`mandala`: Compositional Memoization for Simple & Powerful Scientific Data Management

Aleksandar Makelov
`aleksandar.makelov@gmail.com`

Abstract

We present `mandala`[1], a Python library that largely eliminates the accidental complexity of scientific data management and incremental computing. While most traditional and/or popular data management solutions are based on *logging*, `mandala` takes a fundamentally different approach, using *memoization* of function calls as the fundamental unit of saving, loading, querying and deleting computational artifacts.

It does so by implementing a *compositional* form of memoization, which keeps track of how memoized functions compose with one another. In this way: (1) complex computations are effectively memoized end-to-end, and become 'interfaces' to their own intermediate results by retracing the memoized calls; (2) all computations in a project form a single computational graph, which can be explored, queried and manipulated in high-level ways through a *computation frame*, which is a natural generalization of a dataframe in a way made precise in this paper.

Several features implemented on top of the core memoization data structures – such as natively and transparently handling Python collections, caching of intermediate results, and a flexible versioning system with dynamic dependency tracking – turn `mandala` into a practical and radically simpler tool for managing and interacting with computational data.

1 Introduction

Numerical experiments and simulations are growing into a central part of many areas of science and engineering (Hey et al., 2009). Recent trends in fields such as machine learning point towards ever-increasing complexity of computational pipelines, and increasingly safety-critical application domains such as autonomous driving (Bojarski et al., 2016), healthcare (Ravì et al., 2016; Abramson et al., 2024) and large language model autonomous agents (Yang et al., 2024).

These developments impose opposing constraints on the tools used to manage the resulting computational artifacts: on the one hand, they should be simple and easy to use by researchers, with a minimal learning curve and unobtrusive syntax and semantics; on the other hand, they should be powerful enough to enable high-level operations Wickham (2014), full data & code provenance auditing (Davidson & Freire, 2008) and reproducibility Ivie & Thain (2018) in complex projects. Rules and practices that help with these requirements exist and are well-known (Sandve et al., 2013; Wilkinson et al., 2016), but still require manual work, attention and discipline to follow. Researchers often operate under time pressure and/or need to quickly iterate on their code, which makes these best 'practices' a rather impractical time investment.

Thus, ideally we would like a system that (1) does not require learning a complex new language/semantics/syntax, (2) provides powerful high-level data management operations over complex computational projects, and (3) incorporates best practices by design and without cognitive overhead.

[1] `https://github.com/amakelov/mandala`

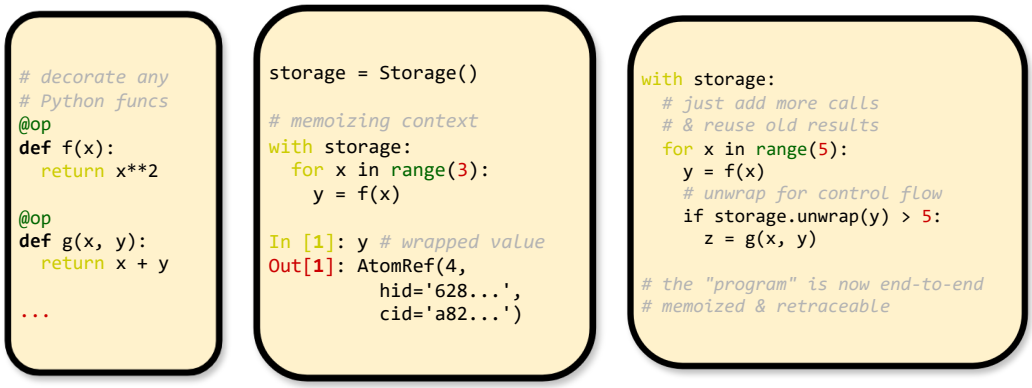

Figure 1: Basic imperative usage of `mandala`. **Left**: add the `@op` decorator to any Python functions to make them memoizable. **Middle**: create a `Storage`, and use it as a context manager to automatically memoize any calls to `@op`-decorated functions in the block. Memoized functions return `Ref` objects, which wrap a value with two pieces of metadata: a *content ID*, which is a hash of the value of the object, and a *history ID*, which is a hash of the identity of the `@op` that produced the `Ref` (if any), and the history IDs of the `@op`'s inputs. **Right**: the storage context allows simple incremental computation and recovery from failures. Here, we add more computations while transparently reusing already computed values.

