Margo: Margin Notes for Computational Notebooks

Margo: Margin Notes for Computational Notebooks

Jake Kara

A Thesis in the Field of Software Engineering

for the Degree of Master of Liberal Arts in Extension Studies

Harvard University

May 2021

Abstract

Jupyter Notebooks combine prose, code and execution output, making them a popular choice over plain source code for communicating computational methods. However, notebooks cannot be reused as modules, and the recommended practice of writing module code in plain source files forces developers to abandon the notebook format. This thesis offers the rationale, design, and implementation details for a system for importing Jupyter Notebooks as Python modules. One challenge we face is that notebook code is written differently from module code — Jupyter Notebooks often include non-reusable code. Therefore, we must be able to define a module view for a notebook that excludes particular cells. To solve this, we develop a scheme of specially formed code comments. We call these comments “margin notes,” and we develop a flexible syntax for margin notes called “Margo.” The margin note system opens up additional ways to extend notebooks. Two specific applications that leverage Margo notes extensively are presented as exemplars: A command line tool to define arbitrary interfaces into notebooks for applications such as pip and GNU Make; and a notebook editor UI concept that supports hierarchical cell relationships.

Acknowledgments

I wish to thank my thesis director, Dr. Bakhtiar Mikhak, for so many hours of discussion over this past year, without which this project would not have succeeded.

I would also like to acknowledge the guidance and assistance I received throughout this process from my research advisor, Dr. Hongming Wang.

Table of Contents

List of Figures

List of Tables

Chapter I.

Introduction

This project is predicated upon the need to extend Jupyter Notebooks. In this chapter we provide evidence for that need. An exploration of the evolution and current state of the art in Jupyter Notebook technology shows that notebooks are being used in ways that extend beyond the use cases envisioned by its maintainers. We discuss the important role modules play in organizing and comprehending software and the inhibiting effect that Jupyter Notebooks' lack of modules has on the software written in them. We also place our modular notebook use case in context as one among many ways Jupyter Notebooks are being extended. We propose that with so much activity around extending notebooks there is a need for general patterns to emerge to facilitate this development, and margin notes are one such pattern with general applications.

1.1. Jupyter Notebook Technology

Jupyter Notebook technology has its origins in IPython, or "interactive python," a command line interpreter for a superset of the Python programming language that added features optimized for human-readable scientific computing (Perez & Granger, 2007). IPython Notebook, launched in December 2011 as part of the IPython 0.12 release, is a web application for interactive computing with a cell-based workflow (Perez, 2012). Cells contain either Markdown prose, IPython code, or execution output. In 2014, IPython Notebook began to support other languages in addition to IPython by implementing interpreters for different languages as kernels that share a common interface. At this point IPython Notebook became Jupyter Notebook — a combination of three programming language names: Julia, Python and R.

Jupyter Notebook documents are serialized as JSON documents that use the file extension ".ipynb," an abbreviation for the predecessor "IPython Notebook." Notebooks contain document-level metadata and an array of cell objects, each with its own metadata and content. An example of information stored as document-level metadata is the name of the programming language kernel used to interpret code cells and generate outputs. An example of cell-level metadata is a numeric execution counter value that indicates the order of cell execution (Jupyter Development Team, 2015).

Jupyter Notebook authoring tools consist of a frontend interface for creating, updating and deleting notebook document cells, as well as a backend consisting of a kernel session. The front-end sends code cell contents to the kernel and receives execution output over a messaging protocol. The execution output received from the kernel session is typically displayed in the frontend. There are a number of Jupyter Notebook authoring tools, two of which are developed by Project Jupyter: Jupyter Notebook and JupyterLab.

The concepts in this paper may apply to other notebook systems. Jupyter is not the first or only notebook system, but it is a very popular one. It is also perhaps the only notebook system that is programming language independent, and it has kernels that support an incredible number of languages.

1.2. Modules and Jupyter Notebooks

Notebooks cannot be imported the same way plain source code files can be. The orthodox approach to writing modular Python code is to write modules in plain *.py* files and import them into a *.ipynb* file. This approach is not ideal for an author who would prefer to write all or at least more of their code in notebooks. In addition, this approach doesn't appear to be used often in practice: In a large-scale analysis, Pimentel et al. (2019) found that 91% of notebooks imported modules, but only 10% imported local modules defined in the same repository as the Notebook.

Another approach is to convert notebooks to a plain source representation and then import the resulting document. This can be done with tools such as nbconvert or Jupytext, which can produce an intermediate *.py* file. Alternatively, the modular representation can be created in memory at the time of import by extending Python's import machinery. The latter approach was demonstrated to import Jupyter Notebooks in Python by the Jupyter Project (Jupyter Team, 2015) and has been adapted in packages such as nbimporter (Sturm, 2019).

Whether producing an intermediate *.py* file or producing one in memory during import, loading every code cell in a notebook is often not ideal. The Margo Loader application we develop in Chapter VI uses this approach of converting a document at import time, but it adds a step of using Margo notes to determine which cells should be imported and how.

1.3. Literate Programming

Jupyter Notebooks' combination of prose and code is often compared with the literate programming paradigm proposed by Knuth (1984), but this resemblance may be superficial. Knuth asserts that the chief concern of a software developer is to communicate with other humans first and communicate in the terse language of computers second. He created a paradigm of programming that embedded code macros within prose. This prose is the developer's description of the software system in a way that is first comprehensible to a human, rather than to a computer. Knuth's WEB tool implemented literate programming with TeX markup for prose and Pascal for code. WEB could preprocess code for computer comprehension (compilation) or render a human comprehensible narrative explanation of the system's inner workings.

While making the case for the WEB system, Knuth simultaneously made a bigger argument: Software development can and should be made comprehensible for the humans who write software, even if that means aggressively questioning assumptions in current practices and tools. Even though the paradigm he proposed has not become widely adopted in software engineering, the concept of literate programming remains present in the literature of the software engineering research discipline, even as an aspirational ideal. Knuth's paper proposing literate programming had been cited more than 2,200 times as of April 2020, according to a search on Google Scholar.

Jupyter Notebooks, like Knuth's literate programming system, prioritize the human experience and human understanding first in their design, particularly with features such as enabling arbitrary code execution order and combining markup prose and code. But there are critical differences between the philosophies motivating Jupyter

Notebooks and literate programming. Knuth's aim was making it easier to maintain and comprehend substantially complex software systems. WEB was in spirit a modular system for engaging in the practice of software architecture — organizing software as a system of discrete components. Without a notion of modules, Jupyter cannot support this kind of software organization. Perez and Jupyter Project have discussed Jupyter Notebook's resemblance to — and contrasts with — Knuth's concept:

> "Knuth's concept of Literate Programming, where the emphasis is on narrating algorithms and programs. In a Literate Computing environment, the author weaves human language with live code and the results of the code, and it is the combination of all that produces a computational narrative." (Perez & Granger, 2015)

If literate programming is a way to narrate a software system, Jupyter is seen by its maintainers as a way to narrate an interactive computing session. At least, that is the scope Jupyter's maintainers choose to focus on. This is not to say that Jupyter is actually used within this limited scope, as we will see in the "Jupyter Notebooks Extended" section.

1.4. Software Organization as a Reflection of Thought

Knuth's Literate programming is part of a recurring theme in the software engineering research discipline: that building software systems requires organizing smaller units of code into larger systems. Designing a software system and implementing its individual components are discrete intellectual tasks. DeRemer and Kron (1976) articulated this, calling the act of implementing a module's source code "programming in the small," in contrast with connecting these modules in a system, which the authors called "programming in the large." These two activities are complementary. The ability to organize code into modules supports thinking about the whole software system at times, and about individual components at other times. To be sure, these labels of "system" versus "component" are not absolute — a large software system could itself be a component in another system. Modules are fundamental to any substantive software design (van der Hoek, A., Lopez, 2011). Without modules, it would be very difficult or impossible to understand software if we could not organize it this way. Dijkstra (1982) articulated the importance of being able to maintain these small and large views of a thing in order to understand it well:

> "We know that a program must be correct and we can study it from that viewpoint only; we also know that it should be efficient and we can study its efficiency on another day, so to speak. In another mood we may ask ourselves whether, and if so, why, the program is desirable. But nothing is gained – on the contrary! – by tackling these various aspects simultaneously. It is what I sometimes have called "the separation of concerns", which, even if not perfectly possible, is yet the only available technique for effectively ordering one's thoughts that I know of."

Despite being visually expressive documents, the lack of notebook modularity limits how an author can organize and in turn think about their software. Only software which may be represented in a single document can be fully expressed in a notebook. Some developers may not find this to be a problem at all, but those with a habit of continuously modularizing their software into components may quickly find the notebook environment confining. Furthermore, developers who do not approach programming with this modularization habit may be less likely to ever acquire such a habit if most or all of their programming is done in Jupyter Notebooks — they may be confined in their methods without ever realizing it.

1.5. Jupyter Notebook Use Cases

An analysis of 1.4 million Jupyter Notebooks found 24 percent executed from start to finish without throwing exceptions, and only 4 percent produced expected outputs (Pimentel et al. 2019). This falls quite short of the goal of reproducible methodology documents. However, it is important to realize that this is not always the goal of a notebook's author. After all, notebook use cases can be as diverse as the use cases for physical notebooks, which might be used to write the great American novel, jot down a grocery list, or sketch idly in the margins. Kery (2018) identified three over-arching modes for notebook use:

1. In the "scratch pad use case," authors only meant to write preliminary code, perhaps to test a few lines of syntax.
2. In the "production pipeline use case," code developed in a notebook would eventually be extracted into a production system.
3. The third use case, "sharing," was when notebook authors reported taking the most care with code and making it comprehensible.

These use cases are not always isolated. Code may begin as scratch code, end up copied into a pipeline when it is deemed useful enough, and eventually shared with others. We should keep in mind that this "migration" between use is another common activity among Jupyter Notebook authors.

1.6. Jupyter Notebooks Extended

While the scope of the Jupyter Project does not envision notebooks as a replacement for conventional integrated development environments (IDEs), notebooks are used in robust software systems in ways that mirror, borrow from, and augment more traditional software engineering tools. The software systems we discuss here are among those that demonstrate a strong demand for using notebooks outside of use cases envisioned as part of the core Jupyter development community's scope. We can generally think of extended notebook uses as falling into one of two broad categories:

- "feature extensions": adding features, particularly IDE-like features to conventional notebook authoring tools; and
- "context extensions": using notebook documents outside of conventional notebook authoring environments altogether.

In this section we will discuss a few projects that demonstrate the diverse ways in which developers are using and extending Jupyter notebooks.

Gather

Notebooks can become "messy," with orphaned cells that do not affect the rest of the notebook. A tool called Gather was developed to allow users to select a particular output and generate "clean" notebooks containing only the cells upon which that output depends (Head et al., 2019).

Variolite

Maintaining different versions of code cells is important during the iterative experimentation sessions that notebooks enable. Conventional versioning tools have trouble with creating meaningful diffs at least in part because notebooks may contain binary attachments. It's also cumbersome to manage small pieces of code in existing version control systems. Researchers built Variolite, a cell-level version control system that allows notebook authors to rapidly iterate on code while keeping a running history of changes (Kery et al., 2017).

SOS Notebook

A Jupyter Notebook can only run code in one kernel and therefore one programming language. SOS Notebook, short for "Script of Script," adds a number of features to Jupyter Notebook, including the ability to execute code with a different kernel for each cell (Peng et al., 2018). The related SOS Workflow project extends Jupyter Notebook further, enabling cells to be executed as a workflow with a runtime signature specifying inputs and outputs. Unlike Gather and Variolite, SOS is a fork of Jupyter Notebook and not intended to integrate with it because it changes so many fundamental aspects of Jupyter Notebook.

Jupytext

Jupytext allows conversion from Jupyter Notebooks to plaintext notebook formats — plain source code files that use specially formed comments, such as `(#%)` to signify new cells. This allows a developer to use a notebook authoring tool but save code in a format that is more compatible with conventional version control systems, or to use a plain-text editor to write documents that may be converted into Jupyter documents automatically (Wouts, 2018). Microsoft Visual Studio Code supports these plain text notebooks, further blurring the notebook-IDE line. This use case deserves particular attention, as it illustrates that notebooks are in fact well suited as enhanced stand-ins for plain source documents, and in fact, notebooks may *be* plain source documents.

Papermill

Papermill uses notebooks as executable tasks. Input parameters may be defined in cells that are marked with the nbformat cell metadata tag “parameters.” Notebooks may be executed from a command line or programmatically with a Python library (Seal, 2019).

General Patterns

During a closing Keynote at JupyterCon 2020, Fernando Perez said that while Jupyter excels at interactive, “human-in-the-loop” computing, it falls short in more traditional, automated software engineering contexts. He acknowledged that notebooks are being used beyond this interactive scope, particularly in a transitional use case — where users begin work in an interactive mode but have a need to transition their work into different contexts (Perez, 2020). These comments and the applications we have just discussed demonstrate there is a need for general patterns around extending Jupyter notebooks.

As we develop Margo and demonstrate its applications, we consider how a standard syntax for margin notes can support general patterns for extending notebooks. Feature-extending software may currently store housekeeping data about cells and notebooks internally that could be exposed and serialized in Margo notes. (We develop such a reference implementation in Chapter VIII). Context-extending software that interprets notebooks may infer properties of a notebook that could instead be made explicit through Margo notes. (We demonstrate this in Chapter VI).

1.7. Overloading Comments

Margo notes are intended to be embedded inline within code comments and specially formed markup to differentiate them from the body contents of their respective code or markup cells. Margo notes could be embedded in cell-level metadata within the underlying *.ipynb* document's JSON structure, but inline notes were chosen instead. First, the Jupyter community discourages applications from using metadata for application-specific, non-well-known functionality. Second, not all authoring tools expose document metadata for viewing or editing. Third, by placing Margo in line with the contents of cells, the same approach can be used in other notebook systems, or even in non-Jupyter plain source code files. While source code comments are primarily used as notes for humans, they are often overloaded with machine-readable syntax in various applications.

Unix Shell Scripts

Unix shell scripts may begin with a specially formed “sha-bang” line that begins with `#!` to specify an interpreter for the script. For example, the following are both common approaches to make a script execute using the Bourne Shell (Cooper, 2014):

```
#!/bin/sh

#!/usr/bin/env sh
```

Python Character Encoding

Python files may use the following syntax to indicate the character encoding of the document. (Lemburg et al., 2001). For example:

```
# -*- coding: <encoding name> -*-
```

Comment Docs

Specially formed comments, often called comment docs, may be interpreted by software to automatically generate rich PDF or HTML documentation. Javadoc is an example of software that interprets comment docs in Java code (Oracle, 2020). Below is an example of the syntax:

```
/**
 * Returns a greeting.
 *
 * @param     name  whom to greet
 * @return    string.
 */
public String sayHello(string name) {
...
}
```

Conditional Comments in HTML

Microsoft's Internet Explorer has used conditional comments to include or exclude segments of HTML documents based on the version of the browser interpreting it. (Microsoft, 2015). For example:

```
<!--[if IE 6]>
<p>You have a really old browser!</p>
<![endif]-->
```

Chapter II.

Project Overview

Along with this paper, this thesis includes several software deliverables, which are enumerated in this chapter.

2.1. Software Deliverables

This thesis project involved developing six deliverable work products, which are discussed in subsequent chapters. The first is the Margo specification and the remaining five are software. All of the source code for this software, as well as live instances which can be run in a browser, are available online. See Appendix 1 to access the source code submitted with this thesis.

Chapter IV, "Margo Specification," discusses the only non-software deliverable. It is a syntax specification for Margo. Margo is a data serialization syntax that has no reserved keywords, but applications built on top of it do require the use and reservation of keywords with agreed-upon meaning, we outline a process for keyword proposal that is observed in subsequent sections.

Chapter V, "Margo Parser," discusses a Python library for parsing Margo syntax. Margo Parser is a dependency of the two subsequent software deliverables: Margo Loader and Margo Tool.

Chapter VI, "Margo Loader," discusses a Python library that extends Python's import machinery to load Jupyter Notebook documents as Python modules. Margo

Loader interprets preprocessor directives stored in Margo statements to allow Jupyter Notebook authors to explicitly define a notebook's module view.

Chapter VII, "Margo Tool," discusses a command line application that extracts Margo notes from Jupyter Notebooks in order to integrate them with other software systems.

Chapter VIII, "Margo Editor," discusses a prototype of a notebook editor user interface that supports cell relationships, which are not a part of the Jupyter Notebook document format. This demonstrates how Margo can be used to serialize metadata within notebook documents in order to extend their features beyond what is supported by the underlying format.

Chapter IX, "Results," discusses the significance of the deliverables above. It contains a case study discussing how modular notebooks have been used by this author to write modular methodology software entirely in notebooks for an experiment of the Yale Digital Humanities Lab. This suite of modular notebooks performs image processing and color palette extraction. It is the final software deliverable included with this thesis submission. A live version of this notebook collection is also available online.

Chapter III.

Technology Selection

This chapter discusses why the technologies used this project were selected.

