of the 2nd Workshop on the Theory and Practice of Provenance*, 2010.

[11] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy. Characterizing and predicting which bugs get fixed: An empirical study of Microsoft Windows. In *ICSE '10*, pages 495–504, May 2010.

[12] M. A. Hammer, U. A. Acar, and Y. Chen. CEAL: a C-based language for self-adjusting computation. In *PLDI '09*, pages 25–37. ACM, 2009.

[13] K. Heimerl, J. Vasudev, K. G. Buchanan, T. Parikh, and E. Brewer. Metamouse: Improving multi-user sharing of existing educational applications. In *International Conference on Information and Communication Technologies and Development*, 2010.

[14] A. Heydon, R. Levin, and Y. Yu. Caching function calls using precise dependencies. In *PLDI '00*, pages 311–320. ACM, 2000.

[15] P. Heymann and H. Garcia-Molina. Contrasting controlled vocabulary and tagging: Do experts choose the right names to label the wrong things? In *ACM International Conference on Web Search and Data Mining, Late Breaking Results Session*, February 2009.

[16] D. Horn, E. Cheslack-Postava, B. F. Mistree, T. Azim, J. Terrace, M. J. Freedman, and P. Levis. To infinity and not beyond: Scaling communication in virtual worlds with Meru. In *Stanford Computer Science Technical Report CSTR 2010-01*, May 2010.

[17] D. Kelly, D. Hook, and R. Sanders. Five recommended practices for computational scientists who write software. *Computing in Science and Engineering*, 11:48–53, 2009.

[18] A. Ko, R. Abraham, L. Beckwith, M. Burnett, M. Erwig, J. Lawrence, H. Lieberman, B. Myers, M. B. Rosson, G. Rothermel, C. Scaffidi, M. Shaw, and S. Wiedenbeck. The state of the art in end-user software engineering. *ACM Computing Surveys*, 2011.

[19] D. Michie. "Memo" Functions and Machine Learning. *Nature*, 218:19–22, 1968.

[20] N. Nethercote and J. Seward. How to shadow every byte of memory used by a program. In *Virtual execution environments*, VEE '07. ACM, 2007.

[21] W. S. Noble. A quick guide to organizing computational biology projects. *PLoS Computational Bio.*, 5(7), 07 2009.

[22] A. Oliner and J. Stearley. What supercomputers say: A study of five system logs. In *DSN '07*, Washington, DC, USA, 2007. IEEE Computer Society.

[23] D. Saff and M. D. Ernst. An experimental evaluation of continuous testing during development. In *ISSTA 2004, Proceedings of the 2004 International Symposium on Software Testing and Analysis*, pages 76–85, Boston, MA, USA, July 12–14, 2004.

[24] A. Salcianu and M. C. Rinard. Purity and side effect analysis for Java programs. In *VMCAI*. Springer, 2005.

[25] C. Scaffidi, M. Shaw, and B. Myers. Estimating the numbers of end users and end user programmers. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 207–214, 2005.

[26] C. E. Scheidegger, H. T. Vo, D. Koop, J. Freire, and C. T. Silva. Querying and re-using workflows with VisTrails. In *SIGMOD '08*. ACM, 2008.