In this paper we present `mandala`, our proposal for such a system. At its core, it **integrates best data management practices** – such as full data provenance tracking, idempotent & deterministic computation, content-addressable versioning of code and its dependencies, and declarative high-level data manipulation – **into Python's already familiar syntax and semantics** (Figures 1 and 3). The integration aims to be maximally transparent and unobtrusive, so that the user can focus on the *essential complexity* (the scientific problem at hand), rather than on the *accidental complexity* (the data management tools used to solve it) (Brooks, 1987).

The rest of this paper presents the design and main functionalities of `mandala`, and is organized as follows:

- In Section 2, we describe how memoization is designed, how this allows memoized calls to be composed and memoized results to be reused without storage duplication, and how this enables the *retracing* pattern of interacting with computational artifacts.
- In Section 3, we introduce the concept of a *computation frame*, which generalizes a dataframe by replacing columns with a computational graph, and rows with individual computations that (partially) follow this graph. Computation frames allow high-level exploration and manipulation of the stored computation graph, such as adding the creators/consumers of given values to the graph, deleting all computations that depend on the calls captured in the frame, and restricting the frame to a particular subgraph or subset of values with given properties.
- In Section 6, we describe some other features of `mandala` necessary to make it a practical tool, such as optionally storing elements of Python collections as separate values (enabling transparent reuse of individual elements), caching of intermediate results, and a flexible versioning system with dynamic dependency tracking.

2 Core Concepts

2.1 Memoization and the Computational Graph

Memoization is a technique that stores the results of expensive function calls to avoid redundant computation. `mandala` uses *automatic* memoization (Norvig, 1991) which is

applied via the combination of a decorator (`@op`) and a context manager which specifies the `Storage` object to use (Figure 1). The memoization can optionally be made persistent, which is what you would typically want in a long-running project. Any Python function can be memoized (as long as its inputs and outputs are serializable by the `joblib` library); there is no restriction on the type, number or naming scheme (positional, keyword, variadic) of the arguments or return values.

`Call` and `Ref` objects, and content/history IDs. `Refs` and `Calls` are the two main data structures in `mandala`'s model of computations. When a call to an `@op`-decorated function `f` is executed inside a storage context, this results in the creation of

- a `Ref` object for each input to the call. These wrap the 'raw' values passed as inputs together with content IDs (hashes of the Python objects) and history IDs (hashes of the memoized calls that produced these values, if any). If an input to the call is already a `Ref` object, it is passed through as is; if it is a raw value, a new `Ref` object is created with a 'empty' history ID that is simply a hash of the content ID itself.
- a `Call` object, which has pointers to the input and output `Refs` of the call to `f`, as well as a content ID for the call (a hash of the identity of `f` and the content IDs of the input `Refs`) and a history ID (by analogy, a hash of the identity of `f` and the history IDs of the inputs).
- a `Ref` object for each output of the call. These are again assigned content IDs by value, and history IDs by hashing the tuple (history ID of the call, corresponding output name[2]).

The `Refs` and the `Call` are then stored in the storage backend, and the next time `f` is called on inputs that have the same *content* IDs, the stored `Call` is looked up to find the output `Refs`, which are then returned (possibly with properly updated history IDs, if the call exists in storage by content ID only). The combination of all stored `Calls` and `Refs` across memoized functions form the **computational graph** represented by the storage (Figure 2). Importantly, the 'interesting' structure of this graph is built up automatically by the way the user composes memoized calls.

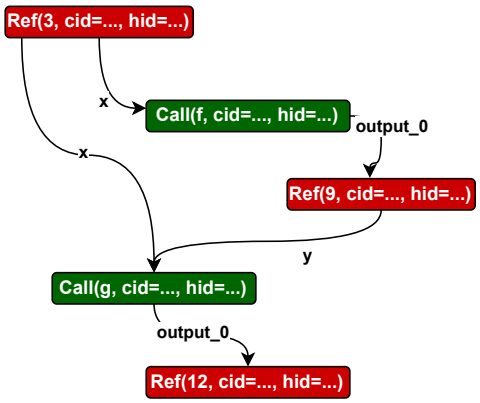

Figure 2: A part of the computaitonal graph built up by the calls in Figure 1. The nodes are `Call` and `Ref` objects, and the edges are the inputs/output names connecting them.

The simultaneous use of content and history IDs has a few subtle advantages:

- it allows for the *de-duplication* of storage, as the same content ID can be used to store the same value produced by different computations. For instance, there may be many computaitions all producing the value 42 (or a large all-zero array), but only one copy of 42 is stored in the backend.
- at the same time, the history IDs allow us to distinguish between computations that produced the same value, but in different ways. This avoids 'parasitic' results in declarative queries. For example, a call to `f` may result in 42, and we may be interested in all computations that were called on this particular 42 returned by `f` and not on any other 42. Without history IDs, it would be impossible to make this distinction in the stored computational graph.