3.1. Jupyter

The concept of margin notes applies generally to most computational notebook systems and programming languages, and the Margo specification is general enough for reuse in different systems. For implementation, this document assumes Jupyter Notebooks with a Python kernel.

Jupyter was chosen for its vast and growing popularity across diverse disciplines, and its programming language-agnostic design. Because it is programming language-agnostic, the solutions proposed here for using Jupyter Notebooks as a plain source code replacement can be leveraged to a greater effect. Jupyter Notebook documents also lend themselves to reuse in alternative contexts outside of the notebook authoring environment because of the system’s modular software architecture: The execution environment, front-end, back-end and document format are all well-defined and isolated components. Finally, Jupyter is an open project. It has transparent governance, and its code is published as free software. This makes it easier and more productive to extend.

3.2. Python

The Python programming language is chosen because it is popular across many disciplines due to its relatively beginner-friendly syntax and enforcement of good programming practices, such as meaningful indentation. Additionally, Python is the most popular language used in Jupyter Notebooks. Finally, and quite importantly, Python is designed to be easily extended. It has an API to extend its import machinery in ways that may be possible but significantly more difficult in other languages. It is also possible in Python to create submodules programmatically rather than only by using the language's import statement. These features enable the rapid development of Margo Loader without focusing on implementation details of a less hackable language.

3.3. Software Licensing

All software developed for this thesis is free and open-source software licensed under the GNU General Public License version 3.0 (GPL 3). This software is built upon the shoulders of other open-source projects and specifications, and it should be free for modification and reuse to anyone who wishes to expand upon it.

Chapter IV.

Margo Specification

This chapter may be considered a stand-alone document that aims to describe and define the syntax of Margo completely and unambiguously. It is not meant to serve as a practical introduction to using Margo. While example uses appear in this chapter, they are only to clarify the syntax and decisions about the syntax.

4.1. Overview

Margo is a syntax for annotating computational notebook cells, including both code and markup cells, with "margin notes." Margin notes may be thought of as metadata. Each note is a statement, one of either:

- a directive statement: a name only; or
- an assignment statement: a name with an associated value.

A value may be formatted as a comma-separated array of JSON scalars (numbers, strings, true, false, null) or as a single valid string of a supported external format, such as JSON, YAML or raw text. Margin notes written in Margo will from here be referred to as "Margo notes," while "margin notes" will be used when referring generally to the concept of margin notes regardless of the syntax.

Margo is best thought of as a serialization format such as JSON or YAML. It has no logic or reserved keywords with semantic meaning that a programming language would have. Agreement upon semantic meaning of keywords is out of scope for this specification and is intended to be determined by the user community. A recommended process for reserving well-known keywords is discussed in the “Keyword Proposal” section.

Layered Model

We can think of Margo notes as a layer within a stack, similar to the layers of the TCP/IP or OSI models.

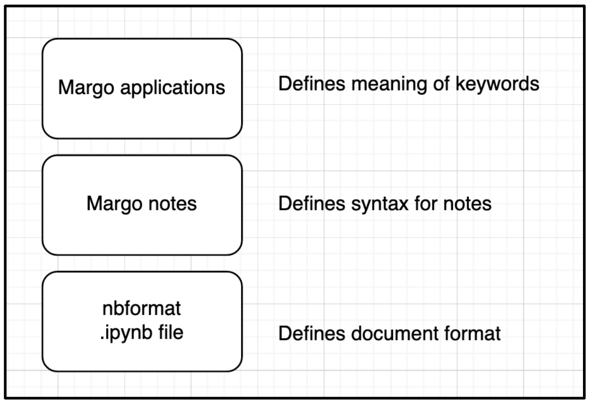


Figure 1. Layered Model of Margo

Lower layers in the stack are “unaware” of higher layers.

In this layered model, lower layers have no knowledge of higher layers. Standard notebook documents have no concept of Margo notes. Margo notes are embedded in comments and other syntax that are compatible but not meaningful to the nbformat layer. Similarly, Margo notes have no concept of the semantic meaning of the notes; it is up to the application layer to determine the meaning of Margo annotations.

Programming Language Interoperability

Because margin notes are designed to be embedded within code comments and markup cells, the practical application of Margo requires a language-agnostic core syntax and a language-specific embedding syntax for each possible code or markup language in which Margo notes may be embedded. The core syntax does not change no matter where it is embedded. The embedding syntax is not explicitly defined in this document as it must vary based on the programming language of code cells and the markup language of markup cells. Examples of Python and Markdown embedding syntaxes are described in this document to serve as a reference for implementation in other languages. When necessary for clarity, we will refer to either the "core syntax" or the "embedding syntax," but when we refer generally to "Margo" we refer to the complement of both syntaxes.

4.2. Versioning

This document specifies Margo version 0.0.1. This specification uses semantic versioning. Major version 0 is considered an unstable alpha specification and is subject to breaking changes.

4.3. Notation

This document uses both plain language and a BNF-like grammar notation to describe the syntax with as much clarity and precision as possible:

```
block ::= (statement endblock)*

endblock ::= "::"
```

The above example indicates that a block is composed of zero or more groupings of a statement followed by an endblock, which is composed of two colon characters.

Railroad diagrams are used sparingly to illustrate the grammar. Below is example that summarizes the block grammar listing above:

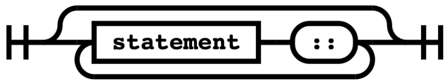


Figure 2. Sample Railroad Diagram

By following the railroad lines from left to right, we can see that a block is made up of zero or more statements, each terminated by a sequence of two colon characters. Note that rounded rectangles represent terminals and standard rectangles represent non-terminals.

4.4. Core Syntax

This section describes the core Margo syntax. Because Margo notes are embedded in comments or specially formed markup, a necessary embedding syntax is described in a later section.

Blocks

A Block of Margo is composed of zero or more Margo statements, each ending with an endblock.

```
block ::= (statement endblock)*

endblock ::= "::"
```

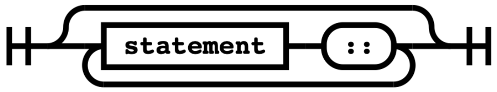


Figure 3. Margo Block

A Margo Block consists of zero or more statements, which each end in an endblock.

Statements

Each statement is either a directive or an assignment.

```
statement ::= directive
            | assignment
```

Directives

Directive statements consist of a name keyword.

```
directive ::= keyword
```

Below is an example of a directive statement:

```
ignore-cell ::
```

Keywords

Keywords are used in Margo where names are required, such as in the directive statements described in the preceding section and the assignment statements described in the next section. They are strings which may include only alphanumeric characters, period `(.)`, underscore `(_)` and hyphen `(-)`.

```
keyword ::= [a-zA-Z0-9\._-]+
```

Assignments

Assignment statements connect names to values. We have already seen names must be valid keywords. Values may be represented in two general ways: Margo Value Format (MVF) or an External Value Format (EVF). MVF assignments, described in the next section, is the default format. EVF allows the use of other externally defined formats, including JSON, YAML and raw text.

```
assignment ::= mvf_assignment
             | evf_assignment
```

Margo Value Format Expression

An MVF expression is a JSON array with two differences. First, the enclosing square brackets are omitted (they are unnecessary because MVF values are always arrays). Second, they may contain only scalar types, not collections — that is, they may contain numbers, strings, true, false, or null, and they may not contain arrays or objects. We summarize this in grammar notation and in the diagram below.

```
mvf_value ::= scalar ("," scalar)*
```

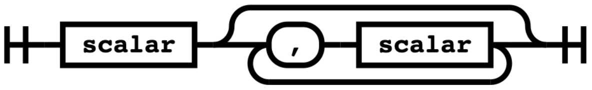


Figure 4. Margo Value Format Expression

The default format for specifying Margo values is Margo Value Format (MVF). It is a JSON array of scalars without enclosing square brackets. It may contain only scalars, not objects or arrays.

Margo Value Format Assignment

An MVF assignment consists of a keyword and an MVF expression, separated by a colon.

```
mvf_assignment ::= keyword ":" mvf_value
```

Below is an example assigning an array containing the string "population.csv" to the name "interface.input".

```
interface.input: "population.csv" ::
```

Scalars

Scalars in MVF are identical to scalar values in JSON.

```
scalar ::= "true"
         | "false"
         | "null"
         | <number>
         | <string>
```

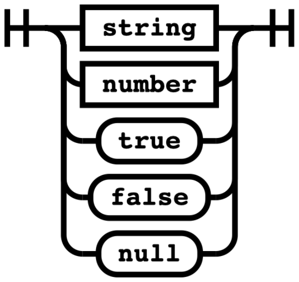


Figure 5. Margo Scalars

Margo scalar values are the same as JSON scalars, they include numbers, strings, "true," "false," and "null."

External Value Format Assignment

Assignments may alternatively be made using other notation schemes we collectively refer to as “External Value Formats.” An EVF assignment expression consists of a name, a format identifier, and a string that must be a valid instance of the specified format. The EVF identifiers reserved in Margo 0.0.1 are listed in the table below.

Table 1. Margo External Value Format identifiers

Format identifier	Format name
json	JavaScript Object Notation
yaml	YAML Ain’t Markup Language
raw	Plain, unstructured text

Format identifiers for Margo Extended Value Format assignments

EVF assignments consist of a name keyword, a format identifier keyword and a value string.

```
evf_assignment ::= keyword "[" evf_id "]" : <string>

evf_id ::= keyword
```

Below is an example of a YAML EVF assignment:

```
interface [yaml] : "
input: population.csv
output:
  - report.html
  - report.pdf
" ::
```

Combined Grammar Diagram

Now that we are familiar with the components of Margo, we can succinctly illustrate the entire grammar with the following railroad diagram.

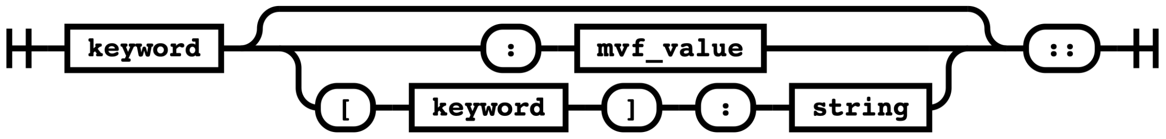


Figure 6. Complete Margo Diagram

Tracing the three railroad lines, we see that a statement always contains a keyword and terminates with an endblock. ***Top line:*** *No other tokens are present; this is a valid directive.* ***Middle line:*** *The keyword is followed by a colon and an MVF value; this is a valid MVF assignment.* ***Bottom line:*** *The initial keyword is followed by another keyword (the format identifier), enclosed in brackets, a colon, and a string — this is a valid EVF assignment.*

Core Syntax Grammar Listing

This listing combines all components of the Margo language described in the previous sections into a complete grammar:

```
block ::= (statement endblock)*

statement ::= directive
            | assignment

directive ::= keyword

assignment ::= mvf_assignment
             | evf_assignment

mvf_assignment ::= keyword ":" value ("," value)*
evf_assignment ::= keyword "[" evf_identifier "]"
: <string>

evf_identifier ::= keyword

keyword ::= [a-zA-Z0-9\._-]+

value ::= "true"
        | "false"
        | "null"
        | <number>
        | <string>

endblock ::= "::"
```

4.5. Embedding Syntax

Because Margo notes are intended to be embedded within code and markup cells of notebooks, they must be differentiated from the rest of a cell's content. This section describes an embedding syntax that wraps Margo notes within valid syntax in the cell's language. Because notebooks support different markup languages in markup cells and different programming languages in code cells, the embedding language will differ depending on the language of the cell. Defining an embedding syntax for every language is beyond the scope of this document. It is up to the parser implementation to define the embedding syntax. This section will discuss a general strategy for defining an embedding syntax and define a reference syntax for Markdown cells and Python.

Code Cells

Code cells should define Margo notes within specially formatted comments, such as a line comment character plus an endblock, to signify the start of each line of Margo.

Below is an example of a Python cell with a Margo assignment statement, split over two embedded lines because of its length.

```
# :: interface.input: "population.csv",
# ::                  "infections.csv" ::
```

Note that each line of Margo begins with `(# ::)` regardless of where statements end. The Margo interpreter removes these characters prior to evaluating the core Margo syntax described in the previous section.

Markup Cells

For markup languages that have a code element, markup cells should use this code element as a container for Margo notes. Further, for markup languages that support identifying the language of a code element, “margo” should be used as the language identifier. Code elements are preferred for containing Margo because they:

- are typically preformatted, rendered in a monospace font;
- may support syntax highlighting;
- and they are typically less obtrusive to the rendered version of a markup cell than other types of elements, such as headings or paragraphs.

Below is an example of markdown cell with Margo assignment:

~~~
```margo
# :: documents.cell: `infection_rates` ::
```

# Cell heading

The rest of the cell is not interpreted as Margo.
~~~

4.6. Margo Preamble

The “Margo preamble” refers to any Margo statements conforming to the appropriate embedding syntax that appear before non-comment code. The applications developed in subsequent chapters only interpret Margo code that is contained in the preamble. If Margo notes appear later in a cell, they will not be interpreted by these applications.

4.7. Keyword Proposal

Margo does not reserve any keywords, but for interoperability between Margo-aware applications, there must be reserved keywords with agreed-upon semantic meaning. There are two levels of interoperability:

- Merely reserving keywords avoids collisions between two different applications.
- Agreeing upon meaning makes it possible for two different applications to interpret the same notes.

The exact format for proposing and accepting keywords is not prescribed entirely in this section. Instead, it is the hope that these practices should be developed organically by the Margo user community, not by a pre-ordained set of inflexible rules.

Keywords that are not yet well-known should be prefixed with some unique identifier, such as a username or domain name, to prevent collisions with current or future well-known keywords. Proposing a well-known keyword is the act of asking the Margo community to reserve a keyword and agree upon its meaning. Agreeing upon its meaning does not mean agreeing upon how applications that support these keywords should behave.

In order to propose a well-known keyword, a Keyword Proposal document should be written, discussed, modified and ultimately accepted or rejected. The Python Enhancement Proposal should be seen as a model for this process (Coghlan et al., 2000). The proposal document should be sufficiently detailed to create a shared understanding. Eventually a standard format should be developed. A repository to organize keyword proposals and discussion around them is available online. See Appendix 1.

Chapter V.

Margo Parser

5.1. Overview

This chapter serves as both a software design document and a discussion about the development of Margo Parser. Margo Parser is a Python library for parsing a string of Margo code and providing an object-oriented API abstraction representing the code as a collection of zero or more statements. Source code for the Margo Parser implementation is included in this thesis submission and available online. See Appendix 1.

5.2. Requirements

1. Margo Parser must parse individual core Margo statements as well as blocks of multiple statements.
2. Margo Parser must provide convenience methods for extracting embedded Margo from Python code cells and from Markdown cells and may provide similar support for other languages .
3. Margo Parser must provide an object-oriented abstraction that requires the client to have no knowledge of Margo syntax or the internal tokenization.
4. Margo Parser must be a complete pip installable package.

5.3. Use Cases

The following section discusses use cases for the Margo Parser software library. The user in these cases is the client software.

Parse Statement

Given a string, determine if it is a valid Margo statement. If so, parse it and return an object representation of the statement.

Parse Margo Block

Given a string, determine if it is a valid block of Margo statements. If so, parse each statement and return an object representation of the block and its statements.

Parse Python Cell

Given a string, extract any embedded Margo blocks from a Python code cell and return an object representation of the block, discarding non-Margo Python code.

Parse Markdown Cell

Given a string, extract any embedded Margo blocks from a Markdown code cell and return an object representation of the block, discarding any non-Margo Markdown code.

5.4. Architecture

Margo Parser is composed of two internal components, an API and a tokenizer. The tokenizer does the bulk of the work: parsing strings of Margo code into a private, intermediate representation. The API abstracts this intermediate representation into an object-oriented representation.

Tokenizer

The tokenizer package exposes a single function, *tokenize*, which takes a block of Margo code as a string and returns a JSON representation or throws a *MargoParserException* if the string is not valid Margo.

The implementation uses the Lark Parser library (Shinan, 2021) to parse input strings and transform them into a JSON. The exposed *tokenize* function creates an instance of Lark with the grammar defined in the next section and transforms it into a JSON representation, which it returns to the caller.

Lark was selected because it is easy to use, actively maintained, well-documented and used by a large number of projects. Additionally, the syntax for defining grammars is very readable, resembling BNF, so the Lark grammar is intuitively self-documenting.

Lark Grammar

The following source code implements the Margo syntax specification in a Lark grammar. We can see that it resembles the BNF grammar presented in the Chapter IV.

```
block: (statement ENDBLOCK)*

statement:
    | mvf_assignment
    | evf_assignment
    | directive

evf_assignment: KEY "[" KEY "]" ":" QSTRING

mvf_assignment: KEY ":" value ("," value)* (",")*

directive: KEY

value:

    | "true" -> true
    | "false" -> false
    | "null" -> null
    | SIGNED_NUMBER -> number
    | QSTRING

ENDBLOCK: "::"
DQUOTE: "\""
SQUOTE: "'"
MULTILINE_TEXT: /.+?/s
DQSTRING: DQUOTE MULTILINE_TEXT DQUOTE
SQSTRING: SQUOTE MULTILINE_TEXT SQUOTE
QSTRING: DQSTRING | SQSTRING


MODULE_NAME: /[a-zA-Z]+[a-zA-Z0-9\_]*/

KEY: /[a-zA-Z0-9\._-]+/


%import common.WS

%import common.SIGNED_NUMBER

%ignore WS
```

API

The API exposes a Block class which can be instantiated with the Margo source code string as the only parameter. This class contains a collection of Statement objects that are either Directive or Assignment subclasses. Both directives and assignments have a name property, and Assignments have an additional value property.

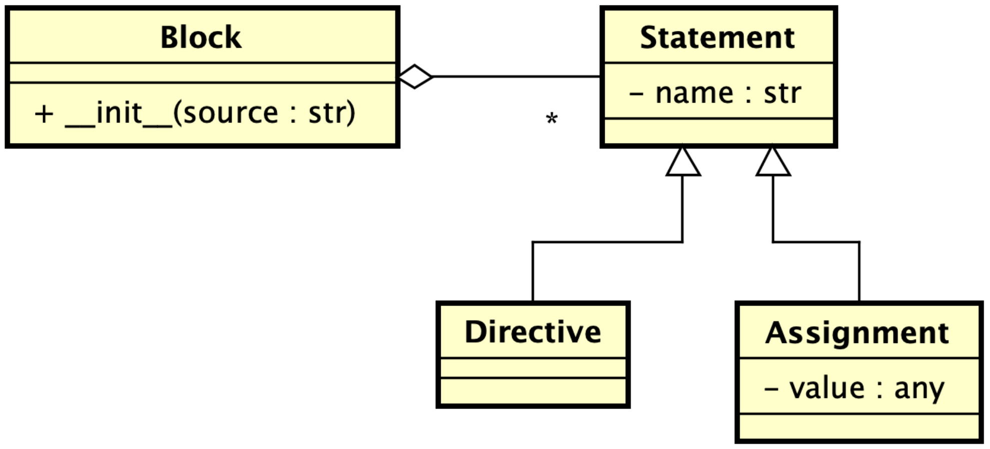


Figure 7. Margo Parser API Class Diagram

The API exposed to client software provides an object-oriented abstraction of the parsed Margo code that requires no knowledge of this implementation on the client's part.

Chapter VI.

Margo Loader

6.1. Overview

This chapter serves as both a software design document and a discussion about the development of Margo Loader. Margo Loader is a package that extends Python to enable importing Jupyter Notebooks. This library uses Margo notes to define preprocessor directives. Complete source code for Margo Loader is available online and submitted as part of this thesis submission. See Appendix 1.

6.2. Module Views of a Notebook

Most programming languages have some construct of modules that allow external code to be imported (Calliss, 1991). In C, a source code file referenced in an `#include` preprocessor directive is copied in place of the directive prior to compilation. Different languages may have different features that give developers more control over how modules are imported and exported. Some languages may require an explicit export statement to expose a named entity, while others may expose all entities except when it is marked as private. While the rules may vary from language to language, they generally work the same way — making code from one source file available in another.

As Calliss (1991) described, a module has a “client view” and a “supplier view.” The client view is “what the subsystem does” and the supplier view is “what the subsystem does and how it does it.” A client view may also be called an interface — it is what is exposed outside the module.

If we think of a notebook's supplier view, we need to expand the definition to: "what the subsystem does and how it does it and a bunch of other scratch code." That's because notebooks are not conventionally written as modules, and so they may contain code that is well-suited for reuse alongside code that is not, as we discussed in Chapter I, Section 5. For example, a function to return five RGB color values to summarize an input image is reusable, but an invocation of that function on a specific image file serves no purpose outside of a notebook context.

This complicates the question of a notebook's client view. The most intuitive approach would be to concatenate all of its cells. Applications that import notebooks, such as nbimport, or execute them as Python scripts, such as nbconvert and Papermill, infer the notebook's client view this way. But that leaves us importing "scratch" code along with the reusable module code we want.

The solution has so far been behavioral, not technological: Notebooks written for use with these tools are typically written more like modules and less like notebooks. They tend to contain less code that might make a useful demonstration in a notebook context because the author intends them to be useful in a module context. This solution is sensible, but it begins to curb some of the characteristic features that make us want to use notebooks in the first place. Including example invocations of a function in the same document where it is defined can be an effective way to document how it works. So why give that up? We propose using a Margo keyword "ignore-cell" to skip a cell during import. This is intended to allow notebook modules to be written with all of the flourish of standalone notebooks but still give the author enough control to make them useful as modules.

6.3. Requirements

1. Margo Import must be compatible with Python 3.4 or later importlib API and may be compatible with earlier versions.
2. Margo Import must import Jupyter Notebooks.
3. Margo Import must support importing notebooks using Python’s built-in import statement and support `from ... import` syntax.
4. Margo Import must be a complete pip installable package.
5. Resolving file paths to notebook modules must follow the same rules as resolving paths to *.py* files, except with files ending in *.ipynb*.
6. Margo import must support directives to ignore cells and organize cells into virtual submodules.

6.4. Use Cases

Use cases fall into two general categories: client and module. The client notebook imports the module notebook.

Import Use Cases

The following use cases describe how Margo Loader is used to import Jupyter Notebooks.

Import a Notebook Module

The Margo Loader library is first imported with:

```
import margo_loader
```

Then a notebook may be imported using Python’s standard import syntax:

```
import NotebookName
```

Import Virtual Submodule

When importing a Jupyter Notebook, cells that belong to *VirtualSubmoduleA* may be imported with:

```
from NotebookName import VirtualSubmoduleA
```

Export Use Cases

The following use cases describe how to prepare a notebook to be imported by Margo Loader.

Defining the Module's __docstring__

In Python, the __*docstring*__ of a module is a documentation string that is displayed by the *help* function. If the first cell in a notebook is a Markdown cell its contents will be stored in the notebook module's __*docstring*__ during import.

6.5. Keyword Proposal

This section proposes that the keywords discussed in this chapter be reserved.

The "ignore-cell" Directive

This keyword indicates that a cell should be excluded from a Notebook's modular view, or any other view of a notebook outside a conventional notebook authoring tool. This keyword must always be used as a directive and may never be used as the name of a Margo assignment. Below is an example use of this keyword:

```
# :: ignore-cell ::
# This cell will not be included in the
# module representation of this notebook

say_hello(to="Mars")
```

The "submodule" Assignment

This keyword identifies one or more submodules in which to include a code cell. This keyword must always be an assignment and may never be a directive. The value assigned must only contain strings that conform to the module naming convention of the Notebook kernel.

In the following example we define two submodules:

```
# greetings.ipynb - cell 1
# :: submodule: "grumpy" ::

def say_hello(to="world"):
    return f"Oh, uhh, hi {to}..."


# greetings.ipynb - cell 2
# :: submodule: "nice" ::

def say_hello(to="world"):
  return f"Hello, {to}! Nice to see you."
```

It is up to the application to determine whether and how to behave upon encountering this keyword. In the case of Margo Parser, the above code results in creating two submodules of *greetings* called *grumpy* and *nice*, each with their respective *say_hello* function. This is illustrated below:

```
>>> import margo_loader
>>> from test_notebooks.greetings import nice
>>> from test_notebooks.greetings import grumpy
>>> nice.say_hello()

'Hello, world! Nice to see you.'

>>> grumpy.say_hello()

'Oh, uhh, hi world...'
```

6.6. Architecture

Python supports extending its import machinery, and Margo Loader leverages this API. The general approach was demonstrated in Python by the Jupyter Project (Jupyter Team, 2015) using an earlier version of this API and has been adapted in code libraries (Sturm, 2019). The module loading machinery in Python is well-defined in the standard library (Python Software Foundation, 2021). When an import statement is evaluated, Python must perform two steps: Finding the module in the file system and loading it. To add custom import behavior, Margo Loader implements two abstract classes from importlib: MetaPathFinder and Loader.

NotebookFinder

Finding a module is achieved by creating NotebookFinder as a subclass of importlib.abc (Python Software Foundation, 2021), which must implement *find_module* and the newer *find_spec* method. Both methods are responsible for taking a Python path and a module name and resolving it to a file on the path if it exists. Finding a Jupyter Notebook module on the file system is the same as finding a standard Python module except the file has the extension ".ipynb" instead of ".py". If underscores are used in a module name but no corresponding file is found, NotebookFinder will then look for a file with spaces in the name.

NotebookLoader

NotebookLoader is responsible for deserializing the notebook file into an array of cells and preprocessing it based on Margo Notes. Loading follows this procedure:

1. Load the cell contents of the notebook using the nbformat library.
2. Create the module in *sys.modules*.
3. If the first cell is Markdown, set *module.__docstring__* to the cell's contents
4. For each cell in the notebook:
    a. Call either the Markdown or code cell preamble parser from the *margo_parser* library.
    b. If the preamble contains an *ignore-cell* directive, break
    c. For each submodule assignment in the preamble:
        i. If a submodule with the given name does not exist in module, create submodule
        ii. *exec* cell in submodule
    d. *exec* cell in module

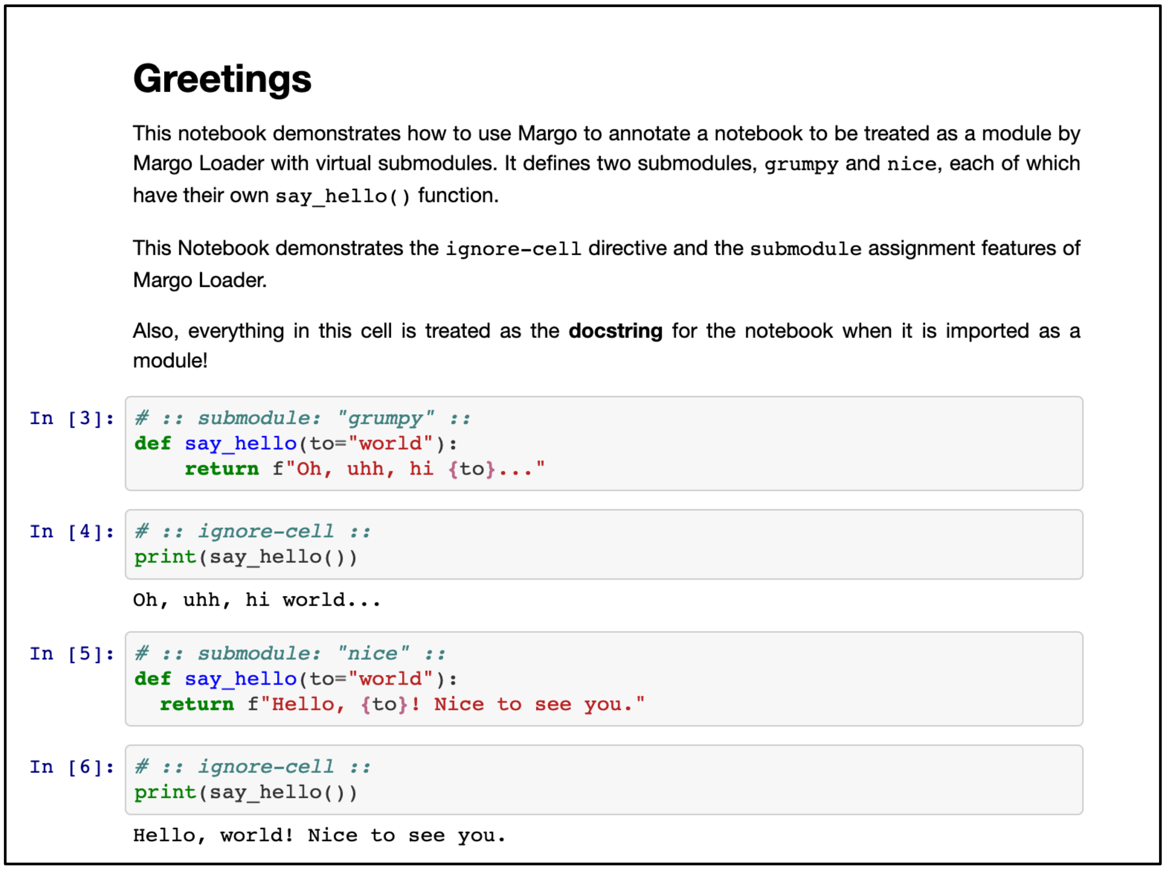
Step 4.d. is under consideration, regarding whether a submodule cell should be executed in both the top-level module and in the submodules to which it belongs. It's not clear without further community input and testing which option is most intuitive or useful. The decision was made for this version to execute in the top-level always, plus submodules.

6.7. Results and Implementation

This section demonstrates Margo Loader in use across two notebooks — a module notebook and a client notebook. To access these notebooks, see Appendix 1.

Module notebook

This screenshot demonstrates a notebook enhanced with Margo annotations.



Greetings

This notebook demonstrates how to use Margo to annotate a notebook to be treated as a module by Margo Loader with virtual submodules. It defines two submodules, `grumpy` and `nice`, each of which have their own `say_hello()` function.

This Notebook demonstrates the `ignore-cell` directive and the `submodule` assignment features of Margo Loader.

Also, everything in this cell is treated as the **docstring** for the notebook when it is imported as a module!

In [3]:
```
# :: submodule: "grumpy" ::
def say_hello(to="world"):
    return f"Oh, uhh, hi {to}..."
```

In [4]:
```
# :: ignore-cell ::
print(say_hello())
```
```
Oh, uhh, hi world...
```

In [5]:
```
# :: submodule: "nice" ::
def say_hello(to="world"):
  return f"Hello, {to}! Nice to see you."
```

In [6]:
```
# :: ignore-cell ::
print(say_hello())
```
```
Hello, world! Nice to see you.
```

Figure 8. Example of a Notebook Module

This is a screenshot of the complete greetings.ipynb notebook included in the test_notebooks directory of the margo-loader repository, as rendered by GitHub.com.

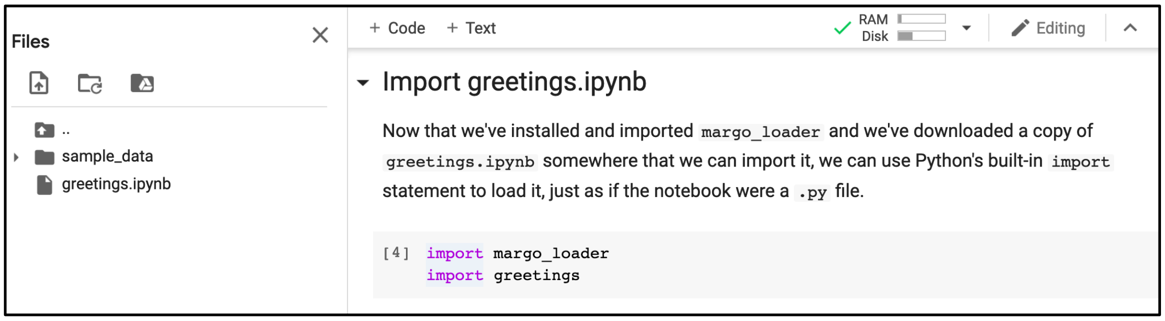
This notebook contains five cells. Let's examine how this notebook is processed and loaded by Margo Loader.

1. Cell 1 (Markdown): This cell's contents are assigned to the module's *__docstring__* property because it is the first cell and it is Markdown.

2. Cell 2 (Code): This cell’s preamble assigns the value “grumpy” to the name “submodule”. The name “submodule” is a keyword reserved by Margo Loader. It indicates the cell should be included in a submodule of “greetings” named “grumpy.” The code contents of this cell define a function called *say_hello*. In the notebook context, the margin notes have no effect on execution, so this function is globally available to the notebook. However, when it is imported, a submodule “grumpy” will be created during import and this function will be scoped within it.
3. Cell 3 (Code): This cell’s preamble contains an “ignore-cell” directive. This “ignore-cell” keyword is also reserved by Margo Loader. It means that this cell will be dropped during preprocessing and prior to executing.
4. Cell 4 (Code): This cell is like Cell 2, except its preamble indicates that it belongs to a new submodule called “nice.” Therefore, the *say_hello* function defined in this cell is accessible within the “greetings” submodule “nice.”
5. Cell 5 (Code): This cell is identical to Cell 3, yet the output is different because the *say_hello* function in Cell 4 replaced the existing *say_hello* function in the notebook context. We will see in the next section when this notebook is imported that both *say_hello* functions are accessible to the client because they are each contained in their respective submodules.

Client Notebook

The following figures demonstrate how a client notebook can import the notebook defined in the previous section. The figures below are screenshots from a Google Colab notebook.