[2]Since Python functions don't have designated output names, we instead generate output names automatically using the order in the tuple of return values.

Why memoization? Memoization is an unusual choice for data management systems, most of which are based on *logging*, i.e. explicitly pointing to the value to be saved and the address where it should be saved (whether this is some kind of name or a file path). Basing data management on memoization means that the 'address' of a value is now implicit in the code that produced it, and the value itself is stored in a shared storage backend. This has several advantages:

- it **eliminates the need to manually name artifacts**. This eliminates a major source of accidental complexity: names are arbitrary, ambiguous, and can drift away from the actual content of the value they point to over time. On the other hand, names are not strictly necessary, because the composition of memoized functions that produced a given value – which must be specified anyway – is already a canonical reference to it.
- it **organizes storage activities around a familiar and useful interface – the function call**. This automatically enforces the good practice of partitioning code into functions, and eliminates extra 'accidental' code to save and load values explicitly. Furthermore, it synchronizes failures between computation and storage, as the memoized calls are the natural points to recover from.

The trade-off is that referring to values without reference to code that produced or used them becomes difficult, because from the point of view of storage the 'identity' of a value is its place in the computational graph. We discuss practical ways to overcome this in Section 3.

2.2 Retracing as a Versatile Imperative Interface to the Stored Computation Graph

The compositional nature of memoization makes it possible to build complex computations out of calls to memoized functions, turning the entire computation into an end-to-end-memoized interface to its own intermediate results. The main way to interact with such a computation is through **retracing**, which means stepping through memoized code with the purpose of resuming from a failure, loading intermediate values, or continuing from a particular point with new computations. A small example of retracing is shown in Figure 1 (right).

This pattern is simple yet powerful, as it allows the user to interact with the stored computation graph in a way that is adapted to their use case, and to explore the graph in a way that is natural and familiar to them. It also simplifies the management of state in an interactive environment such as a Jupyter notebook, because it makes it very cheap to re-run cells.

3 Computation Frames

In order to be able to explore and manipulate the full stored computation graph, patterns like retracing are insufficient, because they require the complete code producing part of the graph to be available. Instead, we introduce the concept of a *computation frame*, which is a high-level declarative interface to the stored computation graph.

3.1 Definition

A computation frame (Figure 3) consists of the following data:

- **computation graph**: a directed graph $G = (V, F, E)$ where V are variables, F are `@op`-decorated functions, and labeled edges E connect from variables to functions using these variables as input (with the edge label being the input name), and from functions to variables that are outputs of these functions (with the edge label being the name of the output). An example is shown in Figure 2.
- **instance data**: for each variable $v \in V$, a set of history IDs H_v of `Refs` corresponding to this variable, and for each function $f \in F$ a set of history IDs C_f of `Calls` to f corresponding to this node.

```
In [1]:
  # get the computation frame for f
  storage.cf(f).\
    # add all computations reachable
    # from calls to f
    expand().\
      # extract as a dataframe
      eval()
Out [1]: Extracting tuples from the
computation graph:
  output_0 = f(x=x)
  output_1 = g(y=output_0, x=x)
```

	x	f	output_0	g	output_1
0	4	Call(f, cid='05e...', hid='f87...')	16	Call(g, cid='4c0...', hid='c95...')	20.0
1	1	Call(f, cid='845...', hid='a55...')	1	None	NaN
2	3	Call(f, cid='f41...', hid='dab...')	9	Call(g, cid='00e...', hid='0ba...')	12.0
3	2	Call(f, cid='77b...', hid='1cb...')	4	None	NaN
4	0	Call(f, cid='82f...', hid='b98...')	0	None	NaN

Figure 3: Basic declarative usage of `mandala` and an example of computation frames. **Left**: continuing from Figure 1, we first create a computation frame from a single function `f`, then expand it to include all calls that can be reached from the memoized calls to `f` via their inputs/outputs, and finally convert the computation frame into a dataframe. We see that this automatically produces a computation graph corresponding to the computations found. **Right**: the output of the call to `.eval()` from the left subfigure used to turn the computaiton frame into a dataframe. The resulting table has columns for all variables and functions appearing in the captured computation graph, and each row correspond to a partial computation following this graph. The variable columns contain values these variables take, whereas function columns contain call objects representing the memoized calls to the respective functions. We see that, because we call `g` conditional on the output of `f`, some rows have nulls in the `g` column.