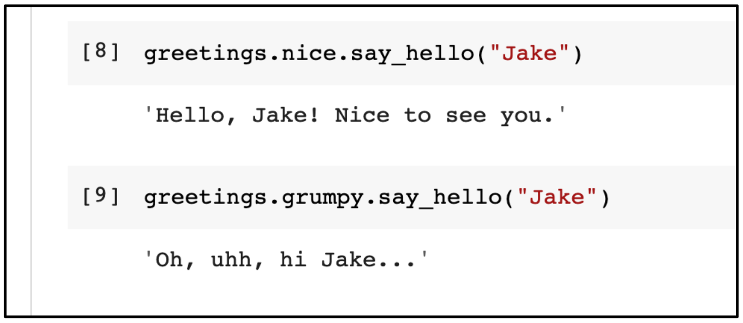
Import greetings.ipynb

Now that we've installed and imported `margo_loader` and we've downloaded a copy of `greetings.ipynb` somewhere that we can import it, we can use Python's built-in `import` statement to load it, just as if the notebook were a `.py` file.

```
[4] import margo_loader
    import greetings
```

Figure 9. Importing a Notebook into a Notebook

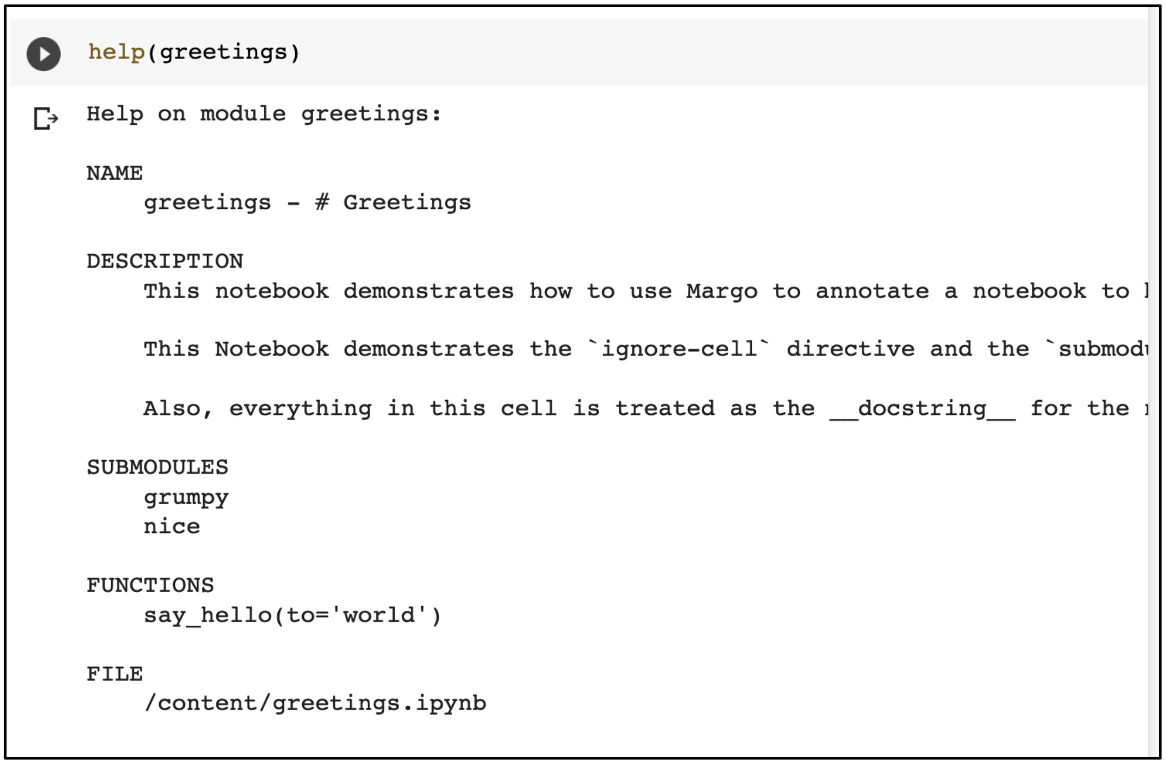
First, we import margo_loader and the greetings.ipynb file (seen in the files list).



```
[8] greetings.nice.say_hello("Jake")

    'Hello, Jake! Nice to see you.'

[9] greetings.grumpy.say_hello("Jake")

    'Oh, uhh, hi Jake...'
```

Figure 10. Accessing a Notebook's Virtual Submodules

We are able to execute greetings.nice.say_hello() and greetings.grumpy.say_hello(), which we can see produce different results with the same input.



```
help(greetings)
```

```
Help on module greetings:

NAME
    greetings - # Greetings

DESCRIPTION
    This notebook demonstrates how to use Margo to annotate a notebook to l

    This Notebook demonstrates the `ignore-cell` directive and the `submodu

    Also, everything in this cell is treated as the __docstring__ for the r

SUBMODULES
    grumpy
    nice

FUNCTIONS
    say_hello(to='world')

FILE
    /content/greetings.ipynb
```

Figure 11. __docstring__ Defined by Markdown Cell

Because the first cell in greetings.ipynb is a Markdown cell, it is assigned to the module's __docstring__, and we can thus view it with the help() function.

Future work

Additional Inclusion/Exclusion Directives

There may be many other directives in addition to the two we have implemented in this version. For example, a directive "end-module" might tell the importer to ignore all the subsequent cells in a notebook. So far, we've considered directives that assume a default behavior of including all cells unless otherwise specified. However, a directive could be used to change this behavior, excluding all cells by default except those which contain an "export-cell" directive.

Notebook Type

Another directive called “notebook.type” could indicate whether a notebook is intended to be a “module,” a “script,” a “notebook,” or some combination of the three. This directive could be loosely or strictly enforced by Margo Parser. A “non-module” label could also be used to explicitly prevent a notebook file from being imported as a module.

Chapter VII.

Margo Tool

7.1. Overview

This chapter serves as both a software design document and a discussion about the development of Margo Tool. Margo Tool is a command-line application that extracts Margo notes and cell contents from Notebooks. Margo Tool is implemented in source code within the Margo Loader software package. Source code submitted with this thesis is available online. See Appendix 1.

7.2. Use Cases

Margo Tool is generally useful for defining ad hoc interfaces into notebooks. We consider two use cases where this can be used to integrate notebooks with external tools.

Dependency Management

Python conventionally uses a file called “requirements.txt” to specify package dependencies. It consists of a newline-delimited plain text file. Using Margo notes, we can define a raw text assignment named “requirements.txt” that contains dependencies for the particular notebook. Being able to extract this value with Margo Tool means we can extract a requirements file for a notebook. With a folder full of notebooks, these can be concatenated into one requirements file just prior to installation.

Makefile Generation

Notebooks may be executed by third-party automation systems, such as GNU Make. Make is a system for generating output files based on rules. Rules specify the inputs (prerequisite files), outputs (targets) and recipes (commands executed to produce outputs). Rules to execute Notebook files can be generated if notebooks store a list of the inputs and outputs in Margo notes.

7.3. Keyword Proposal

This section proposes that the keywords discussed in this chapter be reserved.

The "requirements.txt" Assignment

This keyword indicates a notebook's Python dependencies. This assignment must be an assignment in the Margo EVF (raw) format, and the value string must be a valid pip requirements file or subset of one. This keyword must never be a Margo directive. Below is an example use:

```
# :: requirements.txt [raw]: "
# :: package-1 == 1.2.3
# :: package-2 == 3.2.1
# :: package-n
# :: " ::
```

The "interface.input" Assignment

This keyword is used to list any local files that are required to execute a notebook. This keyword must always be an assignment in the Margo MVF format. This keyword must never be a Margo directive.

```
# :: interface.input: "population.csv"
```

The “interface.output” Assignment

This keyword is used to list any local files that are produced during execution of a notebook. This keyword must always be an assignment in the Margo MVF format. This keyword must never be a Margo directive.

```
# :: interface.output: "report.html",
# ::                    "report.pdf" ::
```

7.4. Architecture

Margo Tool consists of a command processor and a collection of subcommands that process notebook files and generate specified outputs. The two subcommands we will implement are “extract” and “makefile.”

The extract subcommand iterates through the cells of a notebook and prints Margo assignments. This will be used to extract a notebook’s dependencies stored in the Margo assignment “requirements.txt”.

The makefile subcommand is more specialized. It generates a Makefile from a notebook based on the values of the “interface.input” and “interface.output” Margo assignments.

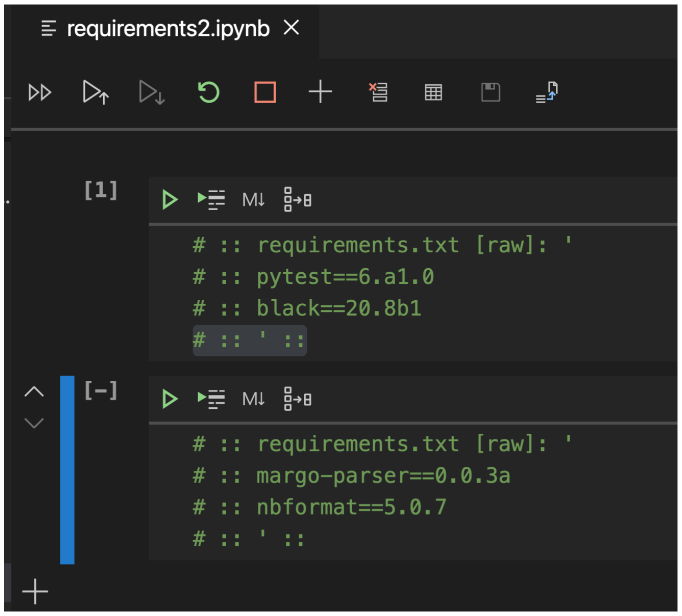
7.5. Results and Implementation

The mechanics of Margo Tool are relatively trivial, but the implications are significant. Margo Tool extracts values of Margo notes from Jupyter Notebooks and prints them in some format to standard output. The power of this application comes in when the outputs from Margo Tool are valid inputs for other tools — effectively allowing notebook authors to create interfaces into Jupyter Notebooks.

Margo Tool implements both a general purpose “extract” subcommand and a more specialized “makefile” subcommand to generate a Makefile from a notebook document. This section demonstrates both subcommands in use.

Generating “requirements.txt”

Margo Tool’s extract subcommand prints Margo assignment values to standard output. One application is integrating a notebook with Python’s package installer so that a notebook’s dependencies may be automatically installed. To accomplish this, we assign to the name “requirements.txt” a raw text string conforming to the requirements file format used by the Python package installer pip.



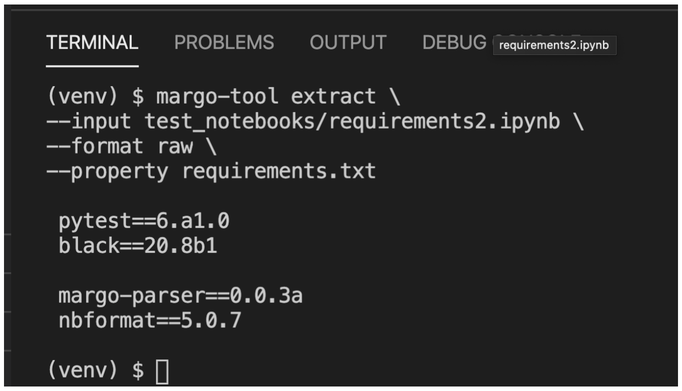
```
# :: requirements.txt [raw]: '
# :: pytest==6.a1.0
# :: black==20.8b1
# :: ' ::
```

```
# :: requirements.txt [raw]: '
# :: margo-parser==0.0.3a
# :: nbformat==5.0.7
# :: ' ::
```

Figure 12. Enumerating Dependencies in Margo

A notebook that defines its dependencies by assigning a raw text value to the name requirements.txt. Note that this requirements.txt assignment is made in two cells. Margo Tool concatenates these values. Screenshot from Visual Studio Code.

With these dependencies stored in Margo notes, we can now use Margo Tool’s extract subcommand to extract the value of “requirement.txt” and print it to standard output.



```
(venv) $ margo-tool extract \
--input test_notebooks/requirements2.ipynb \
--format raw \
--property requirements.txt

 pytest==6.a1.0
 black==20.8b1

 margo-parser==0.0.3a
 nbformat==5.0.7

(venv) $ 
```

Figure 13. Generating requirements.txt with Margo

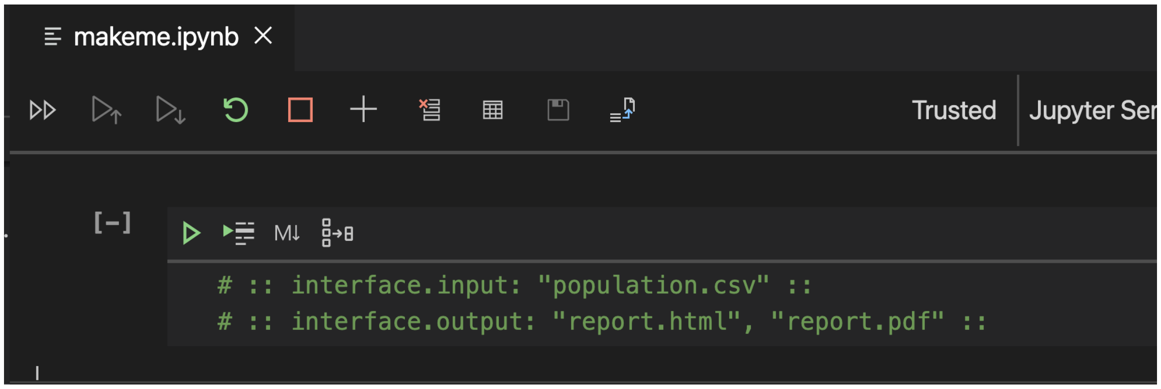
Using the "margo-tool" command to generate a requirements.txt document for a notebook that may be used with a package installer such as pip. This is accomplished with the subcommand "extract" to extract all raw-formatted values for the property named "requirements.txt." Screenshot from Visual Studio Code.

With a folder full of notebooks, one could use this scheme to generate a combined requirements file, or even to create isolated environments for each notebook.

Generating a Makefile

Another use case for Margo Tool is integrating Margo with an automation tool that can orchestrate the execution of component tasks, such as GNU Make. In order to integrate with Make, a notebook must define the "input" files that must be present on the file system for the notebook to run and the "output" files that are generated by executing a notebook.

First, we define the inputs and outputs in Margo. In the previous section on generating requirements.txt, we chose to use Margo's EVF assignment style to format our assignment as raw text. In this case, we choose to use Margo's MVF assignment style because it is compact and expressive. Because names may contain dots, we use a naming convention that implies an object called "interface" with two attributes ("input" and "output") when in fact these are actually two independent Margo assignments.

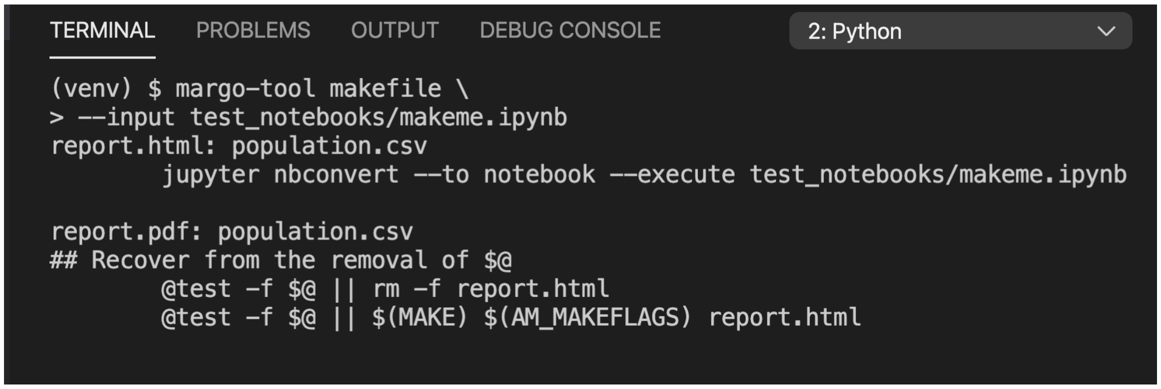


```
# :: interface.input: "population.csv" ::
# :: interface.output: "report.html", "report.pdf" ::
```

Figure 14. Defining a Task Interface in Margo

The above Margo note indicates to Margo Tool that the notebook requires a file "population.csv" to be present prior to execution, and that executing the notebook will generate the files "report.html" and "report.pdf." Screenshot from Visual Studio Code.

The figure below shows how the above notebook is interpreted by Margo Tool's "makefile" subcommand.



```
(venv) $ margo-tool makefile \
> --input test_notebooks/makeme.ipynb
report.html: population.csv
        jupyter nbconvert --to notebook --execute test_notebooks/makeme.ipynb

report.pdf: population.csv
## Recover from the removal of $@
        @test -f $@ || rm -f report.html
        @test -f $@ || $(MAKE) $(AM_MAKEFLAGS) report.html
```

Figure 15. Generating a Makefile

The "margo-tool" command's "makefile" subcommand generates two Make rules for the notebook features in the preceding figure because the document specifies that it produces two outputs: "report.html" and "report.pdf." Screenshots from Visual Studio Code.

Chapter VIII.

Margo Editor

8.1 Overview

This chapter serves as both a software design document and a discussion about the development of Margo Editor. Margo Editor is a user interface prototype that demonstrates how Margo can be used to build applications that implement features that are not supported by the notebook document format.

The feature we implement in this chapter is hierarchical cell relationships — a scheme of linking cells in order to support richer editing applications. Examples of relationship labels might be “tests,” as in “Cell B tests Cell A” or “documents,” as in “Cell D documents Cell C.” We model these relationships in Margo notes and build a user interface on top of this model.

The user interface developed in this chapter is not meant to be complete, realistic production software. It is meant to demonstrate how novel user interface features can be built leveraging the arbitrary extensions of the notebook format made possible by Margo.

Source code submitted with this thesis and a live demo of Margo Editor are available online. See Appendix 1.

8.2 Requirements

1. Cell relationships specified in the editor must be managed internally and serialized with automatically generated Margo notes, which can be deserialized when the notebook is loaded by the editor.

2. Documents that are generated by Margo Editor must be valid Jupyter Notebook documents compatible with software that does not support cell relationships.

8.3. Use Cases

The user interface supports the following user interactions.

- Add cell
- Toggle cell type
- Delete cell
- Run cell
- Save notebook
- Open notebook
- Create child cell
- Rename cell relationship
- Move cell up
- Move cell down

8.4. Cell Relationships

In a notebook document, cells are represented by an array. Cell relationships might ideally be represented with a network data structure, in which cells are nodes and relationships between cells are edges. However, one hard constraint of our design is that in order to preserve backward compatibility we cannot modify the underlying nbformat document — we can only use Margo to layer our features on top. Our approach is to use Margo to encode relationships, using the following general syntax:

```
rel.<relationship-label> : <cell-id> ::
```

A specific example of this general syntax is:

```
rel.tests : "define-hello-function>" ::
```

This example makes reference to a `<cell-id>`, which corresponds to a value defined in the receiving cell. The general syntax for defining a cell ID is:

```
cell-id: <string> ::
```

A specific example of this general syntax is:

```
cell-id: "say_hello"
```

8.5. Cell UI Representation

In this section we develop an approach to representing cells with hierarchical relationships to one another. A conventional notebook authoring UI represents cells in a linear fashion, as illustrated below.

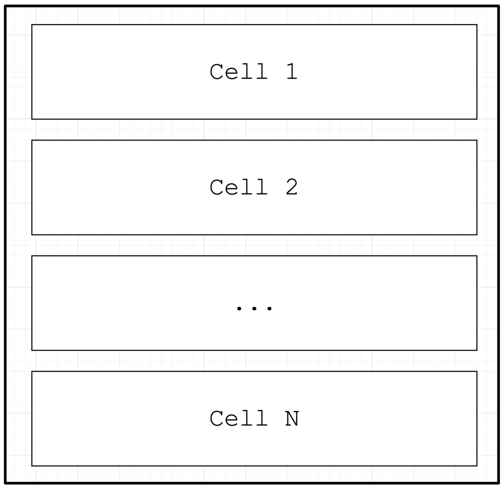


Figure 16. Flat Representation of Cells

Conventional Jupyter Notebook authoring environments display cells in a single vertical column with no notion of hierarchy.

We can see the ways this design fails to represent the relationships described in the previous section. For example, if Cell 2 were labeled as a "test" of Cell 1, and Cell 1 were a "documentation" for Cell N (for N > 2), in what order should cells be listed? Is a vertical column of cells even an appropriate representation of these relationships? This is a challenging question, which could yield interesting UI solutions. We will sidestep a more thorough exploration of these questions. For our proof of concept, we will impose some limitations that eliminate these ambiguities:

1. All cells are either parents or child cells

2. A child has exactly one parent cell
3. A parent cell has zero or more child cells
4. Child cells may not have child cells
5. A child cell must immediately follow its parent or a sibling (one of its parent's children)

A user interface that preserves this linear listing of cells can introduce the notion of hierarchy by using size and offset of cells:

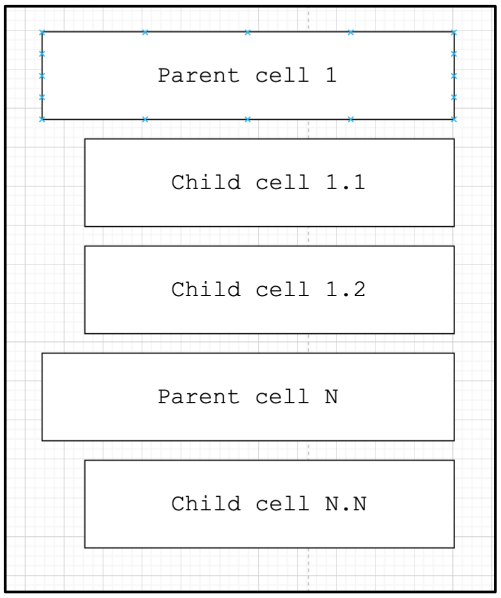
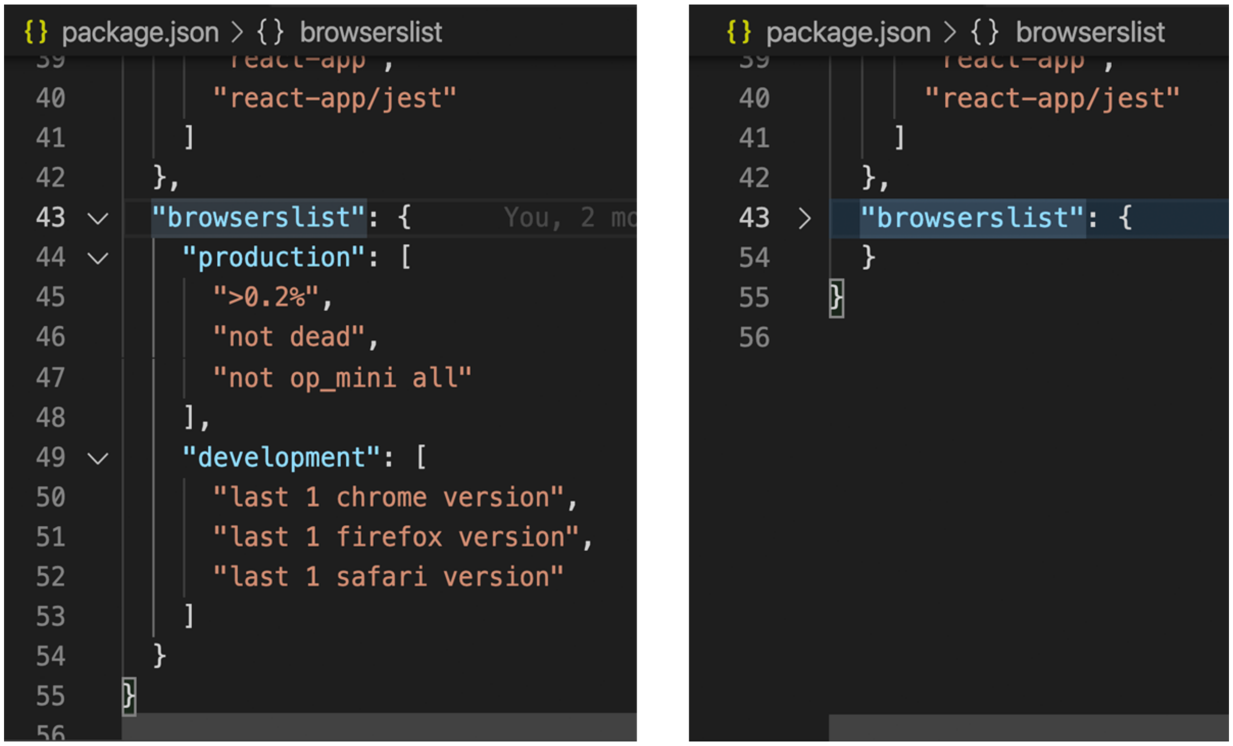


Figure 17. Hierarchical Representation of Cells

Cell hierarchy can be represented with minimal alteration to the flat representation.

8.6. Existing Design Language

Existing IDEs already have a well-established design language for how to represent the hierarchical relationship between lines of code. We will adapt this design language to represent hierarchical relationships between cells, rather than between lines of code.



```
{} package.json > {} browserslist
39          "react-app",
40          "react-app/jest"
41        ]
42      },
43 ∨    "browserslist": {        You, 2 mo
44 ∨      "production": [
45          ">0.2%",
46          "not dead",
47          "not op_mini all"
48        ],
49 ∨      "development": [
50          "last 1 chrome version",
51          "last 1 firefox version",
52          "last 1 safari version"
53        ]
54      }
55    }
56
```

```
{} package.json > {} browserslist
39          "react-app",
40          "react-app/jest"
41        ]
42      },
43 >    "browserslist": {
54      }
55    }
56
```

Figure 18. Code Collapsing and Line Hierarchy

The same JSON object is represented in Visual Studio Code both fully expanded, at left, and then with the "browserlist" object, which spans lines 43 to 54, collapsed.

There are two components of this design illustrated in the above figure that will translate well to the hierarchical display of cells. First, we see vertical through-lines at left that correspond to different levels in the hierarchy. These lines are arranged horizontally, with the number of through-lines at a particular line of code indicating how deep it is within the hierarchy. Second, a chevron icon marks what we will call "parent" lines that have one or more "child" lines. The chevron direction indicates the collapsed state of a parent line: downward when expanded and rightward when collapsed.

8.7. Wireframe

In this section we adapt the design components from the previous section to represent hierarchical relationships between cells rather than between lines of code.

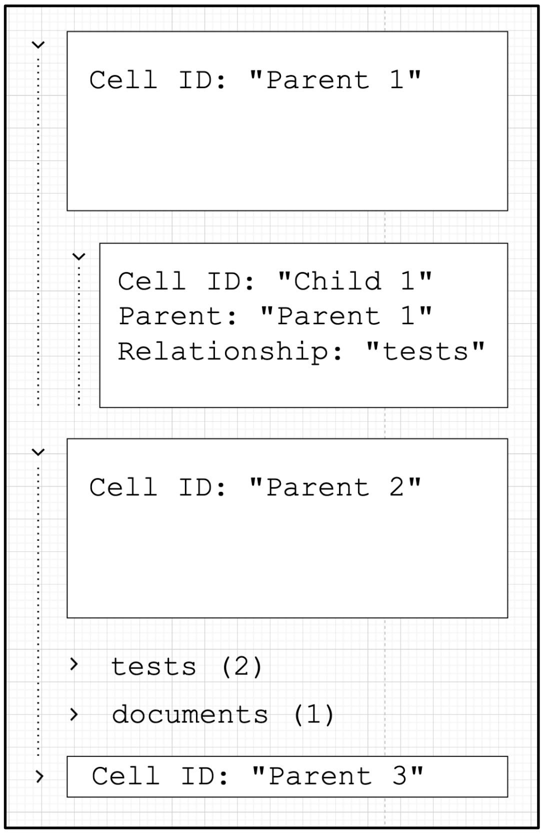


Figure 19. Margo Editor UI Wireframe

The high-level wireframe above illustrates a notebook with seven cells — three parent cells and four child cells. In order they are: An expanded parent cell; an expanded child cell; an expanded parent cell; two collapsed child cells; one collapsed child cell; one collapsed parent cell.

8.8. Technology Selection

Margo Editor is implemented as a web application so that we can demonstrate the interface without requiring installation. Additionally, there has been a proliferation of technology stacks based on web standards, including for the development of native applications, such as React Native and Electron, which make web-based solutions very versatile. This implementation is suitable for integration with other tools, including in Microsoft Visual Studio Code, via its WebView API.

Each cell's editor will be implemented using Monaco Editor. This library was selected because it is one of few mature, feature-rich, web-based code editors. It is a core component of Visual Studio Code, and syntax highlighting is implemented for a vast number of programming languages.

Typescript is used instead of plain JavaScript in order to eliminate classes of programming errors and improve the self-documentation of code.

8.9. Keyword Proposal

This section proposes that the keywords discussed in this chapter be reserved.

The "cell.id" Assignment

This keyword is used to define a cell's ID, independent of the underlying Jupyter Notebook document. This keyword must be used for a Margo assignment using the MVF format to assign a list of exactly one value, which must be a string representing the cell's ID. The ID may be any valid string. This keyword may never be used with a Margo directive. An example is:

```
cell-id: "say_hello" ::
```

The “rel.<relationship-label>” Assignment

This keyword is used to specify a relationship between the cell it is included in and another cell in the notebook. This keyword must be used with an MVF assignment with an array of length one, containing a string that is a cell ID corresponding to another cell in the notebook. An example is:

```
rel.tests : "define-hello-function" ::
```

Additional Uses

Margo Editor uses these relationship keywords to add functionality to a notebook authoring user interface. However, they may be useful in other contexts. The “rel.tests” relationship we have described could be used with a testing suite to automatically run all tests in a notebook. A document generation tool could use a relationship label such as “rel.documents.api” to generate API docs. The label “rel.todo” could indicate that a Markdown cell contains a TODO list.

Validity of State

Because of their interdependence, these keywords raise the possibility for invalid or incomplete states, such as a relationship assignment pointing to a cell that does not exist. Applications should maintain a valid state and handle errors related to invalid state. It is not guaranteed how cell relationships will be interpreted. Cycles and hierarchies with heights greater than two are possible to represent using this notation, but Margo Editor ignores cycles and hierarchies with heights greater than two.

8.9. Architecture

The application architecture consists of two core components, a notebook data model and a front-end.

Notebook Model API

The data model is responsible for serialization, deserialization, and validation of a notebook. It provides an API for so-called CRUD operations to create, read, update and delete cells. This functionality is implemented for standard Jupyter Notebook documents by the JupyterLab project in the *@jupyterlab/cells* and *@jupyterlab/notebook* packages. To support Margo notes, Margo Editor uses a set of wrapper functions to serialize and deserialize Margo Notes and pass the broader notebook management to the two JupyterLab packages.

Front-end Application

The front-end renders the notebook and passes user interactions as messages to the model. The front-end is developed using the React framework.

8.9. Results and Implementation

This section demonstrates the implemented user interface prototype. Source code for this implementation and a live demo are available and submitted as a part of this thesis. See Appendix 1.

Authoring a Notebook

This section walks through the basic actions of creating a notebook with cell relationships in a series of screenshots of the completed application running in the Firefox web browser.

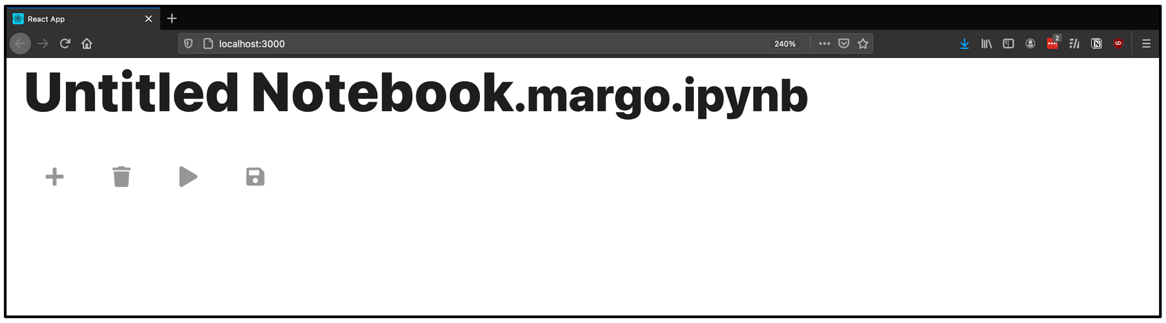


Figure 20. Margo Editor, Initial State

The initial state of the application. The editable document title is at the top, followed by a notebook-level control bar.

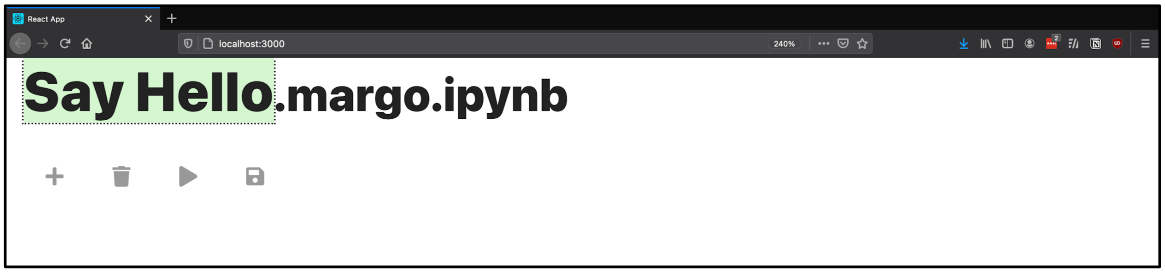


Figure 21. Margo Editor, Rename Document

Type in the name field to edit the notebook document name.