Most importantly, this data is subject to the constraint that, if there exists a call $c \in C_f$ where f is connected via an input/output edge to some variable v, then the value of v corresponding to this call belongs to H_v. In other words, when we look at all calls in $f \in F$, their inputs/outputs must be present in the variables connected to f under the same input/output name.

Intuitively, computation frames are 'views' of the stored computation graph, analogous to database views. In particular, they can be organized in a way that is natural for a specific use case, may contain multiple references to the same `Ref` or `Call` object, and do not necessarily contain all calls to a given function. Formally speaking, there is a directed graph homomorphism (a direction- and adjacency-preserving mapping from vertices to vertices and edges to edges) from a computation frame's instance data to the full stored computation graph.

3.2 Basic Usage

The main advantage of computation frames is that they allow iterative exploration of the computation graph, and high-level grouped operations over computations with some shared structure. For example, we can use them for

- **iteratively expanding the frame with functions that generated or used existing variables**: this is useful for exploring the computation graph in a particular direction, or for adding more context to a particular computation. For example, in Figure 3 (left), we start with a computation frame containing only the calls to `f`, and then expand it to include all calls that can be reached from the memoized calls to `f` via their inputs/outputs, which adds the calls to `g` to the frame.
- **converting the frame into a dataframe**: this is useful at the end of an exploration, when we want to get a convenient tabular representation of the captured computation graph. The table is obtained by collecting all terminal `Refs` in the frame's computational graph (i.e., those that are not inputs to any function in the frame),

computing their computational history in the frame (grouped by variable), and joining the resulting tables over the variables. This is shown in Figure 3 (right). In particular, as shown in the example, this step may produce nulls, as the computation frame can contain computations that only partially follow the graph.

- **performing high-level storage manipulations**: such as deleting all calls captured in the frame as well as all calls that depend on them, available using the `.delete_calls()` method on the frame.

Computation frames are a powerful tool for exploring and manipulating the stored computation graph, and we're excited to explore their full potential in future work.

4 Some Extra Features

4.1 Data Structures

Python's native collections – lists, dicts, sets – can be memoized transparently by `mandala`, using customized type annotations, e.g. `MList[int]` inheriting from `List[int]`, ... By applying this type annotation, the collection is memoized so that its individual elements as well as the collection itself are memoized as `Refs` (with the collection merely pointing to the `Refs` of its elements to avoid duplication).

```
@op
def avg_items(xs: MList[int]) -> float:
  return sum(xs) / len(xs)

@op
def get_xs(n) -> MList[int]:
  return list(range(n))

with storage:
  xs = get_xs(10)
  for i in range(2, 10, 2):
    avg = avg_items(xs[:i])
```

Figure 4: Illustration of native collection memoization in `mandala`. The custom type annotation `MList[int]` is used to memoize a list of integers as a list of pointers to element `Refs`.

This is implemented fully on top of the core memoization machinery, using 'internal' `@ops` like e.g. `__make_list__` which, given the elements of a list as variadic inputs, generates a `ListRef` (subclass of `Ref`) that points to the `Refs` of the elements. In this way, collections are naturally incorporated in the computation graph. These internal `@ops` are applied automatically when a collection is passed as an argument to a memoized function, or when a collection is returned from a memoized function. The upshot is that collection elements can be reused transparently, as shown in Figure 4.

4.2 Caching

To speed up retracing and memoization, it is necessary to avoid frequent reads and writes to the database backend. To this end, `mandala` implements a fairly standard caching system that accumulates results in memory, with the option to explicitly flush values to disk at once when the user decides, or automatically at the end of a `mandala` context. Similarly, to speed up retracing of existing calls, call metadata can be pre-loaded.

4.3 Versioning

It is crucial to have a flexible and powerful code versioning system in a data management tool, as it allows the user to keep track of the evolution of their computations, and to easily recover from mistakes. `mandala` provides the option to use a versioning system with two main features:

- **content-addressed versioning** (Torvalds et al., 2005), where the version of a function is a hash of its source code and the hashes of the functions it calls.
- **dynamic dependency tracking**, where each function call traces the functions it calls. This avoids the need for static analysis of the code to find dependencies, which can

result in many false positives and negatives, especially in a dynamic language like Python.

A full discussion of the versioning system is beyond the scope of this paper, but we refer the reader to a blog post by the author (Makelov, 2023) for details and motivation.

5 Related Work

`mandala` combines ideas from several existing projects, but is unique in the Pythonic way it makes complex memoized computations easy to query, manipulate and version.