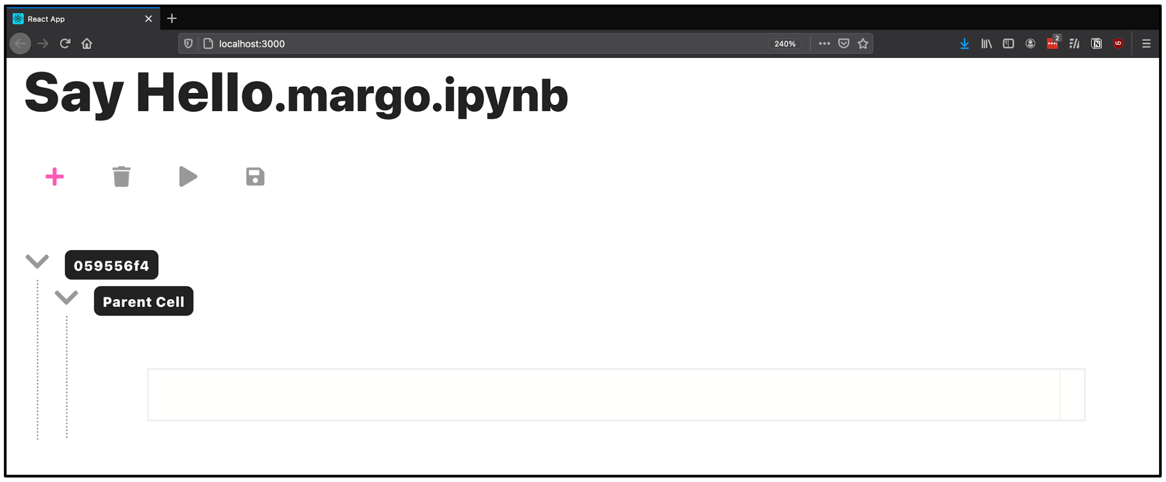
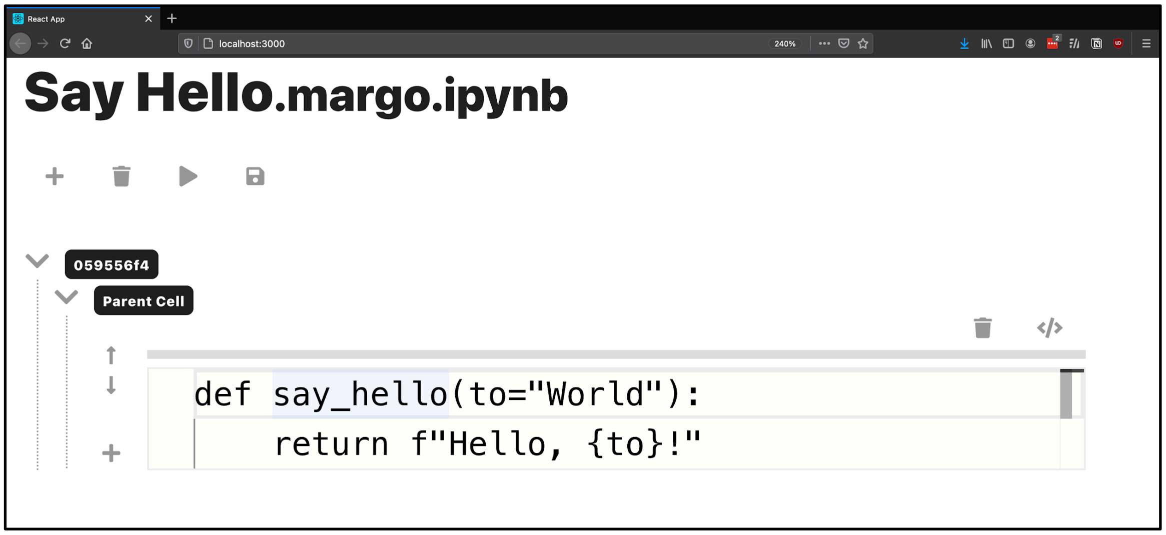


Figure 22. Margo Editor, Add a Cell

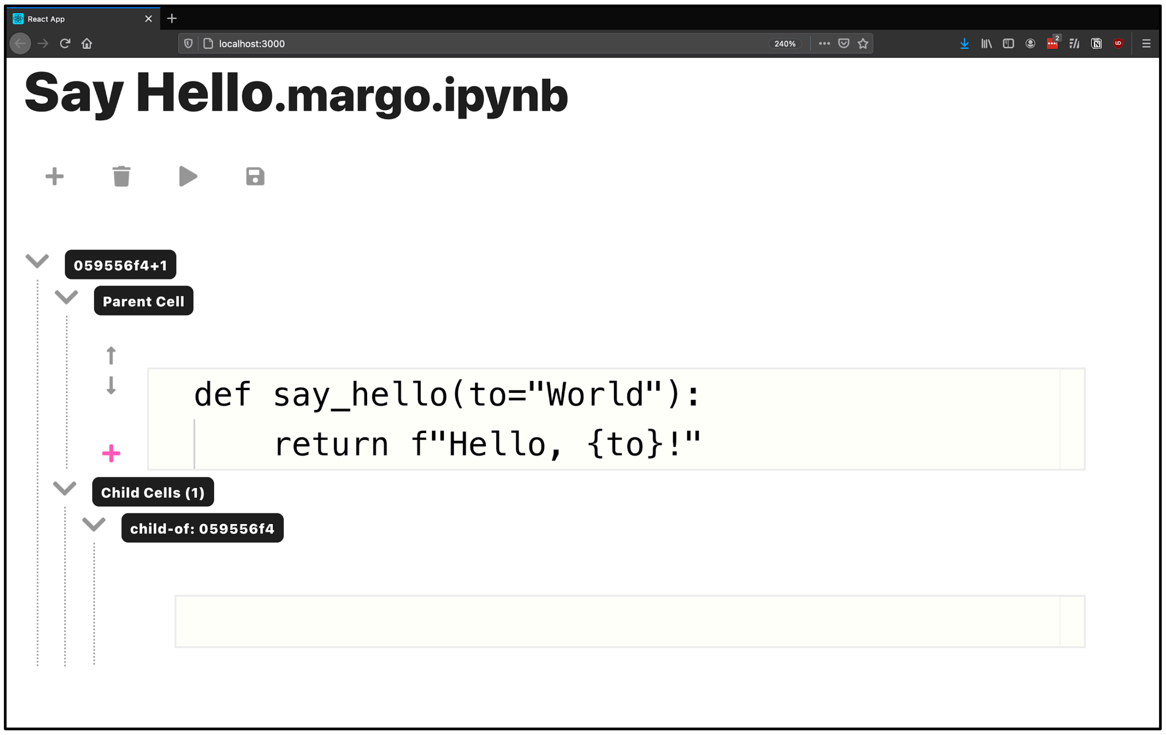
A cell can be appended to the notebook with the notebook-level "add cell" plus sign icon. All cells added this way are parents.



```
def say_hello(to="World"):
    return f"Hello, {to}!"
```

Figure 23. Margo Editor, Enter Code

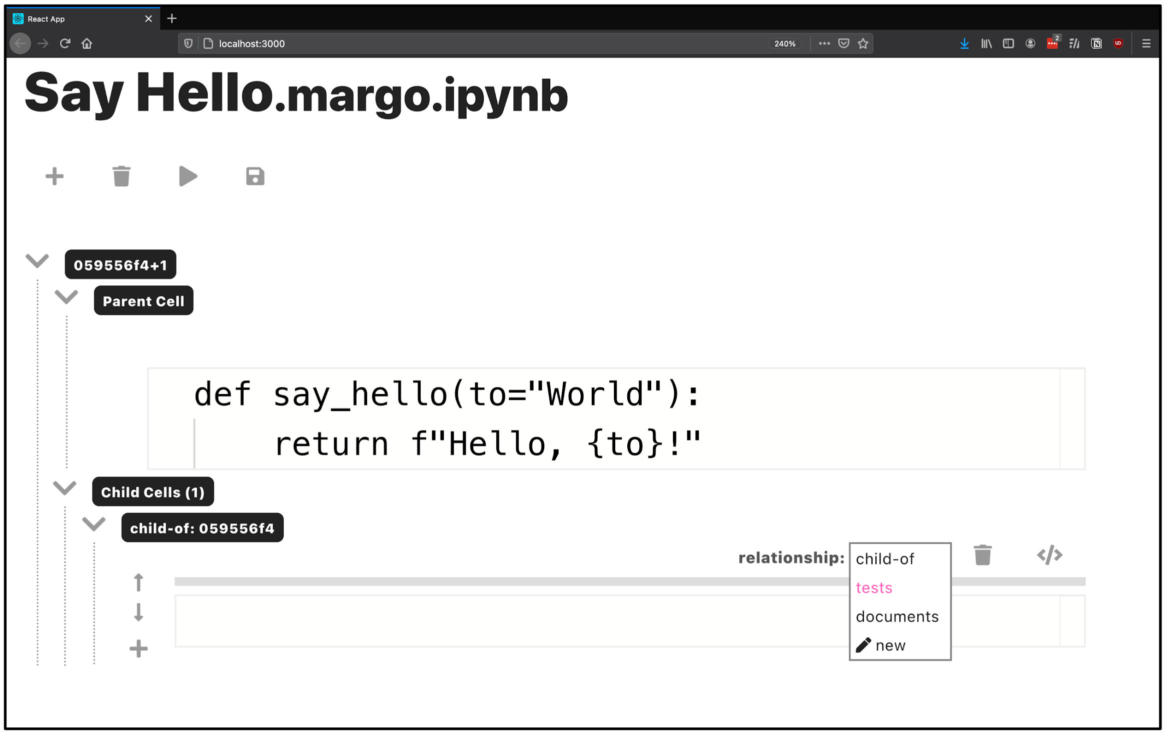
Code is entered into the cell's editor area. In this case we define a say_hello function.



```
def say_hello(to="World"):
    return f"Hello, {to}!"
```

Figure 24. Margo Editor, Add a Child Cell

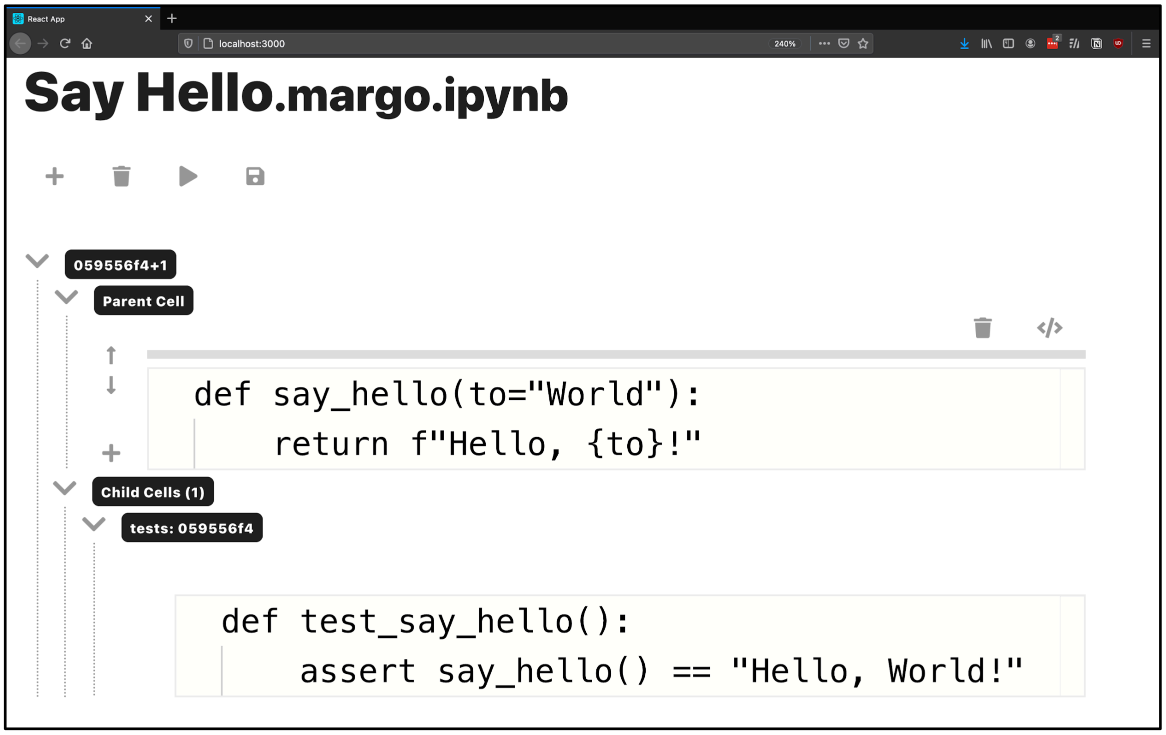
A child cell may be added to a parent cell using the parent cell's "add cell" plus sign icon.



```
def say_hello(to="World"):
    return f"Hello, {to}!"
```

Figure 25. Margo Editor, Set Relationship Type

The relationship label indicating the child's relationship to the parent may be selected from a list of options, or a new label may be added to the list. In this case we set it to "tests" because we intend to write a unit test for the say_hello function.



```
def say_hello(to="World"):
    return f"Hello, {to}!"
```

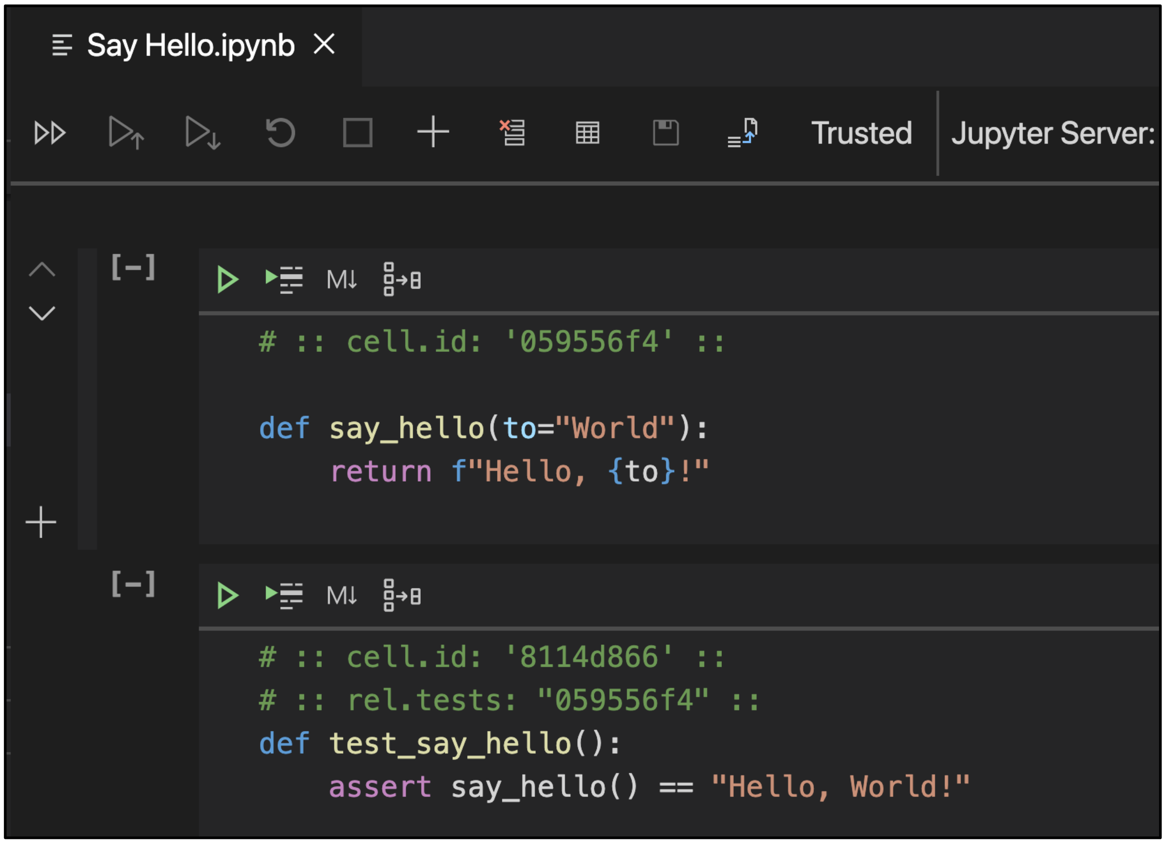
```
def test_say_hello():
    assert say_hello() == "Hello, World!"
```

Figure 26. Margo Editor, Write a Unit Test

We finish editing the notebook by entering the unit test into the child cell's editor.

Inspecting Margo Editor Notebook

In this section we open the document produced by Margo Editor in a conventional notebook editor, which in this case is Microsoft Visual Studio Code. We see that it opens and renders properly. The cell relationships we defined through Margo Editor UI interaction are exposed as Margo notes in Visual Studio Code, which of course lacks support for these notes and displays them as ordinary comments. Because the notes are embedded in code comments, they have no effect on the notebook's operation in Visual Studio Code.



```
# :: cell.id: '059556f4' ::

def say_hello(to="World"):
    return f"Hello, {to}!"
```

```
# :: cell.id: '8114d866' ::
# :: rel.tests: "059556f4" ::
def test_say_hello():
    assert say_hello() == "Hello, World!"
```

Figure 27. Inspecting Margo Editor Document

Both cells have a cell.id Margo assignment, with a value assigned automatically by Margo Editor. Cell 2 has a second Margo note that indicates it tests Cell 1.