Memoization. The `incpy` project (Guo & Engler, 2011) enables automatic persistent memoization of Python functions directly on the interpreter level. The `funsies` project (Lavigne & Aspuru-Guzik, 2021) is a memoization-based distributed workflow executor that uses a similar hashing approach to `mandala` to keep track of which computations have already been done. It works on the level of scripts (not functions), and lacks queriability and versioning. `koji` (Maymounkov, 2018) is a design for an incremental computation data processing framework that unifies over different resource types (files or services). It also uses an analogous notion of hashing to keep track of computations.

Computation Frames. Computation frames are very closely related to functors $\mathcal{F} : \mathscr{C} \to \underline{\mathbf{Set}}$ where $\mathscr{C}$ is a finite category, and this perspective has been explored in e.g. (Patterson et al., 2022).

Versioning. The revision history of each function in the codebase is organized in a 'mini-git repository' that shares only the most basic features with `git` (Torvalds et al., 2005): it is a content-addressable tree, where each edge tracks a diff from the content at one endpoint to that at the other. Additional metadata indicates equivalence classes of semantically equivalent contents. Semantic versioning (Preston-Werner, 2013) is another popular code versioning system. mandala is similar to semver in that it allows you to make backward-compatible changes to the interface and logic of dependencies. It is different in that versions are still labeled by content, instead of by 'non-canonical' numbers. The unison programming language (Lozano, 2017) represents functions by the hash of their content (syntax tree, to be exact).

6 Conclusion

While `mandala` is not very mature yet and still very much a work in progress, it has the potential to considerably simplify the way scientific data is managed and interacted with in Python. The author has already used it extensively to manage several multi-month machine learning projects, and has found it to be a very powerful tool for managing complex computations. We hope that this paper has given a good overview of the core concepts of `mandala`, and that the reader will be interested in exploring the library further.

References

Josh Abramson, Jonas Adler, Jack Dunger, Richard Evans, Tim Green, Alexander Pritzel, Olaf Ronneberger, Lindsay Willmore, Andrew J Ballard, Joshua Bambrick, et al. Accurate structure prediction of biomolecular interactions with alphafold 3. *Nature*, pp. 1–3, 2024.

Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

Frederick P. Brooks. No silver bullet: Essence and accidents of software engineering. 1987. URL `https://api.semanticscholar.org/CorpusID:372277`.

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

Philip J Guo and Dawson Engler. Using automatic persistent memoization to facilitate data analysis scripting. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pp. 287–297, 2011.

Tony Hey, Stewart Tansley, Kristin Michele Tolle, et al. *The fourth paradigm: data-intensive scientific discovery*, volume 1. Microsoft research Redmond, WA, 2009.

Peter Ivie and Douglas Thain. Reproducibility in scientific computing. *ACM Computing Surveys (CSUR)*, 51(3):1–36, 2018.

Cyrille Lavigne and Alán Aspuru-Guzik. funsies: A minimalist, distributed and dynamic workflow engine. *Journal of Open Source Software*, 6(66):3274, 2021.

Roberto Castañeda Lozano. The unison manual, 2017.

Aleksandar Makelov. Practical dependency tracking for python function calls. `https://amakelov.github.io/blog/deps/`, 2023. Accessed: Jun 2024.

Petar Maymounkov. Koji: Automating pipelines with mixed-semantics data sources. *arXiv preprint arXiv:1901.01908*, 2018.

Peter Norvig. Techniques for automatic memoization with applications to context-free parsing. *Computational Linguistics*, 17(1):91–98, 1991.

Evan Patterson, Owen Lynch, and James Fairbanks. Categorical data structures for technical computing. *Compositionality*, 4, 2022.

Tom Preston-Werner. Semantic versioning 2.0.0. `https://semver.org/`, 2013. Accessed: 2024-06-08.

Daniele Ravì, Charence Wong, Fani Deligianni, Melissa Berthelot, Javier Andreu-Perez, Benny Lo, and Guang-Zhong Yang. Deep learning for health informatics. *IEEE journal of biomedical and health informatics*, 21(1):4–21, 2016.

Geir Kjetil Sandve, Anton Nekrutenko, James Taylor, and Eivind Hovig. Ten simple rules for reproducible computational research. *PLoS computational biology*, 9(10):e1003285, 2013.

Linus Torvalds et al. *Git*, 2005. URL `https://git-scm.com/`. Version control system.

Hadley Wickham. Tidy data. *Journal of statistical software*, 59:1–23, 2014.

Mark D Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, Jan-Willem Boiten, Luiz Bonino da Silva Santos, Philip E Bourne, et al. The fair guiding principles for scientific data management and stewardship. *Scientific data*, 3(1):1–9, 2016.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*, 2024.