Future Work

Margo Editor was not intended to demonstrate an ideal user interface. We can begin to imagine how this concept of serializing additional data on top of notebooks could support more useful and innovative user interfaces in an IDE or notebook authoring tool.

<u>Test Status Indication</u>

One improvement to this proposed interface would be a live test status indicator, as illustrated below.

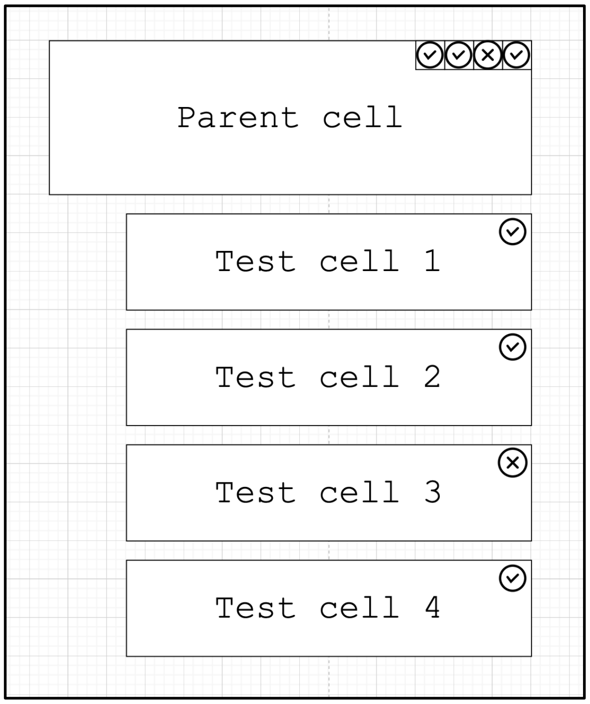


Figure 28. Test Status Indication

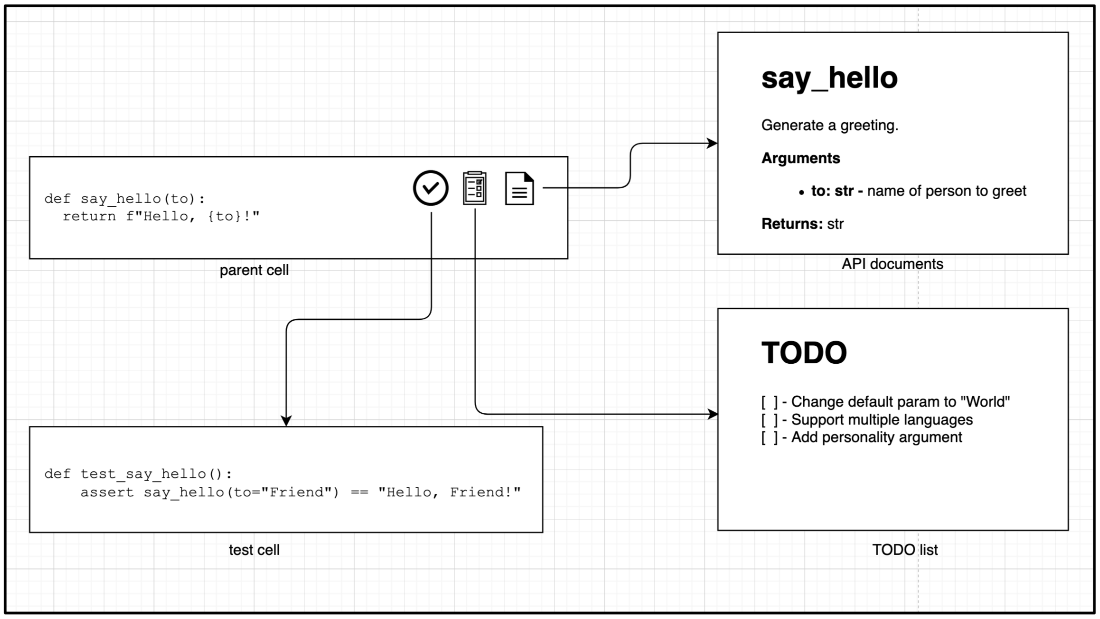
In this figure, tests 1, 2, and 4 are passing; test 3 is failing.

Augmenting IDE Code Navigation

Modern IDEs can infer a great deal about code and support navigation features that reduce the developer's need to manually browse the file system. One example is selecting any named entity and choosing to "jump to" its definition, even if it appears in a different file. Another is an advanced form of autocomplete that can suggest entities which have not been imported into the current context; when you choose one of these, any required import statements may be automatically added to the top of the file.

These tools are very useful, in large part because they allow programmers to focus more on the code without "leaving" the editor to browse the file system. However, they are limited to support relationships between code that can be inferred. With Margo, relationships can be explicitly stated, and even those that are not possible to infer may be supported by these navigation features.

For example, with the cell relationship scheme we have defined, it would be possible to show or hide cells by default based on their relationships, and then display them only through UI interactions similar to autocomplete or "jump to definition." These hidden cells might be a TODO list or CHANGELOG relating to a source code cell, or its API documentation. The following figure shows how a parent cell with three child cells could be represented in such a user interface.



```
def say_hello(to):
  return f"Hello, {to}!"
```

```
def test_say_hello():
    assert say_hello(to="Friend") == "Hello, Friend!"
```

say_hello

Generate a greeting.

Arguments

- **to: str -** name of person to greet

Returns: str

TODO

[ ] - Change default param to "World"
[ ] - Support multiple languages
[ ] - Add personality argument

Figure 29. Augmented Code Navigation

Code navigation that uses explicit rather than implicit relationships between code may be useful for development of IDEs. In this figure, we see a user interface with icons that correspond to a unit test, a "todo" list and a documentation code cell, and clicking each icon opens a view of its corresponding cell.

Chapter IX.

Results

9.1 Overview

In the preceding chapters we have discussed a practical concept of margin notes, developed a syntax for notes called Margo, and demonstrated that Margo can be a useful tool in extending Jupyter Notebooks. We specified the general syntax in Chapter IV, "Margo Specification" and implemented a parser in Python in Chapter V, "Margo Parser."

One way we wanted to extend notebooks was to create a useful implementation of modular notebooks that improves upon prior work. We achieved this in Chapter VI, "Margo Loader." This application allows more lines of code to be written in Jupyter Notebooks without having to leave the notebook authoring environment in order to write modular code. This means more of a methodology can be written in Jupyter Notebooks alone, and the software architecture can be more robust than is possible in monolithic notebooks. Later in this chapter, we will explore a case study that demonstrates modular notebooks to produce software entirely in notebooks for the purpose of a digital humanities research project.

In Chapter VII, "Margo Tool," we demonstrated that Margo can be used to arbitrarily create APIs into notebooks and integrate them with tools such as pip and Make. The idea of creating these ad hoc interfaces to integrate notebooks with other tools provides a great deal of potential for extending the use of notebooks.

In Chapter VIII, “Margo Editor” we demonstrated that Margo may be used to serialize arbitrary data on top of notebooks to support features not supported by the standard notebook format.

In sum we find that Margo is both an elegant and flexible way to enhance notebooks. In the following section we will take a look at a more complete case study of modular notebooks, and we find that we are able to write far more expressive research software entirely in notebooks.

9.2. Modular Notebooks Case Study

In this section we consider a case study for the modular notebooks made possible by Margo Loader: We consider a data processing project by the Digital Humanities Lab (DHLab) at Yale University Library, where this author works as a developer.

This case study is manifested in a body of software that is as significant as any of the software applications developed in Chapters V-VIII and is as substantive a portion of this thesis submission. This software, written entire in notebooks, is available online both as an archive and as a live demo that may be run in the browser on a cloud-hosted virtual machine. See Appendix 1 for links to these online resources.

Digital Humanities Project Overview

In the DHLab, staff software developers work with humanities researchers to translate research questions into computational approaches. In one project, this author worked with a researcher who wanted to extract small 20-color palettes to summarize prints by the artist William Blake.

The processing of images involved at least three component tasks: 1) removal of large areas of paper without ink or paint applied; 2) clustering pixels to form the palette; 3) reconstructing images with the reduced color palettes to demonstrate how the chosen algorithm "sees" the image.

An important aspect of this project was communication with the researcher. There were different ways to approach the component tasks, and so it was necessary to demonstrate several implementations and their outputs. For example, there is no single correct way to extract what human observers would consider the dominant colors of an image. Some algorithms miss small pops of color that might be very important to the print but make up only a very small area of the whole. Some algorithms might produce results that better represent how the human eye sees color — our eyes see colors relatively, so two areas of the same RGB-value might look quite different in two different areas of the same print. It was important to implement palette extraction using several different algorithms and allow the researcher to play a role in deciding which produced the most faithful color summaries for this particular data set.

In addition to collaboration with the researcher, it is also a priority to produce readable methodology documents for that researcher's work, as well as for future researchers, students and any other developers who may be interested in part or all of the computational approach we developed.

Implementation in Notebooks

Because of these needs — communication with the researcher and producing methodology documents — Jupyter Notebooks were a preferred medium for as much of the code base as possible. The development cycle of such a project in the DHLab often includes all three notebook use cases discussed in Chapter I, Section 5: scratch programming, production pipeline, and sharing. At first, in the scratch phase, the developer experiments in notebooks to rapidly iterate on different approaches to data processing. This may involve showing notebooks with outputs to other stakeholders for input and direction. Then when an approach is determined based on a small subset of the entire data set, the code is moved into a production pipeline to process all of the data. This means software is often modularized outside of notebooks. When the processing is finished and the research is published, the sharing phase means making code available for inspection, in a combination of notebooks and plain source modules.

At the time Margo Loader did not exist, so code written in notebooks during scratch phases was removed to plain source modules to be incorporated into a software system that would process thousands of images. The result is that the actual software used in production is not entirely available in notebooks. This is less than ideal because we consider notebooks more readable for programmers and non-programmers alike who may wish to inspect the methodology.

Refactoring in Notebook Modules

Since developing Margo Loader, this author has refactored this color extraction workflow in modular notebooks. Four notebooks each implement and demonstrate a different color extraction algorithm; another notebook demonstrates how to remove background pixels based on lightness; and another notebook coordinates these and other notebook modules into a pipeline for processing images at scale.

Writing the methodology as a system of modular notebooks had many advantages.

1. First, more of the methodology — measured by lines of code — is captured in the preferred notebook format than in plain source modules.
2. Second, there is no need to move code from notebooks to plain source files when going from a “scratch” use case to a “production” use case and then back into notebooks for a “sharing” use case.
3. Third, using the same set of notebooks in production and in sharing means that the methodology documents are a complete and faithful representation of the methodology, not just components that were selectively chosen for demonstration in notebooks.

4. Fourth, different modules can be focused on in their own individual notebooks. Readers may better understand the system as a whole by understanding its components in isolation. This is what Dijkstra (1982) called "being one-and multiple-track minded simultaneously."
5. Fifth, readers may only be interested in a single component, even one not used in the final production pipeline, such as the K Means clustering notebook, which they can read without any of the rest of the methodology. In this way, each module becomes its own methodology document.

In addition to the source code repository and live demonstration links provided above, screenshots of notebooks from this case study are available in Appendix 2 and Appendix 3. In Appendix 2, a notebook called "ExtractorBaseClass.ipynb," defines an abstract base class for a color palette extractor, and then demonstrates how to implement a subclass. In Appendix 3, the notebook called "MMCQExtractor.ipynb" imports the first notebook and defines a subclass that extracts colors using a median cut algorithm.

9.3 Future Work

In this thesis we have designed and implemented a general syntax for margin notes as well as a number of applications. It is our hope that many more will be developed. In this section we describe some more ideas that could warrant further research.

Linked Notebook Documents

One challenge of the Jupyter Notebook format is that execution output is stored as Base64-encoded binary data inside the document's JSON structure. This can lead to very large file sizes, as well as files that cannot be usefully version controlled using conventional version control software like Git. Using Margo, a notebook authoring tool could store cell outputs in separate files and link to them with Margo notes, such as:

```
# :: cell.output : <path_to_file_1> ,
# ::               <path_to_file_2> ::
```

This effectively raises the possibility of "linked" notebook documents, where one notebook is represented across multiple files.

Beyond Notebooks

In addition to *.ipynb* files, Margo Loader supports Jupytext notebooks stored in file names ending in *.nbpy*. This is merely an Easter egg at the moment and is not discussed in the Margo Loader chapter. This means that Margo Loader's directives can be used with plain source code files already. This thesis is concerned only with notebooks, but Margo could more generally be considered a standard format for overloading source code comments. This may be useful considering there are so many non-standardized, machine-readable code comment formats (as described in the section 1.7, “Overloading Comments”).

Appendix 1.

Online Resources

The software developed over the course of this thesis project is as or more important a contribution than this paper. All source code submitted with this thesis is available online, and where possible, live versions of software that run in a web browser without installation are available online as well.

An organization page on GitHub contains all of the software applications developed for this thesis at https://github.com/margo-notebooks. This is where future development, news, documentation and tutorials will be collected. Keyword proposals are organized at https://github.com/margo-notebooks/margo-keywords.

Each software deliverable discussed in the preceding chapters has two online resources linked in the table below: 1) a GitHub repository, representing the ongoing state in that project's development; and 2) a permanent, citable archive of the repository at the time of submission.

Table 2. Online Resources, Source Code

Margo Parser	• https://github.com/margo-notebooks/margo-parser-py • https://doi.org/10.5281/zenodo.4554408
Margo Loader /Margo Tool	• https://github.com/margo-notebooks/margo-loader-py • https://doi.org/10.5281/zenodo.4554406
Margo Editor	• https://github.com/margo-notebooks/margo-editor-ts • https://doi.org/10.5281/zenodo.4554404
Modular notebooks case study	• https://github.com/jakekara/color-extraction-methodology/ • https://doi.org/10.5281/zenodo.4554402

Each software deliverable developed for this project has two corresponding resources in the table above. The first link in each row is a reference to the project's ongoing development state on GitHub. The second is a permanent, archived version of the software at the time of this thesis submission.

In addition to these source code repositories, certain components of this project are available to view live in a web browser with no installation. These resources are listed in the table below.

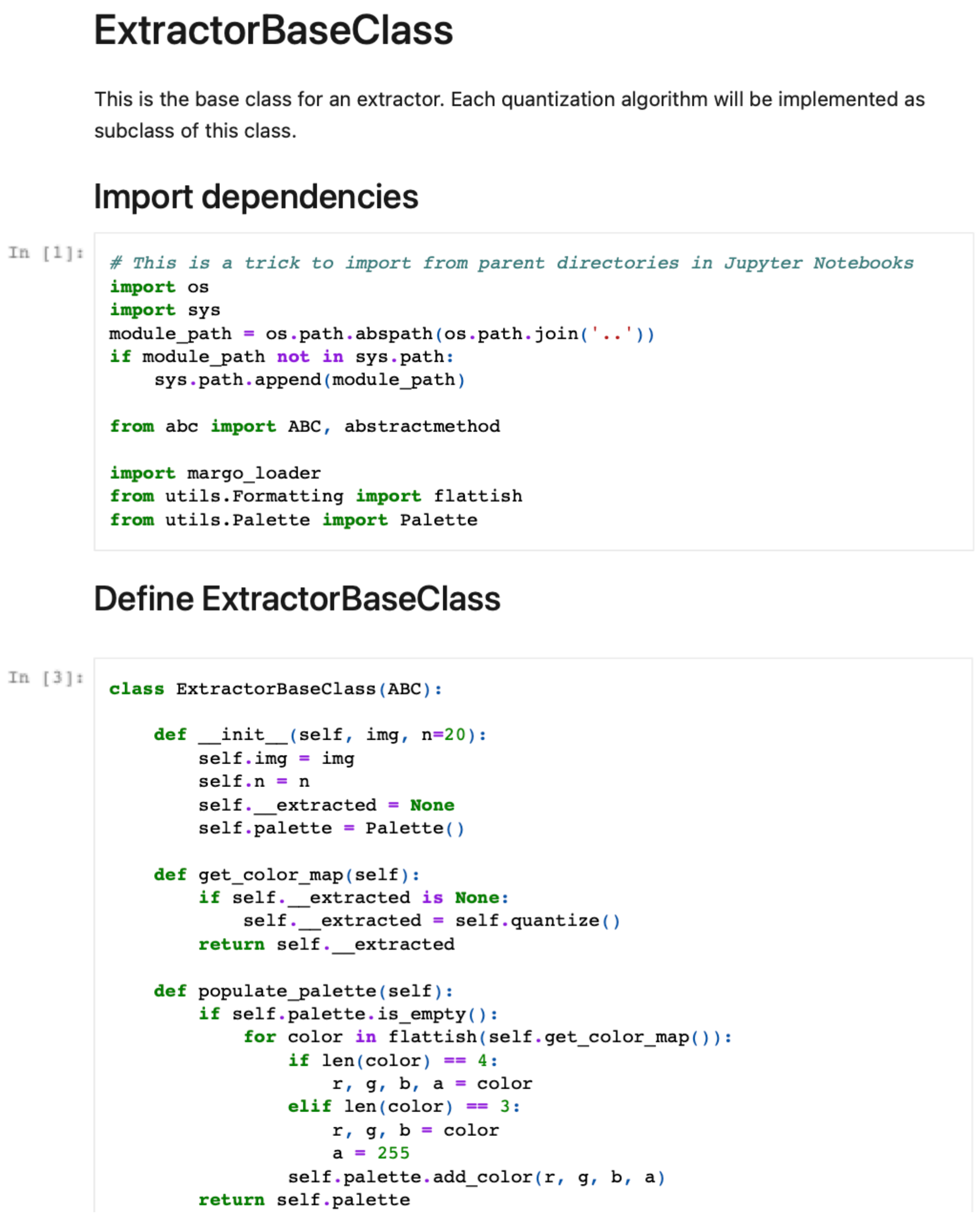
Table 3. Online Resources, Live Demos

Modular notebooks case study	https://mybinder.org/v2/zenodo/10.5281/zenodo.4554402/
Margo Editor	https://margo-editor.netlify.app/

The Modular Notebooks case study runs in the browser with a cloud backend on the MyBinder.org platform. In this environment, all notebooks developed for the case study are available for interactive use.

Appendix 2.

ExtractorBaseClass.ipynb



ExtractorBaseClass

This is the base class for an extractor. Each quantization algorithm will be implemented as subclass of this class.

Import dependencies

In [1]:

```
# This is a trick to import from parent directories in Jupyter Notebooks
import os
import sys
module_path = os.path.abspath(os.path.join('..'))
if module_path not in sys.path:
    sys.path.append(module_path)

from abc import ABC, abstractmethod

import margo_loader
from utils.Formatting import flattish
from utils.Palette import Palette
```

Define ExtractorBaseClass

In [3]:

```
class ExtractorBaseClass(ABC):

    def __init__(self, img, n=20):
        self.img = img
        self.n = n
        self.__extracted = None
        self.palette = Palette()

    def get_color_map(self):
        if self.__extracted is None:
            self.__extracted = self.quantize()
        return self.__extracted

    def populate_palette(self):
        if self.palette.is_empty():
            for color in flattish(self.get_color_map()):
                if len(color) == 4:
                    r, g, b, a = color
                elif len(color) == 3:
                    r, g, b = color
                    a = 255
                self.palette.add_color(r, g, b, a)
        return self.palette
    def get_palette(self):
        if self.palette.is_empty():
            self.populate_palette()
        return self.palette

    @abstractmethod
    def quantize(self, img, n=20):
        raise Exception("Not implemented")
```

In [5]:

```
# :: ignore-cell ::
from utils.ImageFiles import read_img
from PIL import Image

image = read_img("../images/sample.jpg", scale=0.125)

ebc = DoNothingExtractor(image)

Image.fromarray(ebc.get_color_map())
```

Out[5]:

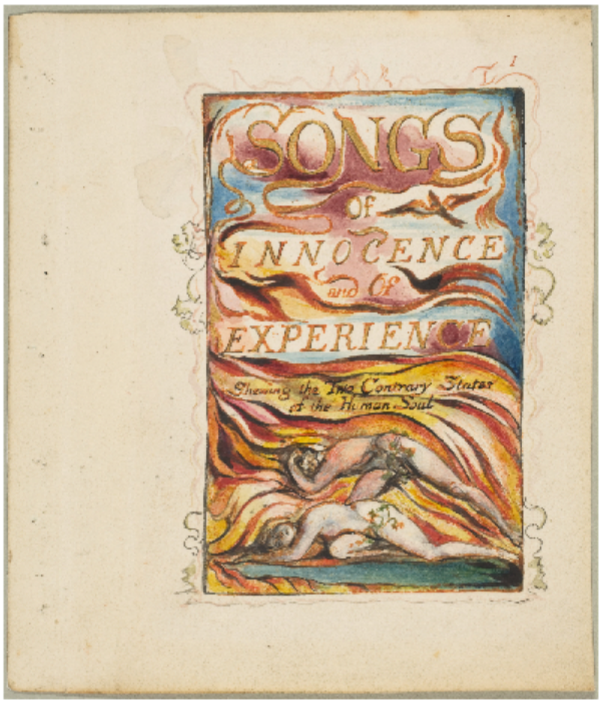


Figure 31. ExtractorBaseClass Notebook

The print included in this notebook is "Songs of Innocence and of Experience, Shewing the Two Contrary States of the Human Soul: Combined Title Page" by William Blake, ca. 1825. This work is in the public domain.

Appendix 3.

MMCQExtractor.ipynb

MMCQExtractor

Median cut extractor using ColorThief's MMCQ implementation. The colorthief version we're using is one that I forked to expose the MMCQ object.

Import dependencies

In [1]:

```
# This is a trick to import from parent directories in Jupyter Notebooks
import os
import sys
module_path = os.path.abspath(os.path.join('..'))
if module_path not in sys.path:
    sys.path.append(module_path)

from colorthief import MMCQ
import cv2
import numpy as np

import margo_loader
from utils.Formatting import flattish, is_visible
from utils.NotebookRenderPalette import notebook_render_palette
from color_extraction.ExtractorBaseClass import ExtractorBaseClass, Palette

from pathos.multiprocessing import ProcessingPool as Pool, cpu_count
```

Define the MMCQExtractor

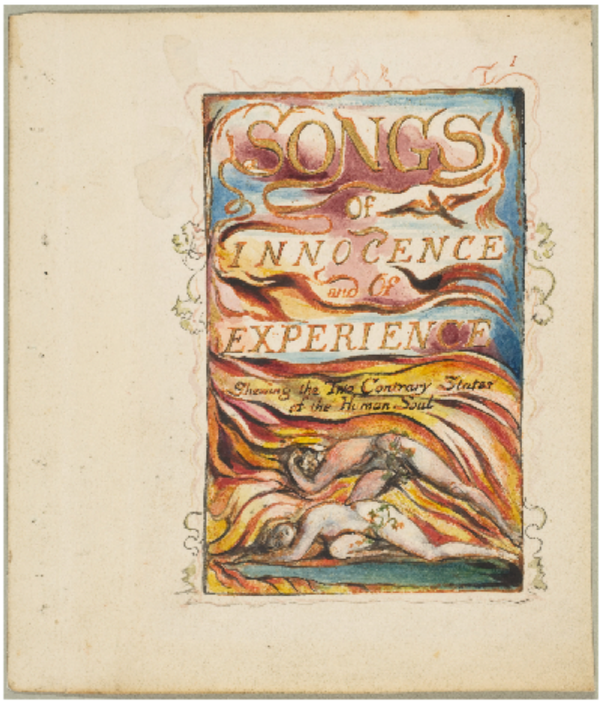
In [2]:

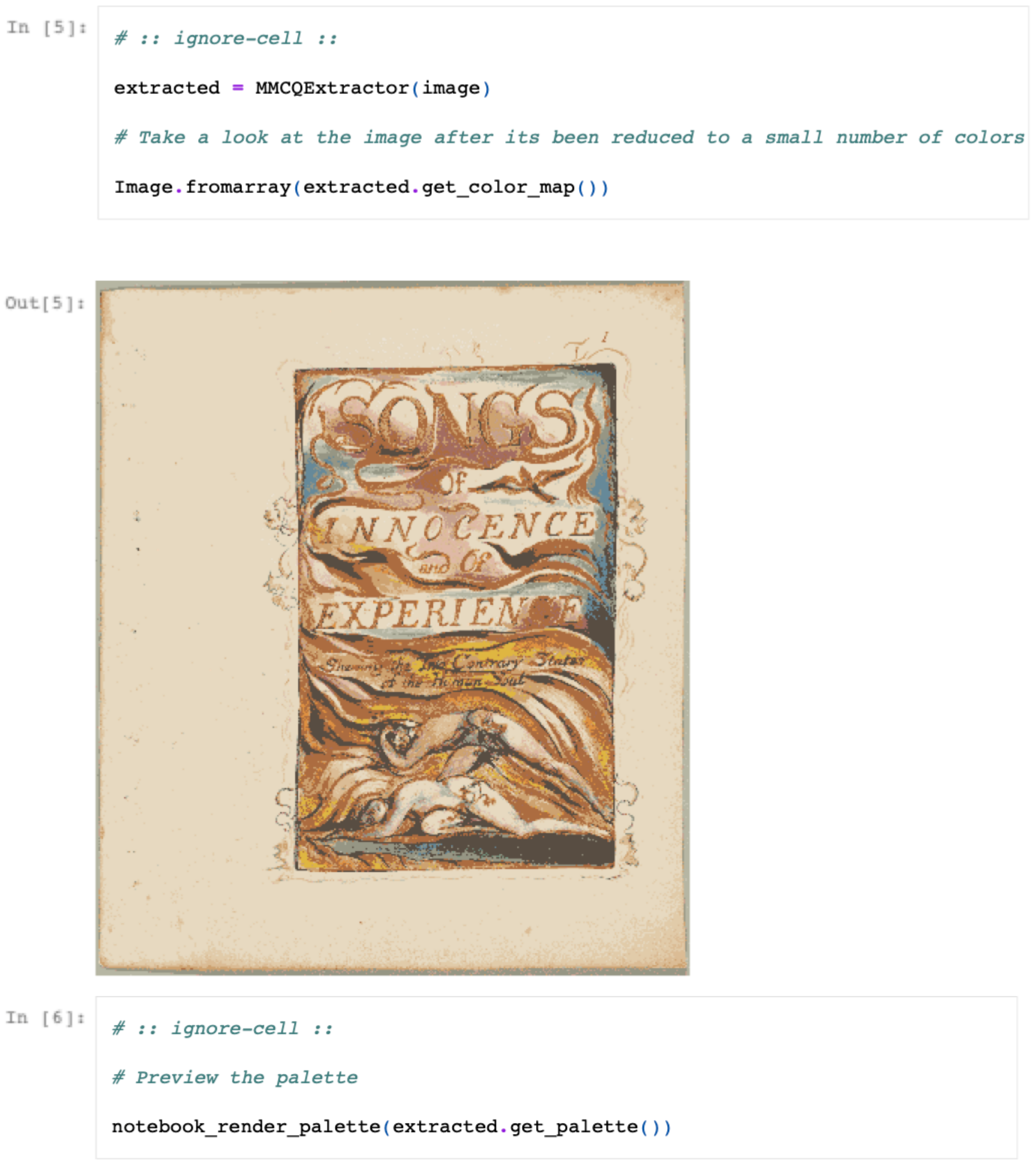
```
def map_nearest(cmap, pixels, process_count):
    with Pool(process_count) as p:
        return p.map(cmap.nearest, pixels)

class MMCQExtractor(ExtractorBaseClass):

    def __init__(self, img, n=20, process_count=int(max(1, cpu_count() / 2)))
        super(MMCQExtractor, self).__init__(img, n=n)
        self.process_count = process_count
        self.cmap = None

    def populate_palette(self):

        # Overrides parent method to take advantage of colorthief's MMCQ cmap
        # keeps track of color count internally. Prevents iterating over all
        # to count them

        if not self.palette.is_empty():
            return

        if self.cmap is None:
            self.get_color_map()

        for i in range(self.cmap.vboxes.size()):
            vbox = self.cmap.vboxes.peek(i)
            r, g, b = vbox["color"]
            count = vbox["vbox"].count
            self.palette.set_color_count(r, g, b, 255, count)
```

In [3]:
```
# :: ignore-cell ::

from PIL import Image
from utils.ImageFiles import read_img
```

In [4]:
```
# :: ignore-cell ::

# Before quantization

image = read_img("../images/sample.jpg", scale=0.125)

Image.fromarray(image)
```

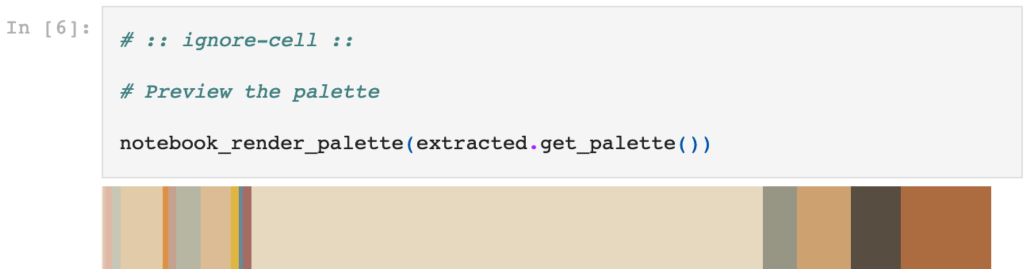
Out[4]:




```
In [5]: # :: ignore-cell ::

        extracted = MMCQExtractor(image)

        # Take a look at the image after its been reduced to a small number of colors

        Image.fromarray(extracted.get_color_map())
```

```
Out[5]:
```

```
In [6]: # :: ignore-cell ::

        # Preview the palette

        notebook_render_palette(extracted.get_palette())
```



```
In [6]: # :: ignore-cell ::

        # Preview the palette

        notebook_render_palette(extracted.get_palette())
```

Figure 31. MMCQExtractor Notebook

The print included in this notebook is "Songs of Innocence and of Experience, Shewing the Two Contrary States of the Human Soul: Combined Title Page" by William Blake, ca. 1825. This work is in the public domain.

References

Calliss, F. W. (1991). A comparison of module constructs in programming languages. *ACM SIGPLAN Notices*, *26*(1), 38-46.

Cooper, Mendel. (2014). Advanced Bash-Scripting Guide: Chapter 2. Starting Off with a Sha-Bang.
Retrieved from: https://tldp.org/LDP/abs/html/sha-bang.html

Coghlan, Nick, Warsaw, Barry, Goodger, David, Hylton, Jeremy. 2001. PEP 1: PEP Purpose and Guidelines
Retrieved from: https://www.python.org/dev/peps/pep-0001/

Dijkstra, E. W. (1982). On the role of scientific thought. In Selected writings on computing: a personal perspective (pp. 60-66). Springer, New York, NY.

Head, A., Hohman, F., Barik, T., Drucker, S. M., & DeLine, R. (2019, May). Managing messes in computational notebooks. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (pp. 1-12).

Jupyter Development Team. (2015). "The Jupyter Notebook Format"
Retrieved from: https://nbformat.readthedocs.io/en/latest/

Jupyter Team. (2015). Importing Jupyter Notebooks as Modules.
Retrieved from: https://jupyter-notebook.readthedocs.io/en/stable/examples/Notebook/Importing%20Notebooks.html

Kery, M. B., Horvath, A., & Myers, B. A. (2017, May). Variolite: Supporting Exploratory Programming by Data Scientists. In *CHI* (Vol. 10, pp. 3025453-3025626).

Kery, M. B., Radensky, M., Arya, M., John, B. E., & Myers, B. A. (2018, April). The story in the notebook: Exploratory data science using a literate programming tool. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (pp. 1-11).

Knuth, D. E. (1984). Literate programming. *The Computer Journal*, *27*(2), 97-111.

Lemburg, Marc-André. von Löwis, Martin. (2001, June). PEP 263: Defining Python Source Code Encodings).
Retrieved from: https://www.python.org/dev/peps/pep-0263

Microsoft. (2015). Internet Explorer for Developers: About conditional comments. Retrieved from: https://docs.microsoft.com/en-us/previous-versions/windows/internet-explorer/ie-developer/compatibility/ms537512(v=vs.85)

Oracle. (2020). javadoc - The Java API Documentation Generator. Retrieved from: https://docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html

Perez, F. (2012). The IPython notebook: a historical retrospective. Retrieved from: http://blog.fperez.org/2012/01/ipython-notebook-historical.html

Perez, F. (2020). Keynote. JupyterCon.

Perez, F., & Granger, B. E. (2007). IPython: a system for interactive scientific computing. *Computing in Science & Engineering*, *9*(3), 21-29.

Perez, F., & Granger, B. E. (2015). Project Jupyter: Computational narratives as the engine of collaborative data science. *Retrieved September*, *11*(207), 108.

Pimentel, J. F., Murta, L., Braganholo, V., & Freire, J. (2019, May). A large-scale study about quality and reproducibility of jupyter notebooks. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)* (pp. 507-517). IEEE.

Python Software Foundation, The. (2020). importlib – The implementation of Import. Retrieved from: https://docs.python.org/3/library/importlib.html

Seal, Matthew. (2019). Notebooks as Functions with Papermill. [Video presentation]. *Data Council.* Retrieved from: https://www.datacouncil.ai/talks/notebooks-as-functions-with-papermill

Shinan, Erez. 2021. Lark - a parsing toolkit for Python. [source code repository] Retrieved from: https://github.com/lark-parser/lark

Sturm, Gregor. 2019. README.md, nbimporter. [source code repository]. Retrieved from: https://github.com/grst/nbimporter

van der Hoek, A., & Lopez, N. (2011, March). A design perspective on modularity. In *Proceedings of the tenth international conference on Aspect-oriented software development* (pp. 265-280).

Wouts, Marc. (2018). Introducing Jupytext. *Towards Data Science.* Retrieved from: https://towardsdatascience.com/introducing-jupytext-9234fdff6c57